A Pattern Language for Human-Computer Interface Design

Jenifer Tidwell

Preface: The Case for HCI Design Patterns

Twenty years ago, Christopher Alexander shook the architectural world with his landmark book *The Timeless Way of Building.* His thesis was that one could achieve excellence in architecture by learning and using a carefully-defined set of design rules, or patterns; and though the quality of a well-designed building is sublime and hard to put into words, the patterns themselves that make up that building are remarkably simple and easy to understand by laymen.

The patterns that he and his colleagues defined -- published in a second volume, *A Pattern Language* -- are an attempt to codify generations of architectural wisdom. They are not abstract principles that require you to rediscover how to apply them successfully, nor are they overly specific to one particular situation or culture. Instead, they are somewhere in between: a pattern describes possible good solutions to a common design problem within a certain context, by describing the invariant qualities of all those solutions.

For example, he recommends using the "Entrance Transition" pattern with homes or any other building that "thrives on a sense of exclusion from the world." The pattern describes what one must do to a doorway so that someone entering it feels as though they are coming into a private, safe space:

> "Make a transition space between the street and the front door. Bring the path which connects street and entrance through this transition space, and mark it with a change of light, a change of sound, a change of direction, a change of surface, a change of level, perhaps by gateways which make a change of enclosure, and above all with a change of view." (From *A Pattern Language*, pg. 552.)

Note that the pattern is not just proscriptive. It describes something positive, something you can try to build, even though you would naturally vary it according to the particular situation. It doesn't simply say, "Never build a doorway without a change of level." Note also that it carries values -- the value of a private space, the value of emotional comfort. Alexander's goal is not to make a building which is merely trendy, or efficient, or even good-looking; he is looking for ways to create a genuinely good experience for people, via their built environment.

In recent years, parts of the software engineering community have enthusiastically embraced the patterns concept, due in no small part to the 1995 book *Design Patterns,* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Like the Alexandrian patterns, these patterns of object-oriented software provide design solutions that are concrete enough to immediately put into practice, with good results, and yet are sufficiently abstract to apply to countless situations, limited only by the imagination and skill of the pattern user.

We badly need the benefits of such a pattern language in the field of HCI design.

The Need for a Human-Computer Interface Pattern Language

There is plenty of good literature out there on the high-level principles of good interface design, and it is getting ever better as this young field matures. We all know by now that we ought to use direct manipulation, immediate feedback, proper affordances, judicious use of sound and animation, protection from accidental mistakes, gentle error messages, and so on. But if you're a novice designer, it's hard even to remember all these principles, let alone use them effectively! And it's difficult sometimes to make the tradeoffs among these principles when they come into conflict; we often have to figure out the best solution by guessing, or by resorting to other means.

1. Test it with users

One excellent way to verify your guesses, of course, is to test your design with potential users. Lots has been written on usability testing and other field methods, and it's all important. Before the design phase begins, we must understand our users' concerns and learn to empathize with them; their feedback guides and inspires us while we explore different design possibilities; and late in a project, they help us refine and build the chosen design. In the field of human-computer interfaces, we have learned -- faster than in many other fields -- the value of an iterative design process that directly involves the end users.

But how do you come up with those initial designs? Once you understand where the user is coming from and what the artifact needs to do, what comes next? What further questions do you ask? What assumptions should you make? How do you put it all together into a design that might work? This creative leap is always harder than it sounds. And it costs us far less, in terms of time and usability testing, to make good guesses and design choices right at the beginning.

2. Follow the style guides

Then there are GUI style guides, both the toolkit-standard ones and custom company-wide style guides. They work fine if you want your company's applications to all look and behave just so, or if you want to make sure you're following the accepted conventions of the toolkit you happen to be working with. Sometimes it's important to know these details. But they are transient -- toolkits, trends, and operating systems come and go, and as soon as the world gets comfortable with one, another arises to take its place. Remember the transition from Windows 3.1 to Windows 95?

Furthermore, by constraining yourself to the relatively small number of tools that most toolkits give you, and to the ways of using them that convention dictates, you limit the expressiveness of your interface to that which is currently acceptable. ("I used a combo box there because that's what everyone does.") And no style guide can infallibly tell you how to strike a balance between two opposing high-level principles.

3. Do what other people do

I have seen inexperienced user interface designers work through design decisions by depending on other people's designs, rather than on their own design skill. They ask themselves questions like, "What techniques or layouts have I seen lately that do what I'm trying to do?" or even "What do the standard Microsoft packages do?" This approach isn't that bad, really -- observation of successful interfaces is part of the learning process, and at least they're not trying to reinvent everything from scratch. (The worst user interfaces reinvent everything in bizarre ways. So do the best ones; take a look atKai's Power Tools.)

But this can be a scattershot approach. The designer's experience may be limited to software of certain types, or made by certain companies, and they may not be closely related to the kind of software being designed. Furthermore, the other interfaces that the designer draws from may not be good ones in the first place. So the designer ends up with an impoverished decision-making ability, despite all good intentions.

(A familiar scenario occurs if the designer is really a software developer by vocation. His or her sphere of experience probably includes mostly developers' tools and the Web, neither of which may have anything to do with the user interface they are designing. We've all seen the results!)

Of course, experienced designers don't entirely escape this mode of thinking, either. Reinventing techniques isn't really practical most of the time -- consciously or subconsciously, they apply what they know, and reuse good solutions they've seen before. The difference lies partly in the depth of experience from which they draw: a seasoned designer has seen, analyzed, or built interfaces of many diverse kinds. And it also lies in the skill with which they apply that experience. They don't clumsily or timidly copy a technique, afraid they'll somehow ruin it by changing it; rather, they understand the design principles and process enough to confidently adapt a good idea to a new use in a new context. They understand what works and what doesn't -- the common ground -- across different media and contexts.

How can the HCI community help inexperienced designers move away from clumsy designs and labor-intensive processes towards this state of confidence and skill, without spending years learning it all the hard way?

To begin with, we could start building a human-computer interface pattern language. A language of this sort is a set of interrelated patterns, which share similar assumptions, terminologies, and contexts. At its best, such a language would both aid individual interface designers in their day-to-day work (as the *Design Patterns* book clearly does for many software engineers), and also help the whole industry develop better tools and paradigms.

More specifically, it would help individuals build better interfaces by:

- Capturing the collective wisdom of other designers in a way that can be immediately used, even by less-experienced designers. When difficult design problems arise, and there are conflicts between basic design principles, a pattern solution may be found that is appropriate for that particular context.
- Giving us a common language that we can speak with our fellow designers, with our interdisciplinary design teams, and with our customers. Participatory design may especially benefit when designers and users can talk about the same concepts, in the same terms, with fewer misunderstandings.
- Allowing one to think "outside the toolkit," by creatively applying familiar patterns in new ways. One of the great paradoxes of design is that one's creativity is often improved by imposing constraints on what one may create. By constraining a designer to work within the pattern, but with flexibility in visual appearance and interaction details, new specific solutions may emerge that are better than those commonly found in today's software.
- Helping to keep one focused on essential values, such as simplicity, fidelity to a consistent model, aesthetic beauty, and emotional comfort.
- Expressing design invariants that can be encoded in software, while allowing the specific solution details to vary as necessary (through design/test/evaluate iterations, for example, or over time as software slowly evolves through different versions). It's too much to expect that all patterns will work this way, since the point is good design, not good programming; but sometimes it works out.

Likewise, a good pattern language can benefit the HCI design community:

- It would be a new, more meaningful vocabulary for talking about why certain interfaces do or don't work. We talk now about the wonders of wizards, popup menus, and combo boxes, when they almost certainly won't be around in twenty years -- but the patterns behind them, that make them work, will still be valid (and were already valid a century ago, in some cases).
- It would enable us to more methodically draw on expertise in related fields, such as book design, consumer electronics, the design of control panels (for cars, airplanes, power plants), video games, the Web and hypertext, and speech-driven interfaces. If a pattern holds true in both UI design and another field, we can learn the specifics of implementation in that other field and imagine how we could apply them to our interfaces. Of course, many of us have already done that, though without the use of the term "patterns" -- but imagine how much more effective we could be if we think about it at the right level of abstraction!
- By isolating the qualities that make certain trendy metaphors, idioms, and widgets work so well, we can learn from them and then move beyond them, without losing their lessons from our collective memory. How many excellent interface ideas have been lost because they became unfashionable, or because they never gained wide acceptance due to economic forces or bad implementations?
- They may serve as a solid practical foundation on which to build new user interface tools or concepts, such as virtual reality, sound-based interfaces, or next-generation Web sites. If we know what patterns work in other diverse fields, then we can better direct our efforts towards creating good artifacts in the new ones. It would be a shame to start developing a new design idea, say a new 3D interface, and waste vast amounts of effort rediscovering the same good old design patterns that have always worked in the familiar 2D spaces.

A Sample Pattern Language

The pattern language presented here is merely a start. It does not yet fulfill all the above goals, though the patterns were developed with them in mind. The ones defined here need refinement, and more patterns should be added over time, since this is far from a complete set.

Each pattern description defines a context of use, a problem the designer needs to solve, a set of "forces" pushing the designer in different directions, and a primary rule -- and sometimes additional secondary rules -- on how those forces might be resolved to best solve the problem. Examples are also provided, both good and bad;

sometimes the bad examples show inappropriate uses of the pattern, and other times they show a situation in which the pattern should have been used but wasn't.

Note that the pattern names and problem descriptions avoid the use of GUI-centric terms whenever possible (e.g. mice, menus, dialogs), so that you may more easily think about them being used outside the GUI world. Most of them do work that way. That was a condition of acceptance into this language: if a pattern is invariant across such different forms as paper, hardware, video games, and desktop GUIs, there must be truth in it. In fact, some patterns, such as User's Annotations, are not even in common usage yet in desktop GUIs.

Please read these patterns actively! Think about other examples that you might have seen, both from the world of desktop GUIs and from other fields. Consider how you would use them to design a new interface, or redesign an existing one (VCRs almost always provide entertaining cases of poor design). Look at an interface you like, and see if what you like about it can be captured by some of these patterns -- keep in mind that a pattern language can serve not only as rules for building a design, but also as a system for deconstructing an artifact and classifying its pieces. Finally, imagine how you might apply the pattern in a fully three-dimensional interface, or in a "Star Trek" interface, or some other new or fantastic technology. Would it work there? Why or why not?

Christopher Alexander posits that good patterns improve with time and widespread use. The object-oriented software development community has discovered that this is true, since there are now lots of people in that field developing their own pattern languages and reviewing them with others. The patterns in the original *Design Patterns* book have been augmented and refined, as is done in John Vlissides' *Pattern Hatching*. There is vigorous discussion going on at conferences, in magazine columns, over mailing lists, and in local special-interest groups worldwide.

The HCI design world could start engaging in similar discussions. If you have thought about patterns as they relate to user interface design or development, write about it. If you have additions to or criticisms of the patterns defined here, speak up, so that these can be improved. Read the literature on patterns and develop your own language, in contrast to this one.

Above all, use these patterns if you find them at all helpful. A pattern language is ultimately worth only what its users can get out of it. There is always room for improvement in the design process for conventional GUIs, and recent developments in HTML and Java are giving us the means to build much more creative Web and desktop interfaces than we had in the past, both technically and in terms of user acceptance of "unusual" interfaces.

If the success of patterns in architecture and software engineering is any indication, both our industry and our customers will benefit greatly from this effort.

Comments to: jtidwell@alum.mit.edu
Last modified May 17, 1999

Table of Contents

Introduction

The patterns contained in this work address the general problem of how to design a complex interactive software artifact. They are intended to be used by people who design traditional user interfaces, Web sites, on-line documentation, video games, and other such things. Others who may be interested include people who implement such artifacts, or test them for usability, or manage teams who design and implement them. The language does not attempt to address implementation issues, however.

These patterns are intended to form an Alexandrian pattern language, as found in Christopher Alexander's book *A Pattern Language,* and not a catalog such as is found in the book *Design Patterns*. This means in part that they are intended to be used together synergistically, in a way such that the whole is more than the sum of its parts. Like other such pattern languages, it does not break new theoretical ground or present innovative new techniques -- it's more than likely that you have seen examples of every pattern in here. Instead, it captures ordinary design wisdom in a practical and learnable way.

The intended goal of this pattern language is deliberately broad: to support high-quality interaction between a person and a software artifact. The artifact may support one or more of a broad spectrum of activities, ranging from the most passive -- absorbing information with little or no interactivity -- to the hands-on creation of other objects. Consider some of the varieties of software out there today:

- Traditional desktop GUI software
- Web sites
- Palmtops and handhelds
- Controllers for industrial processes
- Video games
- On-line documentation
- Multimedia educational software
- Speech-based interfaces
- Design tools for graphics and visual arts
- Scientific visualization
- Collaborative, multi-person work spaces

These are incredibly diverse in their specific goals, their degrees of interactivity, their balance between the verbal and the non-verbal, and other factors. But from a designer's perspective, there's more commonalities among them than you may think. For one thing, many of them bear strong similarities to older, more traditional media. Software is very young, and we're still learning how to take full advantage of its unique characteristics; in the meantime, both users and designers carry over what we already understand about print media (books, paper forms, charts), mechanical things (appliances, cars), physical spaces (architecture, urban design), and other real-world artifacts. Therefore, many patterns in this language draw as much from these other realms as from software.

Another similarity among all these kinds of software (and other media) is their basic purpose. The best ones all provide their users with a successful and satisfying experience, by doing one or both of these things well:

1. They shape the user's understanding of something, through a stylized presentation that unfolds the content to the user in an appropriate way. A successful artifact will enable its users to completely understand and effectively use the content being presented.
2. They enable a user to accomplish a task, by progressively unfolding the action possibilities to the user at an appropriate pace as the user interacts with it. A successful artifact will "flow" so well that it lets its users focus entirely upon the task at hand, causing the artifact itself to fade from the user's awareness.

These twin goals, along with corollary issues such as learnability, user empowerment, and enjoyability, define "high-quality interaction" for the purposes of this pattern language. Some artifacts concentrate more upon one aspect than the other, depending upon their context of use. The two aspects are almost orthogonal, but not quite: something which chiefly provides a set of actions still has to present those actions in a comprehensible way, and something which chiefly presents a set of facts or ideas may have to provide the user with actions they need to interact with the artifact.

In both cases, there is an "unfolding" process going on between the artifact and the user. In something which presents facts or ideas (such as a map), that unfolding may be top-down, in which the "big picture" is shown first and the users work their way down into the details as they need. Or, it might be in the form of a fictional narrative, in which the author uses language and character to let the central themes slowly unfold. The basic shape of the content might take the form of one of these "primary patterns:"

- **Narrative.** Examples include works of fiction, nonfiction books, and news articles written in the "pyramid" form. These are usually linear, and usually verbal, but the unfolding process may work in different ways, as described above.
- **High-density Information Display.** Maps, tables, and charts fall under this category. Users may approach one of these with the intent of getting a big picture, or of finding specific details; the patterns chosen to build it should support both.
- **Status Display.** A wall clock, a car's dashboard, and a VCR display are all forms of a status display; they are used to monitor the state of something that will change, and like High-density Information Display, the user must be able to find specific information quickly as well as get a big picture.

In an artifact which presents actions, such as a GUI "wizard," the user may first be presented with a small range of available actions, one of which is taken; then a new set of possible actions is shown, and the user takes one of those; and so on. Alternatively, the range of actions At any one time might be very wide, as with some direct-manipulation interfaces. Ultimately, the artifact lets the user accomplish some principal task, which may require smaller supporting tasks to be accomplished first, which may in turn require still smaller supporting tasks. The sense of "flow" comes from being able to do these with an appropriately small amount of time and effort, so that the user never loses focus on their principal task (as Brenda Laurel discusses in *Computers as Theatre*).

The primary patterns for actions -- their basic shapes -- might include these familiar genres:

- **Form.** Tax returns and catalog orders are typical examples, and interactive forms are very common in computer interfaces. The user is expected to provide preformatted information, often in a linear fashion. The available actions are fairly narrow, prescribed by cultural expectations.

- **Control Panel.** A lightswitch is a very simple example, offering an extremely limited choice of actions; a TV remote control is more complex; a nuclear power plant's control room may reach the limit of a human being's ability to comprehend complexity. Control Panels are used to set the state of one or more things. Cultural constraints often help define the available actions, but as illustrated by the examples, there is a wide range of action availability, so the unfolding process can take place in many ways.
- **WYSIWYG Editor.** A typewriter or word processor, a drawing program, and a forms designer such as Visual Basic are all typical examples. These offer many possible actions to the user at once, including various kinds of direct manipulation, and the results of those actions are usually immediate -- they thus support quick, iterative, creative work that often leads to the aforementioned sense of "flow.'
- **Composed Command.** Command-line-driven software of all kinds, such as the UNIX operating system or software debuggers, fall into this category; so does one person telling another person what to do, or a person verbally addressing a computer (as in Star Trek). The available actions may be extremely broad, especially if natural language is fully used. This pattern is linguistic where WYSIWYG Editor is graphic: both are highly interactive and user-driven, both offer broad ranges of actions with complex unfolding techniques, and both usually provide immediate feedback. These all work to support complex tasks.
- **Social Space.** This includes the on-line equivalents of "real" social spaces -- newsgroups, email lists, chat rooms, and so on. This pattern is unusual in that the artifact is merely a mediator between people, not a direct participant in a dialogue with the user, nor a passive provider of content. Nevertheless, it still reveals content (the conversations taking place) and provides actions (what the user can do in that space), but in a very stylized manner.

These primary patterns form the backbone of this pattern language. They set the tone for an artifact -- when a user identifies an unfamiliar artifact as belonging to one of these (or others not explicitly defined in this language, such as a spreadsheet), he or she tends to make some initial assumptions about its behavior, based on cultural expectations of how these things usually work. ("What does it do? How am I expected to interact with it? What do I do first? How will it react?") At the end of this introduction, there is a description of each primary pattern's "sublanguage," or a set of interrelated patterns that often work well to support that pattern.

Note that the patterns aren't meant to be straitjackets -- for instance, they can be used in combination with each other, such as putting bits of Narrative into the cells of a Tabular Set, or using a Form as an adjunct to a WYSIWYG Editor -- but the boundaries are basically respected by mainstream artifacts, and for good reason. Usability is improved when users' expectations, subconscious or explicit, are followed. On the other hand, experiments, cutting-edge interfaces, and art often make it a point to violate those boundaries. How you deal with the existing boundaries all depends upon your purpose.

The next set of patterns captures ways of unfolding an artifact's content or available actions. Some apply to content, some apply to actions, some to both; there's no reason to be dogmatic about their use.

- **Navigable Spaces** is highly interactive, letting users move through the artifact at their own pace, and in their own directions.

- **Overview Beside Detail** also lets users work at their own pace, but presents a more structured, two-level view of information or actions.
- **Step-by-Step Instructions** prescribes a sharply limited set of actions to the users, usually in a linear fashion, moving progressively through stages.
- **Small Groups of Related Things** loosely organizes visual content into distinct groups, usually done hierarchically, so that a user can see both the "big picture" and fine detail.
- **Series of Small Multiples** organizes visual content into a single (possibly long) series of discrete images, bringing out differences between them by emphasizing their basic continuity; it can be used to implement a High-Density Information Display.
- **Hierarchical Set** is a way of implementing a High-Density Information Display with a strict tree-like organization.
- **Tabular Set** also implements High-Density Information Display, but as a table; again, users can see both the big picture and fine detail.
- **Chart or Graph** is yet a third implementation of High-Density Information Display, but a more graphic one than the other two.
- **Optional Detail On Demand** lets users reveal hidden actions or content at their own discretion.
- **Disabled Irrelevant Things** blocks certain actions or content according to the current state of the artifact, usually driven by user interaction.
- **Pointer Shows Affordance** temporarily reveals possible actions according to the user's explicit focus of attention.
- **Short Description** temporarily reveals content according to the user's explicit focus of attention.

