Towards a theory of conceptual design for software

Towards a Theory of Conceptual Design for Software

Daniel Jackson
Computer Science and Artificial Intelligence Laboratory
Massachusetts Institute of Technology, USA
dnj@mit.edu

Abstract

Concepts are the building blocks of software systems. They are not just subjective mental constructs, but are objective features of a system's design: increments of functionality that were consciously introduced by a designer to serve particular purposes.

This essay argues for viewing the design of software in terms of concepts, with their invention (or adoption) and refinement as the central activity of software design. A family of products can be characterized by arranging concepts in a dependence graph from which coherent concept subsets can be extracted. Just as bugs can be found in the code of a function prior to testing by reviewing the programmer's argument for its correctness, so flaws can be found in a software design by reviewing an argument by the designer. This argument consists of providing, for each concept, a single compelling purpose, and demonstrating how the concept fulfills the purpose with an archetypal scenario called an 'operational principle'. Some simple conditions (primarily in the relationship between concepts and their purposes) can then be applied to reveal flaws in the conceptual design.

Categories & Subject Descriptors: D.2.1 [Software engineering]: Requirements/Specifications—methodologies; D.2.2: Design Tools and Techniques; H.5.2 [Information Interfaces and Presentation]: User interfaces—theory and methods.

General Terms: Design, human factors, languages.

Keywords: Concepts; purposes; conceptual design; usability.

1. Two Aspects of Software Design

Software design has two aspects. One involves shaping the behavior of the software, how it will be perceived by its users, and what impact it will have in the environment in which it operates. This is *conceptual design* in two senses: first, in the conventional sense in which the term is used in other engineering disciplines—namely, it is where the very idea of the software is first conceptualized; and second, in the sense of 'conceptual modeling', since it requires the invention (or at least recognition) of the fundamental concepts in terms of which the software's behavior will be understood.

The other aspect involves the selection of structures to realize the behavior and concepts of the conceptual design; this might be called *representation design* since it focuses on how abstractions are to be represented as data structures, modules, and so on. Conceptual design is the domain of conventional requirements analysis, project scoping and specification; representation design covers software architecture, module design, and implementation.

Traditionally, conceptual design and representation design have been separated into distinct phases. But the recognition that there are two fundamentally different aspects of design does not presume a particular ordering of tasks, and applies equally to 'agile' processes in which the two are interleaved. Recently, decisions about when conceptual design should take place, and how (or even whether) it is recorded, have been the subject of endless debate. But that conceptual design exists and matters has not been questioned. After all, ideas are the raw material of software, and everyone recognizes that you cannot build good software from bad ideas.

2. Little Research on Conceptual Design

Much research effort has been directed at conceptual design, but very little of it has focused on design itself. Instead, the main concerns has been how conceptual designs should be recorded (for example, as semantic data models or as state machines; with programming languages or specification languages based on sets and relations; with text or with diagrams; and so on). Analysis has been a focus too, and a key motivation for formalizing design models. But this analysis has typically been about exploring details of behavior (for example, with model checkers) or justifying a representation design based on its relationship to a conceptual design (for example, with proofs of correctness), and not about determining whether the design is actually fit for purpose.

More surprisingly, perhaps, despite much talk of the importance of getting the concepts right, little work has been done to clarify what exactly a 'concept' is, how to identify the concepts that underlie a software system, and how to distin-

Onward!'15, October 25–30, 2015, Pittsburgh, PA, USA
ACM. 978-1-4503-3688-8/15/10...
http://dx.doi.org/10.1145/2814228.2814248

guish good concepts from bad ones. If we could do this better, we might end up not only with conceptual designs that work better for the user, but which—due to their clarity and simplicity, amongst other qualities—would provide an easier transition to representation, resulting in more robust and maintainable implementations.

3. Concepts Define Systems

The concepts of a software system are the ideas you need to understand in order to use it. For example, to use a word processor such as Microsoft Word effectively, you need to know what a *paragraph* is (whereas no such concept is required to use a text editor). To use Twitter, you need to understand *tweets*, *hashtags* and the concept of one user *following* another. To use Facebook, you need to understand *posts*, *tags* and *friends*. To use Adobe Photoshop, you need to understand *pixels*, *layers* and *masks* (and *channels*, *profiles* and so on). Some concepts, such as *tweets*, are simple and easy to grasp; some, such as *tags*, are more complicated. Some systems, such as Twitter, have only a few concepts; others, such as Photoshop, have many.

Aside. Throughout the paper, concepts appear *like this*.

Concepts are most evident in applications, but play an important role in infrastructural services too. To use the web, for example, you need to know what a *URL* is, and to be a power user you also need to understand the distinction between a *domain name* and an *IP address*. Concepts distinguish one system from another. The concept of the *trash*, for example, invented by Apple for the Lisa computer in 1982 and carried over to the Macintosh, distinguished that operating system from all others until later copied in Microsoft Windows. Concepts also distinguish families. Thus 'text editors' typically offer *lines* and *character encodings* as their key concepts; 'word processors' offer *paragraphs*, *formatting* and *styles*; and 'desktop publishing programs' offer *stylesheets*, *page templates*, and *text flows*.

Not all software is user-facing, of course. Concepts may still play an important role in such systems. In railway signalling, for example, *routes* were introduced to simplify point and light settings, and reduce the risk of train collisions.

Concepts are not only for the user, but also for the developer. A programmer working on the codebase of a text editor would have to grasp the concept of a *line* (and how exactly a line is terminated). The need to understand a complex of subtle concepts creates entire subspecialties. You cannot work as a programmer in the domain of typography, for example, if you don't know the difference between *kerning* and *letter spacing*; nor can you work in the domain of air traffic control if you are not familiar with the concept of *flight plan* or *waypoint*.

Aside. When discussing concepts, it's convenient to use singular and plural forms interchangeably. So *paragraph* and *paragraphs* will be names for the same concept.

4. Where Concepts Come From

Software systems often embody preexisting concepts. The health care systems that handle doctors' *prescriptions*, for example, did not invent that concept; doctors wrote prescriptions on paper, and they played a similar role prior to computerization. Some old concepts acquire new significance when incorporated in software. The concept of *paragraphs*, for example, goes back to the ancient Greeks, who used a special mark known as the *paragraphos* to divide groups of sentences. But the Greeks did not anticipate the fundamental role that paragraphs play in Microsoft Word—namely to provide the primary unit of formatting.

Most, and perhaps all, concepts are *invented*. Some, like the *paragraph*, were invented in ancient times; some, like the *social security number*, are modern inventions. A software concept may appeal to an analogy with a real-world concept but in fact be a totally new invention, motivated by a new purpose. The Macintosh *trash*, for example, looks like a physical trashcan but plays a very different role. The purpose of a physical trashcan is to make disposal of trash more convenient by staging it (first into the can in the room; then into the building's larger receptacle; then into a dumpster; and finally into landfill). The purpose of the Macintosh trash, in contrast, is to allow deletions to be undone.

Some invented concepts are fundamental enablers: without them, an application would barely work at all. It's hard to imagine a spreadsheet without the concept of *relative references*, for example, which make it possible to cut-and-paste formulas from cell to cell.

5. Abstract Affordances

The psychologist J.J. Gibson recognized that physical objects convey clues about how they are to be used. Through our perception of the world, and our familiarity with the objects we encounter, we are able to sense that a chair can be sat on, or a door opened, without any great cognitive effort. Gibson coined the term 'affordance' to describe this relationship; thus a chair affords sitting, a door affords opening, a screwdriver affords turning, and so on.

Don Norman, in his influential book *The Psychology of Everyday Things* (later renamed *The Design of Everyday Things*) [21], developed a theory of usability based on this notion of affordance. Well designed objects, he asserted, convey their affordances in a clear way. Thus a door handle should be shaped so that it is obvious whether to pull or push; if a sign is needed to tell you which (a 'user manual', as Norman calls it), the designer has surely failed. Norman's notion of affordance became so popular that designers began talking about the features that conveyed affordanc-

es as 'affordances' themselves; thus 'a slider is a kind of affordance'. He therefore suggests (in the latest edition of the book [22]) that we use the term 'signifier' for the feature of the design that signals the affordance.

