Software Construction (Version 0.5)

Terry Bollinger

Software Construction

Software construction is the most fundamental act of software engineering: the construction of working, meaningful software through a combination of coding, self-validation, and self-testing (unit testing) by a programmer. Far from being a simple mechanistic "translation" of good design into working software, software construction burrows deeply into some of the most difficult issues of software engineering. It requires the establishment of a meaningful dialog between a person and a computer – a "communication of intent" that must reach from the slow and fallible human to a fast and unforgivingly literal computer. Such a dialog requires that the computer perform activities for which it is poorly suited, such as understanding implicit meanings and recognizing the presence of nonsensical or incomplete statements. On the human side, software construction requires that forgetful, sloppy, and unpredictable people train themselves to be precise and thorough to the point that at the least they do not appear to be completely insane from the viewpoint of a very literal computer. The relationship works at all only because each side possesses certain capabilities that the other lacks. In the symbiosis of disparate entities that is software construction, the computer provides astonishing reliability, retention, and (once the need has been explained) speed of performance. Meanwhile, the human side provides something utterly lacking on the part of the computer: Creativity and insight into how to solve new, difficult problems, plus the ability to express those solutions with sufficient precision to be meaningful to the computer. Perhaps the most remarkable aspect of software construction is that it is possible *at all*, given the strangeness of the symbiosis on which it is based.

Software Construction and Software Design

To fully understand the role of software construction in software engineering, it is important to understand its relationship to the related engineering concept of *software design.* The relationship is closer than it might seem: Software design is simply what happens when software construction is "parceled out" to more than one developer, usually for reasons of scale or schedule constraints. Since humans are notoriously bad at communicating with each other efficiently and without quarrelling, this act of parceling out construction tasks requires the use of procedural and automated enforcement to ensure that programmers "behave" over the course of completing their tasks. The act of parceling out software construction to create a multi-person software design activity also unavoidably entails a massive drop in productivity, since it is far more difficult for such a group to remain "sane" from the perspective of a literal computer that must run their integrated results.

In practice, nearly all of the skills required in "software design" are also a key part of software construction, as demonstrated by the accomplishments of many individual programmers who have created powerful and well-designed software by directly applying design and construction methods. The idea that all of the hard problems are solved during "design" can lead directly to the common but highly inaccurate impression that software construction is nothing more than the "mechanistic" translation of software designs into actual software. If this were actually the case,

Timestamp: 9908251654

the commercial market would long ago have moved entirely to abstract design tools that did in fact automate the final coding steps out of existence. While progress has indeed been made on this front for some specialized styles of software construction, the actual consequence of such automation is simply to move programming-style activities up to a new and more powerful level of capability. Furthermore, the idea that significant levels of repetitive mechanical translation by people should be tolerated in any strong software engineering process is inherently absurd, as it attempts to make costly and error-prone humans perform the very types of tasks for which the computer is vastly better suited. Sadly, project politics also play in some perceptions of the relative importance and complexity of the design and construction efforts. While it is not at all uncommon for experienced programmers to massively repair, replace, or expand a documented design as the "details" of the implementation become more apparent, it is seldom to their benefit politically to shout too loudly about the naivete or inaccuracy of original design. As a result, the software design summary reports and metrics of projects seldom reflect the exact dynamics of what happened between design and construction, and in more than a few cases they may be vastly and unfairly biased towards touting the results of the "design" phase over construction.

The Spectrum of Construction Techniques

Software construction techniques can be broadly grouped in terms of how they fall between two endpoints: computer-intensive construction techniques, and human-intensive construction ones.

