CHAPTER 4

SOFTWARE CONSTRUCTION

Terry Bollinger
The MITRE Corporation
1820 Dolley Madison Blvd.,
W534 McLean, VA, 22102, USA
terry@mitre.org

Philippe Gabrini, Louis Martin
Department of Computer Science
Université du Québec à Montréal
C.P. 8888, Succ. Centre-Ville
Montréal, Québec, H3C 3P8, Canada
{gabrini.philippe, martin.louis}@uqam.ca

Table of Contents

1. INTRODUCTION

Techniques of software construction are largely craft-based. As we come to understand the techniques better, we can explain them in terms of principles that can be explained as part of engineering knowledge. This description will therefore describe the underlying engineering principles in some detail and treat the specific craft-based techniques more briefly, usually just by naming them.

1.1. Annotated table of contents

This chapter is laid out as follows:

1. Introduction - This provides the road map to explain the overall structure of the chapter.

2. Definition - This defines Software Construction and provides links to other Knowledge Areas.

3. Principles of Organization - This explains the **first and most important method** chosen to break the subject matter into smaller sections, using four principles of software construction. The subject matter proper appears in section 5.

4. Styles of Construction - This explains a **second and less important method** chosen to break down the subject matter in each of section 5 into even smaller subsections, using three styles/methods of software construction.

5. Synthesis – This section contains 4 sub-sections, one for each of the four principles (the major dissection); each section contains 3 sub-sub-sections, one for each of the three styles of construction (the minor dissection).

6. Selected References

7. Additional References

8. Standards

9. References to Justify this Knowledge Area

10. Matrix of Reference Material versus Topics

2. DEFINITION OF THE SOFTWARE CONSTRUCTION KNOWLEDGE AREA

The Guide to the Swebok places the chapter on Construction after the one on Design and before the one on Testing. This does not imply **either** that the design stage must be complete before construction starts **or** that the construction stage must be complete before testing starts. In some development styles – such as the classic waterfall - design, construction, and testing are meant to proceed in that order. In others – such as the spiral method - development proceeds in successive steps, where each step consists of a predefined quantity of design, construction, and testing.

An important part of software engineering is to make a rational choice of development style for a given software project.

Software construction is linked to all other KAs, perhaps most strongly to Design, and Testing. This is because the construction process consumes the output of the Design

process (KA3) and itself provides one of the inputs to the Testing process (KA5).

Software construction is a fundamental act of software engineering: the construction of working, meaningful software through a combination of coding, validation, and testing (unit testing) by a programmer. Far from being a simple mechanistic "translation" of good design into working software, software construction burrows deeply into difficult issues of software engineering. It requires the establishment of a meaningful dialog[1] 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 developers be logical, precise, and thorough so that their intentions can be accurately captured and understood by the computer. The relationship works only because each side possesses certain capabilities that the other lacks. In the symbiosis that is software construction, the computer provides astonishing reliability, retention, and (once the need has been explained) speed of performance. Meanwhile, the human being provides 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.

2.1. Software Construction and Software Design

Software construction is closely related to software design (see *Knowledge Area Description for Software Design*). *Software design* analyzes software requirements in order to produce a description of the internal structure and organization of a system that will serve as a basis for its construction. Software design methods are used to express a global solution as a set of smaller solutions and can be applied repeatedly until the resulting parts of the solution are small enough to be handled with confidence by a single developer. It is at this point – that is, when the design process has broken the larger problem up into easier-to-handle chunks – that *software construction* is generally understood to begin. This definition also recognizes the distinction that while software construction necessarily produces executable software, software design does not necessarily produce any executable products at all.

In practice, however, the boundary between design and construction is seldom so clearly defined. Firstly, software construction is influenced by the scale or size of the software product being constructed. Very small projects in which the design problems are already "construction size" may neither require nor need an explicit design phase, and very large projects may require a much more interactive relationship between design and construction as different prototyping alternatives are proposed, tested, and discarded or used. Secondly, many of the techniques of software design also apply to software construction, since dividing problems into smaller parts is just as much a part of construction as it is design. Thirdly, effective design techniques always contain some degree of guessing or approximation in how they define their sub-problems. A few of the resulting approximations will turn out to be wrong, and will require corrective actions during software construction. (While another seemingly obvious solution would be to remove guessing and approximation altogether from design methods, that would contradict the premise that the original problem was too large and complex to be solved in one step. Effective design techniques instead acknowledge risk, work to reduce it, and help make sure that effective alternatives will be available when some choices eventually prove wrong.)