Because software systems lack the tangibility of physical objects, their affordances tend to be less evident. Even as our devices become more physical and respond to more gestures (such as swiping, tipping or shaking), the problem remains, since the functionality associated with these signifiers is rarely concrete and visible. It is hardly obvious, for example, that shaking an iPhone should undo the last command (although that is what the user interface guidelines dictate).

There is a more fundamental problem with the usability of software in comparison to physical tools, however. The affordances of a physical tool are associated with immediate effects in the environment. Opening a door allows you to walk through immediately; sitting in a chair provides instant support; lighting a match makes a flame (sometimes too quickly!). When the desired purpose is not immediately accomplished, progress towards it is often tangible. Turning a screw progressively tightens a coupling, and even if the tightening has not begun, the user can often see the screw moving towards the surface or item to be coupled.

Software systems, like physical tools, have an impact in the environment in which they are used: causing packages to be mailed, pages to be printed, X-rays to be taken, and so on. And users perform actions in order to produce these effects. But most of the actions performed by users in software systems have no immediate effect in the environment. Sometimes this is due just to delay and failures: sending an email message does not result in it being immediately received; queuing a file to be printed does not mean it will necessarily emerge from the printer. More fundamentally, many actions only directly affect internal state. Thus modifying the definition of a style has no visible effect on a document unless there is a paragraph that has been assigned that style; completing a contact entry in an address book has no effect until that contact is used in some way; storing a bookmark does nothing until the bookmark is later selected and visited.

> *Aside*. Sometimes the indirect effects are surprising and insidious. Tagging a person in a photo on Facebook, for example, changes the accessibility of the photo so that it becomes readable not only by the person who posted the photo and her friends, but also by all the friends of the person tagged (a set that the person performing the tagging may well not have access to).

The affordances of software are thus often *abstract*: the user is offered the capability to perform actions in an abstract world, which may or may not end up having physical consequences.

FIG. 1 A timer embodying the *schedule* concept.

6. Concepts and Complex Systems

This may serve as one definition of a complex system: namely, a system in which the actions of users have direct effects that they cannot see, and indirect effects that they might not be able to anticipate.

How then can a user hope to make sense of such a system? This is where concepts come in: *A concept is a structure that is invented to give a coherent account of the immediate consequences of actions in a complex system*. Thus concepts are rarely ends in themselves, but are means to other ends. The concept of *style* in a word processor, for example, does not provide any direct functionality in terms of the appearance or structure of documents; anything that can be achieved in the final printed document with styles can be achieved without them. Rather, the concept explains all the intermediate effects in which collections of formatting changes can be made to an abstract entity—the style—and then applied (immediately or later) to a set of a paragraphs to which that style is attached.

Sometimes a concept is so familiar that it seems almost silly to explain it. But it's a fun game, and an instructive one. Consider, for example, the timer shown in Figure 1. This kind of mechanical device is still common in the US, despite the encroachment of electronic timers.

The whole device is plugged in to an outlet and a motor rotates its central ring marked with the hours of the day, which passes the little arrow labelled 'time now' at the time marked on the ring. An outer ring of tabs rotates with this inner ring, each tab having two positions. When a tab passes the arrow, a light plugged into the timer will be on if the tab is depressed, and off if the tab is flush with the ring. Thus to set the light to be on all night, for example, one would simply depress all the tabs in the nighttime zone of the ring (the darker side marked with the moon symbol).

More abstractly, we can explain the device with the concept of a *schedule*. The schedule consists of a set of time slots; the timer offers an action to toggle a time slot, which inserts

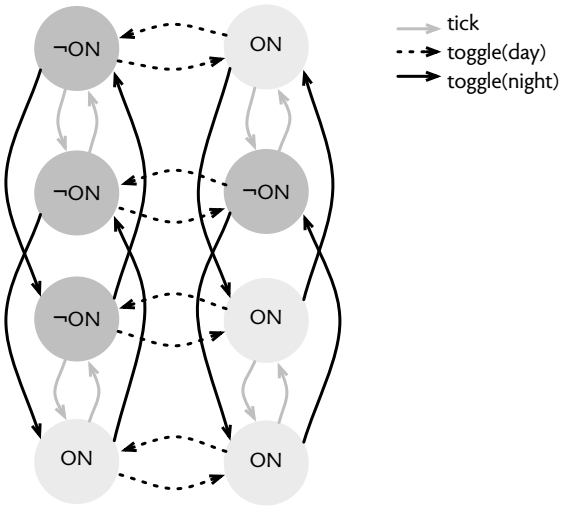

FIG. 2 A conceptless state transition diagram for a timer.

the slot into the schedule or removes it. When a given time slot arrives, the light is on if the slot is in the schedule and off otherwise.

Now imagine trying to understand the timer without the *schedule* concept. The state transition diagram in Figure 2 fully defines the behavior of the timer (albeit simplified with just two slots, one for daytime and one for nighttime, and a twice-daily tick event to represent the transition between the two). And yet it's incomprehensible, lacking the essential structure that allows us to make sense of the behavior.

7. Concepts and Datatypes

Concepts are associated with that programmers would call 'datatypes', and correspond to a set of abstract objects. Thus the *style* concept in a word processor is associated with a set of style objects; *folder* in a file system with a set of folders; *label* in an email application with a set of labels and so on. Some concepts are singletons with only a single object: thus there is only one schedule in the timer, for example. In some cases, the objects of the datatype represent relationships. The objects associated with the *friend* concept in a social networking application, for example, are really 'friendships', created when a friending request is accepted.

But concepts are not quite the same as conventional datatypes. When a datatype is being designed, the programmer considers which behaviors should be assigned to the datatype and which behaviors to other datatypes, aiming to simplify the datatype's interface and minimize dependences in the code. Paradoxically, the very behaviors that motivated the introduction of the datatype may end up not belonging to the datatype itself, but instead be spread across multiple datatypes. For example, in an email client, a programmer may introduce a *Label* datatype to represent labels that can

```
on: bool
time: Slot
schedule: set Slot

inv on = (time ∈ schedule)

tick ≜ time := next(time)

toggle (s: Slot) ≜
   if s ∉ schedule then schedule := schedule ∪ {s}
   else schedule := schedule \ {s}
```

FIG. 3 A timer described using the *schedule* concept.

be attached to messages. Since these labels will be stored as keys in a hash table, they are implemented as immutable objects, and hold no references to messages. The fundamental action of assigning a label to a message is thus not a feature of the *Label* datatype at all.

In contrast, a concept encompasses all the state and behavior that motivates the concept. We can formalize a concept with an abstract state machine model. Such a model for the *schedule* concept is shown in Figure 3. Note that the model includes not only the schedule datatype itself (represented as a set of slots), but additionally the clock (represented by the *time* component) and the state of the light (represented by *on*), since without these, the rationale for *schedule*—turning on and off the light as the clock advances—cannot be conveyed.

Aside. This model is written in the conventional style of model-based specification languages (such as Alloy, B, VDM and Z). The invariant is made to hold magically, updating the state variables in response to the events.

8. The Operational Principle

How should a concept be described? A designer needs to express the concepts of a design, so they can be recorded, shared, evaluated, and so on. One possibility is to construct a formal model (as shown in Figure 3). Such a model has many merits—including abstraction (being free of implementation details), completeness (characterizing all possible behaviors), and precision (lack of ambiguity)—and is also amenable to mechanical analysis.

Aside. A state transition diagram (as shown in Figure 2) provides these advantages too, but lacks the structure (in particular the factoring of the state) that makes the textual model intelligible.

Such a model can be very helpful for exploring the details of a concept's design. But as a primary means of *defining* a concept, it is fundamentally unsuitable. There are two key aspects that are missing.