Computer-Intensive Construction	←	**Human-Intensive Construction**
Task: Configure limited sets of options	←	*Task:* Configure nearly unlimited sets of options
Description: Most knowledge is "in" the computer	←	*Description:* Most knowledge is "in" the programmer
Goal: Move tasks into this category	←	*Goal:* Move tasks out of this category

Computer-Intensive Construction

In the simplest forms of software construction the computer has been previously programmed to take care of the vast majority of algorithmic steps, and the human then takes the simple role of "configuring" software to the special needs of an organization of people and other systems. While this type of construction is far less taxing on the human constructors, it still requires skills such as communication with an organization that are far beyond the capabilities of an unassisted computer, and thus is unavoidably an example of communicating intent to a computer – that is, of software construction. More important, however, is that this simple style of construction should in many cases be the goal to which all construction methods should aspire, since it is vastly more reliable and efficient than nearly any other type of communication-of-intent interface to a computer. Once implemented, configuration-style software construction "swallows" huge chunks of the overall software engineering process and replaces them with a more localized process that is controlled by the computer. It also allows the overall software engineering process to refocus on new and more complex problems that have not yet been so thoroughly automated.

Human-Intensive Construction

At the other extreme of software construction is the development of new and creative algorithms and software mechanisms, in which the programmer must wrestle with unsolved problems or the first-time automation of highly complex processes. One example of this type of programming would be creating a driver for an entirely new class of software peripheral. Human-intensive software construction requires the use of people who are capable of "thinking like computers," and thus are able to provide the computer with the kind of expertise that makes the computer appear to be "smarter" than it was before. This end of the construction spectrum is close to the traditional concept of "coding," although in terms of its burden on the programmer it bears very little resemblance to the idea that programming is simply "translating" higher-level designs into code. On the contrary, this end of the software construction spectrum requires a rich mix of skills including creative problem analysis, disciplined formal-proof-like logical and mathematical expression of solutions, and the ability to "forecast" how the resulting software construction will change over time. The best constructors in this domain not only have much the same skills as advanced logicians, but also the ability to use those skills in a much more difficult environment.

As indicated by the left arrows and goals in the table, all software construction methods should as a general goal work to move tasks away from the human-intensive side and into the computer-intensive side. The rationale for this is straightforward: People are too expensive and error-prone to waste in re-solving the same problems repeatedly. However, moving construction processes to the left can be difficult to accomplish in practice, especially if large numbers of developers are involved or the overall development process does not recognize the need for automation. Even so, the concept of moving towards higher levels of automated, computer-intensive software construction is sufficiently fundamental to good design that it permeates nearly every aspect of the software construction problem. For example, even as simple a concept as assigning a value to a constant at the beginning of a software module reflects the automation theme, since such constants "automate" the appropriate insertion of new values for the constant in the event that changes to the program are necessary. Similarly, the concept of class inheritance in object-oriented programming helps automate and enforce the conveyance of appropriate sets of methods into new, closely related or derived classes of objects.

Computer Languages

Since the fundamental task of software construction is to communicate intent unambiguously between two very different types of entities (people and computers), it is not too surprising that the interface between the two would be expressed in the form of languages. The resulting *computer languages*, such as Ada, Python, Fortran, C, C++, Java, and Perl, are close enough in form to human languages to allow some "borrowing" of innate skills of programmers in natural languages such as English or French. However, computer languages are also very nit-picky from the perspective of natural languages, since no computer yet built has sufficient context and understanding of the natural world to recognize invalid language statements and constructions that would be caught immediately with a natural language.

Construction Languages

Construction languages are a broader set of languages that include not only computer languages such as C and Java, but also pre-constructed sets of parts and terms that have been designed for use together in constructing new software for some well-defined set of problems. Thus the example of sets of simple software configuration options mentioned earlier can also be understood as constituting a small but well-defined construction language for the problem domain of how to configure that software. The concept of construction languages is useful for emphasizing issues such as the need to transition over time from human-intensive computer languages to computer-intensive ones in which the required construction languages are simpler and more problem-focused. In practice the creation and use of specialized construction languages usually begins very early in the construction process, even for one-person efforts. This occurs in part because the creation of simpler, more automated sets of components is an important tool for controlling complexity during the construction process.

