Reflections on Symmetry

Harold Thimbleby
UCLIC, University College London Interaction Centre
26, Bedford Way, LONDON, WC1H 0AP, UK
h.thimbleby@ucl.ac.uk; http://www.uclic.ucl.ac.uk

ABSTRACT

Symmetry is routinely used in visual design, but in fact is not just a visual concept. This paper explores how deeper symmetries in user interface implementations can be 're-flected' in the design of the user interface, and make them easier to use. This deeper application of symmetry for user interface design is related to affordance, and therefore makes that concept constructively applicable. Recommendations for programming better user interfaces are suggested.

"Symmetry, as wide or as narrow as you may define its meaning, is one idea by which man through the ages has tried to comprehend and create order, beauty, and perfection." *Hermann Weyl* [16]

Categories and Subject Descriptors

D.1.5 [**PROGRAMMING TECHNIQUES**]: Object oriented Programming; H.5.2 [**INFORMATION INTERFACES AND PRESENTATION (e.g., HCI)**]: [H.5.2 User Interfaces (D.2.2, H.1.2, I.3.6)]; J.4 [**SOCIAL AND BEHAVIORAL SCIENCES**]: Psychology

General Terms

Algorithms, Documentation, Design, Human Factors, Languages, Theory

Keywords

Affordance, object orientation, statechart, symmetry, user interface design

1. INTRODUCTION

We are most familiar with symmetry in the spatial and visual domains, perhaps most especially as occurring in two dimensional pictures and patterns. For example, a reflection symmetry is a feature of an object that is unchanged when it is reflected in a mirror. A translation symmetry is a feature of an object that is unchanged as it is moved to another position. Wallpaper patterns are familiar objects that combine reflection, translation and sometimes rotational symmetries.

AVI 2002, Advanced Visual Interfaces International Working Conference Trento, Italy. `http://www.diel.univaq.it/avi2002/`

In general, symmetry occurs when a property of an object (e.g., a visual property) is unchanged through a transformation of the object (e.g., a rotation or reflection) [16]. Of course, most transformations change objects in one way or another, but when they do not change some properties a symmetry is involved. Symmetry, then, is a sort of consistency.

In the natural world, attacks and threats to survival are never specialised to the left or right: when one survives a threat from one side, one's chances of survival are improved by assuming the lesson learnt is symmetrical — future threats are equally likely from left or right (or even front and back).

Imagine an image of a human face. Faces have an approximate vertical bilateral mirror symmetry, and more symmetric faces are more attractive. Facial asymmetries, which are not so attractive, arise mainly through imperfections. Imperfections to one's appearance may be caused by disease or trauma, and such accidents rarely have any underlying structure that can maintain symmetries. Somebody who is symmetrical is therefore more likely to be healthy, and therefore likely to be more reproductive. To some extent, evidently, symmetry has evolutionary significance, which goes some way to explaining the widespread appeal of symmetry to humans, including in more abstract domains such as in patterns and rhythms. In human culture, symmetry has acquired æsthetic significance: it is exploited in art in the widest sense, whether the visual arts, music, poetry or rhetoric.

As we develop we learn many symmetries that become obvious and trivial to adults. For example, numerosity is unchanged under many everyday transformations (such as movement, rearrangement, or hiding and revealing). Childhood magic is entertaining because it breaks symmetries: what was empty space in a hat is transformed into a rabbit! Outside of deliberate entertainment, however, breaking everyday symmetries is generally unwanted: as in the pickpocket's artful transformation of a full pocket into an empty one.

Computers can do anything; therefore the behaviour of computers has to be carefully programmed to ensure they behave as required. It follows that if computers are to behave in a useful way (rather than seem magical or irritating) then the appropriate everyday symmetries must be programmed into them. The question is, what are the appropriate symmetries, and how can they be programmed? The answers to this question must be sought from two different directions: what is applicable and effective in programming, and what is applicable and effective for the uses of programs. Programs are not just abstract conceptions, they have meaning when executed.

When users see interactive systems, they do not see all of the systems: they can only see traces. Their user models are therefore incomplete, even if what they do know of the system behaviour is sound. To be able to operate a system beyond their direct experience, they will implicitly use ideas like symmetry: the transformation is changing to another context in the system, and the unchanged property is (hopefully) the same interaction behaviour of the system in the new context. When a symmetric user model is valid, it can be much simpler and smaller than one where each context in the system has to be understood independently.