Other categories of patterns describe an artifact's use of space and other resources, navigational techniques, different kinds of actions, the interrelationships between an artifact's "working surfaces," and so on.

There's one thing you should keep in mind about this language, however, that is atypical of other Alexandrian pattern languages. Most of these patterns can be used at many different levels of scale. A **Form**, for example, may be the dominant pattern in one artifact, while being a minor helper task in another. Likewise for a **High-density Information Display** such as a chart or a table. **Small Groups of Related Things** is recursive by definition, much like Composite in *Design Patterns,* and the concept of a working surface is also recursive -- any single surface can be composed of a set of **Tiled Working Surfaces,** for instance.

Because of this scale issue, I haven't yet been able to draw a coherent diagram of the whole language, nor define clear linear paths through it. I am open to suggestions on how best to do this. For instance, the sublanguages are not much more than suggestions, based on which patterns seem to go well with each other; it shouldn't be interpreted as exclusive or proscriptive.

How to Use This Pattern Language

At one level, this language is a way to describe existing artifacts. Using it as such should be fairly simple: read through the language, and pick out the patterns that you see. The problem statements and forces in the pattern descriptions may help you understand how the artifact does what it does, and what tradeoffs its designer may have been considering.

At a much deeper level, you can use this language as a tool to help you design an artifact. Please understand that it is no substitute for creativity or a good process. It is not a silver bullet. To use any pattern language effectively, you must start with an understanding of the artifact's purpose and audience, as is the case in any design methodology. (How can you pick a design solution if you don't know what the relevant factors are?) And to get the most benefit out of it, I believe you must allow the design to progress iteratively. Allow it to grow organically, by weeding out the bad ideas as you go and letting the good ones flourish. Start with the broad strokes and work down into the fine details; with each iteration, check your designs against reality, so that you can discover the bad choices and get rid of them.

(This reality may take the form of the viewpoints of different stakeholders, such as users, technical writers, testers, customer educators, and marketers. If they "speak" the same pattern language you do, so much the better -- let them in on the fun. They might have some excellent design ideas. In my experience, they almost always do.)

It is common in software to start with a "conceptual model," or a model of objects, relationships, behaviors, and states that describes the artifact independently of the user interface. Upon first reading, these patterns may seem to describe just the surface of an artifact -- the way it looks and the way it behaves. But the patterns may also be used to help design the conceptual model behind the interface. For example, content presented as **Navigable Spaces** might be implemented with one "object" per space, with object relationships corresponding to the links between the spaces. A **Hierarchical Set** should be the visible manifestation of a tree structure. A **Form** may present a series of editors for each property of an object, maybe with subforms for subobjects.

Some designers start from the user's perspective. They first design the interface, based on the user's needs and goals, and then design the underlying model to match it. This works well if you have the luxury of being able to affect the design the whole artifact. Conversely, you could start with the conceptual model -- perhaps it is an unchangeable requirement -- and then choose interaction patterns that have high fidelity to that model, assuming the model is a good one to start with.

The point is, users are going to build their own mental models about the artifact from what they can see. If there is consistency between the underlying model and the interface, and the interface is designed well enough to convey that model effectively to the intended audience, then the users will build mental models which correspond pretty well to the artifact's underlying model. Then the artifact has integrity. The user can more easily predict the artifact's behavior. Errors, if they happen at all, become comprehensible. The interface is easier for the designer to maintain over time, as the model changes.

In any case, I am not going to present a failsafe method for using this pattern language. In the first place, it has not been used enough to generalize from successful examples. Secondly, and perhaps more importantly, I'm not convinced that it will fundamentally change the processes that designers already use to design artifacts. A pattern language should make the processes smoother, more effective, faster, and hopefully with better end results; but there is already a wealth of knowledge out there on how to do design. As pointed out above, user contact and iteration are the keystones of good design.

Just as a proof of concept, here is an example of how the pattern language may be used to build a user interface design.

Example: Order Entry

This is a semi-fictional account of a UI design session using these patterns. The artifact we designed is real -- it is an order-entry application for a phone company, intended as a demo and working example of the use of my company's software. The design session did follow this general framework, although the patterns were never explicitly mentioned.

(Technically, it was a redesign session. A previous version of this software had been designed as basically a set of **Tiled Working Surfaces**, and it hadn't worked very well. The main dialog was too small to allow the contained subforms to grow, for one thing, and it was all getting too complex. The models we had used turned out not to match the way our customers generally thought about the order-entry process, either. Furthermore, the underlying software structures were changing drastically. These factors made it worthwhile to redesign the whole thing from scratch, rather than incrementally modify the existing design.)

Primary Pattern: The users will be typing in preformatted information, and occasionally browsing for information that's already been typed in. The primary pattern we picked was **Form**; no other option seemed to make sense.

Posture: As mentioned above, the application was intended for use as a demo -- it would be seen by potential customers who have short attention spans, and our salespeople would not have long to demo it, and could spend little or no time on setup or long explanations. Therefore, we chose **Helper Posture.** (A real order entry system would require **Sovereign Posture**, to support the people who use the software eight hours a day, five days a week.)

Working-surface Organization: Here we thought carefully about what kinds of information would need to be provided, and how they broke down into obvious groupings -- customer information, order information, services and their various features. We considered how those information groups related to each other, in terms of part-whole relationships. (It turned out to be more or less a hierarchy.) We also thought carefully about use cases -- what happens when a customer calls? What information do they give, and when? What does the order entry person need to tell them? How does the order entry person need to navigate through the application, to keep up with a customer? Our discussions at this point were heavily informed by the feedback we had gotten from the previous design of the interface.

We decided on a **Central Working Surface** where most of the order entry actually gets done (a Java JFrame), with an auxiliary dialog for looking up specific customers and orders. The order entry frame itself was composed of a **Stack of Working Surfaces**, representing the various forms needed to take customer information, order information, etc.

Since these forms related to each other in a hierarchical fashion, we eschewed tabs (a linear presentation) in favor of a **Hierarchical Set** along the left side of the Stack of Working Surfaces.

Information Organization: We wanted to allow a summary of a given customer's entire order to be visible, both for practical reasons and for a more effective demo. But sometimes more than a summary is necessary; for instance, a customer may call and ask about the status of one particular feature of one service of one order, which is getting into very fine detail. Thus, we chose to use **Optional Detail On Demand,** in the form of a Hierarchical Set (which we'd already chosen) in which the nodes could be open or closed by the user.

Actions: We thought about which actions an order entry person would take, such as creating new orders, adding new services to an order, and switching to another customer. We divided these up into sets of **Convenient Environment Actions** and **Localized Object Actions**. Because we had chosen Helper Posture, we decided to create verbiose text buttons for each these actions, and then we decided where to put them (on the outer frame for the Environmental Actions, on the individual forms for Object Actions).

Again, because it was a Helper Posture application, we didn't bother using patterns such as **User Preferences** (no user would use it for very long) or **Actions for Multiple Objects** (potentially confusing and expensive to implement).

Kinds of Information to be Supplied: Each panel for the different kinds of information (customer, order, service, feature) is its own **Form**. For each of these Forms, we had to figure out what logical groups of information existed and in which order the Form would present the information. We sometimes used **Small Groups of Related Things** to visually organize it according to the natural groupings, such as a labeled box around the fields for an address.

At this point, we didn't have enough information about the details of the various model objects (customers, orders, services, features) to go into the next design iteration. But some of the patterns available to us here would be **Forgiving Text Entry**, **Structured Text Entry**, **Choice from a Small Set**, and **Choice from a Large Set**; modifying these still further would be **Disabled Irrelevant Things** and **Good Defaults**.

So, with the design discussion temporarily at an end, we put what we had on paper (in pencil, for easy modification). Some informal usability tests were performed with this paper prototype, to try out the initial design; then an on-line prototype was built, for higher fidelity. The feedback we got from these usability tests indicated that the interface basically performs well. Once people actually tried to play with it, we found that some of the navigation between forms needed to be changed (it was then tested again), but we were happy enough with the final results to begin implementing the interface.

Example: Telephone Interface

(unfinished)

Control Panel - The whole front of the phone.
Small Groups Of Related Things - The various groups of buttons: numbers, function buttons, etc.
Helper Posture - Only in your face while doing dialing, not too imposing on my desk.
Background Posture - The blinking 'Hold' indicator. (also Status Display)
Important Message - The ringer.
Remembered State - Redial button, stored numbers (bound to 'Quick Dial' buttons).
Go Back to a Safe Place - The 'Release' button, or the hook.
Composed Command - The dialing process (sometimes could be better supported by

speech recognition)
Quick Access - 911, 411, *SP, etc.
User Preferences - The 'Quick Dial' buttons and the numbers they are bound to.

Social Space - Conversations and multi-way conversations are supported by the medium the telephone connects to.
Narrative - Voice mail can be seen as a Narrative triggered by the user.
Optional Detail On Demand - Slide-out panel that shows more functionality, in more sophisicated phones.
Iconic Reference - Pictures indicating quick-dial buttons, like fire, police, pizza, etc.
Editable Collection - Voice mail 'unheard' and 'saved' messages.
Step-by-Step Instructions - Automated account information available from the phone company, credit card company, etc.

Sublanguages

Each primary pattern tends to use certain other patterns more than others; they are loosely grouped together here as sublanguages. Some other highly recognizable patterns also have sublanguages that should be familiar to users of these artifacts, particularly Navigable Spaces and Step-by-Step Instructions.

Narrative:
Clear Entry Points, Go Back One Step, Go Back to a Safe Place, Bookmarks, Optional Detail On Demand, User's Annotations, Convenient Environment Actions

High-density Information Display:
Series of Small Multiples, Hierarchical Set, Tabular Set, Chart or Graph, Navigable Spaces, Small Groups of Related Things, Optional Detail On Demand, Disabled Irrelevant Things, Short Description, User's Annotations

Status Display:
Hierarchical Set, Tabular Set, Chart or Graph, Choice from a Small Set, Sliding Scale, Small Groups of Related Things, Optional Detail On Demand, Disabled Irrelevant Things, Important Message, Short Description, Personal Object Space, User Preferences

Control Panel:
Choice from a Small Set, Choice from a Large Set, Sliding Scale, Convenient Environment Actions, Localized Object Actions, Pointer Shows Affordance, Small Groups of Related Things, Optional Detail On Demand, Disabled Irrelevant Things, Good Defaults, Remembered State, Reality Check, Important Message, Short Description, Interaction History, Personal Object Space, User's Annotations

Form:
Choice from a Small Set, Choice from a Large Set, Editable Collection, Sliding Scale, Forgiving Text Entry, Structured Text Entry, Good Defaults, Remembered State, Step-by-Step Instructions, Small Groups of Related Things, Disabled Irrelevant Things, Pointer Shows Affordance, Optional Detail On Demand

WYSIWYG Editor:
Toolbox, Set of Object Actions, Actions for Multiple Objects, Convenient Environment Actions, Optional Detail On Demand, Disabled Irrelevant Things, Pointer Shows Affordance, Short Description, Personal Object Space, User Preferences, Scripted Action Sequence, Remembered State, Reality Check, Demonstration, User's Annotations

Composed Command:
Convenient Environment Actions, Localized Object Actions, Actions for Multiple Objects, Scripted Action Sequence, Forgiving Text Entry, Reality Check, Progress Indicator, Interaction History

Social Space:
Interaction History, Convenient Environment Actions, User Preferences

Navigable Spaces:
Map of Navigable Spaces, Clear Entry Points, Go Back One Step, Go Back to a Safe Place, Interaction History, Bookmarks, Pointer Shows Affordance, Short Description, Disabled Irrelevant Things, Progress Indicator, User's Annotations

Step-by-Step Instructions:
Go Back One Step, Go Back to a Safe Place, Progress Indicator, Map of Navigable Spaces, Interaction History, Optional Detail On Demand, Disabled Irrelevant Things, Convenient Environment Actions, Good Defaults

What is the basic shape of the content?

Narrative

Examples:

- User's guide to a software package
- News article
- Movie or TV show
- Voice-mail message
- Weather forecast
- Traffic report

Context: There is a need to convey information to the user; the information is closely interrelated, but of diverse kinds, and there may be some subjectivity involved.

Problem: In what form should the information be displayed to the user?

Forces:

- Many kinds of information are easier to absorb and remember when they are conveyed via natural language than when they are presented as raw data.
- The artifact's or author's point of view should sometimes be explicitly acknowledged, such as when credibility is an issue; narrative does this well, whereas other forms of data presentation are more anonymous.
- Many people find narrative to be more pleasant than raw data or symbolic representations.
- Natural language can convey subjectivity -- values, interpretations, and opinions -- in a way that no other information presentation can.

- Natural language is difficult for computers to generate and parse, but easy for humans.
- Natural language presented as "straight text" has to be read or skimmed linearly to find specific information.

Solution: **Convey the information via natural language.** Use all you learned in high-school English class about good writing. If users might be skimming the text to find specific data items, use color, fonts, and white space to set off items of interest; for readability in some situations, try using "senselining."

***Notes**:* [Unfinished. I think this is a very important pattern to understand, but I don't understand it yet. Natural language has wonderful, subtle gradations of meaning and emphasis that raw data can't have. How do we decide it's best to, say, give a weather report in a narrative form, rather than as a table? I suspect that some factors are: memorability, subjectivity and biased interpretation (which is not always a bad thing), effort required to absorb and understand the information. ...Maybe we should turn the question around, and ask when we should NOT use narrative, since narrative is the default way that we humans communicate.]

Random other notes...

Storytelling is a huge part of this pattern, but it's really outside the scope of this pattern language

Narrative is declarative, where Step-by-Step Instructions is imperative; both use natural language

Brenda Laurel's quote on the use of narrative in an interactive artifact: "Narrative includes both the story being told (content) and the conditions of its telling (structure and context). ... Within that [narrative] framework, interface designers can adopt strategies from narrative theory, such as including multiple representations of events and information, or using characters as a means of representing material with an explicitly acknowledged point of view." (pg. 182 in my copy)

Give two examples: an image from The Weather Channel's daily weather report (tabular data and icons), and a cut-and-paste from the Mass. weather site (narrative).

Howard Wainer talks about senselining. Strunk & White talks about good writing. Nielsen talks about writing for Web pages. Who talks about colors and fonts and whitespace?...

High-density Information Display

Examples:

- Email software that shows long lists of stored messages, e.g. Netscape or MS Exchange
- Good maps, such as the USGS topographic maps and NOAA charts
- Large, detailed object diagrams for object-oriented software

Context: There is a need to convey lots of information, either of a homogeneous type or interrelated in some way, but all of roughly equivalent importance.

Problem: In what form should the information be displayed to the user?

Forces:

- The user wants to quickly find specific information.
- The user also wants to quickly get a "big picture" of what the information is about.
- The user shouldn't have to do any more thinking, or manipulation of the artifact, than is absolutely necessary.
- The user may be frustrated, or led to a poor decision, by not being able to see all the available information.

Solution: *Pack as much information into one working surface as possible, following the precepts of good graphic design, with an organization that accurately reflects the underlying structure of the information*. Make the information dense enough so that the eyes do not have to move far from one thing to another, and so that scrolling is unnecessary whenever possible. Sparingly use bright color and/or imagery to highlight specific objects or state information relevant to the task at hand, so that the user doesn't have to read text or scan linearly to find important things. Use negative ("white") space or subtle color, rather than boxes or lines, to organize the information. Don't be shy about using large areas, or small fonts, or tiny controls and peripheral things if the information display is the most important task of the artifact.

Resulting Context: This is a rather high-level pattern; you are still left with a decision on exactly how you present the information. The answer partly depends upon the information's structure. If it's a hierarchy, for instance, you could use a **Hierarchical Set**. If the information is geographically structured, a map might be appropriate. Data sets of high dimensionality demand more sophisticated display techniques, often leading designers to combine the three spatial dimensions, color, and time; for very complex data sets, you may want to create **Navigable Spaces** that let the user wander through the complete information space.

The rest of the answer depends on what the user is trying to do -- whether they need qualitative or quantitative information (or both), for instance, or whether it's more important to find one datum, or to see that datum in context with the rest of the data, or to get a big picture. If quantitative information is needed, and the items to be presented have a similar substructure, then a **Tabular Set** may work well (and it can be creatively combined with other kinds of views, such as an outline). Or use **Chart or Graph** to give a visual representation to a "flat" data set, especially when a strong qualitative sense of the data is needed.

Notes: When a great deal of information is well-organized and presented in a visually reasonable fashion, the human mind is amazingly good at viewing it all, finding specific pieces of information in it, and getting the big picture out of it. Try to take advantage of human cognitive skills to enhance the data display.

It's easy to confuse "too much clutter" with "too much information" -- it's likely that most cases of the latter are actually misunderstood cases of the former. If a user complains to you that "There's too much stuff to see at once," see if the problem isn't really that it's just badly presented: too many boxes (imagine a 50x10 grid of white edit boxes on a gray background!), too many or too few colors, a hard-to-read font, poor use of negative space or widgetry, etc.

For good discussions of these issues, see Edward Tufte's books, particularly *The Visual Display of Quantitative Information* and *Envisioning Information.* Ben Shneiderman's latest edition of *Designing the User Interface* has an excellent chapter on information visualization, with plentiful examples from the software world.

Status Display

Examples:

- Status bar on Windows applications
- Train, bus, or airline schedule
- Airplane cockpit
- VCR display
- Computer's performance monitor

Context: The artifact must display state information that is likely to change over time, especially if that state information represents many variables.

Problem: How can the artifact best show the state information to the user?

Forces:

- The user wants one place where they know they can find this state information.
- The information on it should be organized well enough so that the user can find what they need at a glance, and interpret it appropriately.
- It needs to be inobtrusive if the information's not critically important, but...
- It does need to be obtrusive if something important happens.

Solution*: Choose well-designed displays for the information to be shown. Put them together in a way that emphasizes the important things, deemphasizes the trivial, doesn't hide or obscure anything, and prevents confusing one piece of information with another.* Never rearrange it, unless the user does it themselves. Call attention to important information with bright color, blinking or motion, sound, or all three -- but use a technique appropriate to the actual importance of the situation to the user (such as **Important Message**).

Resulting Context*:* If there is a large set of homogeneous information, use **High-density Information Display** and the patterns that support it (**Hierarchical Set**, **Tabular Set**, **Chart or Graph**); if you have a value which is binary or is one of a small set of possible values, use **Choice from a Small Set**. Visually group together discrete items which form a logical group (**Small Groups of Related Things**), and do this at several levels if you have to. For example, date and time are usually found in the same place.

Tiled Working Surfaces often works well with a Status Display, since it hides nothing -- the user does not need to do any window manipulation to see what they need to see. (You might even let the users rearrange the Status Display to suit their needs, using **Personal Object Space.**) If you don't have the space to describe what each of the displayed variables are (e.g. **Background Posture**), or if your users are generally experts who don't need to be told (e.g. **Sovereign Posture**), then use **Short Description** to tell the users what they are.

Notes: Use the positioning of an item within the Status Display to good effect; remember that people born into a European or American culture tend to read left-to-right, top-to-bottom, and that something in the upper left corner will be looked at most often.

What is the basic shape of the actions taken with the artifact?

Form

Examples:

- Tax forms
- Job application forms
- Ordering merchandise through a catalog
- Ordering a pizza over the phone

Context: The user has to provide preformatted information, usually short (non-narrative) answers to questions.

Problem: How should the artifact indicate what kind of information should be supplied, and the extent of it?