Styles of Construction

A good construction language moves detailed, repetitive, or memory-intensive construction tasks away from people and into the computer, where they can be performed faster and more reliably. To accomplish this, construction languages must present and receive information in ways that are readily understandable to human senses and capabilities. This need to rely on human capabilities leads to three major styles of software construction interfaces:

A. Linguistic: Linguistic construction languages make statements of intent in the form of sentences that resemble natural languages such as French or English. In terms of human senses, linguistic constructions are generally conveyed visually as text, although they can (and are) also sometimes conveyed by sound. A major advantage of linguistic construction interfaces is that they are nearly universal among people; a disadvantage is their imprecision.

B. Mathematical: The precision and rigor of mathematical and logical reasoning make this style of human thought especially appropriate for conveying human intent accurately into computers, as well as for verifying the completeness and accuracy of a construction. Unfortunately, mathematical reasoning is not nearly as universal a skill as natural language, since it requires both innate skills that are not as universal as language skills, and also many years of training and practice to use efficiently and accurately. It can also be argued that certain aspects of good mathematical reasoning, such as the ability to realize all the implications of a new assertion on all parts of a system, cannot be learned by some people no matter how much training they receive. On the other hand, mathematical reasoning styles are often notorious for focusing on a problem so intently that all “complications” are discarded and only a very small, very pristine subset of the overall problem is actually addressed. This kind of excessively narrow focus at the expense of any complicating issues can be disastrous in software construction, since it can lead to software that is incapable of dealing with the unavoidable complexities of nearly any usable system.

C. Visual: Another very powerful and much more universal construction interface style is *visual*, in the sense of the ability to use the same very sophisticated and necessarily natural ability to "navigate" a complex three-dimensional world of images, as perceived primarily through the eye (but also through tactile senses). The visual interface is powerful not only as a way of organizing information for presentation to a human, but also as a way of conceiving and navigating the overall design of a complex software system. Visual methods are particularly important for systems that require many people to work on them – that is, for organizing a software design process – since they allow a natural way for people to "understand" how and where they must communicate with each other. Visual methods are also deeply important for single-person software construction methods, since they provide ways both to present options to people and to make key details of a large body of information "pop out" to the visual system.

Construction languages seldom rely solely on a single style of construction. Linguistic and mathematical style in particular are both heavily used in most traditional computer languages, and visual styles and models are a major part of how to make software constructions manageable and understandable in computer languages. Relatively new "visual" construction languages such as Visual Basic and Visual Java provide examples that intimately combine all three styles, with complex visual interfaces often constructed entirely through non-textual interactions with the software constructor. Data processing functionality behind the interfaces can then be constructed using more traditional linguistic and mathematical styles within the same construction language.

Principles of Organization

In addition to the three basic human-oriented styles of interfacing to computers, there are four *principles of organization* that strongly affect the way software construction is performed. These principles are:

1. Reduction of Complexity: This principle of organization reflects the relatively limited (when compared to computers) ability of people to work with complex systems with many parts or interactions.

2. Anticipation of Diversity: The motive behind this principle is simple: *There is no such thing as an unchanging software construction.* Any truly useful software construction will change in various ways over time, and the *anticipation* of what those changes will be turns out to be one of the fundamental drivers of nearly every aspect of software construction.

3. Structuring for Validation No matter how carefully a person designs and implements software, the creative nature of non-trivial software design (that is, of software that is not simply a re-implementation of previously solved problems) means that mistakes and omissions will occur. *Structuring for validation* means building software in such a fashion that such errors and

omissions can be ferreted out more easily during unit testing and subsequent testing activities. One of the single most important implications of structuring for validation is that software must generally be *modular* in at least one of its major representation spaces, such as in the overall layout of the displayed or printed text of a program. This modularity allows both improved analysis and thorough unit-level testing of such components before they are integrated into higher levels in which their errors may be more difficult to identify. As a principle of construction, structuring for validation generally goes hand-in-hand with anticipation of diversity, since any errors found as a result of validation represent an important type of "diversity" that requires will require software changes (bug fixes).

