Concept Design Moves

Daniel Jackson[1]

[1]Massachusetts Institute of Technology, Cambridge, MA, USA; dnj@mit.edu

Abstract. Great designs are rarely inventions without precedent; more often they are skillful adaptations of earlier designs. Designers work by recognizing structures they have previously seen, and taking steps they have taken before. By making such *patterns* and *design moves* explicit, we can educate designers more effectively and promote good design. This paper explains *concepts*, a modular structure for describing software behavior that allows patterns to be recognized, and proposes three pairs of design moves for software design, illustrating their application in some widely used products.

Keywords. Software design, design patterns, design moves, software concepts, modularity.

1 Introduction: Codifying Design Expertise

Accounts of design as a creative process often give the impression that design is mostly about coming up with entirely novel ideas. The fashionable term "ideation" reinforces this view, and suggests that insights emerge *ex nihilo* in the designer's mind. With the assumption that little can be done to make any individual designer more imaginative or creative, we tend to turn to a collaborative process to improve the outcome, for example by encouraging earlier prototyping, or by brainstorming with a diverse group of people. Such practices are helpful, but they are tangential to the substance of design.

1.1 Designers bring prior knowledge

Instead, we might look to how experienced designers think, and try to make explicit (and learnable) the insights that they have gained over years of experience. From my own experience watching software designers in action, and from analyzing the occasions on which I have had design insights myself, I have concluded that design ideas do not appear in a vacuum, but are usually drawn from prior experience. This does not mean that the experienced designer is not creative. Precedents rarely match exactly, so identifying them demands insight (often in the form of an analogy or abstraction), and applying them requires adaptation and skill. Design is thus less about sudden inspiration and more about patient analogizing and adjustment. Innovation is no less important, but becomes less visible, being found not in a wholesale replacement of old ideas with new ones, but rather in subtle (and sometimes unexpected) details of reworking and refinement.

Design expertise, codified in an applicable form, can offer a shortcut to inexperienced designers, who can benefit from the accumulated wisdom of the community, and it can amplify the skills of experienced designers. In a sense, such codification is what thoughtful

To appear:
NASA Formal Methods, (NFM 2022)
Pasadena, CA, May 24–27, 2022

education in all practical areas seeks: to learn by doing, but where much of the doing has already been done (by others, in the past).

1.2 Standard solutions and moves

Different kinds of expertise might be codified, including: (a) standard solutions or *patterns*, which can be adopted in different contexts; (b) *design moves*, in which the solution is not provided, but instead a standard transformation from one solution to another; and (c) *methods* for applying these, e.g. for identifying relevant solutions or moves, and making whatever adjustments are needed to apply them in a new situation.

The first two categories of reusable expertise have been articulated in a variety of design fields. Notable examples include Alexander's design patterns [3, 2], which offer standard solutions in architecture and urban/landscape design, and Altshuller's TRIZ [26], which codified 40 design moves extracted from a study of thousands of patents for physical devices. The third category is addressed tangentially in some of the pattern literature, but has yet to be fully explored.

In software engineering, patterns have been formulated and widely adopted in many areas, including object-oriented programming [10], software architectures [23], enterprise applications [9], and user interfaces [27]. While many pattern collections are available, collections of design moves are harder to come by. Code refactoring [12] is a notable exception. Some of the most influential ideas about program structure might also be seen as design moves, most notably information hiding [21] and decoupling [22].

1.3 Design vs. engineering

All of these examples in the software realm start from the point at which the observable behavior of the software has already been determined, and the problem is how to realize that behavior in code. The most far-reaching decisions—what the functions of the software will be, and how those functions will be organized—have already been made. The user's experience has been set; all that remains is the task of ensuring that the system will deliver its functions reliably and efficiently.

That task, of course, comprises all of programming and software architecture, and arguably user interface design also, so its importance should not be minimized. Nevertheless, our field has tended to focus on it at the expense of the more fundamental task of shaping the software's behavior. As Fred Brooks put it [5]: "The essence of a software entity is a construct of interlocking concepts… I believe the hard part of building software to be the specification, design, and testing of this conceptual construct, not the labor of representing it."

For this reason, I believe it's helpful to distinguish the terms *software design* and *software engineering*, reserving the first for the shaping of behavior and the second for structuring its implementation. This usage would accord with the way the term "design" is used in other fields. Adopting it is not a mere philological exercise, but a serious attempt to recognize the importance of design in its own right. As Mitchell Kapor wrote (paraphrasing slightly): "When you go to design a house you talk to an architect first, not an engineer.

Why is this? Because the criteria for what makes a good building fall outside the domain of engineering. Similarly, in computer programs, the selection of the various components and elements of the application must be driven by the conditions of use. How is this to be done? By software designers." [20]

This paper proposes some design moves for software design. These moves depend on expressing the function of a software system using structures that I call *concepts*. I will explain first what concepts are, and then present the design moves, along with examples of their application. The idea of concepts is presented more fully in a recently published book [15], which also explains the design criteria on which the design moves are based, and includes many of the examples used in this paper—but does not articulate explicitly the idea of design moves.