First, certain actions are connected to one another: the toggling of a time slot and the tick of the clock at which that time slot comes around (and the prior toggling determines

whether the light is turned on or off). The formal model treats these actions entirely independently, and gives no hint of their relationship. Second, these connections are in service of a purpose: to allows the user to establish a schedule in advance for the control of the light. Paradoxically, by describing all possible behaviors, and thus giving them equal weight, the formal model fails to distinguish the behaviors that motivate the concept—for example, in which a time slot is selected and later comes around—from irrelevant behaviors—for example, in which a time slot is toggled twice with no resulting effect.

As another example, imagine a logician from the 19th century—Charles Peirce say—encountering a formal model of *posts* and *friends* in the design of a social networking application. With a little help interpreting the notation, our logician would surely have no trouble understanding the semantics of the model, and could predict the effect of an action in any given state. But this is not the same as *understanding* the model. To understand these concepts, you would need to internalize their purposes, and to see how they fulfill that purpose: for example, that, having friended someone, your posts become visible to them.

The *operational principle* provides a structure for defining concepts to overcome these difficulties. It gives an archetypal scenario that explains how the concept works to fulfill its purpose.

Here are some examples of operational principles for three different concepts:

- *Schedule*: 'When you toggle a slot, the slot is added or removed from the schedule; when that time slot comes around, the light will be turned on or off depending on whether the slots is in the schedule or not.'
- *Bcc*: 'When you add a recipient to a message as a bcc (blind carbon copy), that recipient will get a copy of the message, in addition to any other recipients, but that recipient will not be identified in any visible header so that other recipients will not know that she received a copy.'
- *Style*: 'If you create a style and assign it to one or more paragraphs, then when you make any changes to the format rules of that style, they will be applied to all of the associated paragraphs'.

The operational principle is not a *complete* explanation. The principle that explains *schedule* does not say what happens if you toggle the current time slot; the principle for *bcc* does not say whether the copy of the message saved by the sender names the recipient in its header; the principle for *style* does not tell you what happens when a style is deleted.

This is not a deficiency to be remedied. A description sufficient to predict all behaviors can certainly be constructed, as the kind of formal model just discussed—and indeed doing so can help clarify and refine the design. But such a description must be auxiliary to the operational principle, as it fails to distinguish accidental aspects of behavior from those that *motivate* the invention of a concept.

Aside. In the field of formal methods, a formal model is thought to have a credible 'semantics' because a meaning can be assigned to it systematically, and an informal statement such as the operational principle would be regarded as lacking semantics. But ironically it is the formal model that lacks semantics in the true sense of the word, since it fails to convey the essential meaning of the concept.

9. Operational Misfits

The operational principle is a scenario that explains how a concept fulfills its motivating purpose. But a concept may be flawed, either in its most basic form, or in its elaboration into a fuller set of behaviors. In that case, a negative scenario, a kind of dual to the positive scenario of the operational principle, can explain what goes wrong. Such a scenario is an *operational misfit*.

Take the Macintosh *trash*, for example, one of the key concepts of Apple's Finder. The purpose of the *trash* is to allow deletion of files to be undone. The operational principle explains how: 'when a file is deleted, it is placed in the trash folder; it is only removed irrevocably from the trash when the trash is emptied, and prior to that, can be restored by a simple move to another folder'. The concept of the *trash* isn't quite rich enough, however, to fulfill its purpose in the full context of a modern machine. Here are two operational misfits:

- *More than one drive*: Suppose you mount an external drive intending to copy some files to it, but that drive does not have enough free space. The Finder will tell you that the files cannot be copied. So you delete some files to make space, but of course that will not free the space (since the files remain on the drive, albeit in a special trash folder). To make space, you must empty the trash. But emptying the trash will eliminate not only the files deleted from the external drive, but also all the accumulated deletions from the main drive of the computer. Making room on the external drive will thus make it impossible to later restore files on the main drive.
- *Inadequate metadata*: Suppose you are looking through old directories, cleaning out obsolete files. You realize that a file you just deleted, perhaps a few deletions ago, was actually one you wanted. You can't undo the deletion since you've performed other deletions since then; and you can't find the file in the trash because it contains a large number of files and you can't remember the file's name! You can sort the files by their modification date, but deletion is not a modification, so that won't help.

10. Concept Dependences and Product Families

The concepts of a software system can be arranged in a graph, with an edge from a concept *C1* to a concept *C2*, read '*C1* de-

pends on *C2*', when *C1* makes no sense without the presence of *C2*. Implicitly, subsets of the concepts correspond to different members of a product family. The dependences tell us which subsets are well formed. Thus a dependence of *C1* on *C2* means that whenever a member of the family includes the concept *C1* it must also include the concept *C2*.

Figure 4 shows a dependence graph for some of the concepts of a social networking app such as Facebook. As one might expect, *post* is a leaf concept: it might conceivably exist alone (although the resulting application would be rather primitive) and all other concepts depend on it, directly or indirectly. In contrast, the concept of *user* depends on the concept of *post*, because without posts, users have no purpose.

Aside. Recall that concepts are more than just conventional datatypes (Section 7). Thus the concept *user* in this example means more than the existence of an identified set of users: in particular, it means that the application records which users authored which posts. (It would be possible to have a concept of *user* just for the purpose of allowing users to set application preferences, for example, but that would be a different concept.) The dependence of *friend* on *user* therefore arises not because friendship is defined over users, but because the concept of friend only makes sense in a context in which authorship is recorded.

The dependence diagram suggests some useful sanity checks. First, there should be no cycles in the graph. If two concepts are mutually dependent, they must be included or excluded together. This suggests that they are not independent of one another, and are instead aspects of a single concept. Second, the operational principle of a concept should mention at most that concept and the concepts it depends on, directly or indirectly. The operational principle for the *friend* concept, for example, will say that when one user becomes a friend of another, she can read that person's posts; this then implies that the concept of *friend* must depend on the concept of *post*.

Aside. Likewise the dependence of *tag* on *friend* is required because the operational principle of *tag* says that tagging an image with a user's name makes the image viewable by the user's friends.

Sometimes a group of concepts can be viewed as variants of a single, abstract concept. A word processor (Figure 5) might have a concept of *style* whose purpose is to achieve consistent formatting, and which (unsurprisingly) depends on the *format* concept. The *style* concept has two variants: *paragraph styles* (for formatting paragraphs, which thus depends on the *paragraph* concept) and *character styles* (for formatting arbitrary segments of the text buffer, and which thus depends on the *text* concept). To show this in the dependence diagram, the abstract concept is shown italicized, and dotted instantiation arrows link the variants to it. A dependence on an abstract concept means that at least one of the variants of the abstract concept must be present. Thus this diagram says that if the *stylesheet* concept is present, then *paragraph styles* or *character styles* must be present too.

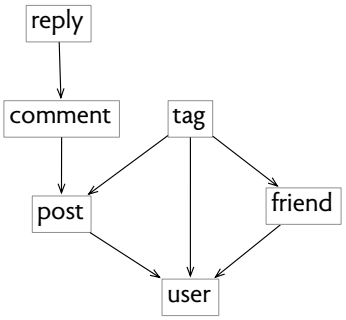

FIG. 4 Part of a concept graph for a social networking app.

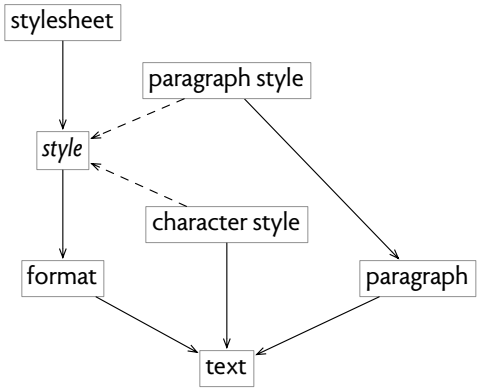

FIG. 5 Part of a concept graph for a word processor.