Forces:

- The user needs to know what kind of information to provide.
- It should be clear what the user is supposed to read, and what to fill in.
- The user needs to know what is required, and what is optional.
- Users almost never read directions.
- Users generally do not enjoy supplying information this way, and are satisfied by efficiency, clarity, and a lack of mistakes.

Solution*: Provide appropriate "blanks" to be filled in, which clearly and correctly indicate what information should be provided. Visually indicate those editable blanks consistently, such as with subtle changes in background color, so that a user can see at a glance what needs to be filled in. Label them with clear, short labels that use terminology familiar to the user; place the labels as close to the blanks as is reasonable. Arrange them all in an order that makes sense semantically, rather than simply grouping things by visual appearance.*

Visually, arrange the blanks and labels according to a spatial grid; align the edges of the stronger graphic elements, like boxes, where you can. Give them a neat look and a good visual rhythm, but don't let a strong geometry overwhelm the meaning of what is written or provided -- big arrays of edit fields (or whatever) look scary! If there are a lot of them, at least separate them into subgroups (**Small Groups of Related Things**); again, do this according to the meaning of the information, not just what looks good.

Provide reasonable default values wherever possible, to lessen the amount of work that the user has to do (**Good Defaults**). If the user provides information that makes some parts of the form irrelevant, disable them (**Disabled Irrelevant Things**). Likewise, if the user provides some clue which enables a software-based form to predict what information will be filled in elsewhere, then have the software do it for them. For example, if a user is providing their address and phone number, the town they provide might only have one area code in it -- so fill in their area code for them. But don't do it wrong; that's worse than not doing it at all, because the user then has to both notice the error and correct the form.

Be forgiving about the format of what the user provides, as much as is possible (**Forgiving Text Entry**). But if the required information simply must follow a certain format, then an empty, featureless text field is the worst possible thing to have to fill in. You never know if you're doing it right. Always offer a clue about what is expected by constraining the user's input in a reasonable way (see **Structured Text Entry** for one example), and don't resort to validating afterwards, giving the user a nasty message about how wrong their input was. Giving an example right there on the form is a better idea, but still not as good as a visual constraint.

(Sometimes, cultural factors make visual constraints unnecessary. A login screen, for instance, usually just has fields for "User name" and "Password" -- the possible input strings are constrained, of course, but 99.9% of its users will understand exactly what is expected, and constraints or extra information will be more trouble than it's worth. See Don Norman's *The Design of Everyday Things* for lengthy discussions of physical and cultural constraints.)

Resulting Context: You must pick controls for each of the pieces of information to be supplied. These may include:

- **Choice from a Small Set**, if the user should pick one or more from a set of 10 or fewer choices (give or take).
- **Choice from a Large Set**, if there are more than 10 choices.
- **Editable Collection**, if the user should construct a list of items.
- **Sliding Scale**, if the possible values vary linearly between an upper and lower bound.
- **Forgiving Text Entry**, if the recipient of the information is capable of parsing whatever text the user provides, which could vary widely.
- **Structured Text Entry**, if the recipient needs the information in a very specific format and cannot parse a potentially wide variety of responses.

Notes: No one ever fills in forms for fun. They do it because they want something (as with a catalog order), or because they are compelled to do it (taxes), or because it's their job (data entry). It's pointless to be cute or clever, and if you make the user have to read extra instructions, or go back to redo something they misunderstood the first time around, or jump around the form to fill in things "out of order," they will be irritated.

If you can find a non-contrived way to use something other than a form, do so. For instance, why ask someone to laboriously fill in a day, month, and year when you can just have them pick a day from a calendar? [Use three pictorial examples here: a date as a Forgiving Text Entry, a date as a Structured Text Entry, and a date as a calendar widget.]

Control Panel

Examples:

- Stove and oven controls
- Airplane cockpit
- Power plant control room
- Physical or electronic CD player

Bad Examples:

- Most remote controls for TVs, stereos, and VCRs
- Cellular phones

Context: The artifact must provide a way for the user to either change its state, or command it to do something.

Problem: How can the artifact best present the actions that the user may take?

Forces:

- The user wants one place where they know they can find the necessary controls, without having to hunt for them.
- The controls on it should be organized well enough so that the user can find what's needed, with minimal effort and with no confusion.
- There might be a lot of controls involved, some of which may be complex.

Solution: *For each function or state variable that is part of the user's mental model, choose one well-designed control that performs the function or displays the variable's value; put them all together such that the most commonly-used controls are the most prominent.* Similar functions may have similar-looking controls, but make sure that they aren't so similar that the user gets confused about which is which, even if their labels are different. (Hence the remote controls and cellular phones that are bad examples. They usually have rows of buttons that all feel alike to one's fingers -- and these are the kinds of things that people like to use without being able to look at them!)

When someone uses the controls, give immediate feedback that something is happening; this could be visual feedback, verbal, aural, tactile, etc. The best controls are often those that also display the current state (thus combining Control Panel with **Status Display**), since it makes sense to people to affect something in the same place they see it.

If the thing(s) being controlled has an obvious and familiar spatial layout, use it in the control panel. In *The Design of Everyday Things*, Don Norman makes an example out of stoves -- their burners are arranged in a 2x2 grid, but the controls for them are usually not, and users have to stop and think about which goes with which. If analogous spatial arrangement doesn't make sense in a given situation, then try instead to group the controls semantically. Ideally, the controls relevant to a given high-level task will end up clustered together (**Small Groups of Related Things**), so the user doesn't have to hunt all over the control panel for the next needed control.

Resulting Context*:* Appropriate controls now have to be chosen. Some patterns you can use are **Choice from a Small Set, Choice from a Large Set, Sliding Scale**, or even an interactive **Chart or Graph**. Make it clear which controls represent artifact-wide actions (see **Convenient Environment Actions**), and which represent actions upon one object (see **Localized Object Actions**) -- and indicate which object is being acted upon, of course!. When a given control is not meant to be used at a given time, disable it (**Disabled Irrelevant Things**).

To help naive users figure out what's what on a counterintuitive display, or on one that's meant for experts, you could employ **Short Description** or **Optional Detail On Demand.** To alert the user to side effects and to unexpected situations, use **Reality Check** and **Important Message**, respectively.

WYSIWYG Editor

Examples:

- Illustrator, PowerPoint, Macdraw, and other drawing software
- Text editors such as Word, WordPerfect, and Netscape Composer
- GUI builders such as Visual Basic or JBuilder
- Paper sketchpad
- Newspaper layout [*what are those pasteup things called?*]

Context: The artifact is used as a tool or environment in which other artifacts may be created (particularly those with visual aspects).

Problem: How can the artifact best present what is being created, and what the user needs to do to create or change it?

Forces:

- People already have the skills to draw, write, sculpt, arrange, etc.
- The user wants to see what they're creating immediately, as they work with it; this instant feedback leads to a strong sense of engagement with the work.
- It's faster or easier for a user to do certain operations visually, rather than programmatically or linguistically -- but not all operations.

Solution: Always show the user an accurate and up-to-date representation of the artifact they are creating ("what you see is what you get"); allow the user to interact directly with it as they add to it, delete from it, modify it, and so on. If additional information or tools are necessary to permit such interaction, such as object handles, then design them so that they don't significantly interfere with the user's ability to see the whole creation.

Sometimes it is easier to let the tool do certain mechanical jobs, such as precise layout, repetitive tasks, complex geometric shapes, image or sound processing, etc. Don't make the user do these by hand unless they choose to; provide easy-to-use automation instead, in a way which is smoothly integrated into the WYSIWYG Editor itself, and which doesn't require a jarring context shift in the user's mind (see Brenda Laurel, *Computers As Theatre*, chapter 5).

Resulting Context: Use a **Toolbox** for creation of different kinds of items or structures, and **Localized Object Actions** (often paired with **Actions for Multiple Objects**) to modify them and operate on them. The **Personal Object Space** pattern will let the user arrange their working surface to fit the way they work best; to preserve the user's settings and state from session to session, use **User Preferences** and **Remembered State**. **Interaction History** allows the user to see how they've modified the artifact lately, and to roll back to a previous version if they wish. **Sovereign Posture** is often used with this pattern, because these kinds of artifacts usually require sustained attention and a non-trivial learning curve.

While visual affordances are a good way to show how parts of the created thing can be manipulated, they have their problems: they usually obscure the user's view to some extent, are part of the tool rather than part of the created thing itself, and contribute to visual clutter. Therefore, **Pointer Shows Affordance** can be used to augment or replace them (and so can **Short Description**, as with tooltips).

Composed Command

Examples:

- UNIX or DOS command-line interface
- SQL
- Spoken instructions to an interface made for visually impaired people
- One human telling another to do something
- Programming and scripting languages

Context: The possible actions to be taken with the artifact can be expressed through commands, which can be composed from smaller parts, in a language-like syntax with precise and learnable rules; and the users are willing and able to learn that syntax.

Problem: How can the artifact best present the actions that the user may take?

Forces:

- Expert users often find linguistic commands to be more efficient than visual representations or direct manipulation.
- Sometimes the available actions cannot, or should not, be expressed graphically (perhaps because there is an intractably large set of them).
- The visual channel may not be available at all to the artifact or to the user.
- The artifact is able to provide feedback on the correctness and appropriateness of the given commands.

Solution: *Provide a way for the user to directly enter the command, such as by speech or by typing it in.* Feedback on the validity of the command, or its results, should be as immediate as is practical. The parts and syntax rules should be easy to learn, and should generate concise commands whose meaning is obvious. Offer a way to do auto-completion or a set of possible interpretations of partially-entered commands, especially if the user is unwilling or unable to learn the language -- but beware the expert user, who may find this irritating! Allow it to be turned off if necessary.

Resulting Context: The possible actions divide neatly into environmental and object actions (see **Convenient Environment Actions** and **Localized Object Actions**); let the object actions accept "wildcards" in place of the object, to effect **Actions for Multiple Objects**. **Forgiving Text Entry** lets the user give commands with a generous margin for error, which is necessary for a natural-language-style interface and pleasant for other kinds. **Reality Check** and **Progress Indicator** generally make dialogue-like actions easier for users to deal with.

Most existing Composed Command systems provide some kind of **Interaction History**. In the linear dialogue that this pattern imposes upon the user, a history can be invaluable; users often need to repeat previous commands, sometimes with minor changes, and they sometimes need to know what they've done recently. Composed Command also takes well to **Scripted Action Sequences** (also common in on-line implementations of this pattern), especially since users are already thinking in terms of grammar and composable parts.

Sociable Space (unwritten)

How does the content or available actions unfold before the user?

Navigable Spaces

Examples:

- The WWW and other hypertext systems
- Myst
- Museum exhibit in a set of physical rooms
- Set of applications in a suite, as with the PalmPilot or a network computer

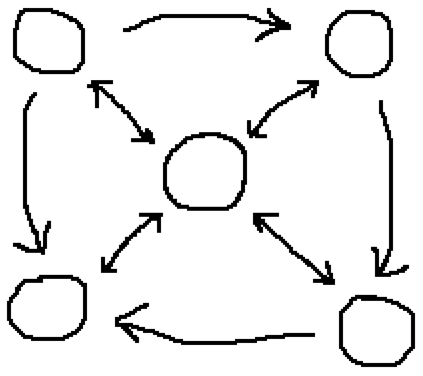
Context: The artifact contains a large amount of content -- too much to be reasonably presented in a single view. This content can be organized into distinct conceptual spaces or working surfaces which are semantically linked to each other, so that it is natural and meaningful to go from one to another.

Problem: How can you present the content so that a user can explore it at their own pace, in a way which is comprehensible and engaging to the user?

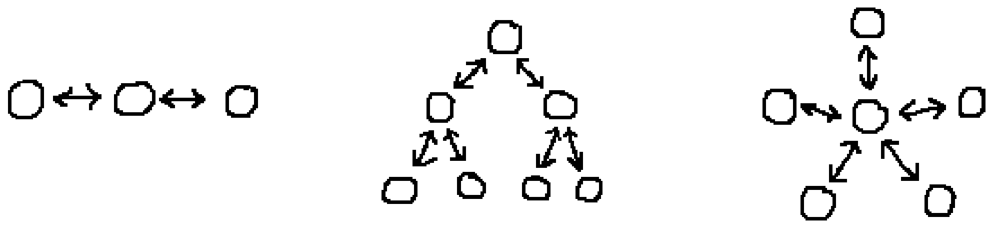
Forces:

- The user wants to know where they can (or should) go next, and how it's related to where they are now.
- The user wants to be able to choose where to go next.
- The user doesn't want to get lost.
- The concept of information spaces is a natural one for people to think about, both because it mirrors the real world and because the WWW is so commonly understood.
- It's delightful to explore a new place, where the user doesn't necessarily know what's "around the corner."

Solution: *Create the illusion that the working surfaces are spaces, or places the user can "go" into and out of.* Start out with at least one top-level or "home" space, to which the user can easily return (**Clear Entry Points**). In each space, clearly indicate how you get to the next space(s), such as by underlined text, buttons, images of doors, architectural features, etc. Use the spatial locations of these links to help the user remember where the links are. Provide a map of how the spaces are interconnected (**Map of Navigable Spaces**), preferably one that allows the user to go directly to the spaces represented on the map. Make sure that the user can easily retreat out of a space (**Go Back One Step**) or return to the home space (**Go Back to a Safe Place**).



The user will build a mental model of the content from the structure of the Navigable Spaces. Therefore, construct the spaces and their interconnections to mirror the model you want to present (which may not be the same as the actual underlying data structure). Chains, trees, and star patterns are common ways to structure Navigable Spaces (see illustration below); they are easy to understand, visualize, and navigate, and they can contain rich content.



Resulting Context: As pointed out above, **Map of Navigable Spaces** should be one of the first patterns you deal with, even if you explicitly choose not to use one; the same for **Go Back One Step** and **Go Back to a Safe Place**. To help show where the links are in the spaces, you can use **Pointer Shows Affordance**; to give additional information about where they go, use **Short Description**.

People using the WWW tend to depend upon their browser's **Interaction History** (the links you've most recently visited, in chronological order) to get around. Not surprisingly, they also depend upon their **Bookmarks** to keep track of places they want to go back to. These two patterns might be especially important in any large or unbounded set of Navigable Spaces, particularly if a map is impractical.

When you're dealing with power users, seriously consider the value of displaying more than one surface at a time, perhaps using **Tiled Working Surfaces**. It's often good to provide the user with the option of being in at least two or three spaces of their choice, especially if a user is likely to be jumping between spaces frequently. This does increase the user's cognitive load, though, so it may not be appropriate for simpler artifacts that require short learning curves.

Notes: With games, part of the fun is in figuring out where you are and where you can go next, so maps and obvious links would actually reduce the user's fun. In a way, the WWW is similar -- who could ever make a map of the WWW anyway? -- but, of course, not everyone uses it for fun.

Notice that chains are structured similarly to **Step-by-Step Instructions**, trees to **Hierarchical Set**, and stars to **Central Working Surface**. All three of these archetypes have very strong, simple geometric properties; they probably warrant further exploration.

Overview Beside Detail

Examples:

- Windows Explorer and many other file-system exploration dialogs
- Web sites that contain a "site contents" frame on the left, and a frame containing the current page on the right
- Email readers that show you a list of messages on one side, and the content of the selected message on the other side
- Many visualization tools for large hierarchies or geographical maps

Context: The artifact contains a large amount of content -- too much to be reasonably presented in a single view. This content may be cleanly partitioned into a top-level set of objects or categories, such as document names and document content, containers and their contents, or objects and their properties; alternatively, the content may be a large, continuous, very detailed data set, in which users may have specific areas of interest.

Problem: How can you present this large amount of content so that a user can explore it at their own pace, in a way which is comprehensible and engaging to the user?

Forces:

- Should always be able to see where you are; keep it within visual field, not hidden or distant
- User wants to move between the detail views quickly, e.g. to compare and contrast
- Not enough space to show multiple detail views simultaneously; may be constrained to one working surface
- Too confusing to show multiple detail views simultaneously
- Lets user concentrate on one object, category, or area of interest at a time
- One working surface is easy to deal with -- no window shuffling or restacking
- Easy navigation, managed by one (often flat) set of objects -- a star pattern

Solution: Show the whole set of objects, or the whole undetailed data set, in one part of the display area, to act as an overview of the content. When the user selects a single object, category, or area of interest within that overview, immediately show its related content -- its detail -- in the remaining space. As the user changes the selection, update the detail area to always reflect the current selection.

Keep the overview and detail areas spatially adjacent, so that the user can easily glance back and forth, using the overview area to drive the decisions about what to look at next.

If you use **Tiled Working Surfaces** to relate them to each other, the user doesn't need to mentally "context switch" or make any extra gestures to go from one area to the other (such as raising one window above another). This enables a sense of flow.

Resulting Context: You have to decide how the user selects the part of the overview that they're interested in. The kind of data you're working with should suggest a graceful solution: a set of objects can be selected one at at time (or perhaps several contiguous ones), while a continuous data set like a geographic map may provide a freely movable "thumbnail," drawn on top of the overview, that shows the current selection.

A set of high-level objects or categories can be linear, hierarchical, or otherwise organized by whatever principle is appropriate -- see **Hierarchical Set**, **Tabular Set**, **Editable Collection**, **Personal Object Space**, and whatever other patterns you find appropriate.

Sometimes you just can't easily partition the content into only two levels of abstraction at a time. You should always retain the high-level overview, but you could then take the detail view and make it act like a second-level overview, so that the user can drill down one more level to see yet more detail. This recursion then gives you a total of three views, and you can continue the recursion for as many levels of depth as you need to. Netscape Messenger does this to show email and news, by the way: an upper-left panel shows the mailbox or newsgroup hierarchy, an upper-right panel shows the list of messages in the currently-selected mailbox or newsgroup, and a large bottom panel shows the message currently selected in that upper-right panel. Stuart Card et. al. recommend that for zooming in on a continuous data set, use a zoom factor between 3 and 30; more than that becomes disorienting for the user.

Sophisticated users may find it too constraining to see only one detail view at a time; if so, consider allowing multiple detail views simultaneously, possibly using **Pile of Working Surfaces.** Just keep in mind the increased complexity of the artifact when you do that -- users may get confused about where they are. I have rarely found it necessary to do this, however.

Notes: This pattern has also been noted, though not in pattern terminology, by Stuart Card, et. al. in their book *Readings in Information Visualization: Using Vision to Think.* A two-paper section called "Overview Plus Detail" is devoted to this concept; in the section introduction, they explain their reasoning:

Having an overview is very important. It reduces search, allows the detection of overall patterns [in the data], and aids the user in choosing the next move. But it is also necessary for the user to access details rapidly. One solution is overview plus detail.

Step-by-Step Instructions

Examples:

- Wizards
- Installation instructions for all kinds of appliances, software, etc.
- Recipes
- Repair manuals
- Getting cash from an ATM

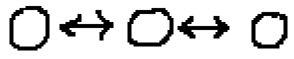
Context: A user needs to perform a complex task, with limited time, knowledge, attention, or space. Alternatively, the nature of the task is step-by-step, and it's meaningless to show all the action possibilities at once.

Problem: How can the artifact unfold the possible actions to the user in a way that does not overwhelm or confuse them, but instead guides them to a successful task completion?

Forces:

- The user doesn't always want, or need, to understand all the details of what they are doing.
- A user presented with a bunch of possible actions, in no prescribed order, may not have any practical way to figure out what to do first, second, etc.
- A user who is afraid of doing something wrong may prefer that the actions they have to perform be explicitly spelled out for them.
- The whole task can't be performed entirely automatically, because it requires choices or information from the user.

Solution: *Walk the user through the task one step at a time, giving very clear instructions at each step.* Use visual similarities in all the steps, e.g. typography and layout, to maintain a rhythm throughout the task; make each step a focal point, both visually and in the user's "attention space." If information is needed from the user, ask for it in simple terms and with brevity; by keeping it short, you can better maintain the user's sense of flow through the whole step-by-step process.