4. Use of External Standards: A natural language that is spoken by one person would be of little value in communicating with the rest of the world. Similarly, a construction language that has meaning only within the software for which it was constructed can be a serious roadblock in the long-term use of that software. Such construction languages therefore should either conform to *external standards* such as those used for computer languages, or provide a sufficiently detailed internal "grammar" (e.g., documentation) by which the construction language can later be understood by others. The interplay between reusing external standards and creating new ones is a complex one, as it depends not only on the availability of such standards, but also on realistic assessments of the long-term viability of such external standards.

These principles of organization in software construction are discussed in more detail below.

Reduction in Complexity

Another major factor in how people convey intent to computers is the severely limited ability of people to "hold" complex structures and information in their working memory, especially over long periods of time. A human-to-computer construction that does not shield people from the nearly unlimited complexity and retention possible within a computer can easily overload the people working with it, potentially leading to serious inefficiencies and a proliferation of errors during the construction process. This need for simplicity in the human-to-computer interface leads to one of the strongest drivers in software construction: *reduction of complexity*. The need to reduce complexity applies to essentially every aspect of the software construction, and is particularly critical to the process of self-verification and self-testing of software constructions.

There are three main techniques for reducing complexity during software construction:

Removal of Complexity: Although trivial in concept, one obvious way to reduce complexity during software construction is to *remove* features or capabilities that are not absolutely required. This may or may not be the right way to handle a given situation, but certainly the general principle of parsimony – that is, of not adding capabilities that clearly will never be needed when constructing software – is valid.

Automation of Complexity: A much more powerful technique for removal of complexity is to *automate* the handling of it. That is, a new construction language is created which features that were previously time-consuming or error-prone for a human to perform are migrated over to the computer in the form of new software features or capabilities. The history of software is replete with examples of powerful software tools that raised the overall level of development capability of people by allowing them to address a new set of problems. Operating systems are one example of this principle, since they provide a rich construction language by which efficient use of underlying hardware resources can be greatly simplified. Visual construction languages similarly provide automation of visual aspects of software that otherwise could be very laborious to build.

Localization of Complexity: If complexity can neither be removed nor automated, the only remaining option is to *localize* complexity into small "units" or "modules" that are small enough for a person to understand in their entirety, and (perhaps more importantly) sufficiently *isolated* that meaningful assertions can be made about them. (A contrasting example: Arbitrarily dividing a very long sequence of code into small "modules" does not help, because the relationships between the modules become extremely complex and difficult to predict.)

Localization of complexity has a powerful impact on the design of computer languages, as demonstrated by the growth in popularity of object-oriented methods that seek to strictly limit the number of ways to interface to a software module. Localization is also a key aspect of good design of the broader category of construction languages, since new feature that are too hard to find and use are unlikely to be effective as tools for construction. Classical design admonitions such as the goal of having "cohesion" within modules and to minimize "coupling" are also fundamentally localization of complexity techniques, since they strive to make the number and interaction of parts within a module easy for a person to understand.

Anticipation of Diversity

This principle has more to do with how people use software than with differences between computers and people. As stated earlier, the main idea is quite simple: *There is no such thing as an unchanging software construction.* Useful software constructions are unavoidably part of a changing external environment in which they perform useful tasks, and changes in that outside environment trickle in to impact the software constructions in diverse (and often unexpected) ways. In contrast, mathematical constructions and formulae can in some sense be stable or unchanging over time, since they represent abstract quantities and relationships that do not require direct "attachment" to a working, physical computational machine. For example, even the software implementations of "universal" mathematical functions must change over time due to external factors such as the need to port them to new machines, and the unavoidable issue of physical limitations on the accuracy of the software on a given machine.