11. Criteria for Judging Concepts

How can we determine, in advance of implementation, whether a concept is well designed? We might simulate the behavior of the concept in our mind, imagining how the operational principle would apply in different contexts. We could also elaborate a model of the concept (as in Figure 3) and explore corner cases of behavior (such as toggling the current time), or formulate properties (for example that toggling is its own undo) and check them. These two kinds of activities are synergistic; a formal analysis is better at finding unusual scenarios that are implicit in the model, but an informal simulation can catch misfits that lie beyond the designer's initial assumptions.

Such analyses are entirely concept-specific. In contrast, there are some general criteria that can be applied to all

concepts. These criteria are neither necessary nor sufficient. They are not necessary because it is possible to violate them and still end up with a good design; they are not sufficient because they do not guarantee a good design. Nevertheless, my experience to date suggests that they can explain a variety of conceptual design flaws in existing products.

There are four criteria, of which the first three all relate concepts to purposes, and do not require detailed analysis of behavior:

- *Motivation*: a concept should have an articulated purpose;
- *No redundancy*: two concepts should not have the same purpose;
- *No overloading*: a concept should not attempt to serve two distinct purposes;

The last criterion requires a comparison of the behavior of different concepts:

- *Uniformity*: variant concepts should have similar behaviors.

The following sections explain each of these criteria in turn, illustrating violations of the criteria through a variety of applications.

12. Unmotivated Concepts

That a concept needs a purpose is hardly surprising; without a purpose, there is no reason for a concept to exist. Or at least there is no reason from the user's point of view: such concepts may be introduced because they solve a problem for the designer or implementer, perhaps making up for the deficiencies of other concepts. A useful rule of thumb is that a purpose should be expressible in a short phrase: 'to allow undo of deletions' (*trash*), 'to maintain formatting consistency within a document' (*style*), 'to classify messages so they can be retrieved easily' (*label*), etc.

Example. The domain name system has a concept of *glue records*; these are special records, which (unlike conventional domain name records) give the IP address of the name server that will resolve the given name rather than the domain name of the name server. Glue records are indeed used to fulfill the purpose of resolving domain names. But that is not the purpose that motivates their existence. Rather, glue records exist because the standard mechanism for resolving names fails when the name server is within the domain of the name being resolved. They exist, therefore, not to meet a purpose, but to patch a flaw in the resolution algorithm (which would otherwise get stuck in an infinite loop).

Example. Likewise, in the Git version control system, the *stash* concept exists to overcome a flaw in the *branch* concept (namely that switching branches has undesired side effects when the old branch contains uncommitted work). Git includes a pervasive concept of *staging area*, a separate area of storage to which files are copied prior to being committed. On first encountering it, users inevitably ask what the staging area is *for*. Expert users, eager to defend a complex and confusing concept, typically respond with a barrage of possible purposes, none entirely convincing.

13. Redundant Concepts

A single purpose should not motivate more than one distinct concept. This criterion is also uncontroversial; the additional concepts bring no additional benefit if they serve the same purpose, and create a burden of needless complexity for user and developer alike.

Example. In Adobe Lightroom, photos have *rating stars* whose purpose is to allow the user to mark the best photos and then find them later. But in addition to rating stars there are also *flags* (with the values select, don't select, and to-be-deleted). One might imagine that *flags* offer a more streamlined way to selecting files for deletion, but Lightroom's metadata search makes it easy to use *rating stars* in the same way. So there appears to be no fundamental difference in motivation between the two concepts. Interestingly, the designers of Lightroom gave the two concepts different scopes (in Lightroom 3, an earlier version): stars were global but flags were specific to collections. This might have been an attempt to differentiate the two with distinct purposes.

Example. Until version 11 of Adobe Acrobat, the text appearing in a file could be modified using a variety of concepts. The *document text* seemed to be a string containing the concatenation of all the text in the document; an *object* was a paragraph or a column that could be selected and moved or deleted; and a *textbox* was like an object, but was something added to the document rather than a preexisting element. These distinctions seemed confusing and unnecessary. They were eliminated in version 11, which instead has a general concept of an *object*, which is either a text object or an image object. Both existing paragraphs and new annotations are treated as text objects, and are edited in the same way, with the same commands.

Example. In 2013, Google added *categories* to Gmail. The avowed purpose was to classify incoming messages automatically, so that users could more easily distinguish personal email from spam. But users were confused since *labels* already existed for the same purpose (with *system labels* being used for automatic assignments by the application itself). It remains unclear why Google did not simply associate the new built-in filters with *labels* rather than inventing the new concept of *categories*. Worse, *categories* and *labels* differ in small but significant ways. The inbox can be sorted into tabs by *category*, for example, but not by *label*.

14. Overloaded Concepts

No one can serve two masters. Either you will hate the one and love the other, or you will be devoted to the one and despise the other. [Matthew 6:24]

The criterion that a concept should not be motivated by more than one purpose is far from obvious. In fact, one might imagine that achieving two purposes with one design component is the very essence of elegant and efficient design. A car's windshield, for example, protects the driver from road debris, provides a view of the road, supports the roof, and helps maintain cabin temperature. It is hard to imagine that a design that separated these purposes into separate components would be an improvement.

Attempting to satisfy multiple purposes at once, however, inevitably creates conflicts. A stronger windshield that will support the roof and protect better against rollovers will be harder to see through, for example. Such designs tend to be highly coupled, with few combinations of the design parameters meeting all requirements. Nevertheless, even for physical systems, it is possible to design the component to minimize the coupling [31].

In the conceptual realm, there are no physical constraints, and thus no reason not to aim for a complete decoupling in which each concept is motivated by at most one purpose.

Example. Acrobat 9 offers a *signature* concept that combines two purposes: including the image of a written signature, and attaching a digital signature to the file. The program won't allow you to include the image without the digital signature; and because creating the digital signature requires visiting Adobe's website to register to a public key, you can't do either without a network connection.

Example. The conference *review* has two purposes: to provide constructive feedback to authors, and to help the program committee select papers. A reviewer who wants a paper accepted will often hold back on constructive suggestions for fear that they will be interpreted as criticisms by other committee members; and authors, when receiving their reviews, are often unsure which comments are intended to be helpful and which actually determined the fate of the paper.

Example. Pamela Zave provides an example from call forwarding [33]. Suppose phone *A* is forwarded to phone *B*, so that calls to *A* are automatically rerouted to *B*. But now suppose that, in addition, *B* is forwarded to *C*. Clearly a call to *B* should be forwarded to *C*, but should a call to *A* be forwarded to *B* or to *C*? In analyzing this (and many related) examples, Zave notes that the answer depends on the *purpose* of the forwarding. If you forward your calls to an assistant, and that assistant forwards calls to another assistant when he is away, then obviously a call should be double-forwarded. But if you forward your phone to a colleague's office that you're using temporarily, and she forwards her calls to the office she's currently using, then it is equally obvious that calls to your office should not be forwarded to your colleague! Zave calls these two scenarios 'follow me' and 'delegate', and it is clear that they are distinct purposes requiring distinct concepts.

Example. The use of a single concept for two different purposes may arise because of 'piggybacking', in which a new purpose is retrofitted onto an existing concept. The printing subsystem of Apple's OS X has the concept of *paper size*, whose purpose is to save (and recall) the dimensions of a sheet of paper, including margins. For their printer drivers, Epson piggybacked onto this concept an additional purpose: saving the choice of media type. Epson's justification was presumably that this now ensures that when you select a paper size you get all the properties of the paper that the printer needs to know. This essentially redefines the purpose of paper size as a printer setting. But paper size is used not only by the printer driver: it's used also to set document sizes, for example. In Adobe Lightroom, printing presets—which are preset layouts defining photo size, margins, and so on—are based on paper sizes. The result is that when you change the media type of the paper you're using (say from matte to glossy) none of your existing presets work!

