CHAPTER 4

SOFTWARE CONSTRUCTION

ACRONYMS

OMG	Object Management Group
UML	Unified Modeling Language

INTRODUCTION

The term *software construction* refers to the detailed creation of working, meaningful software through a combination of coding, verification, unit testing, integration testing, and debugging.

The Software Construction Knowledge Area is linked to all the other KAs, but is most strongly linked to Software Design and Software Testing because the software construction process, itself, involves significant software design and test activity. The process also uses the design output and provides an input to testing (“design” and “testing” in this case referring to the activities, not the KAs). Detailed boundaries between design, construction, and testing (if any) will vary depending upon the software life cycle processes that are used in a project.

Although some detailed design may be performed prior to construction, much design work is performed within the construction activity itself. Thus, the Software Construction KA is closely linked to the Software Design KA.

Throughout construction, software engineers both unit-test and integration-test their work. Thus, the Software Construction KA is closely linked to the Software Testing KA as well.

Software construction typically produces the highest volume of configuration items that need to be managed in a software project (source files, content, test cases, and so on). Thus, the Software Construction KA is also closely linked to the Software Configuration Management KA.

While software quality is important in all the KAs, code is the ultimate deliverable of a software project, and thus the Software Quality KA is also closely linked to the Software Construction KA.

Since software construction heavily involves the use of knowledge of algorithms and of detailed coding practices, it is closely related to the Computing Foundations KA, which is concerned with the computer science foundations that support the design and construction of software products. It is also related to project management, insofar as the management of construction can present considerable challenges.

BREAKDOWN OF TOPICS FOR SOFTWARE CONSTRUCTION

The breakdown of the Software Construction KA is presented below, together with brief descriptions of the major topics associated with it. Appropriate references are also given for each of the topics. Figure 1 gives a graphical representation of the top-level decomposition of the breakdown for this KA.

1. Software Construction Fundamentals

Software construction fundamentals include:

- Minimizing complexity
- Anticipating change
- Constructing for verification
- Reuse
- Standards in construction

The first four concepts apply to design as well as to construction. The following sections define these concepts and describe how they apply to construction.

1.1. Minimizing Complexity

[3]

Most people are severely limited in their ability to hold complex structures and information in their working memories, especially over long periods of time. This proves to be a major factor influencing how people convey intent to computers and leads to one of the strongest drives in software construction: minimizing complexity. The need to reduce complexity applies to essentially every aspect of software construction and is particularly critical to the process of verification and testing of software constructions.

