LECTURE 1: INTRODUCTION TO SOFTWARE CONSTRUCTION

Dr. Ehab Essa
Computer Science Department,
Mansoura University

COURSE SYLLABUS (TENTATIVE)

Lecture1: Introduction to Software Construction

Lecture2: Static Analysis & Mutability and Immutability

Lecture3: Code Review and Testing

Lecture4: Debugging & Exceptions

Lecture5: Designing Specifications & Formal Methods

Lecture6: AI for Software Construction

Lecture7: Traditional Machine learning 1

Lecture8: Traditional Machine learning 2

Lecture9: Deep learning 1

Lecture10: Deep learning 2

TODAY

Introduction to Software Construction

- Software Construction Fundamental
- Margining Software Construction
- Practical Considerations

INTRODUCTION

Software Construction refers to the detailed creation of working software through a combination of coding, verification, unit testing, integration testing, and debugging.

Software Construction knowledge area (KA) is linked to all the other KAs, but it is most strongly linked to **Software Design** and **Software Testing**. The process uses the design output and provides an input to testing.

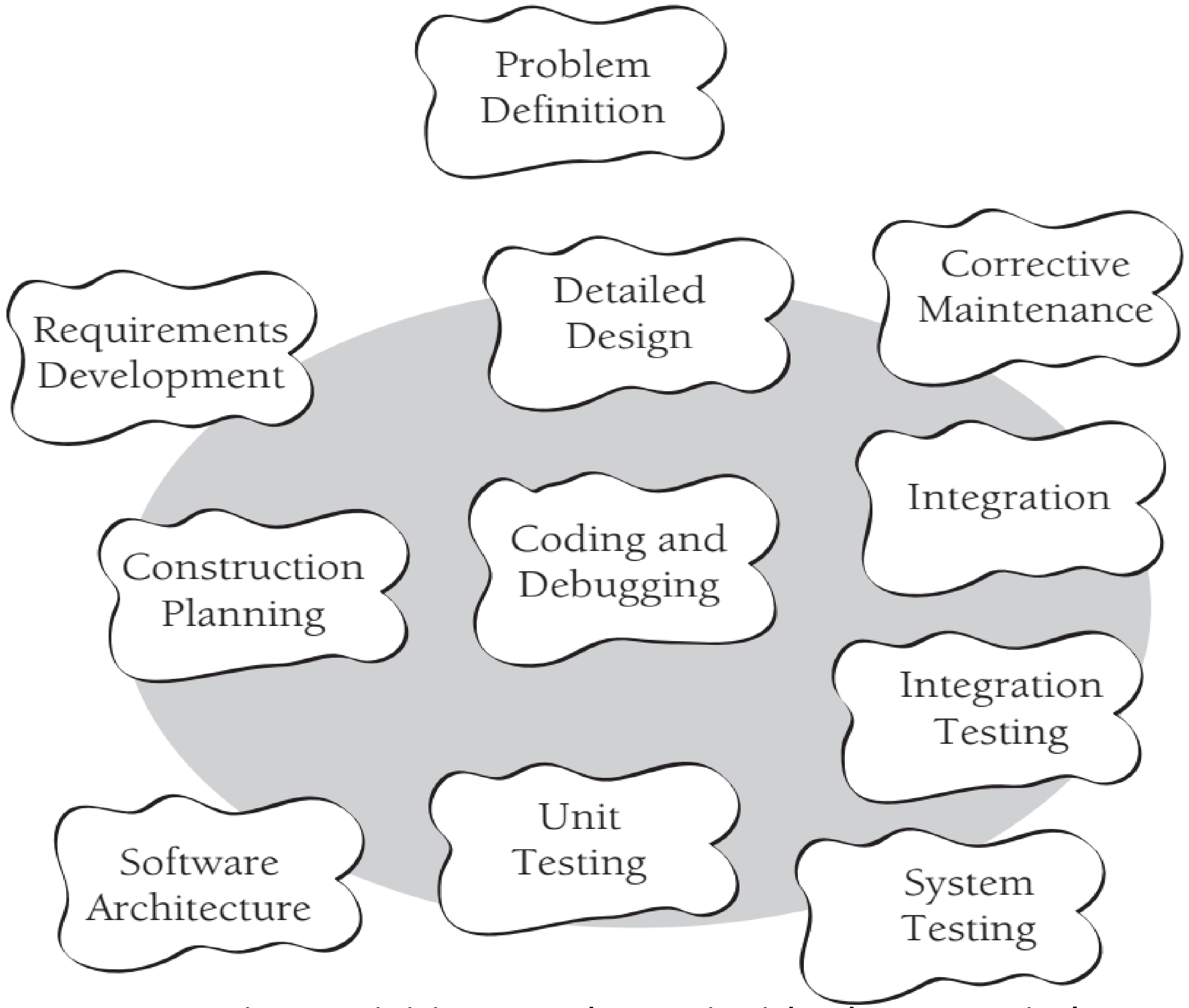
- Although some detailed design may be performed prior to construction, much design work is performed during the construction activity.
- Throughout construction, software engineers both unit test and integration test their work.