Anticipation of the diversity of ways in which software will change over time is one of the more subtle principles of software construction, yet it is vitally important for the creation of software that can endure over time and add value to future endeavors. Since it includes the ability to anticipate changes due to design errors (bugs) in software, it is also a fundamental part of the ability to make software robust and error-free. Indeed, one handy definition of "aging" software is that it is software that no longer has the flexibility to accommodate bug fixes without breaking.

Structuring for Validation

It is not particularly difficult to write software that cannot really be validated not matter how much it is tested. This is because even moderately large "useful" software components frequently cover such a large range of outputs that exhaustive testing of all possible outputs would take millennia or longer with even the fastest computers.

Structuring for validation thus becomes a fundamental constraint for producing software that can be shown to be acceptably reliable within a reasonable time frame. One of the most important implications of this principle is that software must be *modular* in some fashion that allows its behavior to be thoroughly analyzed and validated through testing before it becomes so complex that such validation is no longer feasible. The concept of *unit testing* parallels structuring for validation, and is used in parallel with the construction process to help ensure that validation occurs before the overall structure gets "out of hand" and can no longer be readily validated.

Use of External Standards

With the advent of the Internet as a major force in software development and interaction, the importance of selecting and using appropriate external standards for how to construct software is more apparent than ever before. Software that must share data and even working modules with other software anywhere in the world obviously must "share" many of the same languages and methods as that other software. The result is that selection and use of external standards – that is, of standards such as language specifications and data formats that were not originated within a software effort – is becoming an even more fundamental constraint on software construction than it was in the past. It is a complex issue, however, because the selection of an external standard may need to depend on such difficult-to-predict issues as the long-term economic viability of a

particular software company or organization that promotes that standard. Also, selecting one level of standardization often opens up an entire new set of standardization issues. An example of this is the data description language XML (eXtensible Markup Language). Selecting XML as an external standard answers many questions about how to describe data in an application, but it also opens up the issue of whether one of the growing numbers of customizations of XML to specific problem domains should also be used.

A Taxonomy of Software Construction Methods

The following taxonomy uses the above breakdowns of the software construction problem to suggest categories of tools and techniques needed for effective and well-rounded software engineering during the construction phase. Rather than being delayed until the construction phase itself, decisions about what styles of construction and specific methods will be used should be made early in the engineering process, since they will affect other phases as well.

A. Linguistic Construction Methods

Linguistic construction methods are distinguished in particular by the use of word-like strings of text to represent complex software constructions, and the combination of such word-like strings into patterns that have a sentence-like syntax. Properly used, each such string should have a strong semantic connotation that provides an immediate intuitive understanding of what will happen when the underlying software construction is executed. For example, the term “search” has an immediate, readily understandable semantic meaning in English, yet the underlying software implementation of such a term in software can be very complex indeed. The most powerful linguistic construction methods allow users to focus almost entirely on the language-like meanings of such term, as opposed (for example) to frittering away mental efforts on examining minor variations of what “search” means in a particular context.

Linguistic construction methods are further characterized by similar use of other “natural” language skills such as using patterns of words to build sentences, paragraphs, or even entire chapters to express software design “thoughts.” For example, a pattern such as “search table for out-of-range values” uses word-like text strings to imitate natural language verbs, nouns, prepositions, and adjectives. Just as having an underlying software structure that allows a more natural use of words reduces the number of issues that a user must address to create new software, an underlying software structure that also allows use of familiar higher-level patterns such as sentence further simplifies the expression process.

Finally, it should be noted that as the complexity of a software expression increases, linguistic construction methods begin to overlap unavoidably with visual methods that make it easier to locate and understand large sequences of statements. Thus just as most written versions of natural languages use visual clues such as spaces between words, paragraphs, and section headings to make text easier to “parse” visually, linguistic construction methods rely on methods such as precise indentation to convey structural information visually.

The use of linguistic construction methods is also limited by our inability to program computers to understand the levels of ambiguity typically found in natural languages, where many subtle issues of context and background can drastically influence interpretation. As a

result, the linguistic model of construction usually begins to weaken at the more complex levels of construction that correspond to entire paragraphs and chapters of text.