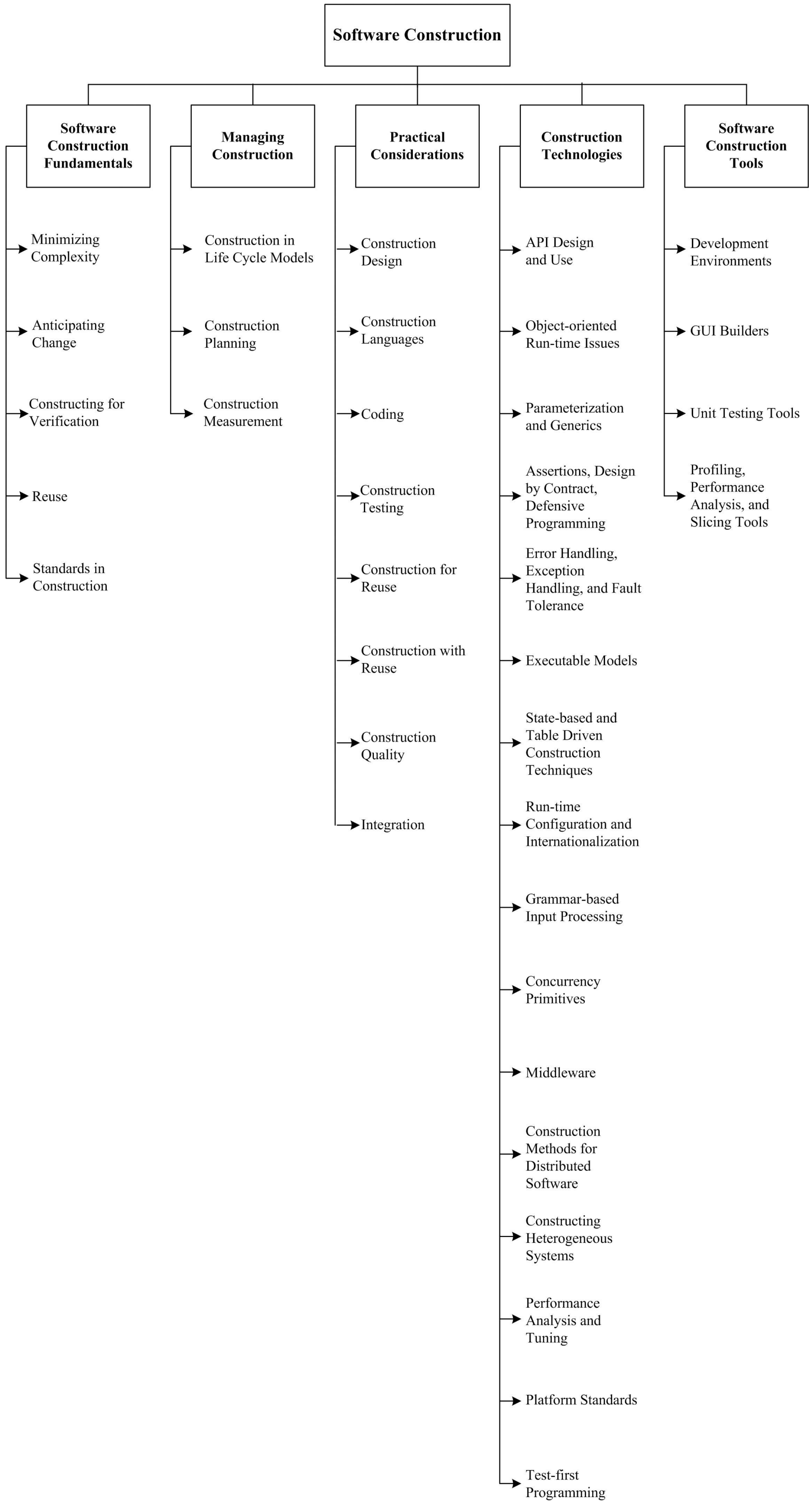

Figure 1. Breakdown of Topics for Software Construction

In software construction, reduced complexity is achieved through emphasizing code creation that is simple and readable rather than clever. It is accomplished through making use of standards (see 1.5), modular design (see 3.1), and numerous other specific techniques (see 3.3). It is also supported by the construction-focused quality techniques summarized in 3.7.

1.1. Anticipating Change
[3]

Most software will change over time, and the anticipation of change drives many aspects of software construction. Software is unavoidably a part of changing external environments; changes in those outside environments affect software in diverse ways.

Anticipating change helps engineers build extensible software, which means that they can enhance a software product without causing violence to the underlying structure.

Anticipating change is supported by many specific techniques, which are summarized in 3.3.

1.2. Constructing for Verification
[3]

Constructing for verification means building software in such a way that faults can be ferreted out readily by the software engineers writing the software as well as testers and users during independent testing and operational activities. Specific techniques that support constructing for verification include following coding standards to support code reviews, unit testing, organizing code to support automated testing, and restricting the use of complex or hard-to-understand language structures, among others.

1.3. Reuse
[7]

Reuse refers to using an asset in solving different problems. In software construction, typical assets that are reused include libraries, modules, components, source code, and commercial off-the-shelf (COTS) assets. Reuse should be practiced systematically, according to a well-defined, repeatable process. Systematic reuse can enable significant software productivity, quality, and cost improvements.

Reuse has two closely related facets: "construction for reuse" and "construction with reuse." The former means to create reusable software assets, while the latter means to reuse software assets in the construction of a new solution. Reuse often transcends the boundary of projects, which means reused assets can be constructed in other projects or organizations.

1.4. Standards in Construction
[3]

Applying external or internal development standards during construction helps achieve the project's objectives for development efficiency, quality, and cost. Specifically, the choices of allowable programming language subsets and usage standards are important aids in achieving higher security.

Standards that directly affect construction issues include:

- Communication methods (for example, standards for document formats and contents)
- Programming languages (for example, language standards for languages like Java and C++)
- Platforms (for example, programmer interface standards for operating system calls)
- Tools (for example, diagrammatic standards for notations like UML (Unified Modeling Language))

Use of external standards. Construction depends on the use of external standards for construction languages, construction tools, technical interfaces, and interactions between the Software Construction KA and other KAs. Standards come from numerous sources, including hardware and software interface specifications (such as the Object Management Group (OMG)) and international organizations (such as the IEEE or ISO).

Use of internal standards. Standards may also be created on an organizational basis at the corporate level or for use on specific projects. These standards support coordination of group activities, minimizing complexity, anticipating change, and constructing for verification.