The task may branch like a flow chart, depending upon what information the user gives it, but the user doesn't necessarily need to know about all the available paths through the task. If it's not likely to confuse the user, show the steps as a **Map of Navigable Spaces**. If possible, allow the user to back out of the steps (**Go Back One Step**, **Go Back to a Safe Place**). If the sequence of steps as seen by the user is too long -- more than ten steps, for example -- try to break it up into manageable sub-sequences, so it doesn't get too tedious for the user. Make sure the sub-sequences relate to each other in a meaningful way, however, or the user may see it as gratuitous or annoying.

Sometimes users may want to know more about what they're doing -- **Optional Detail On Demand** gives you a way to present that extra information. Also, if a user has gone through a lot of steps, they have trouble remembering what they've done and why. At least provide a **Progress Indicator** if the number of steps grows beyond seven or eight, which is the average limit of short-term memory. If a lot of user interaction is necessary, such as for branching decisions, consider providing an **Interaction History**

Resulting Context: **Narrative** is a good choice for presenting the task steps themselves; the use of natural language to describe what needs to be done is intuitively obvious, and puts a user at ease. **Go Back One Step** and **Go Back to a Safe Place**, along with a corresponding Forward control, can be used to move through an interactive task.

To get information from the user, you can use a **Form** or its simpler component patterns, especially **Choice from a Small Set** and **Forgiving Text Entry**. Using **Good Defaults** with them allows the user to move smoothly past the points where extra data entry is unnecessary, again preserving the sense of flow. Finally, a small set of **Convenient Environment Actions** should give the user ways to cancel or suspend the task without having to back out of it one step at a time.

Notes: Be aware that this pattern may irritate experienced users. If a user knows exactly what they need to do, and want to do it quickly, constraint to this step-by-step presentation can feel like a straitjacket! Also, if the task to be accomplished isn't inherently linear -- i.e. you don't really have to do one step first, another step second, etc. -- you might provide an alternative "random access" presentation of the possible actions, such as a **Stack of Working Surfaces**.

Small Groups of Related Things

Context: There are many items or actions to show the user, some of which are more closely related to each other than other things. This context is extremely common, occurring in many of the high-level patterns in this language, including (but not limited to) **High-density Information Display**, **Status Display**, **Control Panel**, and **Form**.

Problem: How should the items or actions be organized?

Forces:

- Large, undifferentiated masses of things can be intimidating and difficult to figure out, especially to someone seeing them for the first time.
- The brain is very good at discerning groups and nested groups of things, and at getting a "big picture" from the grouping structure.
- The items cannot or should not be shown on separate working surfaces; for instance, because they are part of one coherent task.
- People naturally assume semantic coherence where there is visual coherence.

Solution: *Group the closely-related things together, nesting them in a hierarchy of groups if needed.* Keep the number of things in any one group to around ten or fewer, even if the things are other groups. Use repetition and symmetry to keep the groups from becoming visually chaotic (and don't overuse boxes, even the nice etched ones in some GUI toolkits; white space often works just as well). Make sure that the grouping is not arbitrary, but is based on the meaning of what is being shown -- as pointed out in the Forces, a user will naturally try to derive some kind of semantic meaning from the groupings, even if it's wrong.

Resulting Context: To make a visual grouping of things look good, it's tempting to shortchange their individual usability. Try not to make too many sacrifices here. For instance, a common mistake made by **Form** designers is to make a text box too short for the expected input, simply to make it visually fit in with the Small Group of Related Things (e.g. other text boxes) that it belongs to. Also, there's no need to be dogmatic about this pattern -- expert users of a **WYSIWYG Editor**, for instance, may prefer that their **Toolbox** just show all the available tools as densely packed as possible, to save space.

Sometimes a large group of homogeneous items work together to form a single conceptual entity. This is true about many **High-density Information Displays** -- data points on a scatter chart, for instance, or a column of numbers representing a single variable. This pattern shouldn't apply to them. See Edward Tufte's book *Envisioning Information,* in particular the chapter on "Small Multiples," for an excellent set of counterexamples: these large sets of items are meant to show small changes between individual items that share most characteristics, to make those changes stand out. The impact comes from seeing all those similar items next to each other. In many cases, that impact would be completely lost if you broke up those sets of items into small groups!

Notes: This is a very basic way of managing complexity, and is almost more of a principle than a pattern. The "ten or fewer" comes from Miller's number (7+-2), which represents, among other things, the upper limit of someone's ability to "instantly" scan a set of items. Beyond that, the time it takes to read through the items grows linearly.

Series of Small Multiples *(unwritten)*

Hierarchical Set

Examples:

- Windows Explorer
- Class-hierarchy diagrams
- Family tree charts

Context: There are many things to show the user, and they are interrelated in a hierarchy (or can be made to appear that way). This could be in a **High-density Information Display**, or a **Map of Navigable Spaces**, or as the organizing principle for a **Stack of Working Surfaces**.

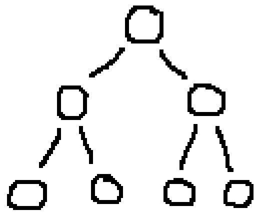
Problem: How should the information be organized?

Forces:

- The user should see the structure of the data.
- Hierarchies are easy to understand.
- They are also quick to traverse, if the user knows where something is in the hierarchy.

Solution: *Show the data in a tree-like structure.* Keep all the nodes at a given depth from the root in the same line, plane, or arc, to emphasize their parallelism. Allow non-leaf nodes to be opened and closed, to give the user control over how much of the hierarchy is visible at any one time. Depending on expected usage, try to balance the demands of having a dense, fully-visible structure with the ability to look at the details of individual nodes; if necessary, use panning and zooming, and other patterns like **Short Description** and **Optional Detail On Demand.**

Resulting Context: You get a form of **Optional Detail On Demand** for free if you let users open and close parent nodes -- if someone wants to see the children of a given node, they can, or they can ignore it. **Pointer Shows Affordance** is a way to indicate that a node can be opened or closed. Remembered State gives you a way to set up the node states according to how the user arranged them last time.



Notes: The Windows Explorer has brought the columnar "outline view" into common usage in desktop GUIs, and every self-respecting toolkit has one; they're great for fitting a hierarchy into a narrow space, but if you don't have that space restriction, try to use something more visually interesting, like an actual 2D drawn tree. These are much better at showing and manipulating large hierarchies than outlines are, because you don't have to stack everything together in one column, then scroll forever to get a big picture. (Doing them right is a non-trivial programming problem, unfortunately.)

Some designers have come up with pretty amazing ways of viewing very large hierarchies. Circular "fisheye" viewers, 3D rotating cylinders... [*find references for these, in CHI proceedings somewhere*]

Beware of using trees if the relationships among the data represent a directed graph instead, such as with multiple inheritance in a class diagram. It can be done, but it's not kind to the user -- they then see one thing in several places in the hierarchy, which may be very confusing. If it's truly a graph, draw the graph and don't oversimplify it. (The term "graph" here refers to its mathematical meaning, in which a set of vertices is arbitrarily interconnected via edges.)

Tabular Set

Examples:

- Spreadsheets
- Stock price tables
- Phone book

Context: There are many homogeneous things to show the user, each of which has similar additional information or subparts. This is often used in a **High-density Information Display.**

Problem: How should the information be organized?

Forces:

- The user should see the structure of the data.
- The user wants to easily look up a specific piece of information.
- Putting similar information into columns facilitates quick comparison of values.
- It's easy, and intuitively obvious, to sort a table according to the data in the columns.

Solution: *Show the data in a table structure.* Order it according to some appropriate organizing principle, such as by the value of some column (if the table's principal use is to compare items by that value) or alphabetically by item (if its principal use is to look up values). Put some white space between columns to set them apart, but not too much; the user's eye shouldn't have to work too hard to go from one column to another.

The columns themselves should be organized logically. Depending upon your purpose, they may be organized with the most commonly needed data immediately after the item name, and in decreasing order of importance as you move from left to right. Or they may be organized in groups, with the group names above the column names (**Small Groups of Related Things**). The best organization will depend upon the data and the user's purposes.

If it's reasonable to do so, allow the user to adjust column widths, column order, and sort order. Many user-interface toolkits for computer applications provide these capabilities.

Resulting Context: If there's no obvious way to indicate that the columns are manipulable (e.g. sortable, or with changeable widths), at least use **Pointer Shows Affordance**. **Remembered State** gives you a way to set up the column states according to how the user arranged them last time.

Notes: Howard Wainer's book *Visual Revelations* has a brief but good chapter about table design. It's worth reading if you will be designing a lot of these.

Chart or Graph

Examples:

- Line charts
- Scatter charts
- Pie charts
- Bar graphs
- Timelines

Context: There is a lot of homogeneous data to show the user, possibly in multiple data sets. This is likely to be needed in a **High-density Information Display**, or a **Status Display**, or even a **Control Panel**.

Problem: How should the information be organized?

Forces:

- The user wants to get a "big picture" quickly.
- Getting specific data out of it is secondary or unimportant.
- Graphs facilitate easy comparison of values to each other.
- Pictures have a bigger immediate, emotional impact than numbers.

Solution: Show the data plotted against time or some other variable. Plot it together with other variables for further comparison. [*unfinished*]

Notes: Tufte's books go into great detail about these.

Optional Detail On Demand

Examples:

- Windows 95 color dialog that opens to show a complete RGB color square
- The "Advanced" or "Other Options" buttons that appear on many dialogs
- The panel on front of some VCRs that hide everything except the basic playing controls
- Footnotes
- The fine print on credit card applications, purchase agreements, etc.
- Visual C++ debugger tooltips that show a variable's value when the cursor hovers over the variable in the source code

Context: A large percentage of the available information or actions (termed "items" for the rest of this pattern) can be considered details, and are unneeded most of the time. This is a very common situation which can happen in any of the primary patterns -- **Narrative** (footnotes), **High-density Information Display** (extra information), **Form** (optional information provided by the user), and so on.

Problem: When should these usually-unneeded items be presented to the user, and how?

Forces:

- The user may be confused or overwhelmed if they are presented with lots of items at once.
- The most-often-used items should be right at hand, and distinct from the details that may be unneeded; this helps orient the user towards what they are most likely to need.
- All the possible items should be easily available to the users that need them.
- Sometimes there isn't enough space to show all the possible items.

Solution: Up front, show the user that which is most important and most likely to get used. Details and further options which won't be needed most of the time -- say 20% or less of expected uses -- can be hidden in a separate space or working surface (another dialog, another piece of paper, behind a blank panel). Mark it clearly so that a user who needs this optional detail can find it immediately. Place the "handle" to it (push button, panel door latch, etc.) very close to the primary working surface, where everything else is, so a user doesn't have to go hunting for it. Make sure that very little effort is needed to get at it.

Resulting Context: You need to figure out which items are most needed, and which are optional. Know your users' needs well. If you can't make a good enough guess on that basis, try putting a few items on the top level and the rest hidden; observe your users using the artifact, and as it becomes clear which items are the most often used, "promote" them to the top level (and "demote" the ones at the top level that never get used). In spite of this hit-or-miss approach, try to make a coherent design out of the items at the top level.

This is a very common, simple way to manage complexity in an artifact. It can sometimes fail, however. Consider a user who might always want to see the optional detail: from the designer's point of view, perhaps only 10% of the total uses of the artifact require the optional detail, but from this user's point of view, it is 100%! Now the user has to take an extra step, every single time they use it, to see what they need to see. If this is likely to happen, think about ways of making the optional detail always visible, if the user takes some small action (like propping open a panel door).

Note that you can nest these inside each other. In fact, some computer applications bury optional details inside chains of dialogs that can be 4 or 5 dialogs deep! This generally doesn't go over well with users. It's easy to lose your place, for instance, and it's hard to remember where something is or explain to someone else how to find it; it also takes a lot of time and effort to reach something that deep.

Notes: If the optional detail takes up very little space, don't bother with this pattern; you might as well fit it all into the primary working surface.

Disabled Irrelevant Things

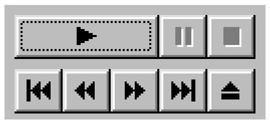
Context: Information or actions that are normally useful become temporarily irrelevant sometimes. This is a common situation in almost all of the primary patterns that use visuals, such as **Control Panel**, **Status Display**, **Form**, and **WYSIWYG Editor**.

Problem: How can the artifact steer the user away from actions that cannot or should not be taken, while still maintaining visual calm and stability?

Forces:

- The artifact should present items in such a way as to allow the user to form a correct mental model of its underlying ideas (for information) or action states (for actions).
- All actions available to the user at a given time should be valid, so that the user doesn't do something erroneous.
- The artifact should be responsible for figuring out what's valid and what's not, to avoid giving the user an unnecessary cognitive burden.
- When the users can trust the artifact to not let them do invalid actions, they will feel more secure about exploring it and trying new things.
- An artifact which is too actively helpful can be distracting, irritating, or confusing.

Solution: *Disable the things which have become irrelevant.* Hide them entirely if the user shouldn't even be aware of them, or "gray them out" (with their main features barely visible) if the user should know they're there but that they just aren't useful right now. If the thing is a manipulable control, don't allow the user to use it.



Resulting Context: Computer interface toolkits normally provide a reasonable implementation of a disabled or grayed-out state. If the item being disabled uses **Pointer Shows Affordance**, however, remember to disable that too, so that the user doesn't get conflicting cues about whether a given control is usable or not.

As the user uses the artifact, different actions may become available to them as they change the artifact's state over time. This pattern provides one way to let the actions unfold to the user, without the disruption of having items appear out of nowhere. Still, it doesn't really tell the user *what* to do -- it only tells them what they *can't* do.

Pointer Shows Affordance

Examples:

- Crosshair or paintbrush pointer in a drawing program
- Arrow pointer over the corner of a resizable window or a movable split pane
- Finger pointer over a hyperlink, especially pictorial links
- Buttons whose borders appear as you move over them

Context: The artifact contains a visual pointer, or "virtual fingertip" (mouse or pen point, for instance) that is the focal point for the user's interaction with the artifact. Patterns that tend to use this a lot are interactive ones with a heavy visual component, including **Navigable Spaces**, **WYSIWYG Editor**, and **Form** (particularly for controls like **Forgiving Text Entry** and **Editable Collection**).

Problem: How can the artifact indicate that a visual entity represents an action that the user may take?

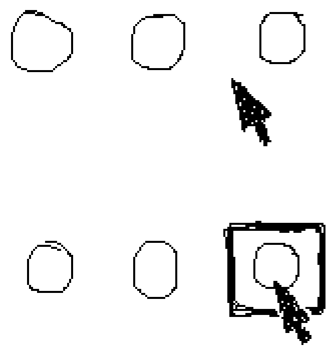
Forces:

- Static visual affordances aren't always enough to indicate the presence of a manipulable control, especially when space is tight or when an elegant-looking graphic design is of paramount importance.
- Multiple affordances can be more effective than one, if they work together properly.
- If the pointer is near the border of a manipulable control, especially something with a small target area, the user should get feedback that instantly tells them whether they're "in" or "out."
- Textual help (e.g. **Short Description**) is not as effective as a simple picture for some kinds of actions.

Solution: *Change the affordance of the thing as the pointer moves over it.* This can be done in one of two ways: by changing the pointer to a small picture illustrating what can be done, or by changing the thing itself to make it stand out visually.

If you change the pointer, use a small picture illustrating what can be done. Use a standard icon if an appropriate one can be found -- crosshairs for drawing, single arrow for selection, I-beam for text entry, hands, pencils, paintbrushes, resize arrows, etc. -- because they are so easily recognized. Keep it small or mostly transparent, so that the user can easily see what's under it.

If you change the thing itself, you have a lot of freedom to experiment. Any visual change may be enough to tell a user that the object is at least clickable; but consider your audience when deciding how flashy or distracting the change is. To be sure that your design actually works, of course, you should test it with potential users.



Resulting Context: Be careful not to use this pattern gratuitously. Now that the tools to implement it are widely available, lots of user interfaces use it as a substitute for static visual affordances. This isn't always wise. Think about the poor user looking at a screenful of borderless icons, some of which are buttons, some of which are moveable objects, and some of which don't do anything at all! The user now has to move the pointer over each object in question to see what it does. **Short Description** has the same problems in these cases.

For those of us stuck with non-tactile interfaces, such as mice, this pattern produces something like a substitute tactile sense. As you run the pointer over the interface, you get visual responses that correlate to physical sensations -- bumpiness (raised button edges), heat (when something turns from a muted color to a bright color), etc. Says David Cymbala:

> "I was cruising the web the other day, and I was using the mouse pointer to 'brush' across an image map that had patches of 'active' areas.The image jumped into my mind of what I was doing: "Feeling" the image map with the mouse. Instead of a 'tactile' sensation, I was correlating the image of the mouse pointer with the movement of my hand through space. I almost 'felt' it physically... The pointer allows me to 'feel' visual space as a replacement for the lost tactile dimension." (From personal correspondence, dated June 17, 1998.)

Notes: Don Norman brought the term "affordance" into the interface designer's vocabulary with his classic *The Design of Everyday Things*. In it, he defines an affordance as "the perceived and actual properties of the thing, primarily those fundamental properties that determine just how the thing could possibly be used."

I find that when I'm working quickly, I depend very heavily on the fact that my pointer changes when I'm over a manipulable control; if I want to resize a window, and I move the pointer towards the window edge, I instinctively start the press-drag motion the instant that pointer changes. I don't actually look hard to see if the pointer is over the control. That zone could extend ten pixels beyond the window edge, for all I care. Conversely, it's very hard to deal with direct manipulation if the cursor doesn't change -- I have to pay far too much attention to the screen, and use better fine motion control; and with small controls, there's always this vague uncertainty that the action will succeed.

For some wonderfully bad examples, take a look at the **Interface Hall of Shame**. Look under the "Visual Elements" section, especially at the Microsoft examples and the first WebZip commentary.

Short Description

Examples:

- Windows tooltips
- Mac bubble-help
- Status bar help

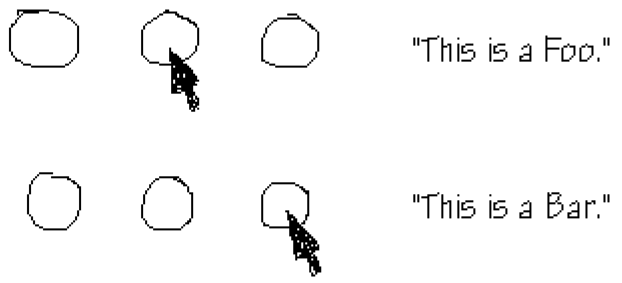
Context: The artifact contains a visual pointer, or "virtual fingertip" (mouse or pen point, for instance) that is the focal point for the user's interaction with the artifact. Nearly all the primary patterns with a visual component can use this to good effect, particularly **Navigable Spaces** for link descriptions, **High-density Information Display**, **Status Display**, **Control Panel**, and **WYSIWYG Editor**.

Problem: How should the artifact present additional content, in the form of clarifying data or explanations of possible actions, to the users that need it?

Forces:

- A short explanation may be all the user needs or wants; something long will be overkill.
- Users generally don't want to leave the artifact and go somewhere else for help, such as a manual; this usually breaks one's concentration and costs too much time.
- There isn't room to put static descriptive text into the artifact, or visual elegance precludes doing so.
- The descriptive text might be useless most of the time, and may become irritating if it is static or hard to turn off.

Solution: *Show a short (one sentence or shorter) description of a thing, in close spatial and/or temporal proximity to the thing itself.* Allow the user to turn it on and off, especially if the description obscures other things or is otherwise irritating; alternatively, don't show it without some deliberate user action on an item-by-item basis, such as pressing a key or hovering over the item for a certain length of time.



