MIT-CSAIL-TR-2013-020 August 8, 2013

Conceptual Design of Software: A Research Agenda

Daniel Jackson

Conceptual Design of Software: A Research Agenda

Daniel Jackson, MIT

Abstract

A research agenda in software design is outlined, focusing on the role of *concepts*. The notions of concepts as "abstract affordances" and of conceptual integrity are discussed, and a series of small examples of conceptual models is given.

What is Conceptual Integrity?

In the field of software design, there is little agreement to be found amongst experts over exactly how software should be structured, what languages and tools should be used, and so on. And yet a handful of general notions—separation of concerns, abstraction, decoupling, representation independence, and so on—have received wide approbation and thus represent the closest we have to some kind of consensus. Amongst these notions, one is of particular interest because it addresses not only the internals of the software but also the behavior experienced by the user, and is thus most relevant to the question of how software should be designed in the larger (and more experiential) sense of "design" used in the more established design disciplines (such as architecture and industrial design).

This is the notion of "conceptual integrity", which has been held up not merely as one of many criteria for good design, but as the single determinant of success or failure. Fred Brooks devoted a large part of his influential book *Mythical Man Month* [3] to discussing this notion (and to our knowledge coined the term), and states, bluntly: "Conceptual integrity is 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."

But what is conceptual integrity? Brooks himself leaves us guessing. At the start of the relevant chapter of *Mythical Man Month*, he tells us "It is better to have a system omit certain anomalous features and improvements, but to reflect one set of design ideas, than to have one that contains many good but independent and uncoordinated ideas." That's the closest he comes to a definition. In a more recent book, *The Design of Design* [4], he cites the earlier book as one of two sources for the idea of conceptual integrity (the other being a book he coauthored on computer hardware architecture [2]), and summarizes the idea as a combination of three principles—orthogonality, propriety and generality—although strangely these terms are not explained in *Mythical Man Month*. Instead, he concentrates on the implications for team structure, arguing that "the design must proceed from one mind, or from a very small number of agreeing resonant minds". "If a system is to have conceptual integrity," he tells us, "someone must control the concepts".

Aware of the risk that this might appear elitist, Brooks admits that this requires an aristocracy of architects, but one "that needs no apology". Nevertheless, the chapter on conceptual integrity in *Mythical Man Month* is devoted precisely to this apology, and the apparent elitism, as Brooks seems to have predicted, did indeed lead to a backlash, represented many years later by Eric Raymond's "The Cathedral and the Bazaar" [20], a manifesto for open source development that contrasted "reverent cathedral-building" (as typified by emacs's development) in which the system is "carefully crafted by individual wizards … working in splendid isolation" with the "great babbling bazaar of differing agendas and approaches" (as typified by Linux) "out of which a coherent and stable system could seemingly emerge only by a succession of miracles." Interestingly, despite countering the central argument of *Mythical Man Month* that increasing the team size has diminishing returns (and otherwise quoting Brooks approvingly), Raymond never addresses explicitly Brooks's argument for conceptual integrity—which is especially surprising since one must assume that Raymond's cathedral analogy was inspired by the image of the cathedral of Reims at the very opening of Brooks's chapter!

The central question, then—of what exactly comprises conceptual integrity—seems to have been pushed aside in favor of arguments about process. And whether conceptual integrity is even coupled to team structure is open to question; Dick Gabriel challenges [6] the idea that design excellence requires the focus and unanimity of a single mind; indeed, he argues that even cathedrals, notably the great basilica in Florence, in contrast to Raymond's characterization, are not produced by (in Raymond's terms) "individual wizards … working in splendid isolation", but rather are produced by a series of great designers, often in teams, each building on the work of others.

Towards a Definition of Conceptual Integrity

Given the apparent lack of consensus about the meaning of the term conceptual integrity, we are taking the liberty to invent our own definition. We do not claim that our definition captures what software researchers and practitioners mean by it; indeed, the lack of consensus suggests no such definition can exist. But our definition is motivated by an attempt to crystallize what is right about the designs that people hold up as exemplars of conceptual integrity (such as the Macintosh computer and the relational database model) and what is wrong about the designs that are criticized for lacking conceptual integrity. In that sense, our definition is descriptive and not prescriptive, albeit indirectly.

We take conceptual integrity to be a judgment about the concepts embodied in a design. By "concepts", we mean the constructs and notions either that preexisting in the world outside the system, such as "bank account" or "airline flight", or that are invented for the purpose of structuring the functions of the system, such as “paragraph style” in a word processor, or “element group” in a drawing application. These concepts, being essential to the design, are both apparent in the interface of the system with the outside world, and in the system’s implementation. In contrast, we do not mean by “concept” anything that exists only at one level, whether it be in the interface (eg, “scroll bar”) or in the implement (eg, “callback”).