2. Managing Construction

2.1. Construction in Life Cycle Models
[3]

Numerous models have been created to develop software; some emphasize construction more than others.

Some models are more linear from the construction point of view—such as the waterfall and staged-delivery life cycle models. These models treat construction as an activity that occurs only after significant prerequisite work has been completed—including detailed requirements work, extensive design work, and detailed planning. The more linear approaches tend to emphasize the activities that precede construction (requirements and design) and to create more distinct separations between activities. In these models, the main emphasis of construction may be coding.

Other models are more iterative—such as evolutionary prototyping, Extreme Programming, and Scrum. These approaches tend to treat construction as an activity that occurs concurrently with other software development activities (including requirements, design, and planning) or that overlaps them. These approaches tend to mix design, coding, and testing activities, and they often treat the combination of activities as construction.

Consequently, what is considered to be "construction" depends to some degree on the life cycle model used. In general, software construction is mostly coding and debugging, but it also involves construction planning, detailed design, unit testing, integrating testing, and other activities.

2.2. Construction Planning

[3]

The choice of construction method is a key aspect of the construction-planning activity. The choice of construction method affects the extent to which construction prerequisites are performed, the order in which they are performed, and the degree to which they are expected to be completed before construction work begins.

The approach to construction affects the project's ability to reduce complexity, anticipate change, and construct for verification. Each of these objectives may also be addressed at the process, requirements, and design levels—but they will also be influenced by the choice of construction method.

Construction planning also defines the order in which components are created and integrated, the integration strategy (for example, phased or incremental integration), the software quality management processes, the allocation of task assignments to specific software engineers, and other tasks, according to the chosen method.

2.3. Construction Measurement

[3]

Numerous construction activities and artifacts can be measured—including code developed, code modified, code reused, code destroyed, code complexity, code inspection statistics, fault-fix and fault-find rates, effort, and scheduling. These measurements can be useful for purposes of managing construction, ensuring quality during construction, and improving the construction process, as well as for other reasons. See the Software Engineering Process KA for more on measurements.

3. Practical Considerations

Construction is an activity in which the software has to come to terms with arbitrary and chaotic real-world constraints, and it must do so exactly. Due to its proximity to real-world constraints, construction is more driven by practical considerations than some other KAs, and software engineering is perhaps most craft-like in its construction area.

3.1. Construction Design

[3]

Some projects allocate more design activity to construction while others to a phase explicitly focused on design. Regardless of the exact allocation, some detailed design work will occur at the construction level, and that design work tends to be dictated by immovable constraints imposed by the real-world problem that is being addressed by the software.

Just as construction workers building a physical structure must make small-scale modifications to account for unanticipated gaps in the builder's plans, software construction workers must make modifications on a smaller or larger scale to flesh out details of the software design during construction.

The details of the design activity at the construction level are essentially the same as described in the Software Design KA, but they are applied on a smaller scale.

3.2. Construction Languages

[3]

Construction languages include all forms of communication by which a human can specify an executable problem solution to a computer. Construction languages and their implementations (for example, compilers) can affect software quality like performance, reliability, portability, and so forth. Especially, they can be serious contributors to security vulnerabilities.

The simplest type of construction language is a *configuration language,* in which software engineers choose from a limited set of predefined options to create new or custom software installations. The text-based configuration files used in both the Windows and Unix operating systems are examples of this, and the menu style selection lists of some program generators constitute 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 or may simply be implied by a toolkit's set of interfaces.

Scripting languages are commonly used kinds of application programming languages. In some software, scripts are called batch files or macros.

Programming languages are the most flexible type of construction languages. They also contain the least amount of information about specific application areas and development processes, and so they require the most training and skill to use effectively. The choice of programming language can have a large effect on the likelihood of vulnerabilities being introduced during coding—for example, uncritical usage of C and C++ are questionable choices from a security viewpoint.

There are three general kinds of notation used for programming languages, namely:

- Linguistic
- Formal
- Visual

Linguistic notations 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 providing an immediate intuitive understanding of what will happen when the underlying software construction is executed.

Formal notations rely less on intuitive, everyday meanings of words and text strings and more on definitions backed up by precise, unambiguous, and formal (or mathematical) definitions. Formal construction notations and formal methods are at the heart of most forms of system programming, where accuracy, time behavior, and testability are more important than ease of mapping into natural language. Formal constructions also use precisely defined ways of combining symbols that avoid the ambiguity of many natural language constructions.

Visual notations rely much less on the text-oriented notations of both linguistic and formal construction, and instead rely on direct visual interpretation and placement of visual entities 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 powerful tool in cases where the primary programming task is simply to build and "adjust" a visual interface to a program, the detailed behavior of which has been defined earlier.