Resulting Context: You get to decide what text to put into the Short Description. There's no point in being redundant with whatever's statically shown in the artifact; if you're going to impinge upon the user's attention with a popup or something, at least add some value with it. You could use it to describe a possible action (as with **Pointer Shows Affordance**), or describe the results of the action, or reveal more data (thus implementing **Optional Detail On Demand**).

Notes: In his **January 11, 1998 Alertbox column**, Jakob Nielsen strongly recommends using link titles to help give the user a preview of where a Web link goes; they add important contextual information to the sometimes-mysterious HTML links. These are effectively Short Descriptions.

I've never seen it done, but this pattern could theoretically be used with speech in a multimodal interface. As you focus your visual attention on some feature, the Short Description for that feature could be spoken aloud to you.

How does the artifact generally use space and the user's attention?

Sovereign Posture

Examples:

- Framemaker, Excel, Photoshop
- Nintendo video games
- Control panel for a power plant
- Car dashboard

Context: The artifact will be heavily used, occupying the user's full attention, and the user is willing to invest time and effort to learn it. Examples of nearly all of the primary patterns can be found using this pattern.

Problem: How should this artifact relate spatially to other artifacts that might share its space, and how can it best use the space it has?

Forces:

- The user wants full access to everything at once, even if the artifact is complex.
- The user will be irritated by small inefficiencies and too much hand-holding.
- There is a limited amount of space to be shared among this and other artifacts.

Solution: *Allow the artifact to take up all the space it needs to get the job done efficiently and gracefully.* However, don't take up too much space or time explaining what things are and what needs to be done, since the amount of time spent learning it will be trivial compared to the time spent using it as an experienced user. Place state information and tools around the edges, within easy reach (**Status Display**, **Convenient Environment Actions**, **Localized Object Actions, Toolbox**), and organize the interface so that a user never needs to do a lot of manipulation to get at things they need often.

Visual clues and affordances, though they shouldn't be neglected, will not be as important in the long run as a clean, efficient interface -- use them sparingly. Note that through its use of space, a Sovereign Posture application can present to the user a very large number of possible top-level actions. This is desirable in some circumstances, but not in others; make sure that that's what you intend to do.

Resulting Context: **Pointer Shows Affordance** and **Short Description** are often used to help cut down on the visual clutter, at the expense of always-visible affordances.

Users of this sort of artifact are likely to use it for long periods of time; they should be able to change their environment to suit them. Therefore, let users rearrange the available working surfaces and tools to their own taste (**Personal Object Space**), and allow them to customize various settings (**User Preferences**). Sufficiently complex artifacts may want to permit **User's Annotations** as well.

Notes: Adapted from *About Face*, by Alan Cooper.

Someone on comp.human-factors came up with the idea of a "clue-clutter index": a single number which indicates the relative importance of heavyweight visual affordances, help, big icons, etc. Let the user pick a point on the "clue-clutter" scale that suits their needs, and the interface adjusts accordingly. It's a fun idea.

This pattern's primary emphasis is on the judicious use of space. What would it mean for a non-visual interface, such as one based on speech? I think that in this context, space can be mapped pretty cleanly into time: a motivated, experienced user with a low tolerance for hand-holding will want to be able to work through the interface as quickly as possible. I've heard blind people literally speed up their speech synthesizers when they're trying to get something done fast (so that it sounds like Alvin and the Chipmunks); they could also recognize a phrase in fractions of a second, cut it off, and move on to the next step. I couldn't understand a thing, but it obviously worked well for them.

Helper Posture

Examples:

- Print dialog
- Interactive kiosk display
- Car radio
- Poster or flyer for an event

Context: The activity supported by the artifact is secondary to other activities, but will occasionally require the user's full attention for a short time.

Problem: How should this artifact relate spatially to other artifacts that might share its space, and how can it best use the space it has?

Forces:

- The user is principally doing something else, and this shouldn't interfere with it.
- The user doesn't want the artifact around except for the short time that it's needed.
- The user has little or no incentive to spend time learning the artifact, so its learning curve should be as short as possible.

Solution: Use as much space as needed to make it comprehensible, but no more; focus tightly on the activity by excluding all but the commonest actions, information, etc. from the top level, but let those common ones take whatever space they need. Use terminology and images familiar to the user to describe these actions. Don't be afraid of verbosity -- no one will go read a manual to figure out a Helper Posture artifact, so what's there has to suffice! As Alan Cooper points out, you may also use brighter colors and larger graphic elements than you would use for a **Sovereign Posture** artifact, because the user won't see it for long enough to become irritated. In fact, they may use it so little that they forget how to use it between times; design it with new users in mind.

Always remember that the user's primary interest is in some other activity. When planning for the user's possible interactions with the artifact, you should assume that the user may stop using it at any time to return to their primary activity. **Remembered State** may be a helpful pattern to use, if the user may lose a significant amount of effort by abruptly stopping or shutting down the Helper Posture artifact.

Resulting Context: As with Sovereign Posture, the resulting context depends heavily upon which primary pattern you're building the artifact around. Still, activities supported by this pattern are generally pretty simple, so you can make the artifact's usage perfectly obvious by using patterns like **Step-by-step Instructions** (a common primary pattern for a Helper Posture artifact) and **Optional Detail On Demand**. Use **Convenient Environment Actions**, of course, and **Disabled Irrelevant Things** can help narrow down the user's available set of actions even more than you've already done in the design of the artifact.

Notes: Adapted from "About Face," by Alan Cooper. He called this pattern "Transient Posture."

Background Posture

Examples:

- Wall clock
- Computer's load indicator
- House thermostat
- Search agent

Context: The activity supported by the artifact is secondary to other activities, and will never need more than a little of the user's attention; but it should stay around for those times the user does need it.

Problem: How should this artifact relate spatially to other artifacts that might share its space, and how can it best use the space it has?

Forces:

- The user is principally doing something else, and this shouldn't interfere with it.
- The user will not bother to expend much mental effort to learn or use the artifact, because the activity's priority is low; its learning curve should be extremely short or nil.
- The user wants the artifact to stay around, in case it's needed

Solution: Make the artifact small, relative to the other primary activities going on at the same time, and keep it inobtrusive. As with Helper Posture, use familiar terminology and images to shorten the learning curve, but tightly restrain your use of space. Keep the number of available actions as small as possible. If space is at a premium, as on a computer screen, strip away any visual detail that may detract from the purpose of the artifact, and in all cases, use graphic design techniques to make it recede into the background (position it in a little-used corner, mute its colors, don't use motion or blinking, etc.).

Resulting Context: It depends entirely upon the artifact's primary pattern. This posture doesn't generally give you enough room to apply most of the patterns in this language, especially those dealing with multiple working surfaces or help. Still, if it is a **Status Display**, put the information right out there for the user to see; if it is a **Control Panel**, or **Form**, make it usable with the smallest possible amount of manipulation by the user.

Notes: Adapted from "About Face," by Alan Cooper.

How is the content or action organized into working surfaces?

Central Working Surface

Examples:

- A universal remote control, for a set of stereo or TV components
- A Web browser window, with its assorted preferences dialogs, Find dialogs, etc.
- A complex Web site's home page
- Someone's desk, office, or workshop

Bad Examples:

- Some Motif applications, designed to have an "application window" with practically nothing useful in it except a menu bar and a status area [did anyone actually like these?]

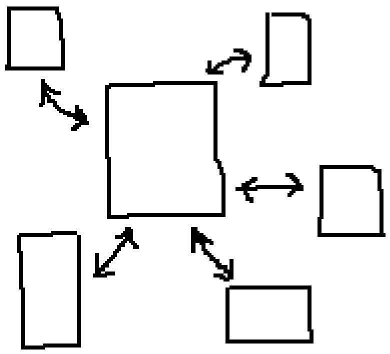
Context: The artifact is composed of multiple working surfaces, and is centered on one particular activity. This could happen in any genre.

Problem: How should the artifact's working surfaces be organized?

Forces:

- The user should have one predictable place where most of their work with the artifact gets done.
- If the artifact is complex, the user may get disoriented as they interact with its various working surfaces; they may then want a familiar "home base" to go to.
- The user needs to know where to find the basic operations of the artifact, such as the existential or state-changing commands (see **Convenient Environment Actions**) or starting points for navigation; they shouldn't have to hunt for them across many working surfaces.

Solution: *Create one working surface where the artifact's major functions are collected together; if most of the work can actually be done there, so much the better.* Secondary functionality may be placed on secondary working surfaces, which can be easily reached from the central one and which allow the user to easily get back to the central one. Place the **Convenient Environment Actions** on this surface.



If you are revising the design of an artifact, find out if users spend a lot of their time going back and forth between two or three different surfaces instead of just one. If that's the case, consider consolidating these most-frequently-used areas into one working surface, or into **Tiled Working Surfaces**, or at least into a **Stack of Working Surfaces**; the idea is to minimize the amount of unnecessary back-and-forth the user has to do. (But don't do this if it ends up inhibiting their work, obviously.)

Resulting Context: The Central Working Surface could be the first thing the user sees upon starting to use the artifact, but not necessarily -- this may be better done by some other working surface, whose express purpose is to orient the new user and direct them towards one subpart or the other. This would be especially true with an artifact which supports multiple activities, which may even require multiple Central Working Surfaces to properly support those activities (and may thus give the impression of being two or more separate artifacts). Consider a multimedia reference book, in which one may either browse casually or do a directed search for certain information. Such an artifact may have two different "home bases" where work gets done -- one for browsing, one for searching.

Notes: This pattern is more analogous to an architectural space than most others in this language. It's meant to be like a workshop, an office, or a living room, and not so much a lobby or reception room -- you do stuff there, it's within easy reach of other places you want to go, etc.

The pattern relates to the concept of a "strong center" as formulated by Christopher Alexander in his upcoming work, *The Nature of Order*. The idea is that a strong center, surrounded and supported by a set of smaller centers (e.g. the secondary working surfaces), is a fundamental organizing principle that the human mind can easily grasp. Central Working Surface evokes this principle both in terms of transition states and visual representation.

Tiled Working Surfaces

Examples:

- Newspaper page
- Car dashboard
- HTML frames
- Docked toolbars
- Multiple Emacs buffers

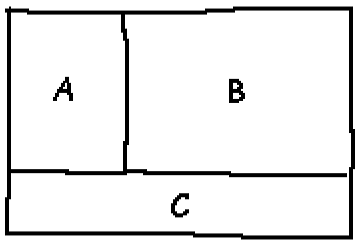
Context: The artifact displays anything visual, and can be split up into multiple working surfaces; there is enough space to show all those working surfaces at once.

Problem: How should the artifact's working surfaces be organized?

Forces:

- The user wants easy access to many working surfaces.
- Some users don't want to (or can't) manipulate multiple working surfaces to see them all.
- The spatial relationships between the working surfaces may be significant, or can be used to good effect.

Solution: *Place the working surfaces together in a plane, such that they do not obscure each other, and show the whole thing to the user.* If the surfaces can be resized or reorganized by the user, allow it, possibly by directly adjusting the borders between them or by dragging the surfaces around (see **Personal Object Space**).



Resulting Context: If two or more working surfaces hold views into **Navigable Spaces**, as with HTML frames, make sure that the navigation controls are associated with the working surface(s), not the whole set of tiled surfaces. This is what trips up so many users of HTML frames -- when they are working in a frame and try to **Go Back One Step** within that frame, the whole browser steps back to the previous site! (This is the difference between the set of environment actions and the sets of object-related actions. Netscape eventually figured this out and put the actions for a given frame onto a popup menu, including "back.")

Consider how many working surfaces a user can really view all at once. The right answer depends entirely on the content of the surfaces and what the user does with them, but make sure you give the user all they need without overwhelming them or letting them lose their place as their attention jumps from one surface to another. [*No doubt there are concrete numbers to be found w.r.t. losing one's place... any to be found in the literature?*]

Stack of Working Surfaces

Examples:

- Tab pages in dialogs
- Book with tabbed sections, like an address book
- Windows startbar with many maximized applications

Bad Examples:

- "Multi-stacked" tab pages, in which the tabs are organized into multiple rows that jump around as you select the tabs.

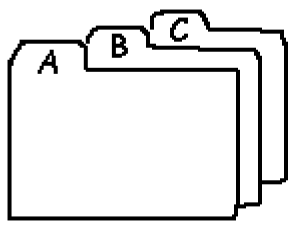
Context: The artifact displays anything visual, and can be split up into multiple working surfaces.

Problem: How should the artifact's working surfaces be organized?

Forces:

- The user wants easy access to many working surfaces.
- There may not be enough space to show them all together.
- Each surface needs, or could at least use, all the space available.
- The user can identify them by name or icon, so that they can be brought to the top when needed.
- Some users don't want to (or can't) manage the working surfaces' positions and sizes themselves.

Solution: Stack the surfaces together. Label each surface with a unique and recognizable name or icon (or let the user pick the label), and visually cluster those labels together near the stack. Provide a very simple means by which a user can indicate via the label "Bring that one to the top," such as a touch with a fingertip or a click with a pointer. If the stack is basically static, don't dynamically rearrange the relative positions of the labels, since the user then has to relearn the layout.



Resulting Context: You need to find an organizing principle for the working-surface labels. Ask yourself how they relate to each other structurally: are they a flat ordered set? A hierarchy? A network? Use a pattern that visually reflects the underlying structure, such as a **Choice from a Small Set** (e.g. tabs) for a flat ordered set, a **Hierarchical Set** for a hierarchy, and so on. In the software world, it is common to nest one Stack of Working Surfaces inside another.

Notes: Effective tab pages on dialogs seem to max out at around 10 to a stack, but no one seems to have trouble with, say, an address book with 26 tabs from A to Z. Why not? Is it a matter of knowing exactly what to expect when you get there, or is it the entirely predictable organization of the labels, or is it just easier to deal with paper tabs than virtual ones?

Pile of Working Surfaces

Examples:

- X-based window managers, and windowing OSes such as Macintosh and Windows

- MDI applications (e.g. Word, Excel, Developer Studio)
- Physical desktop
- Physical bulletin board

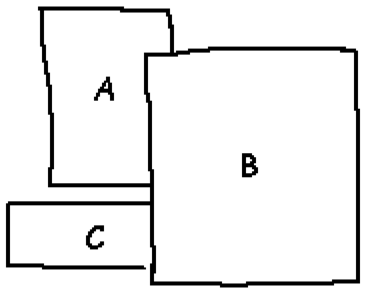
Context: The artifact displays anything visual, and can be split up into multiple working surfaces.

Problem: How should the artifact's working surfaces be organized?

Forces:

- The user wants easy access to many working surfaces.
- There may not be enough space to show them all together.
- Each surface may need a different amount of space, or can be resized independently of the other surfaces.
- The user can identify them by name or icon, so that they can be brought to the top when needed.
- The users generally wants to manage the working surfaces' positions and sizes themselves.

Solution: *Stack the surfaces loosely so that they obscure each other most of the time, but so that one or more surfaces of the user's choosing can be on top*. If the surfaces look alike when stacked close together, label each surface with a recognizable name or icon (or let the user pick the label). Provide a very simple means by which a user can indicate via the label, and by any part of the surface, "Bring that one to the top." Allow the user to work freely in any of the surfaces, even ones that are not topmost.



Though this pattern is most familiar in the "two-and-a-half-D" context of a desktop GUI, it also works effectively in a more fully 3D environment. 2D working surfaces are still important in this context-- they're needed to view documents, or images, or video clips, for example -- but you now have more freedom to scatter them throughout 3D space, with the added benefits of distance cues and (possibly) more freedom of movement to observe them from different viewpoints. Ben Shneiderman's *Designing the User Interface* has a picture of one implementation of such a space, Xerox Parc's WebBook/WebForager (pg. 530). However, he also points out that "[a] three-dimensional desktop is thought to be appealing to users, but disorientation, navigation, and hidden data problems remain." [pg. 528]

Resulting Context: To keep things conceptually simple for the user (and for the programmer), use one single "stacking plane" in which the working surfaces appear. This allows any surface to be topmost, and prevents the miserable confusion caused by

Windows applications that use two stacking planes -- one on top of the other, each containing multiple working surfaces (MDI child windows in one, dialogs in the other), and no obvious clue to the user that they have to move or get rid of all the dialogs before they can get at the MDI child windows underneath. It's awful for novices, who understandably believe that there is just one stack of windows.

How can the user navigate through the artifact?

Map of Navigable Spaces

Examples:

- Book's table of contents
- Web site map
- Geographic map
- Selection area for a suite of applications, as with the PalmPilot or a network computer
- Windows Explorer

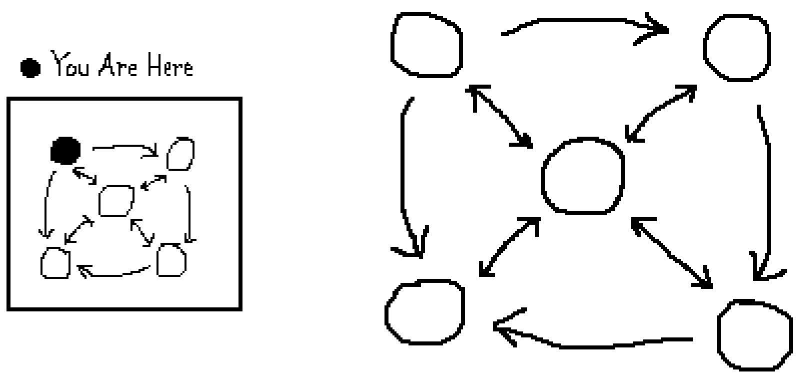
Context: The artifact (or its content) can be organized into distinct spaces or working surfaces which are semantically linked to each other, so that it is natural to go from one to another.

Problem: How can the artifact help a user navigate effectively and remain oriented?

Forces:

- The user should know where they are at any time, in a web of Navigable Spaces.
- The user wants to know where they can go next, and how to get there.
- The user should be able to see the interrelationships between the Navigable Spaces (or objects, as the case may be); this informs them about the artifact's overall structure.
- The user may need a way to jump from place A to place B, but A may not have a direct link to B -- it may be in a totally different part of the artifact.

Solution: Provide a map or diagram of the **Navigable Spaces** relevant to the artifact. Organize it appropriately, and put it where the user can easily get at it; if possible, let it be seen side-by-side with what the user is doing. At all times, show the user where he or she currently is (and there may be multiple places). If possible, allow the user to jump from one place to another by manipulating or gesturing to the map.



Resulting Context: The correct organization of the Map of Navigable Spaces is extremely important. If the spaces are related via a hierarchy, use a **Hierarchical Set**; if they form a flat set, show them in a linear fashion, as with **Choice from a Small Set**; if they form a complicated graph, draw the graph. **High-density Information Display** may give you some ideas.

You also need to figure out how to represent the sites on that map. Good text labels are important, of course; make them accurate, concise, and descriptive, so users can easily find what they need. Pictorial labels are visually interesting, but run the risk of being cryptic; still, they have been used to good effect in Web site maps that show the sizes and types of the pages' content. Try combining them with text labels.

Some sets of spaces, such as Web sites, are so large that you don't want to show them all at once. Interactive hierarchies may let their nodes be opened and closed; this is one way to cope. **Optional Detail On Demand** may also help. In any case, use your judgment and user testing to find the right amount of material to show.

Notes: For Web sites and computer interfaces, if you use the common idiom of putting the map on the left side of the screen and a view onto the chosen "space" on the right, you've effectively created a **Stack of Working Surfaces** out of this pattern! It works well, and many people are familiar with it by now.