When programmers build systems, they want to make their programs more reliable, easier to maintain, and so on. But these seem different criteria, and certainly of different importance, than the criteria that are important to the users of programs. An important way to improve programs is to replace repeated code with function calls, which creates shorter programs that are more manageable, and usually more elegant. Each function call expresses an implicit symmetry: what were different parts of the program — and in fact are different parts when the program is run — become the same.

That the same concept, symmetry, is worthwhile for both programmers and users is highly suggestive.

2. SYMMETRY AS A FORMAL CONCEPT

Define an image of a face by a function $p(x, y)$, which specifies what pixel to paint at co-ordinate (x, y). An idealised face might be mirror symmetric about the vertical line $x = 0$. Here, the transform is x goes to $-x$, and the unchanged property is p, since $p(x, y) = p(-x, y)$.

In general, if $p(v) = p(T(v))$ then a property p is symmetric under the transformation T. Further, T must be invertible, so $p(v) = p(T^{-1}(v))$ as well — e.g., if T is a rotation, the property is unchanged through a rotation, *and* a rotation back. This formal description of symmetry clearly captures the essence of the visual or physical symmetries of objects.

Symmetry allows the description of a function to be compressed. In the simple case of the face p, instead of storing all of p, we could store it only for $x \geq 0$ together with the simple fact of its mirror symmetry. Since we might assume p is an arbitrarily large pixel map, exploiting this symmetry halves the storage requirements.

Other sorts of symmetry can be used to compress p further; for instance, run length encoding exploits the translation symmetry that for many x and y, $\exists k \colon \forall i, 0 < i \leq k \colon p(x, y) = p(x + i, y)$. Each use of this symmetry saves k pixels (say, $32k$ bits) in an explicit representation of p for the smaller cost $\approx \log_2 k$ bits of storing k. Mirror and run length encoding and other symmetries can be used together, and hence obtain further compression.

As a function is compressed, it is described more and more implicitly by the rules it obeys. Thus instead of describing at length all the explicit transitions of a state transition system, one might write a program that represents the transitions more briefly. The program thereby defines (some) symmetries of the state system: the transformation is the change of state the program is applied over, and the unchanged property is the invariant meaning of the relevant fragment of the program. For example, if the program had a statement `state := OFF;` then the symmetry is: however the initial state this is applied to changes, the next-state property (in this case, going to the state `OFF`) is the same. The range of symmetries open to programmers is much more general than the conventional mathematical symmetries, such as group actions.

Symmetry can be exploited at higher levels of programming too. If the source code of a program has a translation symmetry — a pattern of at least two fragments of code that are the same but in different places — then the program can be made shorter by replacing the two pieces of code with common function calls (or using other programming techniques such as inheritance). This can be thought of as text compression, making the program shorter, by exploiting the notation of the programming language. It is well known that as a program is compressed by such techniques, it is likely to become more reliable: there is less code to maintain, and things that are supposed to be the same become guaranteed to be the same. Programs also become more reliable because of an ‘amplification’ effect in debugging: when a bug is fixed in a function, there is a corrective impact on very many parts of the program (everywhere the function is invoked). Conversely, since a single function is exposed to test in more contexts, it is likely to have more of its bugs detected per test.

Since symmetry is so useful, different specialist terms are used to describe the properties involved in different areas; *invariant* being a common term, and one used in programming.

3. SYMMETRY IN USER INTERFACES

The behaviour of a modeless user interface does not depend on what state it is in: so a change in state does not change the property ‘future behaviour.’ Hence modelessness is a symmetry.

There are typically a huge number of states in everyday state machines (such as digital clocks, mobile phones, *etc*) but it is unreasonable, on various grounds, to expect the user’s model to be so large. Therefore, the user *must* rely on symmetries: the user must assume that certain transformations leave the user’s model unchanged.

A digital clock works the same way whatever time it is, though obviously each individual time it displays represents a different state. The way in which the clock can be used is essentially unchanged as states are transformed into other states through the passage of time.