3.3. Coding
[3]

The following considerations apply to the software construction coding activity:

- Techniques for creating understandable source code, including naming and source code layout
- Use of classes, enumerated types, variables, named constants, and other similar entities
- Use of control structures
- Handling of error conditions—both planned errors and exceptions (input of bad data, for example)
- Prevention of code-level security breaches (buffer overruns or array index overflows, for example)
- Resource usage via use of exclusion mechanisms and discipline in accessing serially reusable resources (including threads or database locks)
- Source code organization (into statements, routines, classes, packages, or other structures)
- Code documentation
- Code tuning

3.4. Construction Testing
[3]

Construction involves two forms of testing, which are often performed by the software engineer who wrote the code:

- Unit testing
- Integration testing

The purpose of construction testing is to reduce the gap between the time at which faults are inserted into the code and the time those faults are detected. In some cases, construction testing is performed after code has been written. In other cases, test cases may be created before code is written.

Construction testing typically involves a subset of types of testing, which are described in the Software Testing KA. For instance, construction testing does not typically include system testing, alpha testing, beta testing, stress testing, configuration testing, usability testing, or other more specialized kinds of testing.

Two standards have been published on the topic: IEEE Standard 829-1998*, IEEE Standard for Software Test Documentation,* and IEEE Standard 1008-1987, *IEEE Standard for Software Unit Testing*.

See also the corresponding subtopics in the Software Testing KA: 2.1.1 *Unit Testing* and 2.1.2 *Integration Testing* for more specialized reference material.

3.5. Construction for Reuse

[7]

Construction for reuse is to create reuse opportunities for the future or for other projects with a broad-based, multi-system perspective. It requires the developers to construct general software solutions with reusability. Construction activity for reuse usually is based on variability analysis and design. To avoid the problem of code clones, it is desired to encapsulate reusable code fragments into well designed libraries or components.

The tasks related to software construction for reuse during coding and testing are:

- Variability implementation with proper mechanisms like parameterization, conditional compilation, design patterns, etc.
- Variability encapsulation to make the software assets easy to configure and customize
- Testing the variability provided by the reusable software assets
- Description and publication of reusable software assets

3.6. Construction with Reuse

[7]

Construction with reuse means to create new software with the reuse of existing software assets. The most popular way of reuse is to reuse code from the libraries provided by the language, platform, or tools being used, or the company. And besides these, the applications developed today widely make use of many open-source libraries available all round the world. Reused and off-the-shelf software should meet the same quality requirements (for example, security level) as new software.

The tasks related to software construction with reuse during coding and testing are:

- The selection of the reusable units, databases, test procedures, or test data
- The evaluation of code or test reusability
- The integration of reusable software assets into the current software
- The reporting of reuse information on new code, test procedures, or test data

3.7. Construction Quality

[3]

In addition to faults resulting from requirements, design, poor choices, or use of construction languages, faults introduced during construction can bring serious quality problems—for example, security vulnerabilities. This includes not only faults in security functionality but also faults elsewhere that allow the bypassing of such functionality and other security weaknesses or violations.

Numerous techniques exist to ensure the quality of code as it is constructed. The primary techniques used for construction include:

- Unit testing and integration testing (see 3.4)
- Test-first development (see 2.2 of the Software Testing KA)
- Code stepping
- Use of assertions and defensive programming
- Debugging
- Technical reviews, including security-oriented reviews (see 2.3.2 of the Software Quality KA)
- Static analysis (IEEE1028) (see 2.3 of the Software Quality KA)

The specific technique or techniques selected depend on the nature of the software being constructed as well as on the skills set of the software engineers performing the construction. Especially, constructors/programmers need knowledge of good practices and common vulnerabilities— for example, from widely recognized lists about common vulnerabilities. Useful, automatic static analysis of code for security weaknesses is available for several common programming languages and should be used in security-critical projects.

Construction quality activities are differentiated from other quality activities by their focus. Construction quality activities focus on code and artifacts that are closely related to code—such as small-scale designs—as opposed to other artifacts that are less directly connected to the code, such as requirements, high-level designs, and plans.

3.8. Integration

[3]

A key activity during construction is the integration of separately constructed routines, classes, components, and subsystems into a single system. In addition, a particular software system may need to be integrated with other software or hardware systems.

Concerns related to construction integration include planning the sequence in which components will be integrated, creating scaffolding to support interim versions of the software, determining the degree of testing and quality work performed on components before they are integrated, and determining points in the project at which interim versions of the software are tested.

