Conceptual Integrity

Rawld Gill
ALTOVISO LLC

Introduction

Today I am more convinced than ever. Conceptual integrity *is* central to product quality.
--Fred Brooks, The Mythical Man-Month after 20 years

This is a well-known and accepted cornerstone of software engineering. But, what is conceptual integrity, how is it measured, and how is it achieved? It seems that conceptual integrity is one of those ideas that isn't formally defined but you know it when you see it. We can and must do better; particularly if it is *the* key factor to product quality.

Informal Definitions

The American Heritage Dictionary provides the following definitions:

concept

1. A general idea derived or inferred from specific instances or occurrences.
2. Something formed in the mind; a thought or notion. See synonyms at idea.
3. Usage Problem. A scheme; a plan.

integrity

1. Steadfast adherence to a strict moral or ethical code. See synonyms at honesty.
2. The state of being unimpaired; soundness.
3. The quality or condition of being whole or undivided; completeness.

A reasonable combination of these definitions for the purposes of creating an adjective describing software might be:

conceptual integrity: a complete, undivided, and sound usage pattern.

[*Note:* "whole" could be included with "complete", but this seems redundant. The same can be said for "unimpaired" with respect to "sound" and "plan" and "scheme" with respect to "usage pattern".]

This captures the semantics of "conceptual integrity" nicely, albeit informally.

Griswold makes another attempt at a definition:

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.

Definitions similar to Griswold's are found in several computer science essays and curricula. The definition is more of a heuristic than a formal definition; nevertheless, it helps solidify our understanding.

Brooks gives us advice on how to achieve conceptual integrity:

> Having a system architect *is* the most important single step toward conceptual integrity.

Brooks assumes that if one mind is responsible for the conceptual integrity of the system, then the system ought to be complete, undivided, and sound. This proposition seems terribly flawed. The history of any organization that constructs software is more often than not littered with software products that have had the benefit of a single architect yet been dismal failures, utterly lacking in conceptual integrity. It is not having a single architect that guarantees conceptual integrity, but rather, having a single *enlightened* architect. But what does it mean to be enlightened?

We must also ask the question, is it the singleness of the system architect that is most important or is it the "enlightenedness"? And, if it is insight that is important, can we codify this insight, teach the code to a set of architects, and then divide the labor of system architecture so as to decrease construction time while maintaining product quality?

Formal Definitions

A software product is a set of consumable computer instructions. It exists and is a seeable, touchable entity (e.g., on disk, in chip, etc.). Upon some stimulus, the instructions do something. The range of stimuli recognized by the instructions can be characterized as a language that communicates intentions to the software. The consequence of executing instructions can be characterized as function: i.e., they do *something*, and the "something" is their precise function.

(e1) software product= language + function controlled by the language

Every software product is composed of a combination of a language and function controlled by the language. The language allows consumers to demand function. Consider the following examples:

The C library function strcat	
Consumer	C programmers
Language	char *strcat(char *pDest, const char *pSrc)
Function	Append the source string to the destination string
The Java compiler	
Consumer	Java programmers
Language	The Java language specification and compiler control switches
Function	Transform text into Java byte code
The Windows operating system	
Consumer	System administrators
Language	The user interface to the various applets that control the operating system
Function	Control hardware devices and software programs.
A word processor	
Consumer	The author writing this essay
Language	The user interface to the word processor
Function	Construct a file of formatted text
The micro-code controlling a cell phone	
Consumer	The phone engineering team

Language	The function library interface specification
Function	Control the phone hardware

These examples have been intentionally taken from a wide cross-section of software products to show that, ultimately, every software product has a consumer, a language, and a function.

In discussing conceptual integrity, Professor Brooks offers, “Ease of use, then, dictates unity of design, conceptual integrity.” Brooks cautions that ease of use must be normalized by function: “[…the] *ratio* of function to conceptual complexity is the ultimate test of system design.” A program that adds 2 and 2 and returns the result upon pressing the enter key is trivially easy to use, but offers little functionality. Conversely, e.g., an assembler can be used to demand any function possible on a particular processor, yet providing a secretary with an assembler and instructing him or her to write a letter, while possible, is ludicrous.