1. *Reduction in Complexity (Linguistic)*

   The main technique for reducing complexity in linguistic construction is to make short, semantically "intuitive" text strings and patterns of text stand in for the much more complex underlying software that "implement" the intuitive meanings. Techniques that reduce complexity in linguistic construction include:

   Functions, procedures, and code blocks

   Software templates

   Design patterns

   Component libraries and frameworks

   Interrupt and event-driven programming

   Data structures

   Encapsulation and abstract data types

   Higher-level and domain-specific languages

2. *Anticipation of Diversity (Linguistic)*

   Linguistic construction anticipates diversity both by permitting extensible definitions of "words," and also by supporting flexible "sentence structures" that allow many different types of intuitively understandable statements to be made with the available vocabulary. An excellent example of using linguistic construction to anticipate diversity is the use of human-readable configuration files to specify software or system settings.

   Information hiding

   Embedded documentation

   "Complete and sufficient" method sets

   Object-oriented class inheritance

   Creation of "glue languages" for linking legacy components

   Table-driven software

   Configuration files

   Self-describing software and hardware

3. *Structuring for Validation (Linguistic)*

   Because natural language in general is too ambiguous to allow safe interpretation of completely free-form statements, structuring for validation shows up primarily as rules that at least partially constrain the free use of natural the free use of expressions in software. The objective is to make such constructions as "natural" sounding as possible, while not losing the structure and precision needed to ensure consistent interpretations of the source code by both human users and computers.

   Modular design

   Structured programming

   Style guides

   Unit testing

4. *Use of External Standards (Linguistic)*

   Traditionally, standardization of programming languages was one of the first areas in which external standards appeared. The goal was (and is) to provide standard meanings and ways of using "words" in each standardized programming language, which makes it possible both for users to understand each other's software, and for the software to be interpreted consistently in diverse environments.

   Standardized programming languages

   Standardized data description languages (e.g., XML)

   Standardized alphabet representations (e.g., Unicode)

   Inter-process communication standards (e.g., COM, CORBA)

   Component-based software

B. Mathematical Construction Methods

Mathematical construction methods rely less on intuitive, everyday meanings of words and text strings, and more on definitions that are backed up by precise, unambiguous, and fully formal (mathematical) definitions. Mathematical construction methods are at the heart of most forms of system programming, where precision, speed, and verifiability are more important than ease of mapping into ordinary language. Mathematical constructions also use precisely defined ways of combining symbols that avoid the ambiguity of many natural language constructions. Functions are an obvious example of mathematical constructions, with their direct parallel to mathematical functions in both form and meaning.

Mathematical construction techniques also include the wide range of precisely defined methods for representing and implementing "unique" computer problems such as concurrent

and multi-threaded programming, which are in effect classes of mathematical problems that have special meaning and utility within computers.

The importance of the mathematical style of programming cannot be understated. Just as the precision of mathematics is fundamental to disciplines such as physics and the hard science, the mathematical style of programming is fundamental to building up a reliable framework of software "results" that will endure over time. While the linguistic and visual styles work well for interfacing with people, these less precise styles can be unsuitable for building the interior of a software system for the same reason that stained glass should not be used to build the supporting arches of a cathedral. Mathematical construction provides a foundation that can eliminate entire classes of errors or omissions from ever occurring, whereas linguistic and visual construction methods are much more likely to focus on isolated instances of errors or omissions. Indeed, one very real danger in software quality assurance is to focus too much on capturing isolated errors occurring in the linguistic or visual modes of construction, while overlooking the much more grievous (but harder to identify and understand) errors that occur in the mathematical style of construction.