Clear Entry Points

Examples:

- Portals or home pages on the WWW
- A book's index, introduction, and table of contents
- Entrances to buildings

Context: The artifact is organized as a set of **Navigable Spaces**; in particular, the artifact itself is large or contains a large amount of content.

Problem: How does the user know where to start?

Forces:

- A complex artifact can be very disorienting for a new user.

- Detailed indices or **Maps of Navigable Spaces** may give the user too much information, or information that is too specialized to be immediately useful to them.
- If a new user starts at a randomly-chosen place in a **Navigable Spaces** network, they probably won't have enough contextual information to figure out how best to do what they need to do.
- Users who are familiar with the artifact can't always jump straight to the space where they want to go, either because they can't remember how to get there, or because the artifact prevents them from jumping straight to it.

Solution: Provide a small set of well-defined, clearly-named entry points to the network of **Navigable Spaces**. If the artifact's purpose is narrowly defined, only one entry point may be needed, but if it will be used in different ways, there should be several entry points, specialized for the various ways in which people will use it. When a new user starts using the artifact, make sure they see these entry points first, to orient them. Also make sure that the entry points are easily accessible from anywhere in the network, so a user can return to them (see **Go Back to a Safe Place**).

One function that the entry points may serve is to give the new user the context they need to understand the whole artifact. This may involve instructions to the user, helpful tips, setting up a recognizable identity for the whole artifact, etc.

To help provide this necessary context, you might want to use the entry points to sketch out the structure of the network, to give the user an overall idea of how it's organized (see **Map of Navigable Spaces**). But don't overdo it; if there are too many entry points, the user may be overwhelmed by the number of choices available to them. The point is to use a small number that the user can easily understand and remember.

Resulting Context: *(unfinished)*

Notes: This relates directly to some of Alexander's architectural patterns from *A Pattern Language:* "Main Gateways" (#53), "Circulation Realms" (#98), "Main Building" (#99), "Family of Entrances" (#102), "Main Entrance" (#110), "Entrance Transition" (#122).

I have found that this pattern is also necessary when browsing large networks of other sorts, such as the network of object relationships in an object-oriented programming environment. Once you're in the network, you can follow the relationships (the "links") to view objects related to the one you're currently viewing, but you need to start somewhere... One way to gain initial entry into the network is to provide a simple query mechanism, with which a user can specify what subset of objects they want to start browsing with (based on class, name, attribute values, etc.).

Color-Coded Sections

Examples:

- [rewrite this] From *Information Architects:* pg 83 (Peachpit Press), 168-69 (3M), 182-83 (rainforest poster), 188-89 (Apple multimedia book), 212-213 (chair demo), 222-223 (college search)

Context: The artifact is made up of a large number of **Navigable Spaces**, which is organized into a small number of major subsections.

Problem: How can an artifact both give users a sense of place, and also tell them where they are, within a large network of spaces?

Forces:

- The user has a need to know approximately where he or she is, at all times.
- Sometimes it's impractical to use a **Map of Navigable Spaces** to show where a user is, due to space constraints, look and feel issues, or a lack of appropriate technology.
- A text label saying what section the user is in may be ugly or unnoticeable, and the user has to take the time to read and parse it.
- Iconic labels may have the same problem with being ugly or unnoticeable, and may be far more incomprehensible than text.
- Your visual interface may look better with a little livening up.

Solution: *Use color to identify the major sections of the artifact.* Pick one color per major section, and use it on every space or working surface within that section. It may be a background color, a trim color, or even a text color, but make sure it is used in the same way throughout the artifact. In particular, use it within a consistent visual framework which unifies the artifact; the user should remember the artifact's distinctive look, not the look of a subsection.

If you're using the section color as a background, or otherwise using a lot of it, then use subtler, lighter, or less saturated colors -- vivid colors in large areas make a strong impression. If the color is used in smaller visual elements, be bolder, so that the color gets noticed. Colors chosen from a single hue, but with varying lightness or saturation, may be interpreted as being "more or less of something" -- they imply an order, which may not be what you want. On the other hand, colors from different hues (red, blue, green, yellow, etc. being the primary ones) don't imply any relative order, so they may be better for sections that are random-access, or that are equal in status or importance.

Resulting Context: Don't forget to provide a way to go to other major sections (**Clear Entry Points**), or to the "home space" of the artifact (**Go Back to a Safe Place**). Consider also that you may have color-blind users, so if this is an important navigational feature, you may want to offer a non-color-based alternative in addition to the color, like text labels.

Why it works:

- Color is highly identifiable by the human visual system, probably more so than text labels or icons; a user just sees it, and doesn't have to think about it.
- It is used here as both a unifier (of a bunch of diverse spaces within one section of the artifact) and as a differentiator (between the sections). It therefore lends structure to a set of spaces, from which a user can easily form a correct mental model.
- It also helps create a sense of place for the user, when combined with a consistent visual layout -- it becomes repetitive, familiar, and reassuring, while at the same time making a layout more colorful and interesting.

Notes: --

Go Back One Step

Examples:

- The "Back" button on a Web browser
- The "Back" button on a wizard
- Turning back a page in a physical book or magazine
- The "Undo" feature on some computer applications

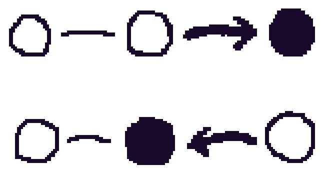
Context: The artifact allows a user to move through spaces (as in **Navigable Spaces**), or steps (as in **Step-by-Step Instructions**), or a linear **Narrative**, or discrete states.

Problem: How can the artifact make navigation easy, convenient, and psychologically safe for the user?

Forces:

- Users tend to explore a navigable artifact in a tree-like fashion, going down paths that look interesting, then back up out of them, then down another path.
- The user may want to temporarily look back at the previous space or state they were in.
- If the user gets into a space or a state that they don't want to be in, they will want to get out of it in a safe and predictable way.
- The user is more likely to explore an artifact if they are assured that they can easily get out of an undesired state or space; that assurance engenders a feeling of security.

Solution: *Provide a way to step backwards to the previous space or state.* If possible, let the user step backwards multiple times in a row, thus allowing them to backtrack as far as they want.



Resulting Context: Having a "back" function implies having a "forward" function; it's more of a convenience than a distinct pattern, but Web browsers have set up this expectation, so your users may be unpleasantly surprised if it's not there. Also, **Go Back to a Safe Place** is a logical pattern to use in addition to this one.

If the user knows they can step backwards multiple times, they may then expect that they can see the history through which they are backtracking -- in other words, their **Interaction History**.

Notes: A 1994 paper on the usage of Web browsers discovered that on average, the use of the "Back" button accounted for 40% of a user's actions. This was second only to following an actual link, which made up 52%. (In contrast, the "Forward" button only accounted for 2%.) Reference: "Characterizing Browsing Strategies in the World-Wide Web," by Lara D. Catledge and James E. Pitkow.

Go Back to a Safe Place

Examples:

- The "Home" button on a Web browser
- Turning back to the beginning of a chapter in a physical book or magazine
- The "Revert" feature on some computer applications

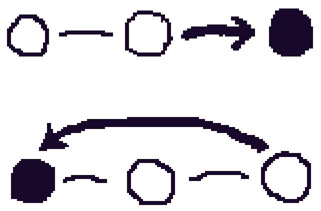
Context: The artifact allows a user to move through spaces (as in **Navigable Spaces**), or steps (as in **Step-by-Step Instructions**), or a linear **Narrative**, or discrete states; the artifact also has one or more checkpoints in that set of spaces.

Problem: How can the artifact make navigation easy, convenient, and psychologically safe for the user?

Forces:

- A user that explores a complex artifact, or tries many state-changing operations, may literally get lost.
- A user may forget where they were, if they stop using the artifact while they're in the middle of something and don't get back to it for a while.
- If the user gets into a space or a state that they don't want to be in, they will want to get out of it in a safe and predictable way.
- The user is more likely to explore an artifact if they are assured that they can easily get out of an undesired state or space; that assurance engenders a feeling of security.
- Backtracking out of a long navigation path can be very tedious.

Solution: *Provide a way to go back to a checkpoint of the user's choice.* That checkpoint may be a home page, a saved file or state, the logical beginning of a section of narrative or a set of steps. Ideally, it could be whatever state or space a user chooses to declare as a checkpoint.



Resulting Context: **Go Back One Step** is a natural adjunct to this pattern, and is often found along with it. For non-Narrative use, **Interaction History** is useful too, almost to the point of making Go Back to a Safe Place unnecessary: it may actually help a "lost" user figure out where they are, for instance, or remind an interrupted user of where they are and what they've done.

What specific actions should the user take?

Convenient Environment Actions

Examples:

- Power switches
- Pausing or resuming a video game
- OK / Apply / Cancel buttons on dialogs
- Minimize / Maximize / Quit buttons on Windows application frames

Bad Examples:

- Applications in which the only way to gracefully quit is an item on a long "File" menu
- Macintosh power switches that were next to CD-ROM drives, making them look like "Eject" buttons
- Windows 95 logout command, which is found first under "Start", then under "Shutdown"

Context: The user can take actions that affect the existence or state of the whole artifact. This is necessary in countless artifacts, with the exception of physical things like books or charts, in which those actions are absent or implicit.

Problem: How should the artifact present these actions?

Forces:

- The user should know exactly how to stop or leave this artifact at any time.
- The user should know what other actions are available.
- The user may already know what they have to do, but they need to find the corresponding action.
- The user may need to perform these in a hurry, or under stress.
- Doing these actions accidentally may be disastrous.

Solution: *Group these actions together, label them with words or pictures whose meanings are unmistakable, and put them where the user can easily find them regardless of the current state of the artifact.* Use their design and location to make them impossible to confuse with anything else. Set them up so they are not easy to trip accidentally -- hardware devices can use physical barriers to do this, but software's more difficult. Confirmation dialogs are clumsy but somewhat effective; use them until someone invents something better. If a state change or shutting down can cause disaster, at least try to make the effects reversible. Disable state-change actions whenever they become irrelevant or impossible (**Disabled Irrelevant Things**), but never do this to controls that close or quit the artifact.

Resulting Context: Icon buttons are often used for these. As long as familiar symbols are used, they work well; use the standards for the artifact's intended domain and culture. ("X" for quit, "?" for help, check-mark for OK, and 0/1 for off/on are common ones.)

When the icons for these basic functions are incomprehensible, or if you hide them in an unexpected place, the user has to carry a heavier memory burden.

Notes: Is this whole pattern just a consequence of clumsy design in the first place? Something about the "books and charts" comment in the Context makes me think about the necessity of explicit Environment Actions in software and electronics. If we could design these so well that they no longer need explicit commands for on/off, save state, change mode, etc., then we might get truly dramatic gains in usability. I don't know how to do this, of course. If I did, I would be off designing things that way, not writing this pattern.

Localized Object Actions

Examples:

- Popup (context) menus
- Direct manipulations, such as drag and drop
- Up/down controls for each of a car's windows

Context: The artifact contains multiple real or virtual objects, such as files, or CDs, or car windows. This often happens in Control Panel, WYSIWYG Editor, and Composed Command.

Problem: How should artifact present the actions that may be taken on those objects?

Forces:

- The user should know what actions they can take that affect the chosen object(s).
- The user may already know what they have to do, but they need to find the corresponding action.
- Different objects may have different sets of actions.

Solution: Group object actions together, even more so than for **Convenient Environment Actions**, and spatially localize them to the object. Separate them visually from any other kinds of actions, to avoid potential confusion. If there are actions that aren't done via buttons or other obvious visual controls -- drag-and-drop or keyboard manipulation, for instance -- make sure that there is some indication, somewhere, that they exist! (Sometimes, as with dragging objects in a **WYSIWYG Editor**, there is no need, since we "know" those actions exist, culturally and instinctively. If in doubt, test with your users.)

Resulting Context: As with **Convenient Enviroment Actions**, use **Disabled Irrelevant Things** when needed. If your users may need to perform one action on several objects at one time, then use **Actions for Multiple Objects**, keeping in mind that it may make the artifact a little bit more complex.

Notes: In desktop GUIs, users can sometimes infer which buttons on a panel are object actions and which are environment actions by watching how the actions enable and disable as they select and deselect objects. It works, but it's not a great solution, since its success depends upon (1) the users' willingness to change the selection set, (2) how alert they are to the state changes, and (3) whether they can even make an empty selection set to cause the disabling! (You can't with some container-type widgets, such as some list box implementations.) Again, separate the two classes of actions; and if you have space, reword the object actions to make it obvious that they refer to the selected objects.

As with environment actions, convention is strong in this area. Recent desktop-GUI software has really picked up on the popup menu, which is an excellent application of this pattern. Perhaps we can extend the idea by making it not just a column of verbal commands, but a popup "card" with pictures, visual cues, and interesting 2D layout...

Actions for Multiple Objects

Examples:

- Moving a set of graphic objects on a WYSIWYG editor
- Copying a batch of files from one place to another
- Locking all four car doors from the driver's seat

Context: The artifact contains multiple real or virtual objects, such as files, or CDs, or car windows. There are actions to be performed on those objects, and users are likely to want to perform actions on two or more objects at one time.

Problem: How can the artifact make repetitive tasks easier for the user?

Forces:

- People aren't good at repetitive tasks; machines are.
- Users don't want to spend time performing the same action to every single object separately.
- It might also be error-prone to do this, if the action is complex or requires precision.
- Different objects may have different sets of actions.
- Putting together a collection of objects may take time and mental effort.

Solution: *Allow the action to be performed "in parallel" across a set of user-selected objects.* Make it easy to put that collection together. Multiple-selection is the typical way to do it today in computer-based artifacts: you choose objects by shift-clicking on them or lassoing them, for instance, and then you perform the action (**Localized Object Actions**). You could also provide a separate facility for performing an action on all instances of a certain kind of object, such as a "Clear All" button on a drawing program.

What if the user has a set of multiple objects that don't share the same actions? You could disable the actions that aren't relevant to all of them (**Disabled Irrelevant Things**), but the user may not understand why they're disabled. A crystal-clear model of the object types and their available actions would help the user avoid this situation in the first place. Start there.

Resulting Context: If the action is difficult or complex, you might want to treat the set of objects to be acted upon as a form of **Editable Collection**; but it would be overkill for really simple actions. In short, don't make the user spend more effort putting together the collection of multiple objects than they would have spent performing the action on each individual object!

Choice from a Small Set

Examples:

- Set of radio buttons
- Combo box (drop-down list)
- "Circle one: Mr / Mrs / Ms" on a paper form
- Old-fashioned car radio with the mechanical buttons
- Automatic transmission shift
- Set of toggled light switches (not the push-button kind)

Bad Examples:

- Some modern car radios that don't show you which button was last selected

Context: The artifact shows, or allows the user to set, a value which is one out of a small set of possible values (10 or fewer). This often happens on **Forms** and **Control Panels**, and sometimes on **Status Displays**; it is very similar to **Choice from a Large Set**.

Problem: How should the artifact indicate what kind of information should be supplied?

Forces:

- The user should see all the possible values, to put the actual value in context.
- If the user needs to set the value (not just look at it), they should know what choices are available.
- Small numbers of things can be taken stock of quickly, and don't take up much space.

Solution: *Show all the possible choices up front, show clearly which choice(s) have been made, and indicate unequivocally whether one or several values can be chosen.* Provide a choice for "Other" or "None of the above," if that will ever be an issue -- don't prevent a user from providing correct information, if they're in a better position to know what's "correct" than you are.

Resulting Context: **Good Defaults** may let the user look at the default value, judge it to be OK, and move on without even bothering to set the value. If the choices are pictorial, or are cryptic in some other way, **Short Description** may be needed to describe the choices further.

With physical or electronic artifacts (i.e. not paper), a single selection can be enforced by causing the previous choice to "unselect" when the next choice is made. Old car radios did this, and GUI radio boxes do it as well. A user will normally discover and understand this very quickly.

Notes: The "10 or fewer" comes from Miller's number (see **Small Groups of Related Things**). For such a small number, it is often pointless to hide the choices, such as in a combo box -- if it won't cost a huge amount of space, you might as well show all the possibilities so the user can see them without going through an extra step to reveal them.

Choice from a Large Set

Examples:

- Scrolled list box
- Combo box (drop-down list)
- 50-piece socket wrench set
- Book index

Context: The artifact shows, or allows the user to set, a value which is one out of a large set of possible values (more than 10). This often happens on **Forms** and **Control Panels**, and sometimes on **Status Displays**; it is very similar to **Choice from a Small Set**., and shares much of its context with **Sliding Scale**.

Problem: How should the artifact indicate what kind of information should be supplied?

Forces:

- The user should see all the possible values, to put the actual value in context.
- If the user needs to set the value (not just look at it), they should know what choices are available.
- The user should be able to find the value they want quickly.
- Large numbers of things take a long time to read and take up lots of space.

Solution: *Clearly show the selected value up front; organize the set of possible values, but hide them nearby if they take up too much space.* Put them on a separate working surface, for instance, or in a scrolled area or combo box in a GUI environment, where they are only a single gesture away. Organize them in the way most appropriate to how the user will be searching -- alphabetically if looking for names, numerically for a font size, most-often or most-recently used for a document to edit, etc. Allow a user who knows exactly what they want to directly enter the choice, as by typing, rather than by laboriously searching it out.

Resulting Context: **Good Defaults** may let the user look at the default value, judge it to be OK, and move on without even bothering to set the value. If the choices are pictorial, or are cryptic in some other way, **Short Description** may be needed to describe the choices further.

Notes: Scrolled combo boxes are really only necessary if the dropdown list is going to run off the edge of the screen -- it's easy to miss the choices beyond the visible area, and it's awkward for many people to drop down the list, then move to the scroll bar (or buttons), then scroll up, then down, etc. It takes the bad features of scrolled lists and makes them worse, first by making you show the list and then by shrinking the scrollbar.

Sliding Scale

Examples:

- Volume knob on a stereo
- Car's gas gauge

- Dimmer switch for a light
- Scroll bars

Context: The value to be shown or set is scalar (not necessarily numeric), appears to the user to be continuous, and is bounded at both ends. This is often found in **Status Displays**, **Control Panels**, and **Forms**, but may be used for scrolling a working surface in any GUI context.

Problem: How should the artifact indicate what kind of information should be supplied?

Forces:

- The user should see all the possible values, to put the actual value in context.
- If the user needs to set the value (not just look at it), they should know what choices are available.
- A display of all the possible discrete values (as in **Choice from a Large Set**) may take up too much room, or may be too difficult to manipulate, or may provide more precision than is actually desired.
- It's often easier to directly manipulate a value than to enter it via text.

Solution: *Show the range of values visually, as with a line or arc; show the current value as a location in that line or arc, and if the value is settable by the user, make that location directly changeable by the user, as with a slider or knob -- or by simply touching or clicking on the desired value.*

If high precision is needed (e.g. "73.6" on a scale of 1 to 100), two additional features become necessary: a display of the exact current value (usually text), and the ability to precisely adjust the value without superhumanly fine motor control. This is sometimes achieved by the workings of the control itself; for instance, large circular knobs (like a volume knob) allow for very fine control of a value, while also facilitating large value changes. Linear sliders aren't nearly as good at this. In the computer world, tiny value changes are sometimes made by clicking on buttons in or near the control, like the arrows at the tops and bottoms of scrollbars.

On the other hand, if precision is unnecessary, then there's no point in cluttering up a clean, simple implementation with extra displays and controls.

Resulting Context: **Good Defaults** may let the user look at the default value, judge it to be OK, and move on without even bothering to set the value. If you're not using a textual display of the current value, but the user might sometimes want to see it, **Short Description** could be used to show it on demand (as a form of **Optional Detail On Demand**). **Pointer Shows Affordance** can help indicate to the user that the control on the scale is manipulable.