Software construction typically produces the highest number of configuration items that need to be managed in a software project (source files, documentation, 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 closely linked to the Software Construction KA.

Since software construction requires knowledge of algorithms and of coding practices, it is closely related to the **Computing Foundations** KA.

SOFTWARE CONSTRUCTION



Construction activities are shown inside the grey circle

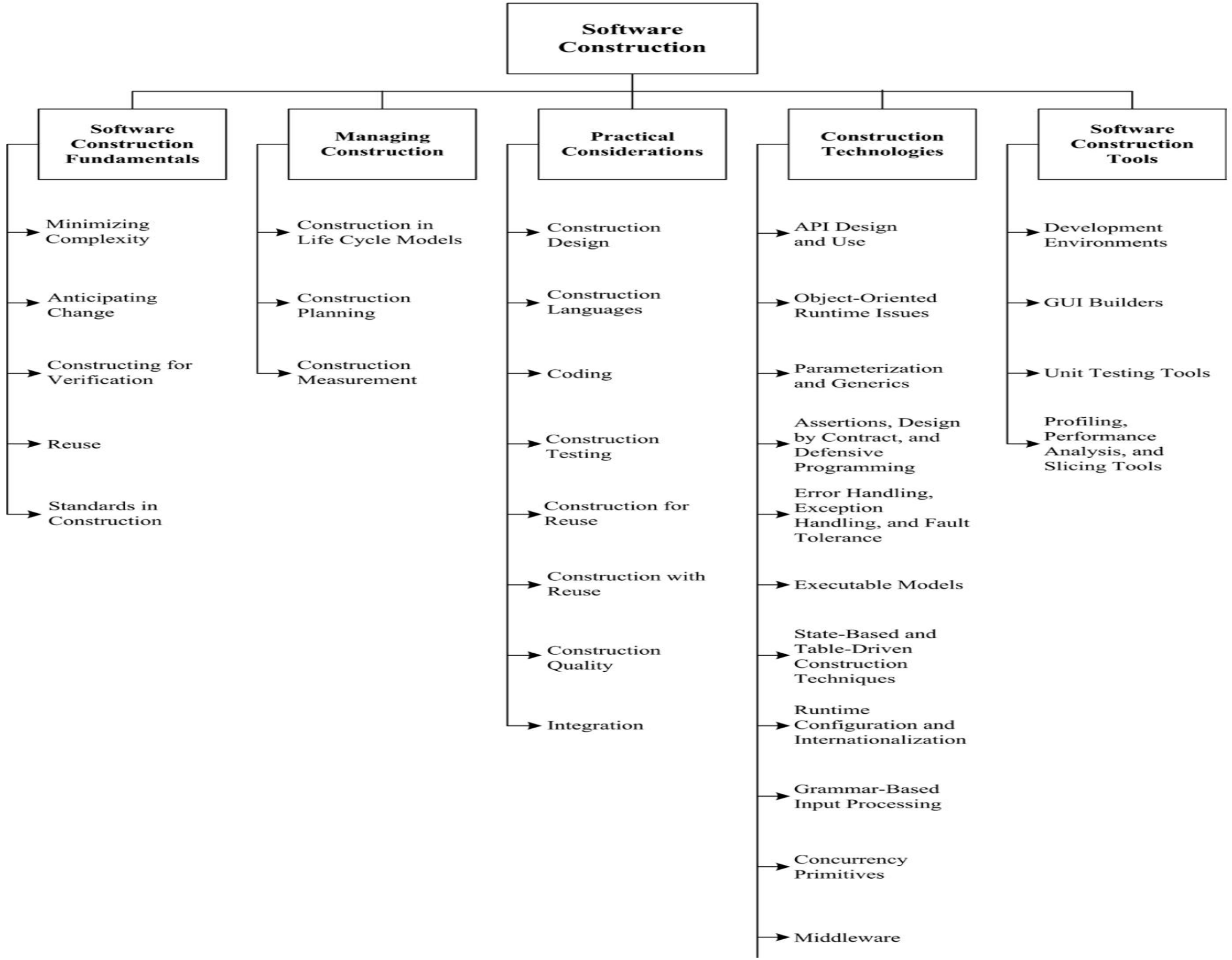
THE GOALS OF SOFTWARE CONSTRUCTION

The primary goal is learning how to produce software that is:

Safe from bugs. Correctness (correct behavior right now) and defensiveness (correct behavior in the future).