Example. Overloading can also arise from the desire to simplify the user interface, so that a single concept does double duty, often at the expense of confusing the user. The designers of Dropbox, for example, understandably wanted to base their application on the concepts of the existing file system interface. As a result, *deletion* of a file or folder serves two very different purposes, distinguished according to context. If Alice shares a folder with Bob, and Bob deletes it, then the folder will no longer be accessible to Bob. But if Alice places a second folder inside the shared folder, and Bob deletes that second folder, it will be no longer be accessible to either Alice or Bob! Two different purposes are conflated here: one is allowing Bob to make space on his machine by no longer sharing Alice's folder; the other is allowing Bob to help Alice clean up by deleting folders. I wonder if this conflation is partly responsible for the many reports of accidental deletions of shared Dropbox folders.

Example. My camera (a Fuji x100s) has two related menus: (1) an *image quality* menu that offers options such as raw (the full, unprocessed sensor data), various levels of JPEG encoding, and a combination of a raw and JPEG file, and (2) an *image size* menu that allows you to select different aspect ratios (such as a square ratio, or a traditional 2×3). Unfortunately, these two concepts are coupled: you can only choose a non-standard aspect ratio when the *image quality* is set to one of the JPEG options. This is not due to any underlying technology problem; if you select the combination raw/JPEG option, along with a square ratio, the raw file actually includes a (non-destructive) crop corresponding to the aspect ratio. The root of this problem, it turns out, is revealed if you look more carefully at the *image size* menu. As its name suggests, it actually provides different image sizes

in terms of pixel dimensions; the different aspect ratios are achieved by offering, for a given height, a choice of different widths. The concept *image size* thus has two purposes that are coupled—choosing the resolution and choosing an aspect ratio—and since resolution only applies to JPEG, it is not possible to select a non-standard aspect ratio in a JPEG-less mode.

Example. The *friend* concept in Facebook originally served two purposes: to allow users to control which other uses could read their posts, and to help users manage a potentially overwhelming number of posts to read by limiting posts to certain favorite authors. Coupling these two purposes meant that a user had to choose the same set of users in both cases. The latter purpose motivates Twitter's *follower* concept, and it was adopted by Facebook in 2011; now you can be someone's friend (and thus let them read your posts) without being a follower (and thus avoiding reading their's).

15. Uniformity of Concepts

When two concepts are variants of the same abstract concept, they are expected to have similar operational principles. When this is not the case, users are likely to surprised and frustrated, and modularity will suffer in the implementation.

Example. In banking, the concept of a *deposit* into a bank account has multiple variants, including *transfer deposits* and *check deposits*. The operational principle for a *transfer deposit* says that the owner of the sending account initiates the transfer and at some later point, that balance of the receiving account is augmented and the balance of the sending account is reduced accordingly. But the principle for a *check deposit* is more complicated: sometime after the check has been received, the balance of the receiving account is augmented with 'available funds' that the account's owner may spend; but at a later point, the check may bounce and the balance will be subsequently reduced. This confusion is the basis of a widespread scam.

Sometimes the operational principle is the same, but the variants differ in their full behaviors in ways that are unrelated to their variant purposes.

Example. In Apple's iCal calendar application, one can associate *alerts* with calendar events. There are different variants of *alert*, such as *email alert* and *message alert*. An alert can be chosen as a default for all new events—for example, always to display a message the day before. But *email alerts* are not included in the default options.

A related violation of uniformity occurs when a concept that depends on an abstract concept makes distinctions between its variants.

Example. Many cameras offer the ability to save and recall one or more *custom setting banks*, whose purpose is to make it quicker to switch the camera between usage modes. There is a concept of *setting* with many variants. But on most cameras, only a subset of the *settings* are selectable in *custom setting banks*. On my x100s, for example, you can include the choice of white balance but not the image size. Similarly, many cameras offer a *custom function* button. On the x100s, it can be bound both to settings (for example, set the image quality to raw) and to menu actions (for example, open the ISO selection menu), but it is limited to a few particular settings and a few particular menu actions.

Example. In Apple Keynote (v5.3), the concept of *group* depends on the abstract concept *element*, which includes *text box*, *shape* and *connection line*. Unfortunately, however, *connection lines* cannot be grouped.

Finally, two different concepts may have different purposes but may achieve them using a shared subpurpose. Ideally, this subpurpose should be realized in the same way, preferably as a concept in its own right.

Example. In Apple Mail, *searches* are used to find previously received or sent messages that match some criteria, and *rules* are used to set automatic treatment for incoming messages. Both concepts involve a step in which the user provides criteria to define a set of messages—in one case to be acted upon, and in the other to be displayed. Yet the criteria in the two cases are different. Preferably, the behavior would not only be made consistent but would be factored out into a concept in its own right. Strangely, such a concept already exists: the *smart mailbox*, which like a *label* in Gmail, offers a way to name a set of messages that satisfy some criteria (in fact, yet a third set, incomparable with the other two).

16. Familiarity and Concept Idioms

Old problems may welcome new solutions; and new problems may demand them. But when an old solution is perfectly fit for purpose, and no better solution is proposed, novelty should give way to familiarity. This holds especially for concepts, since new concepts do not come free, either for the designer (who has to explore their consequences) or for the user (who has to learn what they are for and how they work). And new things tend to be wrong: the misfits and limitations of a concept may be discovered only after extensive use.

Concept idioms are reusable concepts that have applications in many different contexts. Each idiom identifies a central concept, its purpose and operational principle, and known complications arising in the use or implementation of the concept. I am currently building a catalog of such idioms to act as a repository of conceptual design expertise. The table in Figure 6 lists the idioms currently in the catalog.

Consider the concept of *style*. In the context of a word processor, a style is associated with one or more paragraphs, and allows the formatting of those paragraphs to be updated all at once. But the essential concept here is a generic one, and has applications in many contexts. Color *swatches* in Adobe Indesign are *styles*; so are *themes* in Microsoft Pow-

concept	*purpose*
consistency	
style	*achieve consistent formatting of a set of elements*
master	*achieve consistent structure and format of composite elements*
stencil	*use archetypal objects for consistency and time saving*
style buffer	*reformat another object like a previous one to save effort*
preset	*allow setting of many properties at once*
organization	
folder	*organize collection of items into a hierarchy*
REST	*organize collection of resources by simple path names*
group	*group items so they can be treated as a single item*
layer	*allow easy inclusion/exclusion of sets of items*
stack	*place items in stacking order for precedence*
selection	*apply action in aggregate to many items at once*
label	*add labels to items so they can be found later*
alias	*address one or more items with a shorthand name*
filter	*allow filtering of set of objects by their features*
property	*describe an object with properties that have values*
metadata	*sort and search for items using associated data*
navigation	
history	*keep past actions for audit, undo, visibility*
buffer	*provide temporary storage area for quick modification*
cursor	*provide shortcut entry into traversable document*
access	
access token	*control access to a resource in an easy way*
reservation	*allocate resources efficiently and prevent conflicts*
OOBA	*authenticate user with 'out of bound' channel unique to user*
communication	
message	*communicate in discrete packets between endpoints*
posting	*share a short communication by broadcast*
notification	*let a user know when something happens*
community	
friend	*mediate communications by preapproved relationships*
clique	*create subcommunity within larger community*
invitation	*predicate relationship on approval*
account	*centralize user-specific information*
karma	*incentivize users to contribute to an online community*
rating	*crowdsource evaluation of items*

FIG. 6 Some idioms from an initial catalog.

erpoint, and *classes* in CSS. In all these cases, the fundamental idea is the same: to introduce an indirection between elements and their properties. By attaching properties to a style rather than to the element itself, an update that would have required changes to multiple elements can be achieved with an update to a single style instead.

The *style* idiom has many subtleties. For example, since elements may have properties applied directly, it is usually desirable to be able to define styles that assign only *select* properties. Surprisingly, some applications do not properly support this. In both Microsoft Word and Adobe Indesign, the properties of a style are initially unassigned, and become assigned only when selected by the user. But unfortunately, in both applications, there is no simple way, once a property has been set, to unset it. Word experts suggest using Visual Basic underneath the hood. Indesign has offered a 'reset to base' function since 2007, which obliterates all the properties of a style. Amongst current text processing applications, only Apple Pages solves this problem definitively: the dialog for setting properties includes not only a field for the value of the property, but also a checkbox for whether the property should be included or not.