This pattern works because it sets up a natural mapping between the value and what the user sees. It's very powerful; you thus need to be careful that the visual analogy is correct. Consider what would happen with a value that changes logarithmically, for instance -- if you presented a linear slider from 1 to 1,000,000, but numbers like 1, 10, and 100 are just as likely to be picked as the numbers at the other end of the scale, those small numbers will be impossible to pick out. In this case, you could transform the problem to show a linear scale of the exponents of 10: 0, 1, 2, 3, 4, 5, 6. (In fact, **Choice from a Small Set** might work even better.)

Editable Collection

Examples:

- Shopping carts, both real and Web-based
- Email and voice-mail boxes
- Events in an electronic calendar

Context: The user should build or modify an ordered set of things, possibly (but not necessarily) chosen from a larger set.

Problem: How should the artifact indicate what the user is supposed to do with that collection?

Forces:

- The user should know what the collection currently has in it.
- The user should be able to easily add and remove things to it, and reorder the collection if they wish.
- [unfinished]

Solution: *Show the collection to the user, along with obvious ways to remove or change the position of each item. To add an item, make it eminently clear whether the user should obtain the item before or after the "add" command or gesture.* Most of the time, this is clear from context -- if a user is shopping, obviously they have to pick the item out before they put it in their cart -- but other times it's not, so indicate it with a good metaphor, or good labeling (e.g. "Add..." with a subsequent dialog), or by the imposition of constraints (e.g. **Disabled Irrelevant Things**). If duplicate items in the collection aren't meaningful, then gently disallow them.

In a visually-oriented artifact, direct manipulation is an excellent way of dealing with addition, removal, and ordering. Today's desktop GUIs offer drag-and-drop for this. If you use it, be sure to (1) offer a "dumping ground" for removed items, such as the familiar Mac trash can, if they don't get returned to their source; and (2) give some visual indication that you can do this, if there's no strong cultural indication to do so. (Software that use D&D as their primary means of interaction, such as drawing programs, have such cultural indications; others generally don't.)

Notes: Obtaining an item before or after the "add" gesture -- this is part of the old "noun-verb vs. verb-noun" debate. (Any takers?) A good reason to use "noun-verb" is multiple selection: if you want to add or remove multiple items, the user needs to be able to pick them first, then do the operation on them (see **Actions for Multiple Objects**).

Forgiving Text Entry

Examples:

- Someone's name
- Phone number, possibly with country code and/or area code, possibly without
- URL text field on most browsers

Context: The user should enter information, as on a **Form** or in a **Composed Command**, that may be formatted in any one of several ways.

Problem: How does the artifact indicate what kind of information should be supplied?

Forces:

- Physical and cultural constraints on the allowable input prevent unnecessary interpretation errors from being made.
- Too much constraint can be annoying to the user.
- The user doesn't need the cognitive burden of figuring out what specific format is acceptable.
- Users are diverse, and will likely have different cultural or personal preferences for formatting various pieces of information (especially things like names, dates, and times).
- Both computers and humans -- depending on who's interpreting the entered information -- are good at figuring out what a given text entry is intended to be, if it even vaguely resembles some accepted format.

Solution: *Allow the user to enter text in any recognizable format for that context.* Be forgiving of formatting idiosyncracies, formatting mistakes, or recognizable "typos." Place the burden of normalizing the input (i.e. making it fit one format, for use or storage) onto the recipient of the information, whether that be a human or a computer.

In theory, it's difficult to make a computer do this correctly, but in practice it often turns out to be not so bad. To illustrate, consider the fact that most Web browsers these days allow URLs to be entered in several different ways. An URL such as "http://www.foo.com" can be entered that way, or as "www.foo.com", or even just "foo". There is enough context for the browser to figure out what the full URL ought to be; there's no point in making the user type it all in!

Resulting Context: **Good Defaults** may let the user look at the default value, judge it to be OK, and move on without even bothering to set the value; it may also help suggest what kind of input is allowed.

Structured Text Entry

Examples:

- Someone's name, specifically as last, first, middle initial
- Phone number with country code and/or area code
- Digit fields on a bank deposit form

Context: The user should enter information, as on a **Form**, but that information must be in a very specific format; **Forgiving Text Entry** is not a viable option.

Problem: How does the artifact indicate what kind of information should be supplied?

Forces:

- Physical and cultural constraints on the allowable input prevent unnecessary interpretation errors from being made.

- Too much constraint can be annoying to the user.
- The user doesn't need the cognitive burden of figuring out what specific format is acceptable.
- If the user sees exactly what is expected of them, they don't have to be uncertain about what they're entering into the text field.

Solution: Rather than letting a user enter information into a blank and featureless text field, put structure into that text field. Divide it into multiple fields with different relative sizes, for instance, or superimpose a faint visual design on it (like dividers or decimal points). Be careful not to constrict the input so much that it makes things too complicated, or so that it no longer fits the possible input values that users may need to give it! Do user testing as needed to judge whether or not it's too annoying.

Resulting Context: **Good Defaults** may let the user look at the default value, judge it to be OK, and move on without even bothering to set the value; it may also help suggest what kind of input is allowed. Also, if the Structured Text Entry follows a culturally familiar pattern -- such as a column of numbers lined up by their decimal point, with currency signs nearby -- the user might get enough contextual clues from it to know what has to be provided, with little or no explanatory comments.

Toolbox

Examples:

- Shape palettes in Powerpoint, Illustrator, Macdraw, etc.
- Toolbar buttons for font styles and text alignment in Word, Netscape Composer, etc.
- Component libraries in Visual Cafe and other software development environments
- Artist's set of pens, paintbrushes, pastels, etc.
- A garage workbench

Context: The artifact supports the creation of other artifacts, as in a WYSIWYG Editor.

Problem: How should the artifact present the actions that the user may take?

Forces:

- Tools used to create things are different in kind from ordinary actions performed upon existing objects.
- A user who is learning the artifact should know what tools are available.
- An experienced user needs the tools ready to hand.
- If the user brings in new tools, they need to know where they will go.

Solution: Keep tools together, put them physically close to the user's working surface, and make sure they are distinct from other actions that the user may take. If the tools can be represented well as icons and not words, use icons; they usually represent visual objects or modes anyway. Let the users position the tools where they want (**Personal Object Space**).

Organize the tools according to how they are used, so that the most common ones are the most prominent (for instance), and localize them by task, so that the user doesn't have to jump around a lot as they're performing common complex tasks. Users can generally deal with seeing large sets of them -- no one minds a well-organized 20-item palette, and exposing them all means the user doesn't have to spend effort getting to them -- but break them up by function if necessary (**Small Groups of Related Things**).

Allow the user to introduce other tools from outside the artifact, if possible. Put the foreign tools with the others, so that the user doesn't have to deal with an artificial separation between "ours" and "theirs."

Resulting Context: In a software artifact, use **Short Description** or **Pointer Shows Affordance** if there's any doubt that the user will recognize the tools as they are statically represented. Try not to depend upon them, though -- it's easy to get lazy and use poor icons, knowing that a user can just use the tooltip for something they don't understand. The problem, of course, is that a user then has to move the mouse over every single tool to figure out what they all are. This is far from ideal.

Notes: The first force in this pattern is just my intuition, based on empirical evidence. I freely admit that don't have much of a basis for it. Unfortunately, the pattern's distinctiveness hinges on it, so any help on this front would be welcome...

How can the user modify the artifact?

User Preferences

Examples:

- Word, Excel, Netscape, and other desktop GUI applications
- Cars that save "user profiles" with seat positions, mirror angles, etc.
- Java's multiple look-and-feels, for use in GUI software

Context: The artifact will be used by people with differing abilities, cultures, and tastes.

Problem: How does the artifact present the actions that the user may take?

Forces:

- The user may not be able to use this artifact because of the language / font size / etc.
- The user may want some degree of aesthetic control over the artifact.
- Users get a sense of ownership and control over an artifact by modifying it.
- The user probably doesn't want to have to reconstruct their preferences every time they start a new session with the artifact.

Solution: Provide a place or working surface where users can pick their own settings for things like language, fonts, icons, color schemes, and use of sound. Allow users to save those preferences, so that they don't have to spend time setting them up again, but do this per user if multiple people will use it. Build the artifact itself to support such preferences. Devise a set of alternative "canned settings" that users can choose between, if they don't like the default and don't want to spend hours picking out good combinations.

Consider users who deal with these common issues, among others:

- Primary languages other than English
- Colorblind
- Visually impaired (most are not 100% blind; large fonts and high contrast help)
- Hearing impaired
- RSI (repetitive stress injuries) -- many people cannot easily use their hands

Resulting Context: These are commonly presented as a **Form** of some kind, or as a **Control Panel**.

Notes: There's a lot to be said about assistive technologies, particularly as they relate to computer artifacts, but space is short. It would be interesting to study successful uses of them and see what patterns can be found. (Are they already in this language?)

Personal Object Space

Examples:

- A physical desk, especially a messy one
- Bookshelf
- Public bulletin board (again, a physical one)
- Iconified applications in some windowing systems and OSes

Bad Examples:

- The Windows 95 task bar, which also contains iconified applications

Context: There are many things that the user needs ready access to, such as working surfaces, documents, objects, or tools. This is often useful in **Sovereign Posture** applications, especially those that are **WYSIWYG Editors** or integrated development environments; also, **Pile of Working Surfaces** is really a specialized example of this pattern.

Problem: How should the items in question be organized?

Forces:

- The user should be able to arrange things in a way that works best for them, since they know more about how they work than the artifact's designer does.
- If the user arranges a large set of items, they can better remember where things are than if the items were arranged for them.
- The user wants a sense of ownership and control over the artifact, and substantial customization contributes to that sense.
- It's tedious for the user to do all the item placement themselves, especially if they want precision or a sorting order.

Solution: *Allow users to place things where they want, at least in one dimension but preferably in two.* Start out with a reasonable default layout, however. Permit stacking, moving, grouping, aligning, "neatness" adjustments, sorting, and other layout operations. Do not capriciously rearrange the user's space -- only do automatic layout if the user specifically requests it!

Resulting Context: The artifact should maintain the user's layout between uses, so **Remembered State** is a natural next step; **User Preferences** and **User's Annotations**, other customization patterns, go hand-in-hand with this one. **Good Defaults** lets you design the initial layout well. **Actions for Multiple Objects** give the user greater efficiency as they manipulate items within the Personal Object Space.

Notes:

Scripted Action Sequence

Examples:

- "Eager," a Programming By Example system for Hypercard
- UNIX shell scripts and aliases
- Emacs keyboard macros

Context: The user needs to perform the same sequence of actions over and over and over again, with little or no variability. This often happens in artifacts which present a very wide spectrum of actions to the user, such as **WYSIWYG Editor** (in the visual realm) and **Composed Command** (in the linguistic realm).

Problem: How can the artifact make repetitive tasks easier for the user?

Forces:

- People aren't good at repetitive tasks; machines are.
- The user's flow of action through an artifact is helped by making action sequences as short as possible.
- Artifact designers can't build in shortcuts for every possible repetitive action sequence, partly because there are too many possible combinations, and partly because the repeated sequences vary too much from one individual user to the next.
- Customization of the artifact gives the user a sense of power and control over it.

Solution: Provide a way for the user to "record" a sequence of actions of their choice, and a way to easily "play them back" at any time. The playback should be as easy as giving a single command, or pressing a single button, or dropping the action object onto a control of some kind. The user should be able to give the sequence a name of their choice. Let the user review the sequence somehow, so that they can check their work or revisit a forgotten sequence to see what it did (as in **Interaction History**).

The action sequence itself could be played back literally, to keep things simple; or, if it acts upon an object which can change from one invocation to another (see **Localized Object Actions**), allow the sequence to be parameterized (e.g. use a placeholder or variable instead of a literal object). Also let them act on many things at once (**Actions for Multiple Objects**). Finally, make it possible for one action sequence to refer to another, so that they can build on each other.

Resulting Context: How the names of the scripts (or the controls that launch them) are presented, in the context of the running artifact, will likely depend heavily upon the nature of the application, but consider putting them in with the **Convenient Environment Actions** or the **Localized Object Actions** rather than making them second-class citizens. If possible, allow the user to save these actions, for later use, perhaps as part of the **User Preferences**.

The ability to save these sequences, plus the facility for sequences to build upon each other, create the potential for an entirely new linguistic or visual grammar to be invented by the user -- a grammar which is finely tuned to their own environment and work habits. This is a very powerful capability. In reality, it's programming; but if your users don't think of themselves as being programmers, don't call it that or you'll scare them off! ("I don't know how to program anything; I must not be able to do this.")

Notes: GUIs make this difficult to do well, for a variety of reasons. Composed Command interfaces generally make it easy.

This pattern was inspired by an email conversation with Mike Anderson.

User's Annotations

Examples:

- Handwritten comments on a page margin
- Sticky notes on an appliance panel

Bad Examples:

- User-written comments in Windows Help. I know it can be done, but I've never found out how, mostly because I almost never use Windows Help -- it's not easy enough to find specific pages, nor is it close enough to where I need the help most.

Context: The artifact is complex and difficult to learn, but will be used again by the same user or by others. This is common in **Sovereign Posture** artifacts, especially ones that are used infrequently.

Problem: How can the artifact help preserve the user's hard-won understanding from one use session to the next?

Forces:

- For infrequently-needed details, knowledge "in the world" is better than knowledge in the user's head -- it is more reliable, and less burdensome to the user.
- Users know better what's memorable to them than the artifact does (or the artifact's designer).
- The artifact's designer cannot possibly predict all the context or information needed for it to be used most effectively by certain users.
- Users get a sense of ownership and control over the artifact by modifying it.
- The proper spatial placement of help text can increase its effectiveness.

Solution: *Support ways for users to add their own comments and other annotations to the artifact.* Allow the users to place those annotations physically close to where they are needed, and if possible, allow for simple drawings in addition to text. Let users write private comments, for their own eyes only, and also let them write public ones that other users can read. Save the annotations from session to session, as a part of the artifact's **Remembered State**.

Resulting Context: Make sure the comments are extremely easy to use, or users are not likely to bother with it -- if it's too much effort, a user will probably choose to use sticky notes instead, or scribble a note in the manual's margin or something.

Notes: The truly ambitious designer may even want to use some kind of revision history. Be careful not to introduce too much unneeded complexity, of course.

Bookmarks

Examples:

- The "bookmarks" or "hotlist" feature on a Web browser
- Actual bookmarks in a book

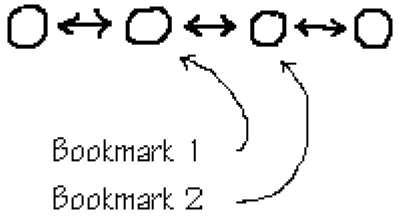
Context: The artifact is large or complex, and allows the user to move freely through it. Navigable Spaces is a common pattern to find this in, as is Narrative.

Problem: How can the artifact support the user's need to navigate through it in ways not directly supported by the artifact's structure?

Forces:

- A user may want to keep track of the places that are most interesting or most useful, for future reference.
- A user may need to temporarily stop using the artifact, but with the intention of coming back later to pick up where they left off.
- A user in "browse mode" may quickly visit a place, decide it's worth coming back to later, and keep going in a different direction; they need to keep track of where they want to return.
- A user may only be able to be in one place at a time, but want to switch between two or more places at once, perhaps because they want to compare those places.
- Users get a sense of ownership and control over the artifact when they can modify it to suit their needs.

Solution: Let the user make a record of their points of interest, so that they can easily go back to them later. The user should be able to label them however they want, since users are in a better position to choose labels that are memorable to them (see also **User's Annotations**). Support at least an ordered linear organization, so that a user can rank them according to whatever criteria they choose; if possible, support a grouping structure of some kind. Save the bookmarks for later use.



Resulting Context: You can use **Editable Collection** as a way to let the user modify the set of bookmarks. If the user can group them (e.g. **Small Groups of Related Things**) or categorize them, consider showing them as a **Hierarchical Set**. **Remembered State** can be used as a mechanism for saving the bookmarks.

Bookmarks can serve well as a set of user-defined **Clear Entry Points** to a large artifact. If you provide a complex enough organizing principle for them, then someone who uses bookmarks extensively may develop something approaching a customized **Map of Navigable Spaces**. This is good, because that user has now completely adapted their surroundings to their unique way of working. Jakob Nielsen points out in his **May 1, 1997Alertbox column** that a user may use bookmarks to make their own "map" of the site, and not use the one provided for the site -- this may cause trouble for the designers of the site, who can no longer assume that a user has entered a site by one single prescribed path.

Notes: The name of this pattern is stolen shamelessly from Netscape Navigator. It seemed to fit better than any other name I could think of.

How can the artifact be made visually clear and attractive?

Iconic Reference *(unwritten)*

Calm Grid *(unwritten)*

Repeated Framework

Examples:

- Many Web sites
- Reference textbooks
- Popular magazines
- This pattern language

Context: The artifact contains a medium or large amount of content through which the user will navigate; that content is subdivided into many pages or working surfaces.

Problem: How should the content be presented in a unified and consistent way, so that the user can easily navigate through it and quickly become familiar with it?

Forces:

- The artifact needs to be visually unified, to look like it's all of one piece. This may be done for commercial reasons, such as to establish a corporate identity, but it also improves the user's experience by helping maintain "visual momentum" as the user moves through the artifact.
- Users should easily be able to find common functionality, wherever they happen to be in the artifact. In particular, they need to find titles and/or headers, navigation controls, and exit points.
- Different content often demands different visual treatment, to make its structure and meaning clear.

- Too much sameness looks boring, and may even cause the user to get lost in the artifact.

Solution: *Design a simple, flexible visual framework for the content, then repeat it on every page or working surface; position the content within that framework, allowing the form of the content to vary as needed.* Put functional elements, such as headers and common controls, in the same place within the framework on each page; these will usually be at the edges. Use the same color scheme (more or less; see **Color-Coded Sections**) and typography throughout -- you're trying to build a look that the user can immediately recognize. Place all the elements of the framework within an invisible grid that establishes the sizes of elements and the distance between them (see **Calm Grid**). Do not make the framework too constraining, however, since all the content needs to fit gracefully in it!

Be sure you understand the structure of the content before you start, so you can take advantage of whatever consistency already exists. For example, a series of dialogs may all have a text box at the top, or a set of reference book sections might all have a single diagram in them; these can be put in the same place within the framework. Let the content speak to you.

If you need a creative boost, consider using some of these elements to construct the framework:

- Dense blocks of text or controls, where it's appropriate for the content
- Images or groups of images
- Strong left or right margins
- Significant areas of white space
- Vertical or horizontal lines
- Sidebars and margin notes, for textual content
- Very bold typefaces for headers
- Text areas that are inverted, e.g. white text on a black background, for an even stronger statement than a bold typeface

Resulting Context: Within the framework, you need to decide where to put **Convenient Environment Actions** and various navigation controls (see **Go Back One Step** and **Go Back to a Safe Place**). You also need to figure out how to handle the different content on each page. This is most commonly done in the middle area of the page, since the edges are usually taken up with framework elements. Try to stay within the grid you established for the framework; other than that, just experiment with it to see what works.

Notes: Mullet and Sano devote a few pages of their book, *Designing Visual Interfaces,* to this technique. In a section entitled "Reinforcing Structure through Repetition," they point out how little repeated structure is really necessary to achieve the effect of a unified framework:

"The programmatic effect of repetition can be based on content or visual characteristics and can be established using virtually any design element. The powerful human tendency to perceive regularity in the display leaves the designer with a wide latitude for choosing an element whose repetition facilitates communication while providing the comforting familiarity of a well-defined program."

Few Hues, Many Values *(unwritten)*

How else can the artifact actively support the user?