Easy to understand. Has to communicate to future programmers who need to understand it and make changes in it (fixing bugs or adding new features). That future programmer might be you, months or years from now.

Ready for change. Software always changes. Some designs make it easy to make changes; others require throwing away and rewriting a lot of code.


Software Construction
Software Construction Fundamentals
Minimizing Complexity
Anticipating Change
Constructing for Verification
Reuse
Standards in Construction
Managing Construction
Construction in Life Cycle Models
Construction Planning
Construction Measurement
Practical Considerations
Construction Design
Construction Languages
Coding
Construction Testing
Construction for Reuse
Construction with Reuse
Construction Quality
Integration
Construction Technologies
API Design and Use
Object-Oriented Runtime Issues
Parameterization and Generics
Assertions, Design by Contract, and Defensive Programming
Error Handling, Exception Handling, and Fault Tolerance
Executable Models
State-Based and Table-Driven Construction Techniques
Runtime Configuration and Internationalization
Grammar-Based Input Processing
Concurrency Primitives
Middleware
Software Construction Tools
Development Environments
GUI Builders
Unit Testing Tools
Profiling, Performance Analysis, and Slicing Tools

SOFTWARE CONSTRUCTION FUNDAMENTALS

1.1 **Minimizing Complexity**

Most people are limited in their ability to hold complex structures and information in their working memories, especially over long periods of time. Complexity makes code harder to understand, test, and maintain.

The need to reduce complexity applies to essentially every aspect of software construction and is particularly critical to testing of software constructions.

In software construction, reduced complexity is achieved through

- emphasizing code creation that is simple and readable rather than clever.
- Abstraction → Hide implementation details, expose simple interfaces.
- Modular design → Break code into small, independent modules.

SOFTWARE CONSTRUCTION FUNDAMENTALS

1.2 Anticipating Change

Most software will change over time. Anticipating change means designing and constructing software so that it can adapt easily when requirements, technology, or environments change.

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

1.3 Constructing for Verification

Constructing for verification means building software in such a way that faults can be readily found by the software engineers writing the software as well as by the testers and users during independent testing and operational activities.

Specific techniques that support constructing for verification include following coding standards to support code reviews and unit testing, organizing code to support automated testing, and restricting the use of complex or hard-to-understand language structures, among others.

SOFTWARE CONSTRUCTION FUNDAMENTALS

1.4 Reuse

Reuse refers to using existing assets 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.

Instead of reinventing the wheel, developers reuse proven components → saves time, reduces cost, enable significant software quality, and increases reliability.

Reuse has two closely related facets:

- "construction for reuse" means to create reusable software assets,
- "construction with reuse." means to reuse software assets in the construction of a new solution.

SOFTWARE CONSTRUCTION FUNDAMENTALS

1.5 Standards in Construction

Standards are agreed-upon rules, conventions, or guidelines that developers follow when writing and organizing code. Applying external or internal development standards during construction helps making code **readable** and **consistent** and Improve **software quality** (fewer bugs, better maintainability).

Standards that directly affect construction issues include

- Coding Standards: Rules for naming, formatting, commenting, and structuring code.
- Documentation Standards: Ensure code is explained consistently.
- Testing Standards: Standard ways to write and organize tests.

External 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).

Standards may also be created on an organizational basis at the corporate level or for use on specific projects (internal standards).

MANAGING CONSTRUCTION

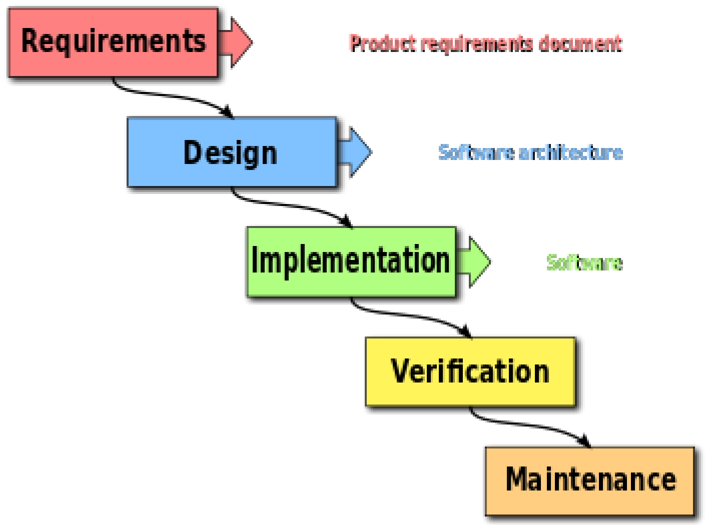
2.1 Construction in Life Cycle Models

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.