This is hardly a cutting-edge technology problem. Back in the 1980s, Framemaker had a special 'as is' value that could be assigned to any property to exclude it from the style. Perhaps if the style idiom had been codified, the systematic solutions to this problem would be more pervasive.

Aside. At the same time, it is always worth questioning traditional concepts. Most users of computers are familiar with the *folder* idiom, in which files are organized into a hierarchy of folders. But perhaps only computer scientists are comfortable with the idea that both the name of a file, and its existence, are properties of the folder in which it is placed. This idea can be attributed to a variant of the *folder* idiom represented by Unix, in which a folder (called a directory in Unix) is a collection of links mapping local names to files. This seems to work well in Unix itself, but it can wreak havoc in other contexts. In version control and backup systems, for example, users may expect changing a file's name to be recorded as a modification of the file, but in many cases it is instead treated as if the file with the old name were deleted and a new, unrelated file with the new name were created. This idea may also explain the confusion amongst some users of Dropbox about the distinction between deleting a top-level shared folder and deleting a folder inside it.

17. Precursors and Inspirations

A project such as this has, of course, many precursors. The essential notions—of concepts and purposes—have been around in various forms for many years, although they have yet to be brought together in a coherent theory.

Some of the most influential ideas succeed because they resonate with sensibilities that many people share, but which have not been fully articulated. It is my hope that conceptual design of software is such an idea. Put another way, I suspect that the very best software designers have been doing these things—identifying purposes and devising concepts to fulfill them—all along. The value of a theory of design is in part just to understand and codify design practices that already work well.

The idea that software design might even exist as an activity distinct from programming was championed by (amongst others) Mitchell Kapor, who argued that software needs designers just as buildings need architects [16].

In his celebrated book *Mythical Man Month*, Fred Brooks coined the term *conceptual integrity*, and claimed that it was 'the most important consideration in system design'. Revisiting the book in an afterword written for an anniversary edition twenty years later, he reiterated: 'I am more convinced than ever. Conceptual integrity is central to product quality.' The book never actually defines the term, but instead focuses on the organizational need for a single designer. Richard Gabriel has challenged Fred Brooks's assertion that coherent designs only emerge from a single mind [10]. Either way, this seems to me a psychological and organizational question that is orthogonal to the substantive question of what a conceptual design *is*, and how it might be judged. A bit more of what Brooks means can be found in his coauthored book on architecture [5], which gives three criteria for conceptual integrity: orthogonality (that individual functions should be independent of one another); propriety (that a product should have only the functions essential to its purpose and no more); and generality (that a single function should be usable in many ways).

Brooks introduced the distinction between 'conceptual' and 'representation' design, and argued that conceptual design is the essence of software development: '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 and testing the fidelity of the representation.' [7]. To Brooks, however, a 'concept' is a looser notion that includes most of the elements of software design—'data sets, relationships among data items, algorithms, and invocations of functions'—and thus conceptual design to him includes aspects that I would classify as representational.

The term 'concept' is used in a subtly different way in the field of conceptual modeling, where it refers to a semantic construct, usually tied to something in the world outside the computer. In this field, construction of a conceptual model is about recording an understanding of the problem world, in a similar manner to the approach known as domain modeling or domain engineering [4]. In contrast, my notion of concept is more operational than ontological, and the work of finding concepts more inventive than descriptive. Closer to my theory (and to my idioms) are Martin Fowler's analysis patterns [9], which are data model fragments for solving common domain-specific problems.

The operational principle comes from the work of the chemist philosopher Michael Polanyi [27, 26], which I was introduced to by Michael Jackson. Prior to discovering Polanyi, Jackson had already used the 'frame concern', an archetypal scenario very similar to the operational principle, to explain how the combination of machine and environment properties satisfy a requirement [15]. Early on, I tried to use Jackson's notion of designation [14] as a way to define concepts, but it was not suitable, since concepts are not observable phenomena but rather artifacts of the design.

My thought experiment of a 19th century logician encountering a formal model of a social networking app is just an expression of one Polanyi's central claims: that the natural sciences cannot explain engineering artifacts. As he explains:

'Engineering and physics are two different sciences. Engineering includes the operational principles of machines and some knowledge of physics bearing on those principles. Physics and chemistry, on the other hand, include no knowledge of the operational principles of machines. Hence a complete physical and chemical topography of an object would not tell us whether it is a machine, and if so, how it works, and for what purpose. Physical and chemical investigations of a machine are meaningless, unless undertaken with a bearing on the previously established operational principles of the machine.' [26]

When we try to be 'scientific' in our pursuit of a deeper understanding of software, we might heed Polanyi's distinction, and recognize that design and engineering knowledge are not reducible to scientific principles. The ascendancy of semantics and logic in programming languages and software engineering research has been very helpful in clarifying the languages that we use and the analyses we apply to them, but it has perhaps drawn attention away from design issues, which are not so readily formalized.

The idea of misfit comes from Christopher Alexander's first book, *Notes on the Synthesis of Form* [1], where he argues compellingly that fitness for purpose cannot be fully characterized. No problem has a 'finite list of requirements', and the best we can therefore do is to check our designs against anticipated misfits. Concept idioms are inspired, of course, by Alexander's design patterns [2], and are perhaps closer to them in spirit than the more implementation-centric patterns of object-oriented programming [11].

David Parnas's *uses relation* and the related notions of program families and minimal subsets [24] are the origin of my concept dependence graphs. For a program, dependences can be defined bottom-up in terms of runtime behavior—although not as easily as one might imagine [13]—and the notion of minimal subsets follows from it. For concepts, however, finding a behavioral characterization has proved hard, so I have used the minimal subset idea not as a consequence of a more basic definition of dependence but as the definition itself.

A behavioural characterization of concept dependence is possible, but it produces a very different relation. Consider a library system, for example, with the concepts of *loan* and *hold*; a user places a *hold* on a book in order to obtain a *loan*. By my definition of concept dependence, *hold* depends on *loan*, but not vice versa: the concept of a *loan* makes perfect sense without the concept of a *hold*, but the concept of

a *hold* exists only to enable *loans*. And yet in the execution of the system, the behavior associated with *loans* will be affected by the behavior associated with *holds*: the very purpose of a *hold* is to suppress *loans* so that the book is available for the party that reserved it. A behavioural dependence graph would therefore include an edge from *loan* to *hold*. This graph seems to be harder and more subtle to construct, but it may be useful to expose design flaws arising from unexpected couplings between concepts.

The concept dependence graph defines a family of applications that could be built with different subsets of the concepts. It can therefore serve as a guide to identifying essential concepts for a minimum viable product. The *uses* relation, in contrast, is defined over the modules of a particular implementation. Even if these modules correspond to concepts (as they are likely to do in a good design), the minimum subsets defined by *uses* will exclude some combinations of modules simply because those particular modules require modules outside the subset, due to the way the code is structured. As a trivial example, in the code of a word processor, a class representing paragraphs may include a field that links a paragraph to its style object; the *uses* relation would then rule out a subset that includes paragraphs but not styles. In the concept dependence graph (Figure 5), on the other hand, there is no dependence of *paragraph* on *style*, representing the fact that the concept of *paragraph* makes perfect sense in the absence of *styles*, and that it should therefore be possible to build an implementation that reflects this.

This suggests that the concept dependence graph might be used as a yardstick to evaluate the module dependence graph of a particular implementation. The modules in the code would mapped to concepts, and the module dependences that are gratuitous (that is, not required by concept dependences) highlighted as potential flaws in the representation design.

My focus on identifying purposes for applications, and subpurposes for concepts, and on emphasizing the uniqueness of such purposes, was inspired (rather incidentally) by Michael Sandel's account [29] of Aristotelian ethics. For Aristotle, everything in nature has its 'telos' or purpose; thus the purpose of an acorn is to become an oak tree, and a human being likewise has a single purpose to fulfill. It struck me that this could be viewed as a principle of design, with the presence of a compelling purpose for a concept being a measure of its clarity.