Design and construction both require sophisticated problem solving skills, although the two activities have somewhat different emphases. In design the emphasis is on how to partition a complex problem effectively, while in construction the emphasis is on finding a complete and executable solution to a problem. When software construction techniques do become so well-defined that they can be applied mechanistically, the proper route for the software engineer is to automate those techniques and move on to new problems, ones whose answers are not so well defined. This trend toward automation of well-defined tasks began with the first assemblers and compilers, and it has continued unabated as new generations of tools and computers have made increasingly powerful levels of construction automation possible. Projects that do contain highly repetitive, mechanistic software construction steps should examine their designs, processes, and tools sets more closely for ways to automate such needlessly repetitive steps out of existence.

2.2. The Role of Tools in Construction

In software engineering, a tool is a hardware or software device that is used to support performing a process. An effective tool is one that provides significant improvements in productivity and/or quality. This is a very inclusive definition, however, since it encompasses general-purpose hardware devices such as computers and peripherals that are part of an overall software-engineering environment. *Software construction tools* are a more specific category of tools that are both software-based and used primarily within the construction process. Common examples of software construction tools include compilers, version control systems, debuggers, code generators, specialized

---

[1] Some reviewers have commented that it is improper even to suggest that computers "understand programs" or "speak languages". However we prefer to retain the language of metaphor to illuminate the material; the reader will understand that such language is metaphorical as opposed to literal.

editors, tools for path and coverage analysis, test scaffolding and documentation tools.

The best software construction tools bridge the gap between methodical computer efficiency and forgetful human creativity. Such tools allow creative minds to express their thoughts easily, but also enforce an appropriate level of rigor. Good tools also improve software quality by allowing people to avoid repetitive or precise work for which a computer is better suited.

2.3. The Role of Integrated Evaluation in Construction

Another important theme of software engineering is the *evaluation* of software products. This includes such diverse activities as peer review of code and test plan, testing, software quality assurance, and measures[2] (see *Knowledge Area Description for Testing* and *Knowledge Area Description for Software Quality Analysis*). Integrated evaluation means that a process (in this case a development process) includes explicit continuous or periodic internal checks to ensure that it is still working correctly. These checks usually consist of evaluations of intermediate work products such as documents, designs, source code, or compiled modules, but they may also look at characteristics of the development process itself. Examples of product evaluations include design reviews, module compilations, and unit tests. An example of process-level evaluation would be periodic re-assessment of a code library to ensure its accuracy, completeness, and self-consistency.

Integrated evaluation in software engineering has yet to reach the stage achieved in hardware engineering where the evaluation is built into the components themselves, e.g. integrated self-test logic and built-in error recovery in complex integrated circuits. Such features were first added to integrated circuits when it was realized the circuits had become so complex that the assumption of perfect start-to-finish reliability was no longer tenable. As with integrated circuits, the purpose of integrated checking in software processes is to ensure that they can operate for long periods without generating nonsensical or hazardously misleading answers.

Historically, software construction has tended to be one of the software engineering steps in which developers were particularly prone to omitting checks on the process. While nearly all developers practice some degree of informal evaluation when constructing software, it is all too common for them to skip needed evaluation steps because they are too confident about the reliability and quality of their own software constructions. Nonetheless, a wide range of automated, semi-automated, and manual evaluation methods have been developed for use in the software construction phase.

The simplest and best-known form of software construction evaluation is the use of unit testing after completion of each well-defined software unit. Automated techniques such as compile-time checks and run-time checks help verify the basic integrity of software units, and manual techniques such as code reviews can be used to search for more abstract classes of errors. Tools for extracting measurements of code quality and structure can also be used during construction, although such measurement tools are more commonly applied during integration of large suites of software units. When collecting measurements, it is important that the measurements collected be relevant to the goals of the development process.

2.4. The Role of Standards in Construction

All forms of successful communication require a common language. *Standards* are in many ways best understood as agreements by which both concepts and technologies can become part of the shared "language" of a broader community of users. In many cases, standards are selected by a customer or by an organization. Project managers should consider the use of additional standards selected to be suitable to the specific characteristics of the project.