Still, if we understand system design is the key to product quality, then Brooks’ equation can help us in our investigation of conceptual integrity:

(e2) product quality≈ system design= function / conceptual complexity

Further, if we compare e1 and e2 we see a direct correlation between “function”. Therefore, it is clear that conceptual complexity is referring to the conceptual complexity of the language defined by the product. This, then, is our thesis:

> As an equation:
>
> (e3) The conceptual integrity of a software product is inversely proportional to the conceptual complexity of the language defined by that product.
>
> As a definition:
>
> conceptual integrity: the measure of conceptual simplicity of the language defined by a product.

The skeptic may argue that some products have been constructed with ingeniously simple languages (e.g., user interfaces) that accomplish great and difficult functionality, yet close examination of the internals of those products reveal an ugly combination of techniques, coding styles and practices, patches, and various other sins. But this argument is invalid because the consumer of the software product doesn’t (or, at least, ought not to) care how the product is constructed so long as when a functional intent is expressed in the language, the function is carried out correctly.

This is not to say that the internal construction is unimportant: it is often critical to ensure proper function. But, the focus of product and consumer must be changed in order to apply our definition of conceptual integrity when examining the internal structure of a program. For example, if the consumer is changed to “the designers and programmers building and maintaining the product”, then the language becomes (e.g.) the type hierarchy and the functions used to make the type hierarchy accomplish functions[1]. And if this language “reveals an ugly combination of techniques, coding styles

[1] The language also includes the rules for combining types and functions.

and practices, patches, and various other sins", then, clearly, the language contains unnecessary conceptual complexity and therefore the product (i.e., the internal design) lacks conceptual integrity.

Indeed, the idea of refocusing the consumer and the product can be applied throughout the software development life cycle from vision to requirements, design, and implementation. Similarly, refocusing can apply to different system partitions, e.g., subsystem-x, subsystem-y, supersystem-z composed of x and y. Each of these phases, subsystems, and supersystems has some measure of conceptual integrity.

Measuring Conceptual Integrity

E3 states that in order to measure the conceptual integrity of a product, we must measure the conceptual complexity of the language defined by the product. There are several well-established metrics by which to measure languages:

Correct	The presence of rules to adjudicate any sentence as valid or invalid; the absence of any conflicting usage rules.
Complete	The capacity for the language to express all desired intents.
Parsimony	The frugality of the language.
Similarity	The similarity of the language with other languages with which the consumer is familiar.
Extendibility	The ability of the language to be extended.

Correct and Complete

Correct is a binary quality: a given language is either correct or not. If the language rules can be applied to every combination of language elements and return a boolean value indicating that the combination is a legal or not, and further, if it is a legal combination, imply exactly one semantic, then the language is correct; it is incorrect otherwise.

Similarly, complete is a binary quality. If every function available in the software product can be expressed through the language, then the language is complete; it is incomplete otherwise.

Extendibility

Extendibility does not apply when adjudicating the conceptual integrity of a single software product because, by definition, a single product never changes. Ergo, it never requires extending. Of course, many (most?) real software products evolve over time. Are different versions of the "same product" actually different products? The answer depends upon your perspective. Users usually view different versions as the same product; implementers see this area as grey; validation and regulatory authorities (e.g., the FDA, when constructing medical devices) view different versions as different products.

But, users don't care about what might happen next year. Maximum product quality, today, is paramount. This implies that a future product revision must take into account past releases. We will see that backward compatibility is a measure of similarity; it is not a measure of extendibility. Thus, in the end, extendibility is a quality that affects the long-term management of a product line rather than the conceptual integrity of any individual product. This is not to say that it is not important. To ignore planning for extendibility is to plan for the inability to achieve conceptual integrity down the road.

Similarity

Similarity is that quality of being familiar and/or natural. Compatibility is a key measure of similarity. As discussed above, a familiar and accepted language that is extended but not otherwise changed possesses a high value of similarity. Consider the various versions of Microsoft Word through the years. Each subsequent version contained additional functionality together with language additions (i.e., user interface enhancements) that provided access to the new functions. Yet most of the language remained constant. Typically, a user can begin using a new version of Word without changing a single usage idiom.