Conceptual design for us, therefore, is about the design of behavior, and not the design of internal software structure. Our contention is that these behavioral concepts are not only the major factor in determining ease of use, but also set the stage for the quality of the code. Poor interfaces and excessive coupling between modules usually reflect not so much an organizational problem at the code level, but rather a more fundamental problem that the concepts on which the code abstractions are built are not sufficiently clear and simple.

A design has conceptual integrity, in our view, if the concepts of the design fit together to achieve a certain integrity or wholeness: if the concepts together provide a basis for rich enough functionality, without needless complexity; if the concepts are largely orthogonal to one another, and can be combined to provide a benefit greater than the sum of the benefits of the individual concepts; if the concepts are faithful when intended to reflect preexisting, external notions, and are intuitive and robust when invented anew.

To some, conceptual integrity is about consistent application of design and implementation standards and idioms across a system. As Bill Griswold puts it in a wiki contribution [8]: “Conceptual integrity is the principle that anywhere you look in your system, you can tell that the design is part of the same overall design. This includes low-level issues such as formatting and identifier naming, but also issues such as how modules and classes are designed, etc.” This is not what we mean by the term; we might call this instead “stylistic uniformity”. We do believe, however, that stylistic uniformity is highly desirable, and is likely to be easier to achieve when the design has conceptual integrity (in our sense of the term).

Our definition of conceptual integrity might be criticized for lacking its own integrity; arguably, it says only that the concepts are well chosen and fit together nicely. We intend in our research to redress this flaw by making these judgments more precise. But, following Griswold, we would like to find a characterization of conceptual integrity, some kind of rule of thumb, that captures something of the spirit of the idea.

One candidate is the following: that integrity of design is evidenced by the seeming inevitability of design decisions. In the design of Alloy, our modeling language, we often applied what we called the “fork-in-the-road principle”, which holds that if (when exploring the design space) you reach a fork in the road, where both choices—turning to the left or turning to the right—seem equally plausible, you should turn back. If either choice seemed equally plausible to the designers at design time, they would surely seem equally plausible to users trying to guess how the language is designed (in the case of the language, how to form a construct or what its meaning would be), thus making the language error prone and hard to use. Such a fork in the road might be seen as evidence of a lack of integrity; when a design has a strong core, later decisions often seem to follow almost inevitably, as a consequence of earlier and more fundamental decisions (and of the principles that they established, explicitly or implicitly).

Concepts as Abstract Affordances

In the world of interaction design, the notion of "affordances" (introduced by the psychologist James Gibson [7] and applied to usability by Donald Norman [16]) has become a popular and helpful way to think about what makes an object easy or hard to use. The concepts in our view of design can be viewed as affordances, but of an abstract sort.

In the world today, we are surrounded by designed objects—tools, appliances, software applications, systems, policies and so on—that provide great power at the expense of conceptual complexity. A simple object has no conceptual structure. Rather, it offers affordances: potentials for action that the object communicates more or less directly to the user. Thus a screwdriver affords turning, and a hammer affords banging.

A complex object, in contrast, is less defined by its affordances. Even an old rotary phone affords the action of calling only in an abstract sense; to place a call, the user must understand the concepts of dial tone, phone numbers, busy and ringing signals, and so on.

Software lies at the extreme end of this spectrum, where conceptual structure dominates. Traditional affordances still play a role, but they are rarely associated with actions that had significance in the world prior to invention of the software. The actions afforded by the software become actions in a conceptual universe, conjured up in the mind of the user by the software designer. Sometimes this universe is built by analogy to the real world, and the actions seem familiar (putting items in a shopping cart, sending mail); at other times the connection to the real world is tenuous and actions only make sense in the designer's invented world (grouping objects, reverting files).

Concepts may therefore be viewed as "abstract affordances". The Facebook application, for example, does not offer affordances in the traditional sense, but it does offer abstract affordances—for "tagging", "friending", "posting", and so on—all of which involve state changes in a conceptual world.

Incidentally, the conceptual structure of software exists not only in the mind of the user and designer but also in the code itself. In many software architectures there is an explicit model that represents conceptual relationships directly (in a relational database or object heap, for example). And in an embedded application (such as a car), while most components are oblivious of the structure of the system as a whole, the software controller will likely contain distinct representations of the state and condition of other components—even those it is connected to only indirectly. This seems to be a distinct feature of software that makes it fundamentally different from the technologies of other components.

A Research Program

In the Software Design Group at MIT, we are engaged in a research program to explore conceptual aspects of software design. Our aim is to develop a working theory of conceptual design that would account for the basic notions of concepts and conceptual integrity, and suggest how they can be applied to improve the quality of software design. Our impression is that the best software developers already emphasize the conceptual aspects of their design work; for them, we hope to provide a language for talking more precisely about what they already do, allowing them to articulate their ideas and skills so they can be shared more effectively with others. At the same time, we believe there are many software developers who pay inadequate attention to conceptual design; they rush to implementation, and have trouble thinking or talking about software except in terms of low-level code notions. Our research goal for this audience is to persuade them of the value of conceptual design, and give them a shortcut to the expertise that better software developers have acquired—in much the same way that the design patterns movement gave novice programmers skills with structures that were previously not articulated explicitly.