Programs can be integrated by means of either the phased or the incremental approach. Phased integration is also called “big bang” integration. Incremental integration is thought to offer many advantages over the traditional phased integration—for example, ease to locate errors, early project success, improved progress monitoring, and improved customer relations. In incremental integration, the developers write and test a program in small pieces and then combine the pieces one at a time. By building and integrating one unit (for example, a class or component) at a time, the construction process can provide early feedback to developers and customers. Other advantages of incremental integration include easier error location, improved progress monitoring, more fully tested units, and so forth.

4. Construction Technologies

4.1. API Design and Use

[1]

An application programming interface (API) is the set of signatures that are exported and available to the users of a library or a framework to write their applications. Besides signatures, an API usually involves statements about the program's effects and/or behaviors.

API design should try to make the API easy to learn and memorize, lead to readable code, be hard to misuse, be easy to extend, be complete, and keep backward compatibility. As the APIs usually outlast their implementations for a widely used library or framework, it is desired that the API be straightforward and kept stable to facilitate the development and maintenance of the client applications.

API use involves the process of selecting, learning, testing, integrating, and possibly extending APIs provided by a library or framework (see 3.6.).

4.2. Object-Oriented Run-Time Issues

[3]

Object-oriented languages support a series of runtime mechanisms like polymorphism and reflection. These runtime mechanisms increase the flexibility and openness of object-oriented programs. Polymorphism is the ability of a language to support general operations without knowing until run time what kind of concrete objects the software is dealing with. The program does not have to know the exact type of the object in advance, and so the exact behaviour is determined at run-time (called *dynamic binding*).

Reflection is the ability by which a program can observe and modify its own structure and behaviour at runtime. Reflection allows inspection of classes, interfaces, fields, and methods at runtime without knowing their names at compile time. It also allows instantiation of new objects and invocation of methods by parameterized class and method names at runtime.

4.3. Parameterization and Generics

[2]

Parameterized types, also known as generics (Ada, Eiffel) and templates (C++), enable the definition of a type or class without specifying all the other types it uses. The unspecified types are supplied as parameters at the point of use. Parameterized types provide a third way (in addition to class inheritance and object composition) to compose behaviours in object-oriented software.

4.4. Assertions, Design by Contract, Defensive Programming

[3]

An assertion is an executable predicate that's placed in a program—usually a routine or macro—that allows the program to check itself as it runs. Assertions are especially useful in high-reliability programs. They enable programmers to more quickly flush out mismatched interface assumptions, errors that creep in when code is modified, and so on. Assertions are normally compiled into the code at development time and are later compiled out of the code so that they don't degrade the performance.

Design by contract is a development approach in which each routine is considered to have pre-conditions and post-conditions. When pre-conditions and post-conditions are used, each routine or class forms a contract with the rest of the program. Assertions are a useful tool for documenting and verifying pre-conditions and post-conditions.

Defensive programming means to protect a routine from being broken by invalid inputs. Common ways to handle invalid inputs include checking the values of all the input parameters and deciding how to handle bad inputs. Assertions are often used for defensive programming to check input values.

4.5. Error Handling, Exception Handling, and Fault Tolerance

[3]

The way in which errors are handled affects the software's ability to meet requirements related to correctness, robustness, and other non-functional attributes. Assertions are usually used to handle errors that should never occur in the code. For other errors that may occur, other error handling techniques—like returning a neutral value, substituting the next piece of valid data, logging a warning message, returning an error code, or shutting down the software—may be used.

Exceptions are a specific means by which code can pass along errors or exceptional events to the code that called it. They can also be used to straighten out tangled logic within a single stretch of code. The basic structure of an exception is that a routine uses *throw* to throw an exception object, and code in some other routine higher up the calling hierarchy will *catch* the exception within a *try-catch* block. Exception handling policies should be carefully designed following common principles such as including in the exception message all information that led to the exception, avoiding empty catch blocks, knowing the exceptions the library code throws, considering building a centralized exception reporter, and standardizing the project's use of exceptions.

Fault tolerance is a collection of techniques that increase software reliability by detecting errors and then recovering from them if possible or containing their bad effects if not. The most common fault tolerance strategies include backing up and retrying, using auxiliary code, using voting algorithm, and replacing an erroneous value with a phony value with benign effect.

4.6. Executable Models

[4]

Executable models abstract away both specific programming languages and decisions about the organization of the software. Different from traditional software models, a specification built in an executable modeling language like xUML (executable UML) can be deployed in various software environments without change. An executable model compiler (transformer) can turn an executable model into an implementation using a set of decisions about the target hardware and software environment. Thus, constructing executable models can be regarded as a new way of constructing executable software.