Similarity often extends far beyond compatibility into customs and culture. For example, infix arithmetic notation is more similar to most than postfix: is there any doubt that "5 * (x + 3)" is easier to understand then "5 x 3 + *". On the other hand, those of us (count the author in this group) who have been keying Hewlett Package postfix calculators for 30+ years often have to think twice after picking up a Texas Instruments infix calculators. Thus the medium of expressing the language (e.g., writing versus keypunching) as well as the user community are both critical factors when adjudicating similarity.

Similarity can be quantified—at least for the purposes of comparing two competing designs.

Let U be the population of users of a software product P.
Let L be the language associated with P.
Let e be an element or rule of the L.
Let D(e) be the definition of e.
Let V_s be the value of similarity for L.

Then the following algorithm can be used to provide a reasonable approximation for V_s.

```
$V_s = 0$
for each u ε U
    for each e ε L
            if u currently uses e is some product as D(e)
                    $V_s = V_s + 2$;
            else if u understands e to have only semantics D(e)
                    $V_s = V_s + 1$;
            else if u understands e to have semantics D(e) as the $i^{th}$ possibility of D(e)
                    $V_s = V_s + (1 / i)$;
            else (u has no idea what e implies)
                    no-op
```

If we apply the algorithm to a 100% backward compatible language that all users are actively using, then V_s is maximized. Excluding backward compatibility issues, if we maximize the elements of L that the maximum of users understand, then V_s is also maximized.

Parsimony

Once again, we begin by turning to the American Heritage Dictionary:

Parsimony

1. Unusual or excessive frugality; extreme economy or stinginess.
2. Adoption of the simplest assumption in the formulation of a theory or in the interpretation of data, especially in accordance with the rule of Ockham's razor.

Certain aspects of this definition can be immediately formalized. For example, we can apply Ockham's razor to a language, and assign a point for every unnecessary element that is cut. The parsimony value of the language then is inversely proportional to the points awarded during this exercise.

But parsimony is more than eliminating unnecessary elements: it is defining the optimal necessary elements. Two different languages may be defined that communicate the same intents with neither language containing any unnecessary element, yet one language may be significantly more frugal/economical than the other. For example if one compares SQL to Date and Darwen's Tutorial D, we find Tutorial D much more frugal and economical when compared to SQL, yet Tutorial D provides more functionality than SQL over the same function domain. There can be no doubt that Date and Darwen are clearly "enlighten" architects. But, what is it, exactly, that they did when they sat down to design Tutorial D? Or, conversely, what was not done when designing SQL?

In addition to applying Ockham's razor, any parsimonious language ought to be:

Orthogonal	Each language element and rule ought to be independent of all other elements and rules.
Consistent	Like elements ought to be used similarly; like rules ought to be formulated similarly.
General	Elements and rules ought to apply to the greatest universe of language constructions.

[*Note:* others, notably Jon Bentley in his essay "Little Languages", group parsimony, orthogonality, consistency, and generality as separate, co-equal attributes at the same level of decomposition.]

Orthogonality

Orthogonality is a fairly well-understood design concept. Achieving orthogonality requires progressively refactoring a language until the elements are independent.

Orthogonality is, with careful analysis, easily determined. For every par of language elements and rules, look for any intersecting semantics. When an intersection is found, determine the best way to refactor:

- Option 1: one element is completely subsumed by the other; thereby eliminating an element.
- Option 2: a third element is factored out of the two overlapping elements; thereby adding an element.

When no intersection is found between any element, the language is orthogonal. This implies that the orthogonality of a language can be given a weight. For example, a point can be assessed for each intersecting component. The orthogonality value of the language is then inversely proportional to the points awarded during this exercise.

Consistency

Consistency is achieved by minimizing the number of rules defined by a language. Consider Figure 3. drawing-attribute values are used in different ways depending upon the position of the value in **`draw`**. This results in two rules. And drawing-attribute is used inconsistently (once as a color, once as a resolution).

Elements	
Shape::=	**`circle	square`**

drawing-attribute::=	**`red`** | **`green`** | **`blue`** | **`low_resolution`** | **`hi_resolution`**
draw::=	**`draw(`**shape,drawing-attribute$_1$**`,`**drawing-attribute$_2$**`)`**
Rules	
draw, *drawing-attribute$_1$*	must be one of **`red`** | **`green`** | **`blue`** from drawing-attribute.
draw, *drawing-attribute$_2$*	must be one of **`low_resolution`** | **`hi_resolution`** from drawing-attribute.