1. *Reduction in Complexity (Mathematical)*

   As is the case with linguistic construction methods, mathematical construction methods reduce complexity by representing complex software constructions as simple text strings. The main difference is that in this case the text strings follow the more precisely defined rules and syntax of mathematical notations, rather than the "fuzzier" rules of natural language. Reading and writing such expressions and constructs them generally more training, but once mastered, the use of mathematical constructions tends to keep the ambiguity of what is being specified to an absolute minimum. However, as with linguistic construction, the quality of a mathematical construction is only as good as its underlying implementation. The advantage is that the precision of the mathematical definitions usually translates into a more precise specification for the software beneath it.

   Traditional functions and procedures

   Functional programming

   Logic programming

   Concurrent and real-time programming techniques

   Spreadsheets

   Mathematical libraries of functions

2. *Anticipation of Diversity (Mathematical)*

   Diversity in mathematical construction is handled in terms of precisely defined sets that can vary greatly in size. While mathematical formalizations are capable of very flexible representations of diversity, they require explicit anticipation and preparation for the full range of values that may be needed. A common problem in software construction is to

use a mathematical technique – e.g., a fixed-length vector or array – when what is really needed to accommodate future diversity is a more generic solution that anticipates future growth – e.g., an indefinitely variable-length vector. Since more generic solutions are often harder to implement and harder to make efficient, it is important when using mathematical construction techniques to try to anticipate the full range of future versions.

Functional parameterization

Macro parameterization

Extensible mathematical frameworks

3. *Structuring for Validation (Mathematical)*

Since mathematics in general is oriented towards proof of hypothesis from a set of axioms, mathematical construction techniques provide a broad range of techniques to help validate the acceptability of a software unit. Such methods can also be used to "instrument" programs to look for failures based on sets of preconditions.

Assertion-based programming (static and dynamic)

State machine logic

Redundant systems, self-diagnosis, and failover methods

Hot-spot analysis and performance tuning

4. *Use of External Standards*

For mathematical construction techniques, external standards generally address ways to define precise interfaces and communication methods between software systems and the machines they reside on.

POSIX standards

Data communication standards

Hardware interface standards

Standardized mathematical representation languages (e.g., MathML)

Mathematical libraries of functions

C. Visual Construction Methods

Visual construction methods rely much less on the text-oriented constructions of both linguistic and mathematical construction, and instead rely on direct visual interpretation and placement of visual entities (e.g., "widgets") that represent the underlying software. Visual construction tends to be somewhat limited by the difficulty of making "complex" statements

using only movement of visual entities on a display. However, it can also be a very powerful tool in cases where the primary programming task is simply to build and "tweak" a visual interface to a program whose detailed behavior was defined earlier.

Object-oriented languages are an interesting case. Although object-oriented languages use text and words to describe the detailed properties of objects, the style of reasoning that they encourage is highly visual. For example, experienced object-oriented programmers tend to view their designs literally as objects interacting in spaces of two or more dimensions, and a plethora of object-oriented design tools and techniques (e.g., Universal Mapping Language, or UML) actively encourage this highly visual style of reasoning.

However, object-oriented methods can also suffer from the lack of precision that is part of the more intuitive visual approach. For example, it is common for new – and sometimes not-so-new – programmers in object-oriented languages to define object classes that lack the mathematical precision that will allow them to work reliably over user-time (that is, long-term system support) and user-space (e.g., relocation to new environments). The visual intuitions that object-oriented languages provide in such cases can be somewhat misleading, because they can make the real problem of how to define a class to be efficient and stable over user-time and user-space seem to be simpler than it really is. A complete object-oriented construction model therefore must explicitly identify the need for mathematical construction methods throughout the object design process. The alternative can be an object-based system design that, like a complex stained glass window, looks impressive but is too fragile to be used in any but the most carefully designed circumstances.

More explicitly visual programming methods such as those found in Visual C++ and Visual Basic reduce the problem of how to make precise visual statements by "instrumenting" screen objects with complex (and mathematically precise) objects that lie behind the screen representations. However, this is done at a substantial loss of generality when compared to using C++ with explicit training in both visual and mathematical construction, since the screen objects are much more tightly constrained in properties.