The controversial idea that a concept should be motivated by one purpose, and not two, was inspired by Nam Suh's axiomatic design [31], a theory of design in mechanical engineering. Suh's theory comprises two axioms, the first of which states that functional requirements should be kept independent, in the sense that two functional requirements should not be dependent on the same design parameter. If they are, it will be hard to adjust the design parameters to control the functional requirements independently. This consideration applies equally to concept design, but I have given more emphasis to the argument that a single concept cannot 'serve two masters' well, rather than the argument for modifiability.

In user interface design, the term 'conceptual model' refers to a mental conception of how a system works. In *The Design of Everyday Things* [22], Norman distinguishes two conceptual models: the 'user model' (representing the user's understanding) and the 'design model' (representing what the designer has in mind). Between the two, the 'system image' (as projected by the user interface) conveys the design model to the user, and thus determines the user model. Ideally, the design and user model are perfectly aligned, so the user has a sound understanding of how the system works.

To avoid confusion, I have avoided the term 'conceptual model'. A 'conceptual design' in my theory corresponds to Norman's design model, with an important caveat. My notion of design is 'conceptual' because it focuses on concepts, not because it is conceptualized in the mind. I take the view that because software is an abstract construction, software concepts are no less real than the physical components of a mechanical contrivance.

Norman beautifully illustrates the idea of a faulty conceptual model with a refrigerator that has two temperature controls, one for the fresh food compartment and one for the freezer. It turns out that many refrigerators actually have just a single thermostat in one compartment, and a single compressor. One of the controls sets the thermostat level, and the other sets the proportion of cold air sent to each compartment. Marking the controls as 'fresh food' and 'freezer' is totally misleading, and produces a system image that leads to an incorrect user model. But, as Norman notes, even if the user understands how the system works, the system image is still flawed because it does not tell you which compartment contains the thermostat, and which control is tied to which function.

My analysis would reach a similar conclusion through a different path. There are two concepts, *freezer setting* and *fresh food setting*, corresponding to the two controls, but these concepts have no compelling purposes. Indeed, they can only be explained (as Norman does) by reference to the internal mechanism. So the design is therefore flawed.

Design thinking [eg, 8, 17] and user-centered design [eg, 3] are related approaches that address what I have called conceptual design. But whereas I have been concerned primarily with the *content* of the design, these approaches tend to be concerned with the *process* of design. They typically advocate incrementality, early prototyping, extensive user involvement, multidisciplinary teams, and so on, and have been important in championing an ethos of design that is more responsive to the needs of users. These are all good things that are synergistic with the approach proposed here.

Jakob Nielsen and Rolf Molich's 'heuristic evaluation' [19] is a way to find flaws in user interface designs by hav-

ing reviewers apply usability heuristics. Nielsen has since refined the heuristics [20], and his colleague Bruce Tognazzini has published a larger collection [32] that contains many additional insights but is less suitable as a basis for a heuristic evaluation. Similar collections have been extracted (and can easily be found with a web search) from texts on user interface design by Don Norman [21] and Ben Shneiderman [30]. In contrast to my criteria, these heuristics tend to be more focused on the user interface itself, and can be viewed as addressing general issues in how concepts are concretely realized.

The process of heuristic evaluation might work for concept criteria too, with an important caveat. To apply user interface heuristics, all you need is the interface itself (or a sufficiently detailed design). But to apply concept criteria, you need not only the design proper (comprising the concepts and their associated behaviors) but also the argument (that is, the purposes and the operational principles that link the concepts to their purposes). A poor argument for a design that happens to work well in practice would thus be found lacking. From one point of view, this is a limitation of my approach, since applying it to a preexisting design requires first reverse engineering the argument. But it might also be seen as an merit, since the construction of an argument during design—just like the construction of a safety case during the design of a critical system—should itself produce better results, even without external evaluation.

'User stories' are a way to structure increments of functionality in agile development. Like the operational principle, the user story connects some behavior to the need it fulfills. In contrast, user stories are not archetypal, but enumerate basic functions of the user interface. They therefore act more as an implementation checklist than as a design tool.

Goals in requirements engineering are related to purposes, but are different. Goals are typically expressed in terms of behaviors; in KAOS [18], for example, a goal is defined as an objective to be met that 'prescribes a set of desired behaviours' [28]. A purpose, in contrast, does not imply any particular set of behaviors. Consider, for example, the purpose of aligning shapes in a drawing application. A human user finds such a purpose is reasonably clear, and with little effort could distinguish concepts that are fit for purpose from those that are not. But tying such a purpose down in behavioral terms is much harder than one might at first imagine.

First there is the question of what can be expected from the user. If we assume that the user will select some elements and then invoke a function to align them, we can certainly specify the desired application behavior. A better solution, however, first introduced by Intellidraw and now ubiquitous, involves no such function at all. Instead as the user moves an element, flashing guide lines indicate alignment relationships, and when the user releases the element, it snaps into the position determined by the nearest guideline. It is clearly not possible to specify a set of system behaviors corresponding to the desired purpose that admits this kind of solution without also admitting unreasonable solutions (such as simply requiring the user to align the element manually).

Second, there is the question of whether, having given up on specifying system behaviors, we can even specify a desired end-state. For the alignment problem this might be feasible. For example, we might formalize the user's intent as a set of alignment constraints amongst pairs of elements, with the ordering of each pair indicating which element is to be moved to satisfy the constraint; the end-state is one that satisfies these implicit constraints. But the complexity of specifying a desirable end-state in this manner may take us across the boundary between need and design, causing us to prematurely limit the space of design solutions.

Finally, goals often include amorphous qualities such as 'safety', 'responsiveness' or 'robustness'. In our approach, these would not be distinct purposes but would be qualifiers of functional purposes—an adverb to a purpose's verb.

18. Concepts in Practice

This essay has been more descriptive than prescriptive. Its goal has been to show how concepts and purposes can explain real issues in familiar designs, thus making the case that concepts might provide a powerful framework for software design. From a practical perspective, though, I believe that—despite the lack of a well-defined process—concepts and purposes can already be used prescriptively to guide a new software design.

First, the focus on purpose in motivating not only the application design as a whole but the motivation of the individual concepts provides a constant reminder to justify every step in the development effort in terms of end goals. It makes it harder to careen off down a design path of high complexity and dubious value. After many years spent teaching a software project class to students at MIT, I have reached the conclusion that the primary determinant of success is not how talented the students are or how well they can write code, but how firm is their grasp on the problem that they are trying to solve.

Second, concepts give developers a way to break a larger design problem into smaller pieces. The widespread adoption of agile approaches is in part a response to the recognition that an incremental approach, in which functionality grows slowly rather than arriving in a big bang, generally is less risky and more effective. Yet agile approaches offer no way to ensure that the increments of functionality are coherent. The desire to deliver the most essential functions first may encourage fragmentation, and force a design to proceed piecemeal, without adequate consideration of the integrity of related functions.

Pressure to drive development by purposes rather than ad hoc use cases should also help to protect against the risk of a fragile design that works only for the concrete use cases that were explicitly considered. This seems to be what happened in the design of CSS: features were introduced only when they could be justified by demonstrated use cases (often on the assumption that web pages would correspond to documents), and by the time the world realized that more general, orthogonal and flexible concepts were needed (and that many web pages were applications, and not documents), it was too late.

Third, the concept dependence graph provides a map that can be used to explore the space of designs with reduced functionality. Many projects aim initially to produce a 'minimum viable product'; the dependence graph gives a way to evaluate which concept sets are viable, and might suggest (when there are too many dependences) which concepts need to be altered to make simpler subsets viable.

19. Next Steps

The ideas presented here leave many questions unanswered.

The criterion that a concept should have at most one purpose begs the question of how to tell when an expressed purpose might in fact be composite, representing two distinct purposes. The wording might reveal a conjunctive quality ('ensure A and B'), but clearly a more rigorous test is needed.