Figure 1

Figure 4 corrects the problem. Two rules are eliminated.

Elements	
Shape::=	**`circle`** | **`square`**
color::=	**`red`** | **`green`** | **`blue`**
resolution::=	**`low_resolution`** | **`hi_resolution`**
draw::=	**`draw(`**shape**`,`**color**`,`**resolution**`)`**
Rules	
none	

Figure 2

Professor Date captures why increasing the number of rules required by a language definition tends to make a language more complicated and less powerful:

- The language is more complicated because of the additional rules needed to define and document the exceptions and special cases.
- The language is less powerful because the purpose of those additional rules is precisely to prohibit certain combinations of constructs and hence to reduce the language's functionality.

Date applies the argument to justifying orthogonality as a desirable characteristic. But the argument is valid over all attributes of parsimony.

Consistency can be assigned a weight. For every circumstance where some element exhibits an orthogonal usage pattern assign a point (e.g., in Figure 3, drawing attribute color is orthogonal to resolution). The consistency value of the language is then inversely proportional to the points awarded during this exercise.

Generality

This leaves generality. A simple example is found in operator+ in just about any language that supports integer and floating point arithmetic. Integer operator+ is a completely different function that floating point operator+. Yet, the "+" symbol used between two operands has the same meaning independent of whether the operands are integers or floating point value/variables: calculate the sum of the operands. This semantic is often extended to strings (e.g., operator+ defined for basic_string in the standard C++ library).

Finding errors in generality requires finding, loosely speaking, isomorphic language constructs that are specified by different language elements and rules. For example, strcat is not a general solution to "summing two strings" whereas operator+ is a general solution. Unfortunately finding isomorphic constructs is often a matter of taste. For example, integer operator+ is commutative; basic_string operator+ is not. Does this mean, for the purposes of generality, basic_string operator+ is not isomorphic to integer operator+. No. But, this is not provable. It is a matter of taste and experience.

Assuming we have enough taste and experience to find isomorphic language constructs, generality can be calculated by assigning one point for each pair of isomorphic constructs. The generality value of the language is then inversely proportional to the points awarded during this exercise.

Finding the Optimum Value for Conceptual Integrity

Recapping the last section, we have:

Conceptual integrity is maximized when the language defined by the software product:

- Is correct, absolutely
- Is complete, absolutely
- Maximizes similarity
- Maximizes parsimony by maximizing
    - orthogonality
    - consistency
    - generality

And the designer should keep in mind that extensibility is important if future enhancements are to have any hope of high conceptual integrity.

Of course some of these measures compete some of the time. In particular, placing a high value on similarity may prohibit maximizing certain attributes of parsimony. Indeed, how many times have architects been forced to include ugly language design in order to maintain backward compatibility? Perhaps the most poignant example of this is certain C syntax that is now a part of C++. Stroustrup had good and valid reasons for doing this, and the world isn't perfect. But there are examples of products carrying forward antiquated, expensive, and painful language attributes for foolish (typically, marketing) reasons. The product would be better, and the customers happier (and sales easier)—if similarity to bad ideas was abandoned. These arguments quickly turn to optimizing revenues which is another topic entirely.

It is worthy of note that frequently these measures compliment each other. This is most often realized when refactoring to achieve orthogonality, consistency, and generality. A design modification caused by finding an error in one attribute often leads to further optimizations on the other two.

Finally, it is certainly true that language design ought to be correct and complete. To do otherwise has only negative consequences.

Conclusion

I have shown that maximizing conceptual integrity implies minimizing conceptual complexity, and that conceptual complexity is purely a function of the language component of any software product. Finally, there are objective, rigorous measures of language complexity as well as heuristics for minimizing the conceptual complexity of a language.

Understanding and applying these heuristics is the key to achieving conceptual integrity. It is clear that certain talented and experienced scientists and engineers seem to gain these insights almost magically. But knowing these principles provides less blessed individuals with tools that may be used to achieve conceptual integrity earlier in their career. Perhaps even more important, applying this knowledge with wise management techniques may allow Brook’s advice of one system architect to be ignored with impunity. It may be possible—again, in conjunction with proper management--to design by team so long as all members know and follow the rules.