Software construction is particularly sensitive to the selection of standards, which directly affects such construction-critical issues as programming languages, databases, communication methods, platforms, and tools. Although such choices are often made before construction begins, it is important that the overall software development process take the needs of construction into account when standards are selected.

2.5. Manual and Automated Construction/The Spectrum of Construction Techniques

Manual Construction

Manual construction means solving complex problems in a language that a computer can execute. Practitioners of manual construction need a rich mix of skills that includes the ability to break complex problems down into smaller parts, a disciplined formal-proof-like approach to problem analysis, and the ability to "forecast" how constructions will change over time. Expert manual constructors sometimes use the skills of advanced logicians; they always need to apply the skills they have within a complex, changing environment such as a computer or network.

It would be easy to directly equate manual construction to coding in a programming language, but it would also be an incomplete definition. An effective manual construction process should result in code that fully and correctly processes data for its entire problem space, anticipates and handles all plausible (and some implausible) classes of errors, runs efficiently, and is structured to be resilient and easy-to-change over time. An inadequate manual construction process will in contrast result in code like an

---

[2] The word metrics is commonly used by software developers to denote the activity that practitioners in other branches of engineering refer to as measurement.

amateurish painting, with critical details missing and the entire construction stitched together poorly.

Automated Construction

While no form of software construction can be fully automated, much or all of the overall coordination of the software construction process can be moved from people to the computer – that is, overall control of the construction process can be largely automated. *Automated construction* thus refers to software construction in which an automated tool or environment is primarily responsible for overall coordination of the software construction process. This removal of overall process control can have a large impact on the complexity of the software construction process, since it allows human contributions to be divided up into much smaller, less complex "chunks" that require different problem solving skills to solve. Automated construction is also reuse-intensive construction, since by limiting human options it allows the controlling software to make more effective use of its existing store of effective software problem solutions. Of course, automated construction is not necessarily low cost; sometimes the cost of setting up the machinery is higher than the cost saved in its use.

In its most extreme form, automated construction consists of two related but distinct activities: (1) configuring a baseline system, which means configuring a predefined set of options that provide a workable solution in a typical business context and (2) implementing exceptions in the context of the product's usage. This may include resetting parameters, constructing additional software chunks, building interfaces, and moving data from existing legacy systems and other data sources to the new system. For example, an accounting application for small businesses might lead users through a series of questions that will result in a customized installation of the application. When compared to using manual construction for the same type of problem, this form of automated construction "swallows" huge chunks of the overall software engineering process and replaces them with automated selections that are controlled by the computer. Toolkits provide a less extreme example in which developers still have a great deal of control over the construction process, but that process has been greatly constrained and simplified by the use of predefined components with well-defined relationships to each other.

Automated construction is necessarily tool-intensive construction, since the objective is to move as much of the overall software development process as possible away from the human developer and into automated processes. Automated construction tools tend to take the form of program generators and fully integrated environments that can more easily provide automated control of the construction process. To be effective in coordinating activities, automated construction tools also need to have easy, intuitive interfaces.

Moving Towards Automation

An important goal of software engineering is to move construction continually towards higher levels of automation. That is, when selection from a simple set of options is all that is really required to make software work for a business or system, then the goal of software engineers should continually be to make their systems come as close to that level of simplicity as possible. This not only makes software more accessible, but also makes it safer and more reliable by removing opportunities for error.

The concept of moving towards higher levels of construction automation permeates nearly every aspect of software construction. When simple selections from a list of options will not suffice, software engineers often can still develop application specific tool kits (that is, sets of reusable parts designed to work with each other easily) to provide a somewhat lesser level of control. Even fully manual construction reflects the theme of automation, since many coding techniques and good programming practices are intended to make code modification easier and more automated. For example, even a concept as simple as defining 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.

2.6. Construction Languages

Construction languages include all forms of communication by which a human can specify an executable problem solution to a computer. The simplest type of construction language is a *configuration language,* in which developers choose from a limited set of predefined options to create new or custom installations of software. The text-based configuration files used in both Windows and Unix operating systems are examples, and the menu-style selection lists of some program generators are another. *Toolkit languages* are used to build applications out of toolkits (integrated sets of application-specific reusable parts), and are more complex than configuration languages. Toolkit languages may be explicitly defined as application programming languages (e.g., scripts), or may simply be implied by the collected set of interfaces of a toolkit. As described below, *programming languages* are the most flexible type of construction languages, but they also contain the least information about both application areas and development processes, and so require the most training and skill to use effectively.