Concepts are not the first attempt to identify patterns in software design prior to implementation. This work was inspired and influenced in particular by Michael Jackson's problem frames [18] and Martin Fowler's analysis patterns [8]. Patterns in data models have also been explored [13].

2 Concept structuring

The behavior of a software system can be modeled as a set of interacting *concepts*. Each concept has its own state and a set of actions that update the state. Importantly, and in contrast to modules in the code, both the state and the actions are visible to the user.

The formulation of a concept's behavior is not novel. To a practitioner of formal methods, a concept is just a state machine, conventionally used to define entire systems in languages such as Alloy [16, 17], B [1], VDM [19] or Z [24]. To a software architect, a concept can be viewed as a service with a public API, like a microservice but smaller (perhaps a "nanoservice"), or as a domain (in the sense of domain-driven design [7]), with the state comprising a context that is even more "bounded" than usual. To a psychologist or social scientist, a concept is a little behavioral protocol that a user engages in, just like the protocols we use in everyday life (for example, in the way we add items to a shopping list and then check them off as we find the items in the store).

The Label concept (Fig. 1), for example, provides the functionality associated with labeling items and subsequently retrieving them through their labels. Some details of the description to note:

1. The term *Label* is overloaded to refer to the name of the concept (in the first line) and the set of labels (elsewhere). The concept is parameterized by *Item*, the type of items to be labeled.
2. The state and actions are defined using Alloy notation, augmented with a C-style update operator (and implicit frame conditions). Thus, for example, the formula *i.labels* += *l* in the *add_label* action says that the set of labels of the item *i* has *l* added to it.
3. The *clear_labels* action removes all the labels from an item, effectively removing the item from the concept's state.

```
concept Label [Item]
purpose
  classify items into overlapping categories
state
  labels: Item -> set Label
actions
  add_label (i: Item, l: Label)
    i.labels += l
  remove_label (i: Item, l: Label)
    i.labels -= l
  clear_labels (i: Item)
    i.labels := none
  find (ls: set Label): set Item
    return {i: Item | ls in i.labels}
operational principle
  if add_label (i, l) and no remove_label (i, l),
    find (l) returns items including i
  if no add_label (i, l), then find (l) returns items not including i
```

FIG. 1 *An example concept: Label.*

```
concept Todo
purpose
  track status of tasks
state
  pending, done: set Task
actions
  add_task (t: Task)
    pending += t
  remove_task (t: Task)
    t in pending + done
    pending -= t
    done -= t
  complete_task (t: Task)
    t in pending
    done += t
    pending -= t
  uncomplete_task (t: Task)
    t in done
    done -= t
    pending += t
operational principle
  following add_task (t), t is in pending until complete_task(t),
      after which t is in done
```

FIG. 2 *An example concept: Todo.*

4. The operational principle is one or more scenarios that demonstrate how the concept fulfills its purpose. In this case, the first says that if label *l* is added to item *i* and not removed, then performing a find on that label will return a set of items that includes *i*; the second says that if no addition of such a label occurs for an item, it will not be returned in a find on that label. The operational principle is intended to explain the basic operation of the concept and not all the details. Thus, it does not explain, for example, that a find on a set of labels returns the items that have all those labels, nor does it mention the *clear_labels* actions.

As another example, the Todo concept (Fig. 2) provides the basic functionality of a todo list, namely adding tasks to be displayed and marking them as done. Note that the *Task* type is not treated as a type parameter; the assumption is that task objects are generated by this concept, and that their details are not specified here. In the simplest case, a task would be just a text string.

2.1 Concept independence

Two important properties characterize concepts, distinguishing them from other behavioral structures (such as features [4] or object-oriented classes). First, each concept is *self-contained*: its behavior is defined without reference to other concepts, and concepts do not "use" each other in the way that one module or microservice in the code of a software system may use another, for example by making calls to it.

Second, concepts are *purposive*: each brings its own benefit that can be defined and evaluated without reference to another. These properties are different but nonetheless related; one can think of them both as forms of independence, the former in the realm of behavior and the latter in the realm of the needs or requirements that the behavior is intended to satisfy.

To achieve these properties, concept boundaries have to be drawn in certain ways, and not all increments of function can be described as concepts. In marked contrast, features can be used to organize the codebase of a system in almost arbitrary units. The rationale for the more restrictive notion of concept is that it ensures that concepts can be understood independently of one another, simplifying the user's mental model. Indeed, this independence of elements of a mental model is essential for its cognitive "robustness" [6]. It also allows the same concept to be instantiated in different systems, which brings benefits to both user (in terms of familiarity) and designer (in terms of reuse of design knowledge).

These properties are illustrated by the Label concept (Fig. 1). Self-containment means first that it includes the end-to-end functionality associated with labels: not only adding and removing labels, but actually using the labels to find items. It also means that, as reflected in the polymorphism of the concept, it relies on no properties of items except that they exist and can be distinguished, so it has no reliance on any other concept.