Executable model is one foundation supporting the Model-Driven Architecture (MDA) initiative announced by the Object Management Group (OMG). An executable model is required as a way to specify a Platform-Independent Model (PIM) completely, which is a model of a solution to a problem that does not rely on any implementation technologies. Then a Platform-Specific Model (PSM), which is a model that contains within it the details of the implementation, can be produced by weaving together the PIM and the platforms on which it relies.

4.7. State-Based and Table-Driven Construction Techniques

[3]

State-based programming, or automata-based programming, is a programming technology using finite state machines to describe program behaviours. The transition graphs of a state machine are used in all stages of software development (specification, implementation, debugging, and documentation). The main idea is to construct computer programs the same way the automation of technological processes is done. State-based programming is usually combined with object-oriented programming, forming a new composite approach called *state-based, object-oriented programming.*

A table-driven method is a schema that allows one to look up information in a table rather than using logic statements (such as *if* and *case*) to figure it out. Used in appropriate circumstances, table-driven code is simpler than complicated logic, easier to modify, and more efficient. When using table-driven methods, the programmer should address two issues: how to look up entries in the table and what to store in the table.

4.8. Run-Time Configuration and Internationalization

[3]

To achieve more flexibility, a program is often constructed to support a late binding time of its variable values. Run-time configuration is a technique that binds variable values and program settings when the program is running, usually by updating and reading configuration files in a just-in-time mode.

Internationalization is the technical activity of preparing a program, usually for interactive software, to support multiple locales. The corresponding activity, *localization,* is the activity of translating a program to support a specific local language. Most interactive software contains dozens or hundreds of prompts, status displays, help messages, error messages, and so on. The design and construction processes should consider the typical string and character-set issues (including which character set is used and which kinds of strings are used), maintain the strings without changing code, and translate the strings into foreign languages with minimal impact on the code and the user interface.

4.9. Grammar-Based Input Processing (Parsing)

[3, 5]

Grammar-based input processing is a kind of syntax analysis, or parsing, of the input token stream. It involves the creation of a data structure (called a *parse tree* or *syntax tree*) representing the input data. The inorder traversal of a parse tree usually gives the expression just parsed. The parser checks the symbol table for the presence of programmer-defined variables that populate the tree. After input parsing, the program uses the parse tree as input in the following verification, computation, and processing.

4.10. Concurrency Primitives

[6]

A synchronization primitive is a programming abstraction with a programming interface that facilitates concurrency and synchronization. Well-known concurrency primitives include semaphores, monitors, and mutexes.

A semaphore is a protected variable or abstract data type that provides a simple but useful abstraction for controlling access by multiple processes or threads to a common resource in a concurrent programming environment.

A monitor is an abstract data type that presents a set of programmer-defined operations that are executed with mutual exclusion. A monitor contains the declaration of shared variables and procedures or functions that operate on those variables. The monitor construct ensures that only one process at a time is active within the monitor.

Mutex is a synchronization primitive that grants exclusive access to the shared resource to only one process or thread.

4.11. Middleware

[1, 5]

Middleware is a broad classification for software that provides services above the operating system layer yet below the application program layer. Middleware can provide runtime containers for software components, supporting an application with a transparent location across the network, message passing, persistence, and lifecycle managing. Middleware can be viewed as a connector between the components that use the middleware.

4.12. Construction Methods for Distributed Software

[6]

A distributed system is a collection of physically separate, possibly heterogeneous, computer systems that are networked to provide the users with access to the various resources that the system maintains. Construction of distributed software is distinguished from traditional software construction by issues like parallelism, communication, and fault tolerance.

Distributed programming typically falls into one of several basic architectures or categories: client–server, 3-tier architecture, n-tier architecture, distributed objects, loose coupling, or tight coupling.

4.13. Constructing Heterogeneous Systems

[5]

Heterogeneous systems consist of a variety of specialized computational units of different types, such as DSPs (Digital Signal Processing), micro-controllers, and peripheral processors. These computational units are independently controlled and communicate with each other. Embedded systems are typical heterogeneous systems.

The design of heterogeneous systems may require the combination of several specification languages in order to design different parts of the system—in other words, hardware/software co-design. The key issues include multi-language validation, co-simulation, and interfacing.

During the hardware/software co-design, software development and virtual hardware development proceed concurrently through stepwise decomposition. The hardware part is usually simulated in field programmable gate arrays (FPGAs) or application-specific integrated circuits (ASICs). The software part is translated into a low-level programming language.