Our program has several components:

- *Studying existing applications*. We are studying examples of applications that are exemplars of good design, to understand why they work well, and the role that concepts play in their success. In addition to studying the products themselves, we plan to talk to designers to understand how they conceive their designs, and how they articulate conceptual aspects of design (or choose not to). At the same time, equally importantly, we are looking at examples of bad design—namely applications that frustrate users or have serious reliability or development problems—and analyzing the extent to which poorly chosen (or non-existent) concepts contribute to their failure.
- *Cataloging conceptual idioms*. We are building a catalog of conceptual idioms that appear repeatedly in different settings. One of the reasons that users are able to quickly master new and complex applications is that even new functionality is often built from well-understood idioms. Conversely, when an application fails to use conventional idioms, or includes “near misses” (in which idioms appear to be used, but deviate in small but surprising ways), users may struggle to learn how to use it.
- *Case studies in redesign*. To explore the impact of conceptual design, we are engaging in a series of case studies in which we take an application that seems to have serious problems in its conceptual design, and then rework the design to meet criteria of conceptual integrity. In so doing, we hope (a) to gain experience applying the prescriptive criteria that we’re developing in our theory, and (b) to find out to what extent improvements in conceptual structure actually lead to improvements in ease of use.
- *Theory development*. Underpinning all this work, and the ultimate aim of our research, is the development of a working theory. This will clarify the basic notions of concepts,

concept design and conceptual integrity, and will define criteria for good and bad concept design. We plan to identify appropriate notations for articulating conceptual design, extending and refining them as needed. We may also find opportunity for tool development, although our emphasis in our research program is methodological and focused more on intellectual tools than computational ones.

Describing Conceptual Designs

How should a conceptual design be described? Obviously, the description must be implementation-independent; it should be easy to understand; it should precise enough to support objective analysis; and it should be lightweight, presenting little inertia to the exploration of different points in the design space. For these reasons, we have chosen (initially at least) to use a standard state machine model of computation, in which named actions (performed by the user or sometimes by the application) produce transitions between abstract states. The abstract state space is described by a conventional relational data model, using the variant of extended entity-relationship diagrams developed for the Alloy modeling language [12]. The actions are crudely specified by naming them, listing their arguments, and describing their effects on the state informally. This form of description is pretty conventional; instead, we might have chosen any of the standard "model based" specification languages (such as Z [21], B [1], VDM [14], or Alloy [12]). Our own preference is for a diagrammatic representation of the state space, but it may not be essential.

In our descriptions, concepts correspond to state components. Consider, example, a conceptual model of a word processor. The abstract state might include a set of buffers, each consisting of a sequence of paragraphs, each paragraph being a sequence of lines, and each line being a sequence of characters. Each of these state components (the sets of buffers, paragraphs and lines, and the relations that associate them with each other) can be regarded as concepts. Even the simple notion of a character should be regarded as a concept, which becomes significant in the context of Unicode and non-Latin languages.

The reader may reasonably ask whether this implies that concepts are no more than state components. In our view, concepts have psychological content; they correspond to how users think about the application. One could construct a state machine model that characterizes the behavior just as precisely, but whose state components do not correspond to concepts. We would not call such a description a "conceptual model". For example, our word processor might be described not in terms of paragraphs and lines but instead in terms of a single sequence of characters that includes special symbols for end-of-paragraph and end-of-line (or as two sequences, one preceding the cursor and one following it [22]). In this description the basic concepts are not represented explicitly, but have to be defined in terms of the state components (for example, a line being the sequence of characters between two end-of-line symbols).

To connect the abstract state components with the user's understanding of them as concepts, it is necessary to provide a bridge between the two. To do this, we use Michael Jackson's notion of a "designation" [13]: a necessarily informal statement that acts as a kind of recognition rule. For example, in a conceptual model of an application for managing university course registrations, we would likely need a designation for the concept of "student". Designations are invariably more challenging (and more interesting) than they first appear to be; the students registered for a course, for example, might include not only regular enrolled students but also special students, visitors, and even staff and faculty members. A far more challenging example is provide by the concept of an airline "flight", which, as any frequent flyer knows, is encumbered with all kinds of complexities: a flight need not have a single take off and landing, for example (and confusion about the concept of a "direct flight" has been the source of many disappointments for passengers who thought that a direct flight had to be non-stop).

A final note on our approach to description: a reviewer of a paper we wrote suggested that we might have used state machine diagrams (such as Statecharts [9]) for representing the conceptual model instead. But such a notation wouldn't be suitable for applications with richly structured state, where the concepts are primarily about how the state is structured in terms of objects and their relationships to one another.