2.2 Concept synchronization

Concepts are composed together to form an application. Since no concept uses the services of another, and every concept's behavior must be visible and intelligible to the user,

```
app todo-with-labels
include
  Todo
  Label [Todo.Task]
sync Todo.remove_task (t)
  Label.clear_labels (t)
sync Todo.add_task (t)
  Label.add_label (t, PENDING)
sync Todo.complete_task (t)
  Label.remove_label (t, PENDING)
sync Label.remove_label (t, PENDING)
  Todo.complete_task (t)
sync Label.add_label (t, PENDING)
  Todo.uncomplete_task (t)
```

FIG. 3 *An example synchronization.*

traditional procedure call is not a suitable composition mechanism. Instead, we'll compose concepts by running them in parallel, synchronizing their actions where needed.

Synchronizations have the form "when action *A1* happens in concept *C1*, action *A2* happens in concept *C2*." An action in one concept can lead to any number of actions in other concepts. Synchronizations may also be conditioned on the states of the concepts. Synchronization is thus similar to the mechanism of an event-driven architecture, but there is an important distinction. A concept can refuse an action, and in that case a synchronization that would lead to that action must be blocked in its entirety. A synchronization is thus a kind of transaction, and its execution is all or nothing.

This composition mechanism is borrowed from CSP [14]. In CSP, actions in two processes are synchronized when they have the same name, and in the resulting composition, a single shared action occurs for both processes. When process actions with different names are to be synchronized, a renaming operator is first applied. For concepts, it makes more sense to allow actions with different names to be synchronized, and for a single synchronization to result in multiple actions in the various participating concepts.

Despite this difference, a fundamental property of CSP is retained. Suppose a concept *C* has a specification *S(C)*, which you can think of as the set of permitted histories of actions (called traces in CSP). Let's say that an app conforms to the specification of *C* if every history of actions of the app, when restricted to those actions relevant to *C*, is a permitted history in *S(C)*. Then given two concepts *C1* and *C2*, any composition *C1* || *C2* will conform to both specifications *S(C1)* and *S(C2)*. In other words, composition of concepts implements conjunction of specifications.

This is hardly surprising. Synchronization can never make a concept do an action it would not otherwise allow, and can therefore only restrict which actions happen. If this were not the case, a concept, once embedded in an app, might behave in an unfamiliar way, compromising the user's understanding of that concept as a distinct and separable unit of functionality.

In the absence of synchronization, a concept's actions are unconstrained, and may occur in any order consistent with the concept behavior. In practice, a user interface will limit which actions are available at any time, typically by offering certain groups of actions on certain pages, with traversal actions to navigate from page to page. Such limitations are rarely fundamental, however, and are unlikely to be imposed by the service layer that lies behind the user interface. They could in theory be described as synchronizations, but in most cases (especially for non-critical software) the effort of specifying them carefully will not be worthwhile, and they would be better addressed in the context of wireframing.

We can assemble a little application that combines the two concepts we defined earlier, allowing the user to add labels to tasks, using the label *PENDING* for tasks that are pending (Fig. 3). Some details to note:

1. The instantiation of the Label concept passes in the *Task* type of the Todo concept as a parameter, thus ensuring that the items manipulated by the Label concept are the tasks of the Todo concept.
2. The implicational structure of synchronizations is implicit. Thus, the first synchronization, for example, says that when a *remove_task* action happens in the Todo concept for task *t*, a corresponding *clear_labels* action happens in the Label concept for the same task.
3. The first synchronization is just a bit of book-keeping, ensuring that when a task is removed from the Todo concept, it doesn't remain as a labeled item in the Label concept. If it did, the removed task might appear when the user tries to find tasks by label.
4. The four remaining synchronizations bring a little magic: when a task is added, it automatically gets the *PENDING* label; and when the task is completed, the label is removed. Conversely, adding and removing the label changes the task status. This allows a user to filter tasks by their labels and by their task status at once, since the latter is now expressible with labels. (For simplicity, I've only included a *PENDING* label, but of course we could add a *COMPLETED* label too allowing the user to find completed tasks as well as *PENDING* tasks.)

The synchronizations of actions between the two concepts preserve an invariant: that the tasks classified as *pending* in the Todo concept are labeled as *PENDING* in the Label concept. You might think at first that this redundancy brings little benefit. But actually there's a valuable synergy at play. With the task classification now reflected in the labeling, the user can use the Label concept to perform filterings that might otherwise have needed additional functionality in the Todo concept. Imagine a user interface for this app: a button that filters tasks to those that are *pending* would, prior to this synchronization, have required a special implementation as a query over the state of the Todo concept, but now it can be implemented using the *find* action of Label. The benefit would be even greater if (as would occur in practice) the Label concept were extended to a rich query language, so that the *PENDING* status could be combined with other labels in conjunction and disjunction.