4.14. Performance Analysis and Tuning

[3]

Code efficiency—together with program architecture, detailed design, and data-structure and algorithm selection—influences a program's performance, including both execution speed and size. Performance analysis is the investigation of a program's behaviour, using information gathered as the program executes, with the goal of identifying possible hot spots in the program to optimize.

Code tuning, which improves performance at the code level, is the practice of modifying correct code in ways that make it run more efficiently. Code tuning usually involves only small-scale changes that affect a single class, a single routine, or, more commonly, a few lines of code. A rich set of code tuning techniques is available, including those for tuning logic expressions, loops, data transformations, expressions, and routines. Using a low-level language is another commonly used technique for some hot spots of a program.

4.15. Platform Standards

[5, 6]

Platform standards enable the programmers to develop portable applications that can be executed in compatible environments without changes. Platform standards usually involve a set of standard services and APIs that compatible platform implementations must implement. Typical examples of platform standards are Java 2 Platform Enterprise Edition (J2EE) and the operating system standard POSIX (Portable Operating System Interface), which represents a set of standards implemented primarily for UNIX-based operating systems.

4.16. Test-First Programming

[3]

Test-first programming is a popular development style in which test cases are written prior to writing any code. Test-first programming can usually detect defects earlier and correct them more easily than traditional programming styles. Furthermore, writing test cases first forces programmers to think about requirements and design before coding, thus exposing requirements and design problems sooner.

5. Software Construction Tools

5.1. Development Environments

[3]

A development environment, or integrated development environment (IDE), is a software application that provides comprehensive facilities to programmers for software construction by integrating a series of development tools. The choices of development environments can affect the efficiency and quality of software construction.

In additional to basic code editing functions, good modern IDEs often offer other features like compilation and error detection from within the editor, integration with source-code control, build/test/debugging tools, compressed or outline views of programs, automated code transforms, and refactoring.

5.2. GUI Builders

[3]

A GUI (Graphical User Interface) builder, also known as GUI designer, is a software development tool that enables the developer to create and maintain GUIs in a WYSIWYG (what you see is what you get) mode. A GUI builder usually includes a visual editor for the developer to design forms, windows, and manage the layout of the widgets embedded by dragging, dropping, and parameter setting. The GUI builder can automatically generate the source code corresponding to the visual GUI design.

As current GUI applications usually follow the event-driven design style (in which the flow of the program is determined by events and event handling), GUI builders usually provide code generation assistants, which automate the most repetitive tasks required for event handling. The supporting code connects widgets with the outgoing and incoming events that trigger the functions providing the application logic.

Some modern IDEs provide integrated GUI builders or GUI builder plug-ins. There are also many standalone GUI builders.

5.3. Unit Testing Tools

[3, 7]

Unit testing verifies the functioning in isolation of software pieces (for example, classes, routines, components), which are separately testable. Unit testing is often automated. Developers can use unit testing tools and frameworks to extend and create automated testing environment. With unit testing tools and frameworks, the developer can code criteria into the test to verify the unit's correctness. Each individual test is implemented as an object, and a test runner runs all of the tests. In and after the test execution, those failed test cases will be automatically flagged and reported.

5.4. Profiling, Performance Analysis, and Slicing Tools

[3]

Performance analysis tools are usually used to support code-tuning decisions. The most common performance analysis tools are profiling tools. An execution profiling tool watches the code while it runs and tells how many times each statement is executed or how much time the program spends on each statement or execution path. Profiling the code while it is running gives insight into how the program works, where the hot spots are, and where the developers should focus the code-tuning efforts.

Program slicing is the computation of the set of program statements (i.e., the program slice) that may affect the values at some point of interest, which is referred to as a slicing criterion. Program slicing can be used for locating the source of errors, program understanding, and optimization analysis. Program slicing tools compute program slices by static or dynamic analysis methods for various programming languages.

RECOMMENDED REFERENCES FOR SOFTWARE CONSTRUCTION

1. P. Clements et al., *Documenting Software Architectures: Views and Beyond,* Boston: Addison-Wesley, 2002.
2. E. Gamma et al., *Design Patterns: Elements of Reusable Object-Oriented Software,* Reading, MA: Addison-Wesley Professional, 1994.
3. S. McConnell, *Code Complete*, Redmond, WA: Microsoft Press, 2004.
4. S.J. Mellor. and M.J. Balcer, *Executable UML: A Foundation for Model-Driven Architecture*, Boston: Addison-Wesley, 2002.
5. L. Null and J. Lobur, *The Essentials of Computer Organization and Architecture,* Sudbury, MA: Jones and Bartlett Publishers, 2006.
6. A. Silberschatz et al., *Operating System Concepts*, Hoboken, NJ: Wiley, 208.
7. I. Sommerville, *Software Engineering*, New York: Addison-Wesley, 2006.