Examples of Conceptual Models

To make these ideas more concrete, let's look at a few examples of conceptual models, each chosen to illustrate one or more key points. In each case, the model is described informally in text, and is sometimes accompanied by a diagram in Alloy's diagrammatic syntax [12], a simple notation similar to extended entity relationship diagrams or UML's object models (class diagrams).

Airline flight reservations. Our first example illustrates how a conceptual model provides a focus for identifying essential elements of a problem. In a flight reservation system, we might assume that a customer books tickets; each ticket provides zero or more journeys; each journey consists of one or more flights; and each flight has a departure and arrival airport, start and end times, a carrier, and a number (Fig. 1). At this point, we consider whether this model is sufficiently rich to support the situations we are familiar with. We see immediately that the notion of carrier is inadequate; in a code sharing scheme, the airline selling the ticket is not the one flying the plane. So we refine our model to distinguish operating and marketing carriers. More problematically, we might notice that direct flights are not supported. Unlike a non-stop flight, a direct flight might have intermediate stops, and even changes of plane. This realization is more disruptive; we can't just add a notion of intermediate stops (associating flights with airports) because this would not capture the arrival and departure times at those stops. Rather, we should probably invent a new notion, a leg

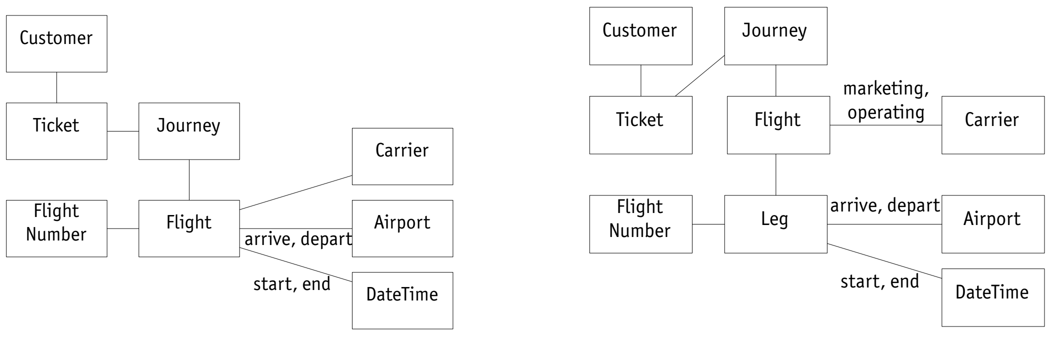


Fig. 1: Airline example, before (left) and after (right) elaboration

say, making a flight a set of legs, each with its own arrival and departure properties. A non-stop flight would then have a single leg, and a direct flight may have two or more.

Note that we are not claiming that the construction of a conceptual model in advance of design and implementation will always result in the uncovering of such subtleties. The conceptual model is a live artifact that captures the designer's evolving ideas, and is valuable for recording ongoing complexities that arise during the development and use of a system. That said, the act of constructing a model does often bring subtle issues to the fore. Simple multiplicity questions turn out to be surprisingly thought-provoking: how many carriers does a flight have? how many flight numbers? how many flight numbers can be associated with one aircraft?

At a more mundane level, the conceptual model defines the vocabulary of a development—in this case, the notions of journey, flight and leg, for example—and encourages the use of names in the code that match names in the problem description. Lack of consistent terminology is rife in software developments and a major cause of unnecessary confusion.

Pasting objects. Our second example illustrates the misconceptions users often have about software and how a clear articulation of the conceptual model might clear up confusions. In most drawing programs, an object has some pixels, each with a color and its own location (Fig. 2). The drawing board consists of some objects, typically in some front to back ordering (whose strucure is an interesting but separate issue) and a grid of pixels whose color (if any) is determined by the pixel of the object at that location in the frontmost layer. Now a novice to Adobe Photoshop will discover that if you select an area in one image, and cut and paste it into another, it can be moved around like an object in a drawing program. Moreover, you can paste again, and the new pixels can be moved around independently of those previously pasted. But when this user tries to go back and move the pixels pasted the first time, there appears to be no way to select them. What has happened is that the user has carried over to Photoshop a conceptual model that does not apply. The Photoshop file is also

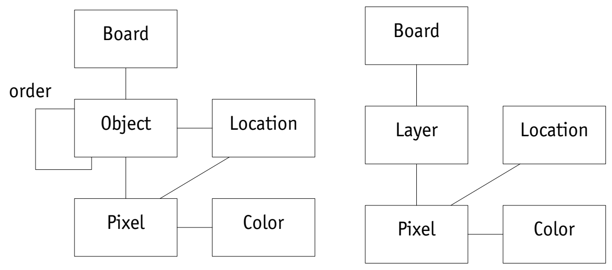


Fig. 2: Pasting example: most drawing programs (left); Photoshop (right)

a set of layers, but there is no notion of object. Pasting a collection of pixels silently adds a new layer and pastes those pixels into that layer. Each subsequent pasting therefore looks as if an object has been added that can be independently moved around, but only because the action that seems to move an object is in fact moving all the pixels in the layer, and that layer only contains the pixels the user associated with an object. In fact, recognizing this problem, Adobe changed the interface of Photoshop Elements so that a layer can be selected by clicking on some pixels appearing only in that layer, thus furthering the illusion of "objects".