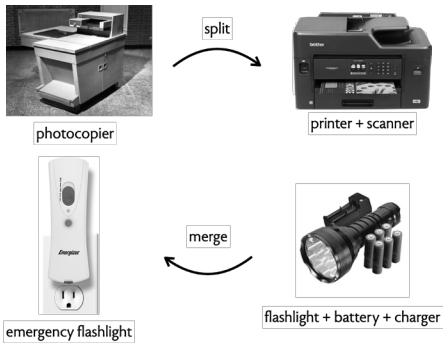

FIG. 4 *Split-merge design moves.*

FIG. 5 *Unify-specialize design moves.*

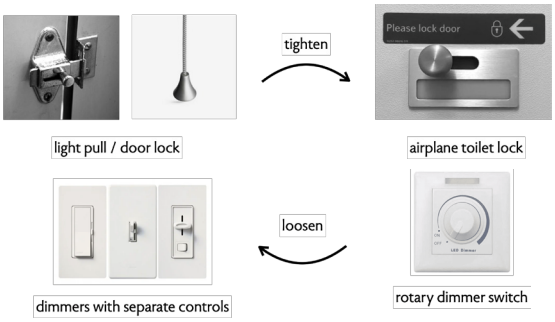

FIG. 6 *Tighten-loosen design moves.*

3 Design moves: mechanical analogues

To introduce the concept design moves in an intuitive way, I'll use some familiar mechanical analogues. There are six design moves, organized into three pairs of duals. Each pair embodies some design tradeoffs; a move in one direction benefits some property P at the cost of some other property P', with its dual in the other direction benefitting P' at the cost of P.

3.1 Split/Merge

The split/merge pair (Fig. 4) trade off simplicity and directness of usage on the one hand with flexibility on the other. The first photocopier machines offered a single concept, Photocopy say. Today's all-in-one printers also provide photocopying, but no longer as its own concept. Instead, there are two distinct concepts, Print and Scan, each with its own collection of controls and customizations. By splitting the Photocopy concept into Print and Scan, the user now has more flexibility. Photocopying is still available as a synchronization of the two, but it is marginally less convenient.

For the dual, consider the emergency flashlight, whose single concept merges the distinct concepts of Flashlight, Battery and Charger. The merging brings a loss of flexibility: you can't use the batteries in another device, for example. But the gain in simplicity is significant, which is important especially for a device designed for use in emergencies. The merged concept also permits some special functionality, for example automatically turning the flashlight on when an outlet in which it has been charging loses power.

3.2 Unify/Specialize

The unify/specialize pair (Fig. 5) trade off generality and specificity. The invention of the adjustable wrench (in the mid-19th century) solved the problem of needing a collection of wrenches to handle nuts of different sizes; one single concept replaced multiple variants. Of course the generality comes at a cost in specificity, since the adjustable wrench isn't quite as good a fit for a given nut as a plain wrench of the exact size.

A macro lens is specialized for taking close-ups. A general purpose lens can be used for close-ups, but not so effectively: both its closest focus distance and smallest aperture are typically larger than for the specialized lens. On the other hand, a macro lens is usually less suitable for other applications (such as portraits), since its widest aperture is typically smaller.

3.3 Tighten/Loosen

The tighten/loosen pair (Fig. 6) trade off automation and control. In an aircraft toilet, the light and the door lock are synchronized tightly: you can't turn on the light without locking the door. The loss in flexibility (of a cleaner being able to keep the light on and door open) comes with the benefit of a passenger avoiding embarrassment of the door being opened while using the toilet.

The dual move, in which concept synchronization is loosened, can be seen in modern dimmer switches. The earliest switches coupled together the basic light switch concept and the dimmer concept: to turn the light off you had to first dim it all the way down. In modern designs, the two concepts can be operated independently, so that a light can be turned on and off while retaining the brightness setting.

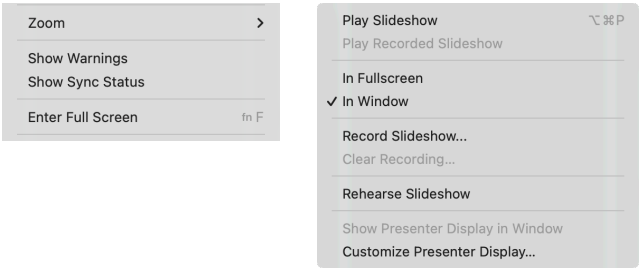

FIG. 7 *Keynote's fullscreen for edit (left) and non-fullscreen for play (right).*

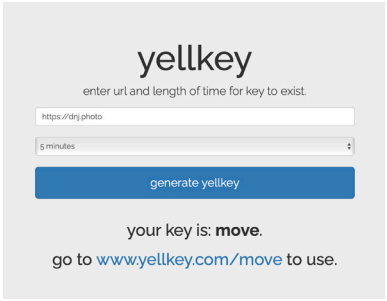

FIG. 8 *Yellkey URL shortener.*

4 Concept design moves: software examples

I'll now illustrate each of the design moves with an example from software, showing how the move was applied successfully in a familiar software product.

4.1 Split: emergence of a concept in Keynote

The Fullscreen concept (Fig. 7) has had a slow emergence. Initially, certain apps used full screen mode only for certain functions. Presentation apps such as Powerpoint and Keynote, in particular, would go full screen when the user switched from editing to presenting. Later, apps began to offer a full screen mode as an option during regular use. The final step came during the COVID-19 pandemic, when users began to make slide presentations in Zoom meetings, and needed a way to present *without* going full screen (so they could continue to see the other participants). Finally, Fullscreen became a concept in its own right that can be fully controlled independently of other concepts.