Matrix of Topics vs. Reference Material

	(McConnell 2004)	(Sommerville 2006)	(Mellor and Balcer 2002)	(Clements, Bachmann et al. 2002)	(Gamma, Helm et al. 1994)	(Null and Lobur 2006)	(Silberschatz, Galvin et al. 2008)
1. Software Construction Fundamentals							
1.1 Minimizing Complexity	c2, c3, c7-c9, c24, c27, c28, c31, c32, c34						
1.2 Anticipating Change	c3-c5, c24, c31, c32, c34						
1.3 Constructing for Verification	c8, c20-c23, c31, c34						
1.4 Reuse		c18					
1.5 Standards in Construction	c4						
2. Managing Construction							
2.1 Construction in Life Cycle Models	c2, c3, c27, c29						
2.2 Construction Planning	c3, c4, c21, c27-c29						
2.3 Construction Measurement	c25, c28						
3. Practical Considerations							
3.1 Construction Design	c3, c5, c24						
3.2 Construction Languages	c4						
3.3 Coding	c5-c19, c25-c26						
3.4 Construction Testing	c22, c23						
3.5 Construction for Reuse		c18					
3.6 Construction with Reuse		c18					
3.7 Construction Quality	c8, c20-c25						
3.8 Integration	c29						
4. Construction Technologies							
4.1 API design and use				c7			
4.2 Object-oriented run-time issues	c6, c7						
4.3 Parameterization and generics					c1		
4.4 Assertions, design by contract, defensive programming	c8, c9						
4.5 Error handling, exception handling, and fault tolerance	c3, c8						
4.6 Executable Models			c1				
4.7 State-based and table driven construction techniques	c18						
4.8 Run-time configuration and internationalization	c3, c10						
4.9 Grammar-based input processing	c5					c8	
4.10 Concurrency primitives							c6
4.11 Middleware				c1		c8	
4.12 Construction methods for distributed software							c2
4.13 Constructing heterogeneous systems						c9	
4.14 Performance analysis and tuning	c25, c26						

	(McConnell 2004)	(Sommerville 2006)	(Mellor and Balcer 2002)	(Clements, Bachmann et al. 2002)	(Gamma, Helm et al. 1994)	(Null and Lobur 2006)	(Silberschatz, Galvin et al. 2008)
4.15 Platform standards						c10	c1
4.16 Test-first programming	c22						
5. Construction Tools							
5.1 Development environments	c30						
5.2 GUI builders	c30						
5.3 Unit testing tools	c22	c23					
5.4 Profiling, performance analysis and slicing tools	c25, c26						

Clements, P., F. Bachmann, et al. (2002). *Documenting Software Architectures: Views and Beyond*. Boston, Addison-Wesley.

Gamma, E., R. Helm, et al. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA, Addison-Wesley Professional.

McConnell, S. (2004). *Code Complete*. Redmond, WA, Microsoft Press.

Mellor, S. J. and M. J. Balcer (2002). *Executable UML: A Foundation for Model-Driven Architecture*. Boston, Addison-Wesley.

Null, L. and J. Lobur (2006). *The Essentials of Computer Organization and Architecture*. Sudbury, MA, Jones and Bartlett Publishers.

Silberschatz, A., P. B. Galvin, et al. (2008). *Operating System Concepts*. Hoboken, NJ, Wiley.

Sommerville, I. (2006). *Software Engineering*. New York, Addison-Wesley.

FURTHER READINGS

T.T. Barker, *Writing Software Documentation: A Task-Oriented Approach*, Allyn & Bacon, 1998.

K. Beck, *Test-Driven Development: By Example*, Addison-Wesley, 2002.

M. Fowler et al., *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.

M. Howard and D.C. Leblanc, *Writing Secure Code*, Microsoft Press, 2002.

W.S. Humphrey, *Introduction to the Personal Software Process*, Addison-Wesley, 1997.

B. Meyer, *Object-Oriented Software Construction*, 2nd ed., Prentice Hall, 1997, Ch. 6, 10, 11.

R. Sethi, *Programming Languages: Concepts & Constructs*, 2nd ed., Addison-Wesley, 1996, Parts II–V.

Common Weakness Enumeration. www.cwe.mitre.org.