Style. Our third example illustrates conceptual idioms: how the same conceptual structure arises repeatedly in different applications. We shall describe it first in abstract terms, and then give examples. A product contains some elements, each of which has some properties. The properties of an element are not given directly by the user, though; instead, the user selects a style for each element, and chooses properties for the style (the user-defined properties in Fig. 3). The implied properties of an element are then determined by the properties of its style. The merit of this scheme is that the user can keep the properties of a large number of elements synchronized by assigning them the same style; changing the properties of that style will then change the properties of all the elements. As David Wheeler famously noted: "There is no problem in computer science that cannot be solved by an extra level of indirection".

The best known application of this idiom, which we might call *Style*, is in word processing and typographic layout programs (such as Microsoft Word, Adobe Indesign and Apple Pages). The elements are typically paragraphs and individual characters; additional complexities include (a) the ability to give elements properties directly, overriding the style properties; (b) arranging the styles in a hierarchy, with property inheritance; (c) storing sets of styles in stylesheets that can be swapped in and out, replacing the properties of a set of styles at once.

Another application of the idiom appears in the notion of a color swatch (Fig. 3, right). An Indesign user colors an element by selecting a swatch; changing the color of the swatch, it turns out, changes the color of all elements to which that swatch was previously applied.

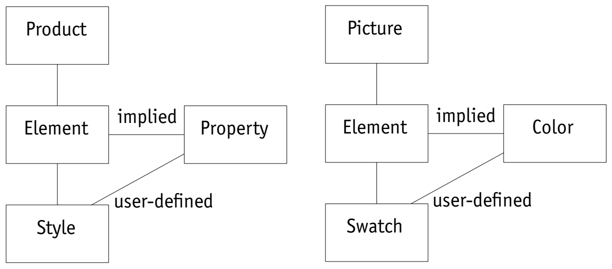


Fig. 3: Style idiom (left) and instantiation (right)

The swatch is therefore a style. Likewise, Powerpoint offers a notion of color scheme, in which the user selects colors for different categories of element (such as titles, body text, etc). In this case, the styles and the mapping of elements to styles is predefined. Apple's color palette, used across applications, appears at first glance to offer a similar mechanism, but while the user can select a swatch to color an element, and can create and delete swatches, there is no way to change the color of an existing swatch.

Articulating conceptual model idioms should have the same kind of advantages that design patterns brought to programming. They give us a way to talk precisely and clearly about complex notions, and to compare and evaluate instantiations. We can observe, for example, that both the Powerpoint and Apple color palette schemes are strictly less expressive, for different reasons, than Indesign's color swatch scheme.

Moreover, designers should be aware that partial or quirky instantiations of idioms are suspect, and can confuse and frustrate users. For example, another widely used and well known idiom might be called *Folder*, in which a collection of resources is placed in a tree of folders. Here are two examples of instantiations in which the normal features of this idiom are violated. In Adobe Lightroom, photos may be placed in collections; these appear to be like directories, but it turns out that a collection cannot be placed inside another collection, but only within a collection set (which itself can be placed inside a collection set, but cannot contain photos). In all IMAP clients, mail folders can form a tree of arbitrary depth. In some, however, mail messages can only be placed in folders that are leaves of the tree (that is, do not contain other folders).

Trash. Our fourth example illustrates the peril of over-generalization. Our critique of Lightroom's instantiation of the *Folder* idiom might be formulated as the observation that the distinction between collections and collection sets is artificial, and a single more general concept might be better. Finding opportunities for generalization is an important part of conceptual design, but it brings risks. In Apple Mail, messages that have been deleted, saved as drafts, or sent, are placed into folders that are, for the most part, no different from user-created folders. This has several unpleasant consequences, the most serious of which

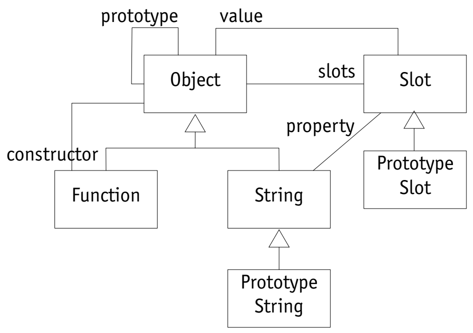


Fig. 4: JavaScript prototypes, slots and constructors

is this: the trash folder, containing deleted messages, records only the times of sending and/or receipt, and does not record the time at which a message was deleted. If an old message is inadvertently deleted, it may disappear into the trash; the user cannot sort the deleted messages by time of deletion to find recently deleted messages. The same issues arises with Apple's trash folder at the operating system level.