4.2 Merge: the Yellkey URL shortener

The ShortURL and Expiry concepts have been known for a long time, although they weren't typically combined. Most URL shortening services generated a short URL that was permanent. The Expiry concept is used in a variety of contexts, and is often under user control (for example, to allow you to limit the access period for a shared document). The

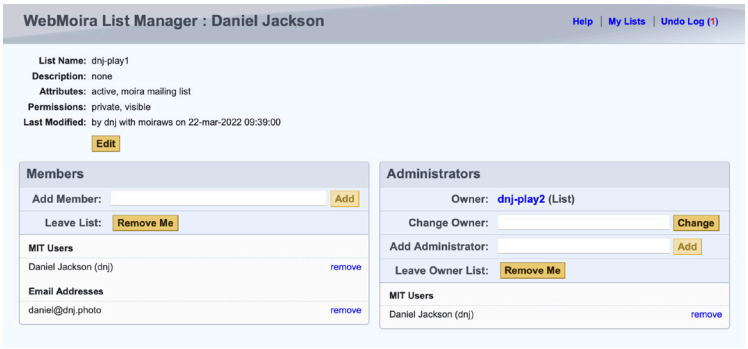

FIG. 9 *MIT's Moira mailing list service.*

Yellkey URL shortener (Fig. 8) brought these two concepts together so tightly that they no longer appear to have any independent existence: when you request a short URL, you enter the long URL and an expiry time. No other action is provided, except of course the redirection in which short URLs are expanded. The benefit of this integration is that, by ensuring very short lifetimes for short URLs, a common word can be used in place of an unreadable sequence of random characters.

4.3 Unify: MITs Moira service

My own university offers a service for mailing lists called Moira (Fig. 9). The owner of a mailing list can be a single user, but it can also be a group of users. This is handy, because it permits the burden of maintaining the group to be spread amongst multiple administrators; it also supports the common case of a professor delegating control to an assistant.

In a deft design move, Moira's designers chose to reuse the mailing list concept for administrative groups too. Essentially, there is only a single concept, List say, which unifies two concepts, MailingList and AdminGroup, which might have been separated, but which share several key actions (notably adding and removing members). The unification simplifies the user interface and its implementation, and also offers the ability to treat an administrative group as a mailing list (so you can provide an email address for the administrators of a list to those who want to join or leave).

But, as with all unifications, the move is not free of costs. A mailing list can include two types of members: internal users (who have MIT accounts, and are identified by their MIT user names) and external users (who do not have accounts, and are identified by their full email addresses). Because all administrative functions require login with an MIT user name, an external user cannot administer a mailing list. If you assign as the owner of a list a group comprising only external members, nobody can edit the list! It is also possible to create cyclic ownership—two mailing lists each of which serves as the administrative group for the other. Because the system is used by a relatively small community of mostly expert users, however, these problems do not appear to matter much in practice.

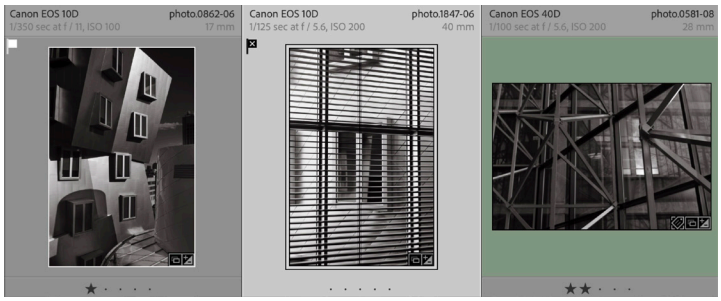

FIG. 10 *Lightroom ratings, flags and color labels.*

4.4 Specialize: three similar concepts in Lightroom

Intentional specializations are harder to find in software, since software designers tend to favor unification over specialization, often to reduce implementation effort.

In Adobe Lightroom, there are three distinct concepts that all serve the purpose of classifying photos into a small number of fixed categories (Fig. 10). The Rating concept lets you rate a photo with some number of stars between zero and five; the Flag concept lets you mark a photo as picked or rejected (or neither); the ColorLabel concept lets you assign one of a few colors to a photo. These three distinct concepts are applied by Lightroom users mostly in one particular scenario: marking uploaded photos by quality prior to deleting some of them.

There are various differences between the concepts. The colors of the ColorLabel concept are not ordered; the Rating concept, in contrast, allows you to filter for photos with more than one star (for example). The Flag concept has a rather baroque built-in action called *refine* whose effect is to cause unflagged photos to be flagged as rejected, and picked photos to be unflagged. There is also a *delete-rejected* action which can then be applied to delete the photos that are now marked as rejected (and were previously the ones not picked).

In an early version of Lightroom, ratings and color labels were associated with files (so that if a photo belonged to multiple collections, any change in its rating or label would be seen in all of them), but flags were scoped by collection, so the same photo could have a flag in one collection but not another. This was a feature with legitimate uses, but it was removed, and flags were brought into line with ratings and color labels, presumably because users found it confusing that flag metadata was not saved to file.

Whether three distinct concepts are necessary here is not clear, but the cost of the additional complexity seems minor, and users seem to appreciate the choice.