Good Defaults

Examples:

- A PC login screen with the last user's name still in the "User name:" field
- A phone menu which, if you don't press any number, puts you through to a human operator who can help you
- Refrigerator controls which give you a "medium coldness" setting by default

Context: The user should fill in information on a **Form** (or change settings via a **Control Panel**), and some of the data fields can be given reasonable default values. This can happen within many different subpatterns: **Choice from a Small Set**, **Choice from a Large Set**, **Sliding Scale**, **Forgiving Text Entry**, etc.

Problem: How does the artifact indicate what kind of information should be supplied?

Forces:

- Filling out forms is not inherently a fun activity; don't prolong the agony by making the user do unnecessary work.
- The user may have no clue what kind of value to supply, from the given context.
- The user may be perfectly happy with the default behavior or values, with no desire to change it; but they may want to know what the default values are.
- "Correct" values for some unfilled fields may be difficult or impossible for the artifact itself to figure out.

Solution: *Supply reasonable default values for the fields in question.* Show these defaults to the user, so that they know they aren't required to fill them in. Indicate clearly that the value can be changed by the user, if they so desire.

Resulting Context: You need to choose the correct default value. The actual value you use will depend entirely upon the particulars of the artifact, of course, but keep in mind such principles as minimal work (pick a default value that most of your users will be OK with), adaptability (change a default value to be consistent with information the user has already supplied), and representativeness (make it a good example of "correct" input).

Remembered State

Examples:

- Text boxes that retain the last text entered into them
- Windows 95 preference settings (background color, etc.) that are remembered from session to session

- Radios with user-definable button settings, remembered even when the radio is turned off
- A physical bookmark in a book

Bad Examples:

- Windows 95 file dialogs in most applications don't remember the last place in a filesystem that a file was opened from, so you have to reestablish a default directory every time you restart the application.

Context: The artifact lets users enter information (e.g. **Form**), set its state in various ways (as with **Control Panel** or **WYSIWYG Editor**), or customize it; and the artifact is likely to be used again soon by the same user.

Problem: How can the artifact help save the user time and effort?

Forces:

- Users don't want to redo their customizations from scratch every time they use the artifact.
- Users tend to develop habits and predictable ways of working with an artifact, but the artifact's designer can't necessarily predict those habits at design time.
- A user may get interrupted in the middle of working with the artifact, and have to temporarily end their session with it; they should be able to restart it later and pick up exactly where they left off, for continuity.
- Things in real life do this automatically -- your car seat stays reclined to the same place, bookmarks stay in books, etc.
- It may take time to restore an artifact's state when a use session begins, and resources to save the state in between sessions.

Solution: *Design the artifact so that it can remember its state from session to session.* If multiple users are likely to use it, make sure the state is saved on a per-user basis. The state should be recalled and reconstructed without any user intervention, so that the illusion of continuity is convincing. Keep in mind, though, that sometimes a user may not want the state to be recalled! Give the user an option to start fresh if they choose.

Chances are, not all state needs to be saved; users may not care about some of it, like keyboard focus or mouse position. Try to balance the time and resources needed to implement Remembered State against the value it adds to the user's experience. Usability testing may help you find this balance.

Resulting Context: All of the user-modification patterns can be implemented with Remembered State: **User Preferences**, to let a user set up simple options like fonts and colors; **Personal Object Space**, for extensive custom layouts; **Scripted Action Sequences**, to record series of actions; **User's Annotations**, for comments and other auxiliary help-style information; and **Bookmarks**, to keep track of landmarks in **Navigable Spaces**. They are all pretty much useless without Remembered State to carry them from session to session. If you've got a good mechanism for this pattern, consider using it for these others, if they make sense in the design.

Remembered State can also be used to establish a set of **Good Defaults** for things like a **Form** or a **Control Panel**. What worked last time for a user is likely to work again next time, under some circumstances.

Notes: -

Interaction History

Examples:

- The "history" or "visited links" feature on a Web browser
- UNIX shell's saved command history
- Logs of exchanged email or other social exchanges

Context: The user performs a sequence of actions with the artifact, or navigates through it. This may happen in **Navigable Spaces**, **Control Panel**, **WYSIWYG Editor**, **Composed Command**, and **Social Space**.

Problem: Should the artifact keep track of what the user does with it? If so, how?

Forces:

- People are forgetful of tedious details; users are not likely to remember just what they've recently done with the artifact, and computers are better at it than people are.
- The user may need to know exactly what they've just done, so they can undo their work or backtrack.
- The user may want a high-level overview of what they've done, to gain understanding that they wouldn't get just from memory.
- Audit logs are sometimes necessary, such as with legal regulatory requirements.
- Highly interactive artifacts may generate huge amounts of recordable detail.
- Describing certain actions in a human-readable way is difficult.

Solution: *Record the sequence of interactions as a "history."* Keep track of enough detail to make the actions repeatable, scriptable, or even undoable, if possible. Provide a comprehensible way to display the history to the user; most artifacts that implement this pattern use a textual representation, especially **Composed Command**, but that's not a requirement. (In fact, a history for **Navigable Spaces** may be better portrayed as a state diagram, showing single steps, backtracks, etc.) If the artifact is capable of saving its state, as with **Remembered State**, give the user the option of saving the history from session to session.

It may not be necessary to record every single transaction. Web browsers keep track of the visited sites, which is what the user presumably wants to know; they don't record printing, saving, or preference changes. But the user should have some control over how big the history gets. This could take the form of a number of history records to keep, or an expire time, or a decision to discard the entire history at the close of a session.

Resulting Context: Now that the artifact has a mechanism to keep track of the history, the user may expect that those actions are scriptable; consider implementing a **Scripted Action Sequence** based on those mechanisms. When found in **Navigable Spaces**, it also provides raw material for **Bookmarks**, if a user can pore over the history and pick out certain points of interest.

Having a history around provides the user with a set of milestones that they can use with **Go Back to a Safe Place** -- but explicitly think about whether you want to actually undo all the history between the "present" and the point in the history that the user wants to fall back to. The answer will depend upon your specific circumstances.

Notes: Jakob Nielsen pleads for better visualization of Web browsers' navigation histories in his November 1, 1997 Alertbox column:

> "Well, we can now sort the history list so that all the pages visited on a given site are listed together, but visualization is still missing. It would be very useful to have active sitemaps that showed the user's movements with footprints, showed additional detail at the current focus of attention while collapsing other regions, and also showed connections to other sites with a preview of the relevant sections of these other sites."

Progress Indicator

Examples:

- Countdown timer on a microwave oven
- Progress bars in desktop GUI software
- The percent-complete message in a Web browser during a download

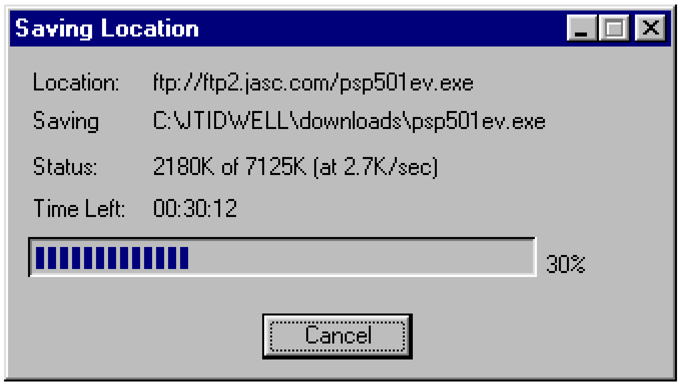
Context: A time-consuming process is going on, the results of which is of interest to the user.

Problem: How can the artifact show its current state to the user, so that the user can best understand what is going on and act on that knowledge?

Forces:

- The user wants to know how long they have to wait for the process to end.
- The user wants to know that progress is actually being made, and that the process hasn't just "hung."
- The user wants to know how fast the progress is being made, especially if the speed varies.
- Sometimes it's impossible for the artifact to tell how long the process is going to take.

Solution: *Show the user a status display of some kind, indicating how far along the process is in real time.* If the expected end time is known, or some other relevant quantity (such as the size of a file being downloaded), then always show what proportion of the process has been finished so far, so the user can estimate how much time is left. If no quantities are known -- just that the process may take a while -- then simply show some indicator that it's still going on.



Location: ftp://ftp2.jasc.com/psp501ev.exe
Saving C:\JTIDWELL\downloads\psp501ev.exe
Status: 2180K of 7125K (at 2.7K/sec)
Time Left: 00:30:12
30%

Animation is often used to good effect in this pattern; motion draws the user's attention, and its cessation implies a new relaxed, stable state of being ("the process is done, so you can relax now"). Sound can also be used this way, for the same reason. Be subtle, though, and be aware of the importance of the process relative to the other things demanding the user's attention. For example, **Background Posture** artifacts sometimes shouldn't have attention-demanding Progress Indicators at all. **Helper Posture** artifacts probably should, and so should **Sovereign Posture** artifacts if waiting on the process is suspending all other activity.

Resulting Context: A user may expect to find a way to stop the process somewhere spatially near the Progress Indicator. It's almost as though the Progress Indicator acts as a proxy for the process itself, in the user's mind. Go with it, and put a "stop" action near or on the Progress Indicator if you can.

Notes: A story about the use of sound as a Progress Indicator: I once had a workstation that had a uniquely noisy disk drive, which worked wonderfully (if unexpectedly) as a Progress Indicator for a lot of my software development activities. It wasn't too loud, fortunately, but it did have distinctive sounds for different classes of activities -- one for copying a large file, one for compiling, and so on. If I was waiting for a long compile to finish, I could work on other things without having to watch my monitor; the sudden cessation of the sound told me it was done, in a nice subtle way.

(Also, if my workstation hung or started behaving abnormally, the sound it was making often gave me a clue what was wrong. I got very familiar with the sound and timing of an 8-meg core dump. But that has little to do with Progress Indicators...)

Here's another story, from Brad Appleton:

> "Back in 1989 when I worked on a commercial version control tool for the PC, a common customer complaint was that the system took too long to start up (almost a full minute - keep in mind this was still in the days of 640K ;-).
>
> We decided it would take some time to profile the system and ferret out the bottlenecks. While we were doing that, we threw in a quick "appeaser": instead of showing nothing while starting up, we decided to show a display window with the product name/logo above a small status area that printed out each major initialization action that was taking place (e.g. "loading xxxxx" and "verifying yyyy"), much in the same way that Emacs does when it starts up.
>
> Well - after we did the above, the complaints about long start-up times stopped. We hadn't made anything faster at all (we were still trying to work on that), but we had satisfied their need to have some clue as to what's happening and when it might be finished." (From personal correspondence, dated August 16, 1998.)

Important Message

Examples:

- Alert dialogs
- Ringing telephone
- Alarms of all sorts

Context: While using the artifact, the user must be informed of something immediately. This often happens with **Status Display** and **Control Panel**, especially when they are used in life-critical situations, but could happen in other primary patterns as well.

Problem: How should the artifact convey this information to the user?

Forces:

- The user is probably paying attention to something else at the time the event happens.
- The user may be short of time or under stress, possibly as a result of the message itself, so they won't be in a good position to stop and think about how to react.
- Normal operation of the artifact after the event happens may be a bad idea.
- Too much repetition of a distracting cue can desensitize the user to it.

Solution: *Interrupt whatever the user is doing with the message, using both sight and sound if possible.* Communicate the message in clear, brief language that can be understood immediately, and provide information on how to remedy the situation, unless the cultural meaning of a non-lingual message is so strong that it cannot be mistaken (like a fire alarm). If the artifact shouldn't be used until the situation is dealt with, disable all actions until the message is acknowledged.

Use different visual and aural cues for different classes of messages, so that a tense and distracted user has some basis for distinguishing between them. Bright colors, motion or flashing, and loud, strident, or shrill sounds all work to get a user's attention. Stop the alarm after acknowledgement, or at least let the user mute anything truly distracting.

Resulting Context: Give the user an obvious way to acknowledge the message. If an audit trail is necessary or desirable, keep track of the messages over time, as with an **Interaction History**.

Notes: This is terribly overused. In truth, it's rarely the case that a user really must acknowledge a message before they can resume normal use. If this pattern is used with a non-critical message, a user's patience will quickly wear thin, and subsequent messages are at risk of being ignored (as in the boy who cried "Wolf") and bad things may happen. Use some form of **Status Display** for things which aren't critically important; don't shove them into the user's consciousness uninvited.

There's got to be good reference material out there on this subject.

Reality Check

Examples:

- When you're editing a Visual C++ source file, and it changes out from under you, Visual C++ gives you a warning message. [*find the exact situation and get screenshot*]

Context: The user is performing an action which may have destructive or nonobvious side effects, especially if that action isn't reversible. This may be one of the **Convenient Environment Actions**, for instance, or one of the **Localized Object Actions**, or a **Composed Command**; less commonly, it may also be a result of direct manipulation through a **WYSIWYG Editor** or a **Control Panel**.

Problem: How can the artifact protect itself and the user from these kinds of actions, while allowing the user to have the final say over whether or not an action is performed?

Forces:

- "Dangerous" actions should not be done without the user's explicit, informed consent.
- The user may be deeply confused by unexpected side effects to their actions (and may not even be able to connect them to the action that caused them), leading to a mixed-up mental model and distrust of the artifact.
- Users tend to not discover hidden or counterintuitive side effects to actions.
- The user may understand the ramifications of the action better than the artifact does; they should have the ultimate choice about whether or not to perform the action.

Solution: Before the action is performed, tell the user what the side effects of the action will be, and ask the user to confirm that that's what they really want to do. Don't simply parrot back the action request -- this won't tell the user anything they don't already know, unless the action request was accidental in the first place. Instead, give them an intelligent analysis of what the action may do, in case they did not anticipate the potential side effects.

Resulting Context: Once the user has seen a few Reality Checks in appropriate places, they may become used to it, and assume that the artifact will tell them whenever they're about to do something bad. On the one hand, this is good -- the user develops trust in the artifact, and feels more free to experiment with it. On the other hand, it raises the bar for the designer: make sure you have Reality Checks wherever they're needed, or the user's trust will be rudely shattered the first time the system lets them do something bad!

Notes: This is a very computer-centric pattern, since other media don't generally have the capability to understand the implications of a given action, nor react to it appropriately. Humans do, though -- what are some good examples? Executive secretaries? Athletic coaches?

Don Norman, in *The Design of Everyday Things*, points out that reflexively asking a user if they really want to do a given action doesn't work. To paraphrase one of his examples:

User:	"Remove file Foo."
Computer:	"Do you really want to remove file Foo?"
User:	"Of course." (That's what I just told you, you idiot!)

Computer: "File Foo removed."
User: "Oops."

Now what if the computer said instead, "If you remove file Foo, you will permanently lose all your custom settings for the application FooMaker. Do you want to do this?"

Demonstration

Examples:

- Video game demos that run automatically between games
- Adobe Illustrator's introductory tour and tutorials
- A friend showing you how to do something
- TV shows that demonstrate cooking techniques or hardware projects

Context: A user needs to understand how to do something complex, usually either performing a task or creating something freeform (as with a **WYSIWYG Editor**).

Problem: How can the user learn how to use the artifact?

Forces:

- The artifact doesn't clearly show the user what to do next.
- The task will be easy to remember once the user sees how to do it.
- The user doesn't have the time or inclination to read written help, such as a manual.
- Written help is useless or absent, or what needs to be done is difficult to put into words.

Solution: *Demonstrate how to do it.* Show a video clip, or drive the software to actually do the task (or something representative of a typical usage) in front of the user. Make clear what exactly is being demonstrated beforehand. Let the user repeat the demo, pause it, move slowly through it, etc. If the task in question is very freeform, demonstrate several variations of the same task, to highlight both the common and the specialized aspects.

Resulting Context: Hopefully the user will learn what they need to know. Support them when the start performing the tasks in question with extra help, such as with **Short Description** for immediate information on objects or actions, and **Step-by-Step Instructions** for very specific kinds of tasks.

Notes: Illustrator's demo tutorials turned out to be not as useful as I hoped they would be. Why not?... They were hard to follow; I didn't always know what they were doing before they did it (a running text commentary would have helped), and the mouse jumped around too much to see where it was going most of the time. I'd love to know what other people's experience has been, with this and other software tutorials.

Quick Access *(unwritten)*

Familiar Quantity *(unwritten)*

Bibliography

Alexander, Christopher. *The Timeless Way of Building.* New York: Oxford University Press, 1979.

Alexander, Christopher, Sara Ishikawa, Murray Silverstein, Max Jacobsen, Ingrid Fiksdahl-King, and Shlomo Angel. *A Pattern Language.* New York: Oxford University Press, 1977.

Alexander, Christopher. *The Nature of Order.* New York: Oxford University Press, at press.

Brown, John S., and Paul Duguid. "Keeping It Simple." In *Bringing Design to Software*, edited by Terry Winograd. New York: ACM Press, 1996.

Catledge, Lara D. and James E. Pitkow. "Characterizing Browsing Strategies in the World-Wide Web." Unpublished (?), 1994. [give URL?]

Cooper, Alan. *About Face: The Essentials of User Interface Design.* Foster City, CA: IDG Books Worldwide, 1995.

Cypher, Allen. "Eager: Programming Repetitive Tasks By Example." In *Proceedings of CHI '91*, ACM, 1991.

Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Reading, MA: Addison-Wesley, 1995.

Laurel, Brenda. *Computers As Theatre.* Reading, MA: Addison-Wesley, 1991.

Mullet, Kevin, and Darrell Sano. *Designing Visual Interfaces: Communication Oriented Techniques.* Mountain View, CA: SunSoft Press, 1995.

Nielsen, Jakob. "The difference between Web design and GUI design." *Alertbox* column of May 1, 1997.

Nielsen, Jakob. "The Tyranny of the Page: continued lack of decent navigation support in Version 4 browsers." *Alertbox* column of November 1, 1997.

Nielsen, Jakob. "Using link titles to help users predict where they are going." *Alertbox* column of January 11, 1998.

Norman, Donald. *The Design of Everyday Things.* New York: Basic Books, 1988.

Norman, Donald. *Things that Make Us Smart.* Doubleday, 1997.

Rheinfrank, John, and Shelly Evenson. "Design Languages." In *Bringing Design to Software*, edited by Terry Winograd. New York: ACM Press, 1996.

Shneiderman, Ben. "A Taxonomy and Rule Base for the Selection of Interaction Styles." In *Human Factors for Informatics Usability*, eds. Shackel, B., and S.

Richardson. Cambridge University Press, 1991. [this doesn't sound right, I think it's older than 1991]

Shneiderman, Ben. *Designing the User Interface: Strategies for Effective Human-Computer Interaction* (3rd edition). Reading, MA: Addison-Wesley, 1998.

Tognazzini, Bruce. *Tog on Software Design.* ???

Tufte, Edward. *The Visual Display of Quantitative Information.* Graphics Press, 1983.

Tufte, Edward. *Envisioning Information.* Graphics Press, 1990.

Wainer, Howard. *Visual Revelations.* ???

Wurman, Richard S. *Information Architects.* New York: Graphis, 1997.

Acknowledgements

John Vlissides, for shepherding this pattern language for PLoP '98

Jim Coplien, for excellent feedback and general advice

Tom Erickson, for early reviews and support

The Boston Patterns Group, for a writer's workshop on some of these patterns

Writer's Workshop Group 3 at PLoP '98, including Linda Rising, Bill Opdyke, Ian Chai, Oscar Nierstrasz, Fernando Lyardet, and Jens Coldewey

The "Gang of Threes" -- David Cymbala, Austin David, Mike Wilson, and Rosemary Michelle Simpson

Dan Palanza, Brad Appleton, Mike Anderson, Bob Chassell, and others that sent in comments

My managers and colleagues at InConcert, Inc., especially Doug Knowles and Becky Spainhower

Comments to: jtidwell@alum.mit.edu
Last modified May 17, 1999