Incidentally, the Macintosh trash has a conceptual model that is arguably out-dated. It was conceived at a time when computers had only one persistent storage device, so the concept of a single trashcan made sense. With the advent of multiple drives, the conceptual model was not elaborated. This produces the following dilemma. You insert a USB keyring into your laptop, and attempt to copy some files to it. The operating system complains of inadequate space, so you delete some files. This makes no difference, however, since the deletion does not actually remove the files, but instead marks them as trashed (that is, moved to the trash). To remove them, you must 'empty the trash', but this operation will remove all the files that have been marked as deleted on the laptop's hard drive as well. The conceptual model seems to be simply inadequate to support a common task; the only remedy is to break through the abstraction barrier and execute a Unix rm command on the offending files (which may be found in distinct trash folders, one per drive, indicating that the conceptual model was at least elaborated at this lower level).

JavaScript Prototypes. Our fifth example shows that conceptual models can capture essential design concepts in programming languages. In our software engineering course at MIT [11], the key concepts of JavaScript (the type hierarchy, objects and slots, namespaces, environments and closures, etc) are described using small conceptual models. Here, we illustrate the use of a conceptual model to explain a small but non-trivial feature.

A JavaScript object may have been constructed with a function (using the keyword new); if so, the function is said to be the constructor of the object. Note that the model diagram (Fig. 4) shows the set of functions to be a subset of the set of objects; this implies, in particular, that a function may itself have a constructor. Every object has some set of slots, each consisting of a property (a string) and a value (another object). Any object that has a constructor also has a prototype; when you look up a property of an object, the value, if any, is obtained first by examining the object's slots, and then, if no matching property is found, the properties of its prototype, and so on, up the prototype chain until a matching slot is found or an object is reached without a prototype.

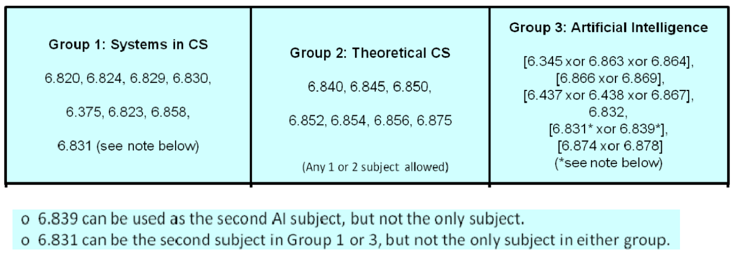
This is all relatively straightforward. The tricky part is understanding how the prototype object is assigned. It turns out that if a function has a slot with the property name "prototype", then any object constructed with that function will acquire the object in that slot at the time of construction as its prototype.

This kind of conceptual model fulfills a different role; it allows us to map out a collection of intertwined concepts, and to address potential confusions. For example, the object in the prototype slot of a function is not its prototype object; changing which object is in the prototype slot of a function will not affect which objects are the prototypes of objects previously constructed; modifying the object in the prototype slot of a function, however, will affect the prototypes of objects previously constructed. (The model diagram is particularly unsatisfying in this case; it doesn't even show that the prototype slot of a function has the prototype string as its property, let alone capturing these mutability issues. Interestingly, textual notations such as Alloy can capture the invariant but not the mutability aspect, at least without specifying the exact effects of operations.)

Apache security. Our sixth example revisits the earlier theme (from the pasting example) of mismatched conceptual models in a more critical setting. A web application in Apache typically consists of some script files and some data files. In the Apache configuration files, a web developer can list which files the web server is allowed to serve in response to HTTP requests. By indicating that only the script files and not the data files should be served, we can prevent secret data from being served directly to users; instead, access to data files containing sensitive information will have to be accessed by scripts. An inexperienced developer might reasonably assume that these configuration settings ensure that only his scripts can access his data, and so long as the scripts are written carefully, sensitive data will not be leaked.

This conceptual model is wrong, however, because it fails to account correctly for how scripts obtain access to data files. A script runs (by default) with the privilege of the web server; if a sensitive file is to be readable by a script, it must give permission to the web server to read it. Now the problem is that a script belonging to another user on the same system will run with the same privilege. So all an attacker need do to read the contents of a sensitive file is obtain an account on the machine, and write a script that accesses the file. A web application that held private student data at MIT suffered from exactly this vulnerability [15].

Sometimes conceptual model mismatches arise because an application is ported from one context to another, with a change in conceptual model. In the Unix file system, a file's protection is independent of its location; when a file is moved, it carries its permission bits with it. In AFS, however, a file's protection is determined by the access controls of its parent



Group 1: Systems in CS	Group 2: Theoretical CS	Group 3: Artificial Intelligence
6.820, 6.824, 6.829, 6.830, 6.375, 6.823, 6.858, 6.831 (see note below)	6.840, 6.845, 6.850, 6.852, 6.854, 6.856, 6.875 (Any 1 or 2 subject allowed)	[6.345 xor 6.863 xor 6.864], [6.866 xor 6.869], [6.437 xor 6.438 xor 6.867], 6.832, [6.831* xor 6.839*], [6.874 xor 6.878] (*see note below)

- o 6.839 can be used as the second AI subject, but not the only subject.
- o 6.831 can be the second subject in Group 1 or 3, but not the only subject in either group.