4.5 Tighten: page scheduling in Hugo

Traditional blogging platforms such as WordPress and SquareSpace offer a Schedule concept that allows a blogger to author a post now but schedule it for publication at some date in the future. Blog posts also have metadata that includes a date which is often set by de-

fault to the date on which the post was created, and thus need not match the publication date.

In the Hugo static website generator, in contrast, scheduling is implicit. Files are written in markdown and contain a preamble giving the value of various metadata fields. The date field determines the date of publication, so all a user need do to schedule a post in the future is to enter the desired date in the file. Every file is treated uniformly, so any file on the website—not just a blog post—can be scheduled in this way.

This mechanism is a synchronization of Hugo's Metadata and Schedule concepts. The Metadata concept in its general form lets you associate properties with an object, retrieve them and sometimes also sort and filter collections of objects by their properties. In this design, the date field of a file is playing two roles: one (the basic Metadata role) is to show the date of a post on the website, and to sort posts by their dates in index pages; the other (the Schedule role) is to determine the date on which the post becomes visible. In fact, many of the other metadata fields in Hugo play additional roles of this sort via synchronization.

While elegant and flexible, the design is not without problems. Because the publication date of any file can be set in this way, it is possible if one is not careful to introduce inconsistencies in which a file and the files it references are published at different times, leading to broken links.

4.6 Loosen: expert control in ProCamera

Most digital cameras offer a rich Focus concept, often with many modes and settings, that sets the focal distance of the lens automatically, and an Exposure concept that sets the exposure (the aperture, shutter speed and sometimes also the ISO speed). In most cameras, the Focus concept offers an option in which the user can select a focal point somewhere in the image, and move it around (for example with a little joy stick on the back of the camera). In more advanced cameras, exposure can either be set by averaging over the scene, or by sampling a particular point (allowing the photographer to ensure that a face, for example, is correctly exposed). In almost all cameras, the Exposure point coincides with the Focus point.

Sometimes, however, the photographer wants to focus at one point and set exposure at another. In most cameras, this can be achieved by moving the focus point to the first point and setting "focus lock," and then moving it to the second point to set the exposure (or vice versa, using "exposure lock").

In ProCamera, a camera app for the iPhone, this complexity is eliminated by loosening the synchronization between the Focus and Exposure concepts. Unlike the conventional design, in which the target point of the Focus concept is used as the target point of the Exposure concept, each concept has its own point, so that focus and exposure can be sampled independently.

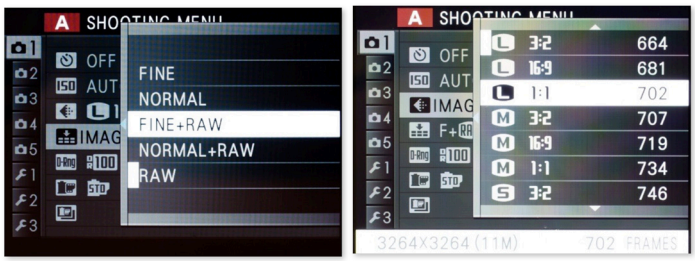

FIG. 11 *Fujifilm image quality menu (left) and image size menu (right).*

5 Solving problems with design moves

To show how design moves might be used to fix design problems, let's consider some troubled designs.

5.1 Aspect ratio in Fujifilm cameras

Fujifilm makes a range of digital cameras that are widely admired for their physical design. Their cameras have been lauded especially for their manual controls, which allow almost all adjustments to be made directly by turning a dial or ring, and without having to navigate through menus. This is especially significant because, as in most cameras, the user interface design for the virtual controls is not as refined as the mechanical design.

One example can be found in the way one selects the aspect ratio (which can only be done by menu). Most mirrorless cameras (and some digital SLRs too) let you choose an aspect ratio that differs from the sensor's native ratio. This means wasting pixels, but allows a photographer to employ a different framing, and to visualize that framing in the viewfinder. The most common non-standard ratio is probably 1:1, since it matches Instagram's preferred ratio.

On Fujifilm cameras (Fig. 11), the ratio is set in the Image Size menu (whose name already suggests a problematic design). This menu is used also to set JPEG resolution. Thus, if you want a 1:1 ratio, for example, the menu offers three options for that ratio, combining it with L, M and S settings (for large, medium and small numbers of pixels in the recorded JPEG). A separate menu, called Image Quality, lets you choose to record just a raw file, or just a JPEG file, either in fine or normal quality, or to record both raw and JPEG, with either of the two JPEG quality options.

If you choose the option to store only raw files, and no JPEGs, the Image Size menu is greyed out, and the aspect ratio reverts to the default. You might imagine that this is because custom ratios are achieved by cropping the JPEG in-camera (which is true), and that they therefore play no role for raw files (which is false). In fact, these cameras helpfully store a non-destructive crop in the raw file. But because of the strange design, if you want only raw files but with a custom ratio, you need to switch to the option to store JPEGs too, and then throw them away.

The remedy here seems straightforward. Ratio is not a proper concept in its own right; it has been merged into the Image Size concept as an additional feature. This is what I call "overloading by piggybacking": it seems as if the developer needed to find a place to insert the ratio feature and "piggybacked" it onto another concept. Applying a split and making Ratio a concept in its own right, so that the user could select the ratio independently of the JPEG size, would eliminate the problem of having to generate spurious JPEGs. It would also allow Fujifilm to support a larger number of ratios. Users are always asking for more (and there is even an online petition) but Fujifilm is presumably reluctant to do so, because of the combinatorial explosion it would produce in the Image Size menu (with its current design).