1. *Reduction in Complexity (Visual)*

   Especially when compared to the steps needed to build a visual interface using text-oriented linguistic or mathematical construction, visual construction can provide drastic reductions in the total effort required to building a working graphical interface to a program. It can also reduce complexity by providing a simple way to select between the elements of a small set of choices.

   Object-oriented programming

   Visual creation and customization of user interfaces

   Visual (e.g., visual C++) programming

   "Style" aspects of structured programming

2. *Anticipation of Diversity (Visual)*

   Provided that the total sets of choices are not overly large, visual construction methods can provide a good way to configure or select options for software or a system. Visual construction methods are analogous to linguistic configuration files in this usage, since both provide easy ways to specify and interpret configuration information.

   Object classes

   Visual configuration specification

   Separation of GUI design and functionality implementation

3. *Structuring for Validation (Visual)*

   Visual construction can provide immediate, active validation of requests and attempted configurations when the visual constructs are "instrumented" to look for invalid feature combinations and warn users immediately of what the problem is.

   "Complete and sufficient" design of object-oriented class methods

   Dynamic validation of visual requests in visual languages

4. *Use of External Standards (Visual)*

   Standards for visual interfaces greatly ease the total burden on users by providing familiar, easily understood "look and feel" interfaces for those users.

   Object-oriented language standards

   Standardized visual interface models (e.g., Microsoft Windows)

   Standardized screen widgets

   Visual Markup Languages

References

1. Bentley, Jon, *Programming Pearls.* Addison-Wesley, 1989.

2. McConnell, Steve, *Code Complete*. Microsoft Press, 1993.

3. Henricson, Mats, and Nyquist, Erik, *Industrial Strength C++*. Prentice-Hall, 1997.

4. Brooks, Frederick P. Jr., *The Mythical Man-Month.* Addison-Wesley, 1995.

Very important note from the Knowledge Area specialist and the editors. Suggestions for book and article references are very much welcome for this Knowledge Area Description. Here is a reminder of the specifications regarding references:

1.1 Criteria and Requirements for selecting Reference Material

a) Specific reference material must be identified for each topic. Each piece of reference material can, of course, cover multiple topics.

b) Proposed Reference Material can be book chapters, refereed journal papers, refereed conference papers or refereed technical or industrial reports, or any other type of recognized artifact. They must be generally available and cannot be confidential in nature.

c) Proposed Reference Materials must be in English.

d) A maximum of 15 Reference Materials can be recommended for each Knowledge Area.

e) If deemed feasible and cost-effective by the IEEE Computer Society, selected reference material will be published on the Guide to the Software Engineering Body of Knowledge Web site. To facilitate this task, preference should be given to reference material for which the copyrights already belong to the IEEE Computer Society or to the ACM. This should not, however, be seen as a constraint or an obligation.

f) A matrix of reference material versus topics must be provided.

In order to facilitate your contribution to this section, here is table containing the breakdown of topics suggested above by the Knowledge Area Specialist. Please make your suggestions of reference material which cover one or more of the following topics, while respecting the above specifications.

We do not expect you to make reference material suggestions for each topic, especially if you don't agree with the breakdown or some part of it. If you suggest other topics, please include relevant material suggestions for these new topics.

Topics	Proposed reference material
A. Linguistic Construction Methods	
1. Reduction in Complexity (Linguistic)	
2. Anticipation of Diversity (Linguistic)	
3. Structuring for Validation (Linguistic)	
4. Use of External Standards (Linguistic)	
B. Mathematical Construction Methods	
1. Reduction in Complexity (Mathematical)	
2. Anticipation of Diversity (Mathematical)	
3. Structuring for Validation (Mathematical)	
4. Use of External Standards (Mathematical)	
C. Visual Construction Methods	
1. Reduction in Complexity (Visual)	
2. Anticipation of Diversity (Visual)	
3. Structuring for Validation (Visual)	
4. Use of External Standards (Visual)	