MANAGING CONSTRUCTION

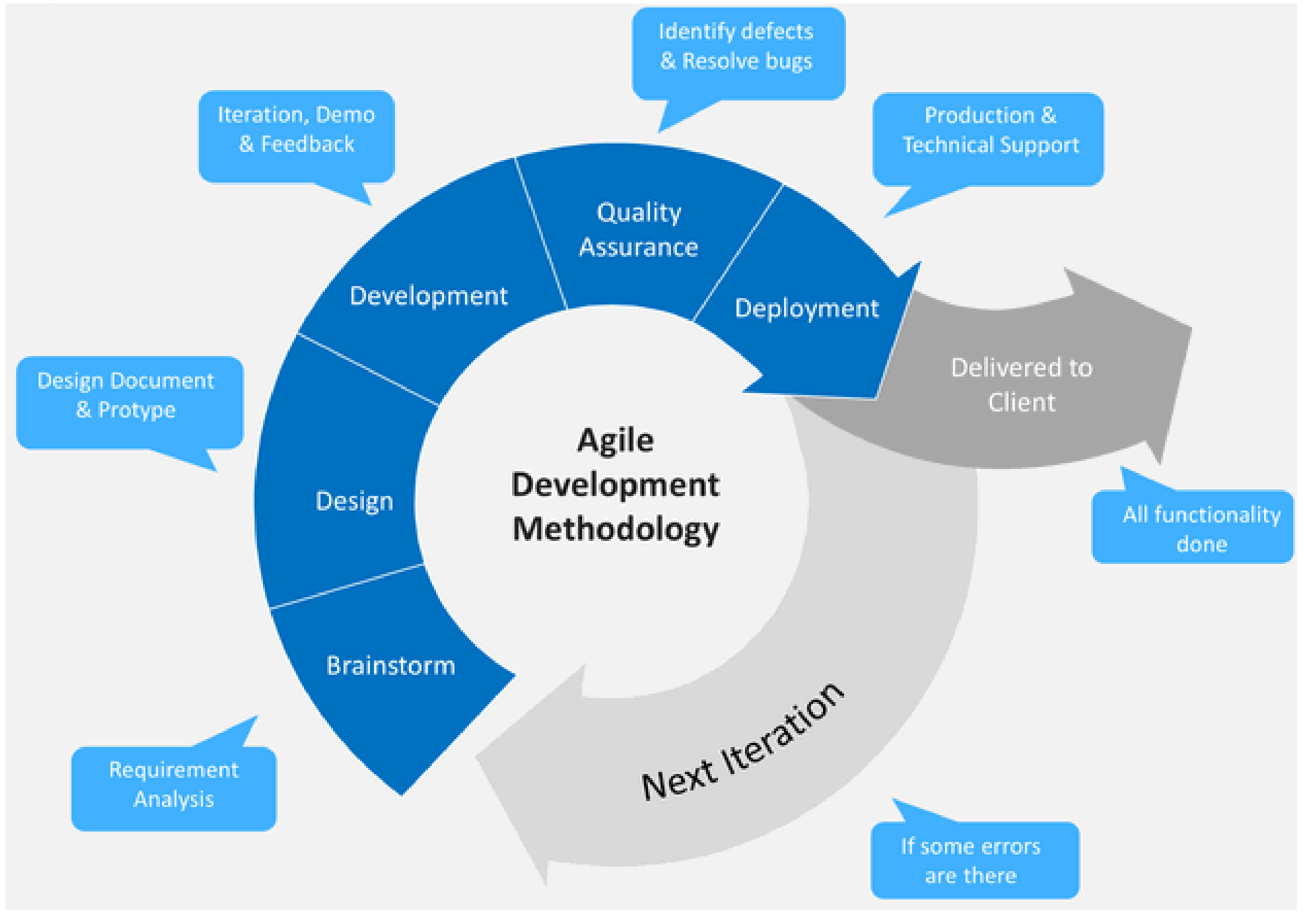
2.1 Construction in Life Cycle Models

Other models are more iterative—such as evolutionary prototyping and agile development.