The Image Quality menu has a bad smell too in its mixing the choice of format (JPEG vs raw) and JPEG quality. There seems to be an opportunity to rework the entire menu system of digital cameras in a more systematic, concept-structured way.

5.2 Message filters in Apple Mail

Apple Mail, the default mail client on macOS, includes several strongly related concepts: Rule, which lets you define a processing rule that when matched on a message performs some action (such as moving it to a given mailbox); Search, which lets you search for messages from particular senders or with certain subjects (amongst other things); and SmartMailbox, which lets you define a mailbox containing messages that meet certain criteria.

All three concepts involve filtering a set of messages using defined criteria. For Rule, the set comprises either incoming messages (by default), or the messages in a specified mailbox; for Search, the set comprises either all messages or the messages in selected mailboxes; and for SmartMailbox, it comprises all messages.

There is no fundamental reason (as far as I can tell) that this filtering should be specialized to the three concepts. And yet each concept has not only its own user interface, but also its own filtering options, and these options are incomparable. Thus, only Rule lets you filter on whether messages are encrypted or not; only Search lets you select messages in a mailbox whose name contains a given string; and only SmartMailbox lets you choose messages by the date when last viewed. Furthermore, Rule and SmartMailbox let you conjoin or disjoin multiple conditions; Search does not.

Applying the unify move to create a single, general filtering concept might improve this design. It would allow a single user interface for the filtering aspect of all three concepts, and it would allow more powerful filtering (especially for Search, which is very limited). It would also support converting a search into a rule or a smart mailbox.

As with all unifications, there would be rough edges to handle: most notably, some warning would be desirable when a rule is applied only to incoming messages but contains a condition that does not apply to them (such as belonging to a given mailbox, or having been viewed already).

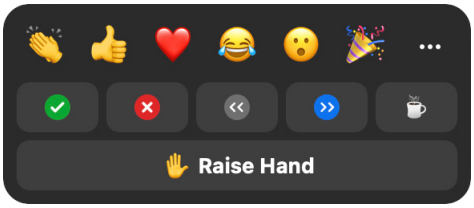

FIG. 12 *Zoom reaction options.*

5.3 Event deletion in calendars

Calendar apps such as Apple Calendar and Google Calendar synchronize two concepts together: an Event concept (which lets you store and view upcoming events) and an Invitation concept (which lets you send a request to a group of participants and receive replies). Obviously, the two concepts need to work in concert together, but in practice the synchronization has sometimes been too tight.

For years, Apple Calendar's predecessor iCal suffered from an amusing problem—although hardly amusing to users. If an event had an associated invitation, then deleting it would automatically send a reply that the invitation had been declined. This created a Catch-22 when you receive calendar spam: if you deleted the spam event, the reply would reveal to the spammer that your email address was legitimate, but if you left it, your calendar could fill up with spurious events. Various workarounds (such as moving the spam events to a new calendar and deleting the calendar in its entirety) were developed until Apple solved the problem by loosening the synchronization and offering users the option of deleting an event without declining the associated invitation.

This problem seems to have persisted longer in Google Calendar. A few years ago, seminars in our lab would routinely appear to be canceled, only for the organizer to assure people that the seminar was in fact going ahead. The problem, it turned out was that members of the lab mailing list received the seminar announcement in their email client, which in some cases automatically installed the event in their calendar. If someone then deleted their copy of the event, a message saying the event was canceled was sent to all (nearly 1,000!) members of the seminar mailing list, which was specified as a single participant in the seminar event.

It's unclear whether this problem remains in Google Calendar. In the official documentation for "Delete an event," there is no mention of deleting without canceling or sending a notification. Google Calendar does seem to present a "delete without notify" option in some cases, but according to one forum thread, only if the deletion is being requested by the owner of the event/invitation and not if by a recipient.

5.4 Sticky hands in Zoom

Finally, here is an example that seems to call for a tighten move. In Zoom, the RaisedHand concept operates entirely independently of other concepts, notably the Mute concept. And yet a common protocol for meetings has participants mute themselves until

they want to talk; then raise their hand; then unmute when called upon. Unfortunately, people often forget to lower their hands, so the moderator is uncertain of whether or not to call on them again.

Introducing some synchronization here might help. One possibility would be to offer a special meeting mode in which only one participant who is not a host can be unmuted at a time; participants raise their hands to speak, and when a host selects someone, they are simultaneously unmuted, their hand is lowered, and the previous speaker muted.

The key point here is not that the design move is obvious or trivial. On the contrary, there are many pitfalls in designing this kind of behavior while trying to maintain a balance between simplicity and flexibility. What the design move offers is not a panacea but a way to frame the design space, by encouraging the designer to separate the design of the concepts and their actions from the way in which those actions are synchronized.