2.7. Programming Languages

Since the fundamental task of software construction is to communicate intent unambiguously between two very

different types of entities (people and computers), the interface between the two is most commonly expressed as languages. Programming languages are more literal than 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 in a natural language. As will be discussed below, programming languages can also borrow from other non-linguistic human skills such as spatial visualization. The particular requirements of an application domain can give rise to the development or use of a specialized, *domain-specific language* such as lex, yacc, PHP, TCL, or TK.

Programming languages are often created in response to the needs of particular application fields, but the quest for more universal or encompassing programming language is ongoing. As in many relatively young disciplines, such quests for universality are as likely to lead to short-lived fads as they are to genuine insights into the fundamentals of software construction. For this very reason, it is important that software construction not be tied too greatly on any programming language or programming methodology. Adherence to suitable programming language standards, and avoiding proprietary feature sets helps avoid language obsolescence.

3. BREAKDOWN OF TOPICS FOR SOFTWARE CONSTRUCTION[3]

3.1. Principles of Organization

The first and most important method of breaking the subject of software construction into smaller units is to recognize the four principles that most strongly affect the way in which software is constructed, namely

- Reduction of Complexity
- Anticipation of Diversity
- Structuring for Validation
- Use of External Standards

These are discussed below.

3.1.1. Reduction of Complexity

This principle of organization reflects the relatively limited ability of people to work with complex systems that have many parts or interactions. A 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. 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

---

[3] An alternate, more traditional, breakdown is presented in Appendix B.

complexity applies to essentially every aspect of the software construction, and is particularly critical to the process of self-verification and testing of software constructions.

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

3.1.1.1 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.

3.1.1.2 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 in 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 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 the construction of software that otherwise could be very laborious to build.

3.1.1.3 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. This might even lead to components that can be re-used. However, one must be careful, as 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 programming 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, even though that might end up making components more dependent. 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.

3.1.2. Anticipation of Diversity

This principle has more to do with how people use software than with differences between computers and people. Its motive is simple: *There is no such thing as an unchanging software construction.* Any useful software construction will change in various ways over time, and the *anticipation* of change drives nearly every aspect of 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, formal mathematical constructions and formulas 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 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 in software, it also helps 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.

There are three main techniques for anticipating change during software construction:

3.1.2.1 Generalization

It is very common for software construction to focus first on highly specific problems with limited, rather specific solutions. This is common because the more general cases often simply are not obvious in the early stages of analysis. Generalization is the process of recognizing how a few specific problem cases fit together as part of some broader framework of problems, and thus can be solved by a single overarching software construction in place of several isolated ones. Generalization of functionality is a distinctly mathematical concept, and not too surprisingly the best generalizations that are developed are often expressed in the language of mathematics. Good design is equally an aspect of generalization, however. For example, software constructions that use stacks to store data are almost always more generalized than similar solutions using arrays behaving as stacks, since fixed sizes immediately place artificial (and usually unnecessary) constraints on the range of problem sizes that the construction can solve.

Generalization anticipates diversity because it creates solutions to entire classes of problems that may not have even been recognized as existing before. Thus just as Newton's general theory of gravity made a small number of formulas applicable to a much broader range of physics problems, a good generalization to a number of discrete software problems often can lead to the easy solution of many other development problems. For example, developing an easily customizable graphics user interface could solve a very broad range of development problems that otherwise would have required individual, labor-intensive development of independent solutions.

Anticipating diversity by using generalization is effective only when the developer finds generalizations that actually correspond to the eventual uses of the software. Developers may have no particular interest (or time) to develop the necessary generalizations under the schedule pressures of typical commercial projects. Even when the time needed is available, it is easy to develop the wrong set of generalizations – that is, to create generalizations that make the software easier to change, but only in ways that prove not to correspond to what is really needed.

For these reasons, generalization is both safer and easier if it can be combined with the next technique of *experimentation.* Change experimentation makes generalization safer by capturing realistic data on which generalizations will be needed, and makes generalization easier by providing schedule-conscious projects with specific data on how generalizations can improve their products.