The way a video recorder is used depends only very weakly on the position of the video tape: the transformation represented by $\boxed{\texttt{PLAY}} \cdots \boxed{\texttt{STOP}}$ may change the picture, but changes no properties of interaction. Both operations are reversible (e.g., using $\boxed{\texttt{REWIND}}$) even if not self-inverse.

Conversely, a lack of symmetry in a user interface makes it harder to use. Consider a Nokia 8310 mobile phone. Pressing $\boxed{\texttt{MENU}}$ then $\boxed{\star}$ locks the keypad. Unfortunately, and unnecessarily, this rule only works when the phone is in standby. There is therefore no simple symmetry

$$\mathit{meaning}(\boxed{\texttt{MENU}}; \boxed{\star}) = \mathit{meaning}(\text{arbitrary-prefix}; \boxed{\texttt{MENU}}; \boxed{\star})$$

Even in the Nokia's calculator mode, where $\boxed{\texttt{MENU}}; \boxed{\star}$ could have meant something sensible other than keypad lock (i.e., multiply), it changes the phone's state to standby and inserts a star as part of a number to dial — pretty pointless. It appears that thinking about symmetries (even modes: these ideas were formalised over 12 years ago [11]) was not a design issue; one infers that the Nokia's program is larger than it need have been — and hence is less reliable because a larger program will have been harder to manage. These inferences are consistent with what Nokia say about themselves: they use explicit state transition diagrams for storyboarding [7].

The impact for the user is that the Nokia cannot have its keypad locked reliably without first looking at it and getting it into the standby state by whatever mode-dependent means is appropriate. With a symmetric design, the Nokia program itself would have been shorter, the user interface would have been more consistent and easier to use, and the user manual could have been slightly shorter (currently it says, [only] "In standby mode, press ... "); there is no reason for the restriction.[1]

4. AFFORDANCE AND DESIGN

Gibson [4] understood human vision by assuming certain features of vision are invariant with motion and rotation, and that the visual features, despite their physical transformations, are somehow "picked up" by the observer. He claimed the function of the brain was to "detect invariants" despite changes in visual sensations. For instance, as a face is rotated, it still looks like the same face, and this is the invariant. Once recognised, an object may stimulate some or several sorts of action, and when it does so this is in some sense a set of "natural" relations. Hence Gibson assumed the human (*i*) recognises a set of symmetries (*ii*) particular sets of symmetries stimulate particular responses. Together these ideas constitute *affordance.*

[1]If the Nokia was location aware, and for instance knew when it was in a pocket, handbag or holster — which is about the only sort of situation when keypad lock is required — there would be little need for a user interface command for keypad lock. Other mobile phone designs use a physical flap over the buttons, which (*i*) works in any state (*ii*) is itself a physical interlock so the mobile cannot be packed with the keypad unlocked (*iii*) does not require exceptions for dialling 112 and 999.

The classic examples of affordance are the door plate and the door handle, which when recognised stimulate pushing or pulling the door respectively. Door plates (more accurately, their visual images) are said to afford pushing; handles afford pulling. Occasionally one comes across doors with handles that can only be opened by pushing; occasionally one comes across doors with plates that cannot be opened by pushing. The behaviour inconsistent with affordance in each case is frustrating to users.

Affordance is an informal but stimulating concept; this led to subsequent research attempting to pin down the concept adequately to exploit it further. Norman's classic *Psychology of Everyday Things* [9] brought affordance to the attention of designers. Gaver subsequently widened the scope of affordance to the design of graphical user interfaces [3], where an added factor is that display screens can show pictures of objects (such as push buttons) that would have had affordances if actually present. Since user interface designers want to encourage users to follow appropriate courses of action — and this is by no means easy to do — affordance is a key concept for interface design. If interface features consistently "afford" certain actions, and these actions are appropriate, the interface will be easier to use.

Because affordance seems intuitive and represents a generally "good thing" it has become a popular design concept — but has been used increasingly sloppily by user interface designers [10].