A slightly different view of this problem is possible. In the current user interface of Zoom, the "raise hand" option appears when you click the "Reactions" button, along with other options such as displaying a clapping emoji (Fig. 12). This suggests that RaisedHand is not in fact a concept in its own right, but is a feature (comprising just the *raise-hand* action) that is subsumed by the Reaction concept. In this case, the appropriate design move might be first to apply a split so that RaisedHand is made a concept in its own right. That would allow a raised hand to have different behavior from a reaction. For example, a host might choose the next person to speak by clicking a button that changes their raised hand to a different icon (in contrast to reactions, which are controlled solely by the participant).

6 Discussion

The design moves described here should be viewed as an initial proposal. They are undoubtedly incomplete; they do not include, for example, some arguably even more fundamental moves (such as adjusting a novel concept to bring it into alignment with a familiar, existing concept). The distinctions between the moves are not always as clear as they might be. For example, while the general ideas of split and loosen are easy to distinguish, we saw in the context of the Zoom problem that whether one or the other applied depended on whether the raised-hand feature were viewed as a concept in its own right or just part of a Reaction concept.

I have also been a bit sloppy in identifying the exact boundaries of concepts. In the Apple Mail filtering case, for example, it seems likely that the unify design move would apply only to the filtering aspect of each of the three concepts Rule, Search and SmartMailbox, each of which would be composed with a unified Filter concept but would retain its own distinct identity. The same treatment might apply to the Moira case, so that instead of seeing the design as a unifying of MailingList and AdminGroup, it is seen as the factoring out of a unified (and shared) List concept.

Although much remains to be done, my hope is that this initial effort will inspire others to think about design in this way. Design will always remain a creative and uncertain

activity, but a good design language and design structures can empower us to work with greater confidence and clarity.

Acknowledgments

Thank you to Geoffrey Litt, Joshua Pollock and Michael Jackson for helpful discussions about design moves, and to Akiva Jackson, Rachel Jackson and Rebecca Jackson for sharing their experiences and insights about troubled concepts. The author's research was supported in part by the National Science Foundation, under the Secure and Trustworthy Cyberspace (SATC) and Designing Accountable Software Systems (DASS) programs.

References

1. Jean-Raymond Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 2005.
2. Christopher Alexander. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, 1977.
3. Christopher Alexander. *Timeless Way of Building*. Oxford University Press, 1979.
4. Don Batory and Sean O'Malley. The design and implementation of hierarchical software systems with reusable components. *ACM Transactions on Software Engineering and Methodology*, Vol. 1:4, Oct. 1992, pp. 355–398.
5. Frederick P. Brooks. No silver bullet—essence and accident in software engineering. *Proceedings of the IFIP Tenth World Computing Conference*, 1986, pp. 1069–1076.
6. Johan de Kleer and John Seely Brown. Mental models of physical mechanisms and their acquisition. In J. R. Anderson (ed.), *Cognitive Skills and Their Acquisition*, Lawrence Erlbaum, 1981, pp. 285–309.
7. Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley, 2004.
8. Martin Fowler. *Analysis Patterns: Reusable Object Models*. Addison-Wesley Professional, 1997.
9. Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Professional, 2002.
10. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.
11. Saul Greenberg and Bill Buxton. Usability evaluation considered harmful (some of the time). *Proceedings of Computer Human Interaction* (CHI 2008), Apr. 2008.
12. William Griswold and David Notkin. Automated assistance for program restructuring. *ACM Transactions on Software Engineering and Methodology* (TOSEM), Vol. 2:3, 1993, pp. 228–269.
13. David C. Hay. *Data Model Patterns*. Dorset House, 2011.
14. C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.
15. Daniel Jackson. *The Essence of Software: Why Concepts Matter for Great Design*. Princeton University Press, 2021.
16. Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT Press, 2012.

17. Daniel Jackson. Alloy: A language and tool for exploring software designs. *Communications of the ACM*, Vol. 62:9, Sept. 2019, pp. 66–76. At https://cacm.acm.org/magazines/2019/9/238969-alloy.
18. Michael Jackson. *Problem Frames: Analysing & Structuring Software Development Problems*. Addison-Wesley Professional, 2000.
19. Cliff B. Jones. *Systematic Software Development Using VDM*. Prentice Hall, 1990.
20. Mitchell Kapor. A software design manifesto. Reprinted as Chapter 1 of [28].
21. David L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, Vol. 15:12, Dec. 1972, pp. 1053–1058.
22. David L. Parnas. Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering*, Vol. 5:2, March 1979.
23. Mary Shaw and David Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Pearson, 1996.
24. John Michael Spivey. *The Z Notation: A Reference Manual*. International Series in Computer Science (2nd ed.), Prentice Hall, 1992. Full text at https://spivey.oriel.ox.ac.uk/wiki/files/zrm/zrm.pdf.
25. Bruce Tognazzini. *First Principles of Interaction Design*, revised & expanded, 2014. At https://asktog.com/atc/principles-of-interaction-design.
26. TRIZ (Wikipedia article). At https://en.wikipedia.org/wiki/TRIZ.
27. User Interface Design Patterns. At https://ui-patterns.com.
28. Terry Winograd with John Bennett, Laura De Young, and Bradley Hartfield (eds.). *Bringing Design to Software*. Addison-Wesley, 1996.