3.1.2.2 Experimentation

Experimentation means using early (sometimes very early) software constructions in as many different user contexts as possible, and as early in the development process as possible, for the explicit purpose of collecting data on how to generalize the construction. To experiment is to recognize how difficult it is to anticipate all the ways in which software constructions can change.

Obviously, experimentation is a process-level technique rather than a code-level technique, since its goal is to collect data to help guide code-level processes such as generalization. This means that it is constrained by whether the overall development process allows it to be used at the construction level. Construction-level experimentation is most likely to be found in projects that have incorporated experimentation into their overall development process. The Internet-based open source development process that Linus Torvalds used to create the Linux operating system is an example of a process that both allowed and encouraged construction-level use of experimentation. In Torvalds' approach, individual code constructions were very quickly incorporated into an overall product and then redistributed via the Internet, sometimes on the same day. This encouraged further use, experimentation, and updates to the individual constructions. Development environments and

languages that support the rapid prototyping style of development also encourage construction-level experimentation.

3.1.2.3 Localization

Localization means keeping anticipated changes as localized in a software construction as possible. It is actually a special case of the earlier principle of *localization of complexity,* since change is a particularly difficult class of complexity. A software construction that can be changed in a common way by making only one change at one location within the construction thus demonstrates good *locality* for that particular class of modifications.

Localization is very common in software construction, and often is used intuitively as the "right way" to construct software. Objects are one example of a localization technique, since good object designs localize implementation changes to within the object. An even simpler example is using compile-time constants to reduce the number of locations in a program that must be changed manually should the constant change. Layered architectures such as those used in communication protocols are yet another example of localization, since good layer designs keep changes from crossing layers.

3.1.3. Structuring for Validation

No matter how carefully a person designs and implements software, the creative nature of non-trivial software construction (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 important implication 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 will require software changes (bug fixes). It is not particularly difficult to write software that cannot really be validated no 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 eons with even the fastest computers. *Structuring for validation* thus becomes one important constraint for producing software that can be shown to be acceptably reliable within a reasonable time frame. 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.

3.1.4. 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 programming 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. 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 more important. This is a complex issue, however, because the selection of an external standard may need to take account of such difficult-to-predict issues as the long-term economic viability of a particular software company or organization that promotes that standard. Stability of the standard is especially important. 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 raises the issue of whether one of the several customizations of XML to specific problem domains should also be used.

Other examples of external standards include API standards such as mathematics libraries, POSIX and SQL. In addition there are standards such as ISO/IEC 9126 , IEEE Std 1061, and IEEE Std 982, which are used in both Design and Construction.

3.2. Styles of Construction

Section 3.1 explained four principles of organization. A second and less important method of breaking the subject of software construction into smaller units is to recognize three styles/methods of software construction, namely

*   Linguistic

- Formal
- Visual

The traditional hierarchical taxonomy places the items in a tree; each item appears in one place only. Such an approach is not suitable for the items used in software construction because some of the items naturally *belong* in more than one place. In the classification that follows, an individual construction method may appear in many different places, rather than in just one. The number of repetitions indicates its breadth of application, and hence its importance in software construction as a whole. Modularity is one example of a construction method that has such broad impacts.

A good construction language moves detailed, repetitive, or memory-intensive construction tasks away from people and into the computer, where such tasks 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 discussed in the subsections below.

Of course, construction languages seldom rely solely on a single style of construction. Linguistic and formal 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 programming languages. Relatively new "visual" construction languages such as Visual Basic and Visual Java provide examples that 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 formal styles within the same construction language.

3.2.1. Linguistic

Linguistic construction languages make statements of intent in the form of sentences that resemble natural languages such as English or French. 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 the imprecision of ordinary languages such a English, which makes it hard for people to express needs clearly with sufficient precision when using linguistic interfaces to computers. An example of this problem is the difficulty that most early students of computer science have learning the syntax of even fairly readable languages such as Pascal or Ada.

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.

3.2.2. Formal

The precision and rigor of formal 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, formal 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 formal 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, formal 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.

Formal 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 (or mathematical) definitions. Formal 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. Formal constructions also use precisely defined ways of combining symbols that avoid the ambiguity of many natural language constructions. Functions are an obvious example of formal constructions, with their direct parallel to mathematical functions in both form and meaning.

Formal 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 formal style of programming cannot be overstated. Just as the precision of mathematics is fundamental to disciplines such as physics and the hard science, the formal 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. Formal 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 formal style of construction.