Regardless of the psychological validity of Gibson's views, affordance is clearly a valid formal description of perception and action. Understanding affordance more formally, as generated by symmetry, may help avoid its sloppy use in interactive systems design. As Marr put it [8], although Gibson under-estimated the complexity of vision, his ideas attacked implementation bias in vision research, enabling it to focus on what vision achieves rather than, as had been emphasised, on how vision works at the biophysical level. Similarly in user interface design, the user is not interested in how a system is implemented nor even in how they as users respond to it (or its affordances): a more abstract approach is required, and affordance-as-symmetry seems to capture some of the crucial notions of interaction design at the right level.

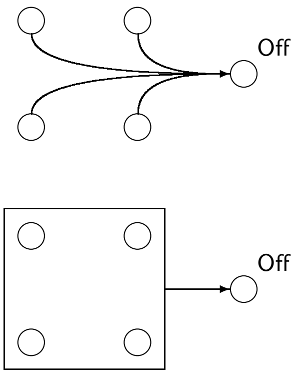
(Of course affordance is a sophisticated concept, which has accrued nuances and caveats over decades; symmetry is not a basis to formalise all of the issues now associated with affordance.)

5. VISUALISING MACHINES

Machines of even modest complexity are very messy to draw as state diagrams, and various schemes have been proposed to make diagrams clearer. Harel's Statecharts [5] are one approach for simplifying state transition diagrams.

In an explicit state transition diagram, a single program rule such as `state := OFF` would be represented in the diagram by as many arrows as states to which it applies. Typically, if a state machine S has N states, this would require $N-1$ arrows just for this one rule (since a machine can usually be switched off in all states where it is not already off).

In a statechart, however, this symmetry can be represented directly by just one arrow. The figure below compares the explicit and statechart representations for the case $N = 5$.



Amongst other symmetries, statecharts can also exploit implicit cross products: if the transitions in S have a nontrivial product structure, then the two diagrams A and B, such that $A \times B = S$, will be smaller than the diagram for S. An explicit state diagram requires the product of the number of arrows: a considerable extra visual complexity, and a worse drawing complexity that will encourage drafting errors.

User interfaces are very often cross products. Since a product $S = A \times B$ can be projected onto A, the behaviour A can be understood independently of the behaviour of other components of S. A simple example is a window system S; if the behaviour of window i is W_i, the overall system behaviour is $S = W_1 \times W_2 \times W_3 \cdots$, where each window has (in the idealisation here) independent behaviour. The symmetry is the unchanged behaviour of all windows $W_i\colon i \neq j$ regardless of the transformations the user makes to the state of window j. In practice this means a user can make changes in one window, and (although this is a transformation on S) expect the properties of all other windows to be unchanged.

The statechart representation of a window system S can represent each window as a separate box in the diagram. Thus a statechart itself exploits symmetries corresponding to those that may be represented in the user interface. If a statechart of a window system has lots of arrows criss-crossing it, then there are interactions between windows: the window system is nasty, and the statechart is nasty. (In general, in any system the arrows crossing between boxes represent non-orthogonality.) It is plausible that ‘neat’ statecharts are easier to use than ‘nasty’ statecharts. If so, then statecharts are a useful, *constructive* tool for designing interactive systems, since improving the statechart diagram improves the user interface.

A statechart makes large parts of a system specification implicit, by exploiting certain symmetries: various notational devices (such as dashed lines) are used to imply symmetry. The crucial insight is that to a large extent what is implicit/explicit in a statechart is similarly implicit/explicit in a user interface. Simplifying a statechart therefore simplifies the user interface, and in turn simplifies the user model.

Thimbleby [12] gives some examples of statecharts representing user interface issues, and [13], more specifically, gives an extended example of statecharts simplifying cross products but applied to a digital alarm clock rather than to a window system

6. DEEPER CONNECTIONS

The physical interface (whether knobs and switches in a strict physical sense, or their visual representation on a screen) presents symmetries to the user that the user may assume are implemented by the system. If button A looks like button B, and A has certain affordances, then button B should have the same affordances. On the screen there is a translational symmetry between the buttons; in a good implementation of the button behaviour, the program code will be the same for each button. In the program, the symmetry is that changing button does not change the code applied. Thus there is a deeper sort of symmetry than the superficial visual screen translation: the symmetries in the visual user interface is connected to the symmetries of the state space — so the affordances are consistent to the implemented behaviours.