Fig. 5: MIT computer science degree course requirements

directory. The SSH application was designed under the assumptions of a Unix file system, and stores both of a user's keys—private and public—in the same .ssh directory. A user who installs keys in the standard Unix manner will expose not only the public key, but the private key too (since the directory must be world readable, to allow the SSH daemon to access the public key prior to authentication).

Organizational Policy. As a seventh and final example, we illustrate the use of conceptual modeling to simplify an organizational policy. Many software systems operate in social contexts in which policies and protocols must be reflected in the software design. Companies that build large enterprise systems call these the "as-is business process" and will often advocate changing to a new business process that will make the software easier to build and more effective. Even when software is not involved, conceptual models can be used to improve how an organization works.

Our example is taken from the doctoral degree requirements in computer science at MIT. A student is required to take courses to acquire some breadth, according to a set of rules specified by a table (Fig. 5), which can be summarized as follows. There are three groups, each with a set of options; the student must select one option from each of two groups, and two options from the other group. Some options are mutually exclusive, and some may only be used as the second option in a group. Finally, there is an unstated rule that none of the chosen options may be for the same course.

These rules comprise a conceptual model that is actually more complex than it needs to be. It can be reformulated as follows. There is a set of courses, each of which is associated with zero, one or more areas. Some courses conflict with each other. A student must take 4 courses in total, including one course in each area, and no pair of course in the selection may conflict. With this reformulation, we can see that the whole concept of a second option in a group was a needless complication; the notion probably arose because it seemed natural to assign courses to groups according to their subject matter, even if the assignment had no significance for limiting the selection.

This example might seem pedantic. But this was exactly how the rules were implemented in a web application used to check them, and the reformulation has allowed a much simpler and cleaner implementation. (Incidentally, the equivalence of the two conceptual models is not obvious, and actually requires some side conditions to hold—for example, that any option appearing in a group must be a second-course-only option if there is another option, for the same course, appearing in another group.)

Afterword: A Counter-Cultural Agenda

On the one hand, this research agenda seems to align with a broad interest amongst software practitioners and researchers in software design. In a series of talks that the author has given about this work to date, the response from audiences has been remarkably enthusiastic. Almost everybody, it seems, is interested in the question of what constitutes the essence of usability, and to date nobody has challenged us on the primacy of conceptual design.

On the other hand, there is a respect in which this agenda is deeply counter-cultural. Both research and teaching in software engineering have always emphasized "hard" over "soft" topics. In a typical software engineering or programming course, for example, almost no attention is paid to questions of how to actually design the behavior of software, or how to design a good interface, or how to decouple modules from one another. Instead, emphasis tends to be placed on topics that can be readily formalized, or which involve concrete technical notions. So rather than explaining how to use languages, we focus on their syntax and semantics; instead of explaining how to design modules, we focus on modularity mechanisms and namespaces; instead of talking about degrees of confidence in software correctness, we focus on definitions of test coverage criteria; and so on. Research conferences in software engineering seem to be moving away from fundamental questions in software design and development to a lower-level, more technical focus on tools, testing and analysis.

A drive in the software engineering community towards greater emphasis on empirical studies [24, 23] drastically exacerbated this problem, in our opinion. While the desire for more quantitative and more empirical research was well intended, the result of this drive was a widespread adoption of a dogmatic viewpoint amongst researchers that devalued intellectual enquiry. A research paper that presented a new idea, argued for it compellingly, and illustrated it with carefully constructed examples (designed to embody the essence of the problem) was no longer regarded as academically respectable, since it lacked "empirical validation". It seems unlikely that many of the great (and most influential papers) in software engineering, such as Parnas's paper on modular decomposition [18], could be published in a top conference today. Also, there seems to be much less enthusiasm amongst researchers for the kind of informal writing about software development that Dijkstra and Hoare engaged in extensively (such as [5]) and which had so much influence on the field.

Of course, this is not to say that we want lots of speculative papers with weak arguments. The problem, rather, is that we no longer seem willing even to consider intellectual arguments that lack empirical backing, as if we have lost the confidence to judge ideas for their intrinsic merit and the quality of the argumentation.

The result of this empirical trend seems to be a general dumbing down of the field. Programme committees have less patience for careful arguments and favor the collecting of data and the construction of tools over intellectual contributions. Although the empiricists hoped that a call for data might open a reconsideration of cultural assumptions, almost the opposite seems to have occurred: it is now harder to question conventional wisdom since the bar for any criticism has been raised so high. And perhaps worst of all, researchers have been side-tracked by a perceived need to perform "empirical studies", often involving inexperienced student developers working on toy systems, giving a spurious sense of validation, or going through the motions of collecting data without any clear hypothesis [19]. And, despite all this effort and the publication of collections of studies [such as 17], we seem to be no closer to a useful empirical assessment of any of the fundamental ideas of the field.