Some concepts succeed precisely because they can be used in so many ways. My sense is that such concepts are not 'multipurpose'—that is, designed to serve several, perhaps unrelated purposes—but rather are 'general purpose', meaning that they are designed with a coherent purpose in mind that happens to be general enough to have many applications. Some of the concepts that appear in my evolving idiom catalog have this quality. It would be good to refine this idea, and account for the way in which the best designed concepts can be combined synergistically so that the combination is greater than the sum of the parts.

The uniformity criterion unfortunately introduces a non-uniformity into the list of concept design criteria, since the others all involve simple properties of the mapping between concepts and purposes. And the criterion itself is sadly non-uniform. Can uniformity be expressed in more basic terms? And can its different aspects be unified into a more compelling and simpler criterion?

Shriram Krishnamurthi noted that the same concept can have multiple forms, at different levels of sophistication. For some users and in some contexts, a simpler version will suffice, but sometimes additional complexities must be understood. For example, to a typical user of a web browser, a domain name is the permanent name of a single machine, and this view will suffice to explain most behaviors that such a user will experience. A more sophisticated user will know that the domain name may be mapped by a load balancer to multiple machines, and that the mapping is neither permanent nor global, so that when a DNS record is changed, a domain name may be resolved for some users and not others. An even more sophisticated user will know that the binding of a domain name may change in the course of a web interaction; this is the view needed to understand some security attacks. A similar story can be told for an image in Photoshop, ranging from a simple matrix of pixels to a richer view that includes channels, transparency, profiles, and so on. How should we account for this and exploit it in design?

Concepts are not unique to software systems, but are found in all kinds of complex systems, physical and organizational. It would be interesting to see whether these ideas might be useful there too.

Acknowledgments

I am indebted to Santiago Perez De Rosso, who has been my partner on this project, discussing the ideas as they emerged, testing them in practice, and helping me refine them. His work on Gitless, a rethinking of the conceptual design of Git, has been a wonderful testbed for the evolving theory. Thank you also to the undergraduates whose project work provided examples for this paper: Dwyane George (Facebook), Nikki Shah (Gmail) and Kelly Zhang (Dropbox).

This work has benefited greatly from the friendly skepticism of many friends: my students Ivan Kuraj, Eunsuk Kang, Matt McCutchen, Aleks Milicevic and Joe Near; my colleagues Shigeru Chiba, Jonathan Edwards, Yishai Feldman, Richard Gabriel, Philip Guo, Emanuel Letier, Michael Maddox, Donald Norman, Julia Rubin, Mitch Wand, William Woods and Pamela Zave; and especially David Faitelson, Bill Griswold, Michael Jackson, Shriram Krishnamurthi and Kevin Sullivan, with whom I enjoyed detailed exchanges that helped expose weaknesses and find new paths forward. Thank you to you all.

Finally, thank you to the International Design Center, a joint project of MIT and the Singapore University of Technology and Design, and to John Fernandez and Chris Magee, its directors at MIT, for supporting this work.

References

[1] Christopher Alexander. *Notes on the Synthesis of Form*, Harvard University Press, 1964.

[2] Christopher Alexander. *A Pattern Language: Towns, Buildings, Construction*, Oxford University Press, 1977.

[3] Hugh Beyer and Karen Holtzblatt. *Contextual Design: Defining Customer-Centered Systems*, Morgan Kaufmann, 1997.

[4] Dines Bjorner. *Domain Engineering: Technology Management, Research and Engineering*. Japan Advanced

Institute of Science and Technology (JAIST) Press, March 2009.

[5] Gerrit A. Blaauw and Frederick P. Brooks. *Computer Architecture: Concepts and Evolution*. Addison-Wesley Professional, 1997.

[6] Frederick P. Brooks. *The Mythical Man-Month*. Addison-Wesley, Reading, Mass, 1975; anniversary edition, 1995.

[7] Frederick P. Brooks. No Silver Bullet—Essence and Accident in Software Engineering. *Proceedings of the IFIP Tenth World Computing Conference*, pp. 1069–1076, 1986.

[8] Nigel Cross. *Design Thinking: Understanding How Designers Think and Work*, Bloomsbury Academic , 2011.

[9] Martin Fowler. *Analysis Patterns: Reusable Object Models*. Addison-Wesley Professional, 1997.

[10] Richard P. Gabriel. Designed as Designer. Essay track, *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages and Applications*, Montreal, 2007. Available at: http://dreamsongs.com/DesignedAsDesigner.html.

[11] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley Professional, 1994.

[12] James J. Gibson. The Theory of Affordances. In *Perceiving, Acting, and Knowing: Toward an Ecological Psychology*, edited by Robert Shaw and John Bransford, Lawrence Erlbaum Associates, 1977.

[13] Daniel Jackson. Module Dependences in Software Design. In *Radical Innovations of Software and Systems Engineering in the Future*, 9th International Workshop, RISSEF 2002, Venice, Italy, October 7-11, 2002, pp.198–203, 2002.

[14] Michael Jackson. *Software Requirements and Specifications: a lexicon of practice, principles and prejudices*. Addison-Wesley, 1995.

[15] Michael Jackson. *Problem Frames: Analysing & Structuring Software Development Problems*. Addison-Wesley Professional, 2000.

[16] Mitchell Kapor. A Software Design Manifesto, in *Bringing Design to Software*, edited by Terry Winograd, with John Bennett, Laura De Young, and Bradley Hartfield. Addison-Wesley, 1996.

[17] Tom Kelley and David Kelley. *Creative Confidence: Unleashing the Creative Potential Within Us All*, Crown Business, 2013.

[18] Axel van Lamsweerde. *Requirements Engineering: From System Goals to UML Models to Software Specifications*., Wiley, 2009.

[19] Jakob Nielsen and Rolf Molich. Heuristic evaluation of user interfaces. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems* (CHI '90), Jane Carrasco Chew and John Whiteside (eds.), ACM, New York, 1990.

[20] Jakob Nielsen. *10 Usability Heuristics for User Interface Design*. At: http://www.nngroup.com/articles/ten-usability-heuristics

[21] Donald Norman. *The Design of Everyday Things*. Originally published under the title *The Psychology of Everyday Things*. Basic Books, 1988.

[22] Donald Norman. *The Design of Everyday Things: Revised and Expanded Edition*. Basic Books, 2013.

[23] David L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, Vol. 15, no. 12, pp. 1053–1058, Dec. 1972.

[24] David L. Parnas. Designing Software for Ease of Extension and Contraction. *IEEE Transactions on Software Engineering*, Volume SE-5, Issue 2. March 1979

[25] Santiago Perez De Rosso and Daniel Jackson. What's wrong with git? a conceptual design analysis. In *Onward!, part of ACM International Conference on Systems, Programming, Languages and Applications (SPLASH)*, pp. 37–52, 2013.

[26] Michael Polanyi. *The Tacit Dimension*, University of Chicago Press, 1966, with foreword by Amartya Sen, 2009; pp. 39–40.

[27] Michael Polanyi. *Personal Knowledge: Towards a Post-Critical Philosophy*, University Of Chicago Press, 1974.

[28] Respect-IT. *A KAOS Tutorial*. At: http://www.objectiver.com/fileadmin/download/documents/KaosTutorial.pdf

[29] Michael Sandel. *Justice: What's the Right Thing to Do?*, Farrar, Straus and Giroux, 2010.

[30] Ben Schneiderman. *Designing the User Interface: Strategies for Effective Human-Computer Interaction*. Prentice Hall, 2009.

[31] Nam P. Suh. *The Principles of Design*. Oxford Series on Advanced Manufacturing (Book 6), Oxford University Press, 1990.

[32] Bruce Tognazzini. *First Principles of Interaction Design (Revised & Expanded)*. At: http://asktog.com/atc/principles-of-interaction-design

[33] Pamela Zave. Secrets of Call Forwarding: A Specification Case Study. In *Formal Techniques for Networked and Distributed Systems (FORTE)*, 1995, Montreal, Canada.