The meaning of a window for the user is defined by the meaning of the set of states and transitions that implement the window (there may be ambiguities in the user interface that the user resolves through world knowledge, so the ‘meaning’ may be non-deterministic). The meaning property is unchanged through a translation from program to screen pixels. In short, there is a connection between the screen and the program; and obviously the strength of this connection will depend on how well the program is constructed to ensure the symmetry. In an object oriented programming language, the programmer would have access to representations of W_i that are independent of other W_j (i.e., thanks to encapsualation), hence connecting to the symmetries the user sees. A reason for the effectiveness of object oriented programming languages for implementing good user interfaces is that simplifying the program also simplifies the user interface (compressing, exploiting invariants, *etc*). Good object oriented programming therefore leads to good user interfaces, a benefit that is not enjoyed by non-object oriented imperative languages.

Many examples of good programming practice can be interpreted as attempts to increase reliable symmetries. As mentioned earlier, if a program has a translational symmetry (the fragment of code *here* is the same as the fragment of code *there*) then a single abstraction (procedure) should be used. With an abstraction, the programming language *guarantees* the symmetry — without an abstraction the exact translational symmetry relied on the accuracy of the programmer exactly replicating the same fragment of code. In an imperative progamming language, the symmetry can only be guaranteed to be preserved at run time if there are no global state variables; in a functional programming language, there are no state variables, so this stronger guarantee is automatically ensured. This leads to the principle: when a programming language supports a particular symmetry, the programming notation cannot refer to it (if it could, the programmer could easily break the symmetry). This is analogous to the elimination of arrows by statecharts: when no arrows are drawn, but which are implicitly represented, the programmer cannot draw them to the wrong states.

7. RECOMMENDATIONS

User interface programmers are concerned with the implementation of user interface features, but the connection between program and user interface is rarely made as explicit as we are suggesting it can be made. As a program is improved, eliminating duplicate code by using inheritance or abstraction, there will be concomitant improvements to user interface.

Statecharts are often used for representing system specifications. The connections with user interface affordance, however, suggests a more constructive role for them: tidying up a statechart diagram — by changing the system specification — is likely to tidy up the user interface in a corresponding way. In other words, a statechart is not just a record of a system specification, but is a design tool that can be used to help improve usability.

Some improvements programmers will be interested in will not be visible at the user interface, and some (e.g., some patterns) will not have a useful impact because they introduce new program objects that have no relevance to the user. The recommendation from this paper is to prioritise program optimisations and improvements that connect to the user interface, so they cause optimisations there. In turn this will make user manuals better, and so on. An optimisation with a widespread impact is a very worthwhile opitimisation.

Once a symmetry is established — whether starting at the user interface with affordances, or starting from the program with objects and classes — other structural connections follow that may be exploited, for instance with the user manuals. Just as the user interface should correspond to the program in obvious ways, the user manual should too, as should the user's model of the system. Repetitions in a user manual can be eliminated, making the manual shorter. If the user manual can be compressed by using symmetries (and remain sound and complete), then it follows that the user model (i.e., the cognitive load for learning and using a system) can also be reduced. As with statecharts, this process may be used constructively: simplifying a user manual is a process that can suggest changes to the system design to make it easier to use. A user manual need not simply be a natural language record of a fixed system specification: writing a good user manual can be an active part of the design process.

Finally, a programmer should examine programs carefully for approximate symmetries. Can the system be modified so that the approximate symmetries become exact? These two parts of the system that are similar, could they be made equivalent? These two similar parts of the user manual, could the system specification be modified so that they were the same?

User manuals do not describe all of a system (unless the system is very simple), assuming the user can cope with implicit symmetries. Yet very often the actual system will not obey the implied symmetries. For example, in many desktop computer applications text is edited in slightly different ways in different contexts (e.g., in paragraphs, tables, diagrams, dialog boxes, file dialogs, number fields ...) — typically, selection, cut and paste, undo and other features are not implemented consistently. If the manual described all these varieties of text editing subsystems faithfully, the manual would be very much longer, and very tedious to read! The success of the interactive system relies on the user coping with the inconsistencies; instead — and certainly for safety critical applications — would it not be far better if the different contexts used the same implementation of text editing: thus making the program smaller, and the faithful user manual shorter?