3.2.3. 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 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.

Visual construction methods rely much less on the text-oriented constructions of both linguistic and formal 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 "adjust" a visual interface to a program whose detailed behavior was defined earlier.

Some argue that object-oriented languages belong in this section because 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., Unified Modeling Language, or UML) actively encourage this highly visual style of reasoning. Others argue that object-oriented languages are no more inherently visual than procedural ones. They remark that SA/SD is a popular visual notation for procedural systems.

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 formal 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 formal 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 formally 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 formal construction, since the screen objects are much more tightly constrained in properties.

3.3. Synthesis

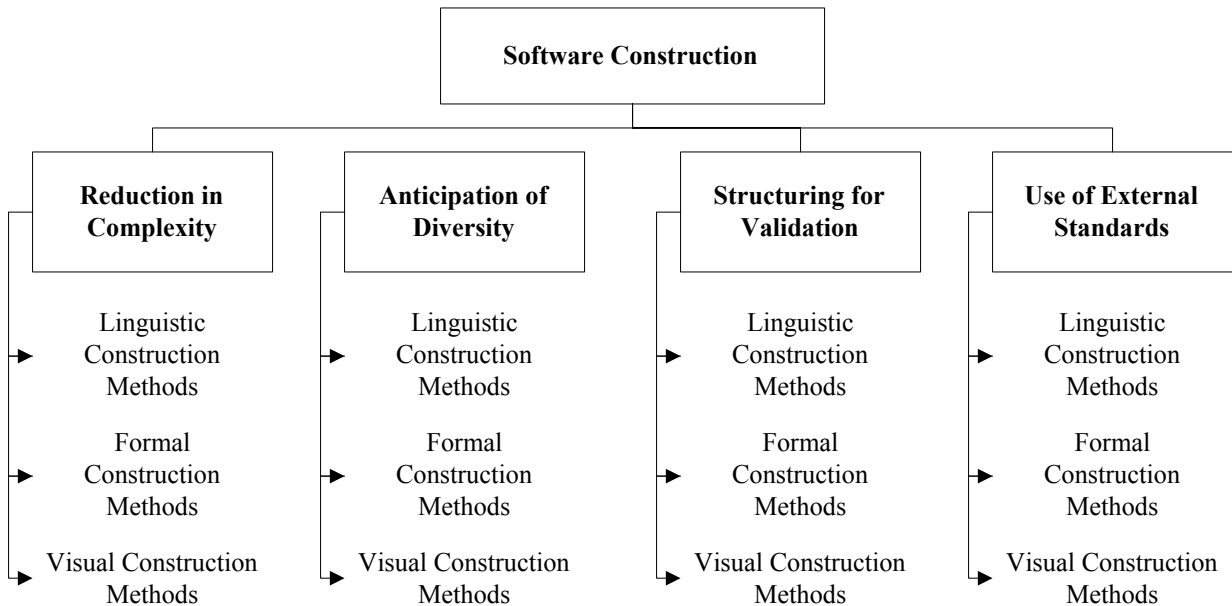The figure that follows combines the four principles of organization with the three styles of construction. Read the diagram by columns to see the principles, by rows to see the styles.

3.3.1. Reduction in Complexity

3.3.1.1 Linguistic Construction Methods

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:

- Design patterns
- Software templates

Software Construction			
Reduction in Complexity	**Anticipation of Diversity**	**Structuring for Validation**	**Use of External Standards**
Linguistic Construction Methods	Linguistic Construction Methods	Linguistic Construction Methods	Linguistic Construction Methods
Formal Construction Methods	Formal Construction Methods	Formal Construction Methods	Formal Construction Methods
Visual Construction Methods	Visual Construction Methods	Visual Construction Methods	Visual Construction Methods

- Functions, procedures, and code blocks
- Objects and data structures
- Encapsulation and abstract data types
- Objects
- Component libraries and frameworks
- Higher-level and domain-specific languages
- Physical organization of source code
- Files and libraries
- Formal inspections

3.3.1.2 Formal Construction Methods

As is the case with linguistic construction methods, formal 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 formal notations, rather than the "fuzzier" rules of natural language. The reading, writing, and construction of such expressions requires generally more training, but once mastered, the use of formal constructions tends to keep the ambiguity of what is being specified to an absolute minimum. However, as with linguistic construction, the quality of a formal construction is only as good as its underlying implementation. The advantage is that the precision of the formal 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
- Program generators
- Mathematical libraries of functions