Consequently, although we expect our research agenda to be exciting and engaging, to resonate with the intuition and passion of developers, and to connect us to the larger community of design researchers, we don't expect it to be easy to publish or obtain funding. While the most important questions seem to us to be methodological and philosophical (what is a concept? what makes a concept good or bad?), our research culture will likely look for value in more technical areas, such as formalization of notations, analysis tools, and construction of code.

A proposal to the National Science Foundation for this project was rejected earlier this year. No doubt much of the blame falls on us for failing to explain and articulate our aims and plans well enough. Nevertheless, the reviews seemed to reflect these culture biases. As the panel summary stated:

> *The proposal does not make clear the linkage between models and implementations. How would these formal notations for concepts help in realizing code? … The examples are compelling, but there needs to be stronger evidence that formal concept notations could translate into better implementations. The panel suggests linking the ideas on concept notations to code, giving a sense of what the notation could be, and what analyses would be enabled by the proposed formal notation.*

Fortunately, while the field of software engineering seems to be moving away from design, the community of researchers in other fields who focus on design as a topic in its own right are gaining ground. A new university in Singapore—the Singapore University of Technology and Design—has been established with the aim of making design central in the engineering curriculum (and our project has been funded in part by a grant from the university's International Design Center [10]).

Acknowledgments

The views in this memo were formed in the context of many discussions with the members of the Software Design Group at MIT (Jonathan Edwards, Eunsuk Kang, Aleks Milicevic, Joe Near, Santiago Perez De Rosso) and our recent visitors (Alcino Cunha, Andrea Mocci), and in particular by our first (and ongoing) redesign case study—a conceptual redesign of the Git version control system led by Santiago Perez De Rosso. The ideas and writings of Michael Jackson have strongly influenced this project. Parts of this memo were written in response to helpful comments from the programme committee of the SPLASH 2013 Onward track, especially Dick Gabriel.

References

[1] J. Jean Raymond Abrial. *The B-Book: Assigning Programs to Meanings*. Cambridge University Press, 2005.

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

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

[4] Frederick P. Brooks. *The Design of Design: Essays from a computer scientist*. Addison-Wesley, 2010.

[5] W. Dijkstra Edsger. On the role of scientific thought. EWD447, 30th August 1974, The Netherlands. Appearing in *E.W. Dijkstra, Selected Writings on Computing: A Personal Perspective*. Springer Verlag, 1982.

[6] 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.

[7] 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.

[8] William Griswold. *Conceptual integrity*. Wiki page from course notes. Available at: http://cseweb.ucsd.edu/users/wgg/CSE131B/Design/node6.html.

[9] David Harel. Statecharts: A visual formalism for complex systems. *Science of computer programming*, 8.3 (1987): 231-274.

[10] International Design Center, Singapore University of Technology and Design. http://www.sutd.edu.sg/idc.aspx.

[11] Daniel Jackson, Jonathan Edwards and MIT Teaching Staff. *6170: Software Studio*, Fall 2012. At: http://stellar.mit.edu/S/course/6/fa12/6.170.

[12] Daniel Jackson. *Software Abstractions: logic, language, and analysis*. MIT Press, 2006; Revised edition, 2012.

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

[14] Cliff B. Jones. *Systematic software development using VDM*. Prentice Hall, 1990.

[15] Eunsuk Kang and Daniel Jackson. *A Model-Based Framework for Security Configu-*

ration Analysis. Unpublished manuscript. Available at: http://people.csail.mit.edu/eskang/papers/security-configuration.pdf.

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

[17] Andy Oram, Greg Wilson. *Making Software: What Really Works, and Why We Believe It*. O'Reilly Media, October 2010.

[18] 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.

[19] Dewayne E. Perry, Adam A. Porter, and Lawrence G. Votta. Empirical studies of software engineering: a roadmap. *Proceedings of the Conference on The Future of Software Engineering*. ACM, 2000.

[20] Eric S. Raymond. The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary. In: *The Cathedral and the Bazaar*, Snowball Publishing, 2010.

[21] J. Michael Spivey. *The Z Notation: A Reference Manual*. Prentice Hall, 1992.

[22] Bernard Sufrin. Formal specification of a display-oriented text editor. *Science of Computer Programming*. Volume 1, Issue 3, May 1982, pp. 157–202.

[23] W.F. Tichy, P. Lukowicz, L. Prechelt, and E.A. Heinz. Experimental Evaluation in Computer Science: A Quantitative Study. *Journal of Systems and Software*, 1995. 28(1): p. 9-18.

[24] M.V. Zelkowitz and D. Wallace, Experimental validation in software technology. *Information and Software Technology*, 1997. 39(11): p. 735-744.