8. FURTHER DIRECTIONS

This is a brief paper (developed from ideas first described in [13]), suggesting a new and as yet not thoroughly developed approach. Elsewhere, we have explored particular user interface symmetries, though not then described as such [15, 14]. One of the clearest discusssions of symmetry in state models is in [2]; a more exhaustive discussion is [1].

We did not attempt to formalise 'connection' in this paper. An exceptionally clear discussion of formal program design, including Galois connections (which are generalisations of reversible transformations) is [6].

9. CONCLUSIONS

Symmetry is a universal and desirable concept at many levels of system design: relevant in program design, program representation (as in statecharts), user interface design (as in visual design), and in perception (as in affordance), and beyond into areas such as user models and user manuals. When connections can be found between different sorts of symmetries, the desirable design goals at each level are unified. Certain sorts of symmetry are desirable for programmers, and when these can be related to the sorts of symmetries desirable for users, then programmers will tend to make user interfaces better, and users will tend to understand the programs better.

From a user interface perspective, affordance, which is already widely accepted as relevant to ease of use, can be interpreted as symmetry. From a program perspective, even at the low level of state transition systems, elegance in statechart representations is a symmetry. Moreover, as parts of user interfaces and parts of statecharts can be put in correspondence, the symmetries are connected. A key contribution of this new understanding of affordance and symmetry will be a reduction in user interface implementation bugs.

Using an appropriate programming notation (of which statecharts are an example), particularly one that is object oriented, to implement the system means that the programmer can work at a higher level about the rules the program obeys in its interaction behaviour. Improving, compressing, programs (at least, ones expressed in suitable notations) improves their user interfaces.

Finally, because symmetry creates connections through implementation to affordance, and hence connections between implementation and use, user interface designers and programmers will be able to work more constructively together (a point Gaver also alludes to [3]).

Acknowledgments

Roland Backhouse, Ann Blandford, Paul Cairns and Peter Ladkin made very helpful comments.

Harold Thimbleby is a Royal Society-Wolfson Research Merit Award Holder, and acknowledges this support.

10. REFERENCES

[1] A. Carbone and S. Semmes. *A graphic apology for symmetry and implications*. Oxford University Press, Oxford, 2000.

[2] E. M. Clarke, Jr., O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, Cambridge, Massachusetts, 1999.

[3] W. Gaver. Technology affordances. In *ACM CHI'91 Conference*, pages 79–84. ACM, 1991.

[4] J. J. Gibson. *The Ecological Approach to Visual Perception*. Houghton Mifflin, Boston, 1979.

[5] D. Harel and M. Politi. *Modeling Reactive Systems with Statecharts: The Statemate Approach*. McGraw-Hill, New York, 1988.

[6] C. A. R. Hoare and H. Jifeng. *Unifying theories of programming*. Prentice Hall, London, 1998.

[7] H. Kiljander. User interface prototyping methods in designing mobile handsets. In *Proceedings Human-Computer Interaction Conference, Interact'99*, pages 118–125. IFIP, 1999.

[8] D. Marr. *Vision*. W. H. Freeman & Company, New York, 1982.

[9] D. A. Norman. *The Psychology of Everyday Things*. Basic Books, New York, 1988.

[10] D. A. Norman. Affordance, conventions, and design. *ACM Interactions*, 6(3):38–43, 1999.

[11] H. Thimbleby. *User Interface Desing*. Addison Wesley/ACM Press Frontier Series, New York, 1990.

[12] H. Thimbleby. Visualising the potential of interactive systems. In *The 10th. IEEE International Conference on Image Analysis and Processing (ICIAP'99)*, pages 670–677, 1999.

[13] H. Thimbleby. Affordance and symmetry. In C. Johnson, editor, *Interactive Systems: Design, Specification, and Verification*, volume 2220 of *Lecture Notes in Computer Science*, pages 199–217, Berlin, 2001. Springer Verlag.

[14] H. Thimbleby. Permissive user interfaces. *International Journal of Human-Computer Studies*, 54(3):333–350, 2001.

[15] H. Thimbleby and C. Runciman. Equal opportunity interactive systems. *International Journal of Man-Machine Studies*, 25(4):439–451, 1986.

[16] H. Weyl. *Symmetry*. Princeton University Press, Princeton, New Jersey, 1952.