3.3.1.3 Visual Construction Methods

Especially when compared to the steps needed to build a graphical interface to a program using text-oriented linguistic or formal construction, visual construction can provide drastic reductions in the total effort required. 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 programming (e.g., visual C++)
- "Style" (visual formatting) aspects of structured programming
- Integrated development environments supporting source browsing

3.3.2. Anticipation of Diversity

3.3.2.1 Linguistic Construction Methods

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. Techniques and methods that help anticipate diversity include:

- Information hiding
- Embedded documentation (commenting)
- "Complete and sufficient" method sets
- Object-oriented methods
- Creation of "glue languages" for linking legacy components
- Table-driven software
- Configuration files, internationalization
- Naming and coding styles
- Reuse and repositories
- Self-describing software and hardware (e.g., plug and play)

3.3.2.2 Formal Construction Methods

Diversity in formal 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 formal 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 indefinite variable-length vector. Since more generic solutions are often harder

to implement and harder to make efficient, it is important when using formal construction techniques to try to anticipate the full range of future versions.

- Functional parameterization
- Macro parameterization
- Generics
- Objects
- Error handling
- Extensible mathematical frameworks

3.3.2.3 Visual Construction Methods

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 (part of design)

3.3.3. Structuring for Validation

3.3.3.1 Linguistic Construction Methods

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 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
- Stepwise refinement

3.3.3.2 Formal Construction Methods

Since mathematics in general is oriented towards proof of hypothesis from a set of axioms, formal 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 fail-safe methods
- Hot-spot analysis and performance tuning

◆ Numerical analysis

3.3.3.3 *Visual Construction Methods*

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

3.3.4. External Standards

3.3.4.1 *Linguistic Construction Methods*

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 (e.g., Ada 95, C++, etc.)

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

◆ Standardized alphabet representations (e.g., Unicode)

◆ Standardized documentation (e.g., JavaDoc)

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

◆ Component-based software

◆ Foundation classes (e.g., MFC, JFC)

3.3.4.2 *Formal Construction Methods*

For formal 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

3.3.4.3 *Visual Construction Methods*

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 screen widgets

◆ Visual Markup Languages

4. MATRIX OF TOPICS VS. REFERENCE MATERIAL

Topics	Proposed reference material
Software Construction and Software Design	[GLA95] Part III, IV [MAZ96] Part IV [McCO93] Chap. 1, 2, 3
The Role of Tools in Construction	[HUN00] Chap. 3 [MAG93] Chap. 4 [MAZ96] Part IV [McCO93] Chap. 20
The Role of Integrated Evaluation in Construction	[HUM97] [MAG93] Chap. 8 [McCO93] Chap. 31, 32, 33
The Role of Standards in Construction	[IEEE]
Manual and Automated Construction / The Spectrum of Construction Techniques	[HUN00] Chap. 3
Construction Languages	[HUN00] Chap. 3 [SET96]
Programming Languages	[SET96]
A. Reduction in Complexity	
1. Reduction in Complexity (Linguistic)	[BEN00] Chap. 2, 3 [KER99] Chap. 2, 3 [McCO93] Chap. 4 to 19

Topics	Proposed reference material
2. Reduction in Complexity (Formal)	[BOO94] Part II and V [MAG93] Chap. 6 [MEY97] Chap. 6, 10
3. Reduction in Complexity (Visual)	[HOR99] Part II [WAR99] Chap. 1, 2, 3, 4, 5, 10
B. Anticipation of Diversity	
1. Anticipation of Diversity (Linguistic)	[BOO94] Part VI [McCO93] Chap. 30
2. Anticipation of Diversity (Formal)	[BEN00] Chap. 11, 13, 14 [KER99] Chap. 2, 9
3. Anticipation of Diversity (Visual)	[WAR99] Chap. 1, 2, 3, 4, 5, 10
C. Structuring for Validation	
1. Structuring for Validation (Linguistic)	[BEN00] Chap. 4 [KER99] Chap. 1, 5, 6 [MAG93] Chap. 2, 5, 7 [McCO93] Chap. 23, 24, 25, 26
2. Structuring for Validation (Formal)	[MAG93] Chap. 3 [MEY97] Chap. 6, 11
3. Structuring for Validation (Visual)	[HOR99] Part IV [MEY97] Chap. 11
D. Use of External Standards	
1. Use of External Standards (Linguistic)	http://www.xml.org/ http://www.omg.org/corba/beginners.html
2. Use of External Standards (Formal)	Object Constraint Language: http://www.omg.org/uml/
3. Use of External Standards (Visual)	http://www.omg.org/uml/

5. RECOMMENDED REFERENCES FOR SOFTWARE CONSTRUCTION

[BEN00] Bentley, Jon, Programming Pearls (Second Edition). Addison-Wesley, 2000. (Chapters 2, 3, 4, 11, 13 14)[BEN00] Bentley, Jon, Programming Pearls (Second Edition). Addison-Wesley, 2000. (Chapters 2, 3, 4, 11, 13 14)

[BOO94] Booch, Grady, and Bryan, Doug, Software Engineering with Ada (Third edition). Benjamin/Cummings, 1994. (Parts II, IV, V)[HOR99]

[KER99] Kernighan, Brian W., and Pike, Rob, The Practice of Programming. Addison-Wesley, 1999. (Chapters 1, 2, 3, 5, 6, 9)

[MAG93] Maguire, Steve, Writing Solid Code – Microsoft's Techniques for Developing Bug-Free C Software. Microsoft Press, 1993.

[McCO93] McConnell, Steve, Code Complete: A Practical Handbook of Software Construction. Microsoft Press, 1993.

[MEY97] Meyer, Bertrand, Object-Oriented Software Construction (Second Edition). Prentice-Hall, 1997. (Chapters 6, 10, 11)

[SET96] Sethi, Ravi, Programming Languages – Concepts & Constructs (Second Edition). Addison-Wesley, 1996. (Parts II, III, IV, V)

[WAR99] Warren, Nigel, and Bishop, Philip, Java in Practice – Design Styles and Idioms for Effective Java. Addison-Wesley, 1999. (Chapters 1, 2, 3, 4, 5, 10)

APPENDIX A – LIST OF FURTHER READINGS

[BAR98] Barker, Thomas T., *Writing Software Documentation – A Task-Oriented Approach.* Allyn & Bacon, 1998.

[FOW99] Fowler, Martin, *Refactoring – Improving the Design of Existing Code.* Addison-Wesley, 1999.

[GLA95] Glass, Robert L., *Software Creativity.* Prentice-Hall, 1995.

[HEN97] Henricson, Mats, and Nyquist, Erik, *Industrial Strength C++.* Prentice-Hall, 1997.

[HOR99] Horrocks, Ian, *Constructing the User Interface with Statecharts.* Addison-Wesley, 1999.

[HUM97] Humphrey, Watts S., *Introduction to the Personal Software Process.* Addison-Wesley, 1997.

[HUN00] Hunt, Andrew, and Thomas, David, *The Pragmatic Programmer.* Addison-Wesley, 2000.

[MAZ96] Mazza, C., et al., *Software Engineering Guides.* Prentice-Hall, 1996. (Part IV)

Standards

IEEE Std 829-1983 (Reaff 1991), IEEE Standard for Software Test Documentation (ANSI)

IEEE Std 1008-1987 (Reaff 1993), IEEE Standard for Software Unit Testing (ANSI)

IEEE Std 1028-1988 (Reaff 1993), IEEE Standard for Software Reviews and Audits (ANSI)

IEEE Std 1063-1987 (Reaff 1993), IEEE Standard for Software User Documentation (ANSI)

ISO/IEC 12207: 1995 Information technology – Software Life Cycle Processes and IEEE/EIA 12207.0, 12207.1 and 12207.2 ISO/IEC 14674:1999 Information Technology – Software Maintenance

ISO/IEC 14674:1999 Information Technology – Software Maintenance

APPENDIX B – A PROPOSED ALTERNATE BREAKDOWN FOR A SOFTWARE CONSTRUCTION KNOWLEDGE AREA

1. Construction Planning
2. Code Design
3. Data Design and Management
4. Error Processing
5. Source Code Organization
6. Code Documentation
7. Construction Quality Assurance
8. System Integration and Deployment
9. Code Tuning
10. Construction Tools

Source: Adapted from Mc Connell, Steve, "Code Complete: A Practical Handbook of Software Construction," Microsoft Press, 1993.