These approaches tend to mix design, coding, and testing activities, and they often treat the combination of activities as construction.

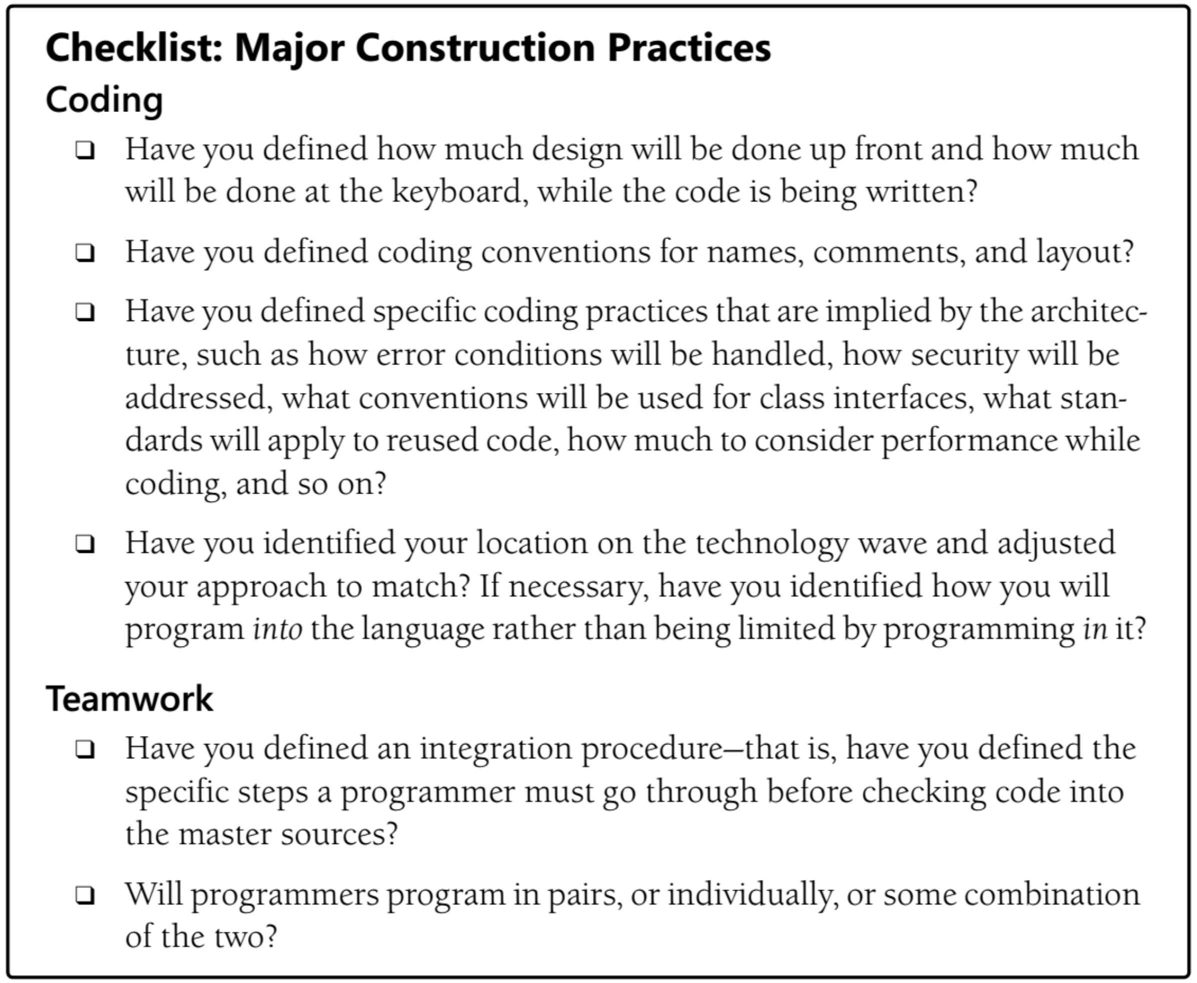
Consequently, what is considered to be "construction" depends to some degree on the life cycle model used.

In general, **software construction is mostly coding and debugging**, but it also involves construction planning, detailed design, unit testing, integration testing, and other activities.



MANAGING CONSTRUCTION

2.2 Construction Planning: refers to the process of defining and coordinating the tasks, resources, schedule, and procedures to construct software as part of the software development lifecycle.



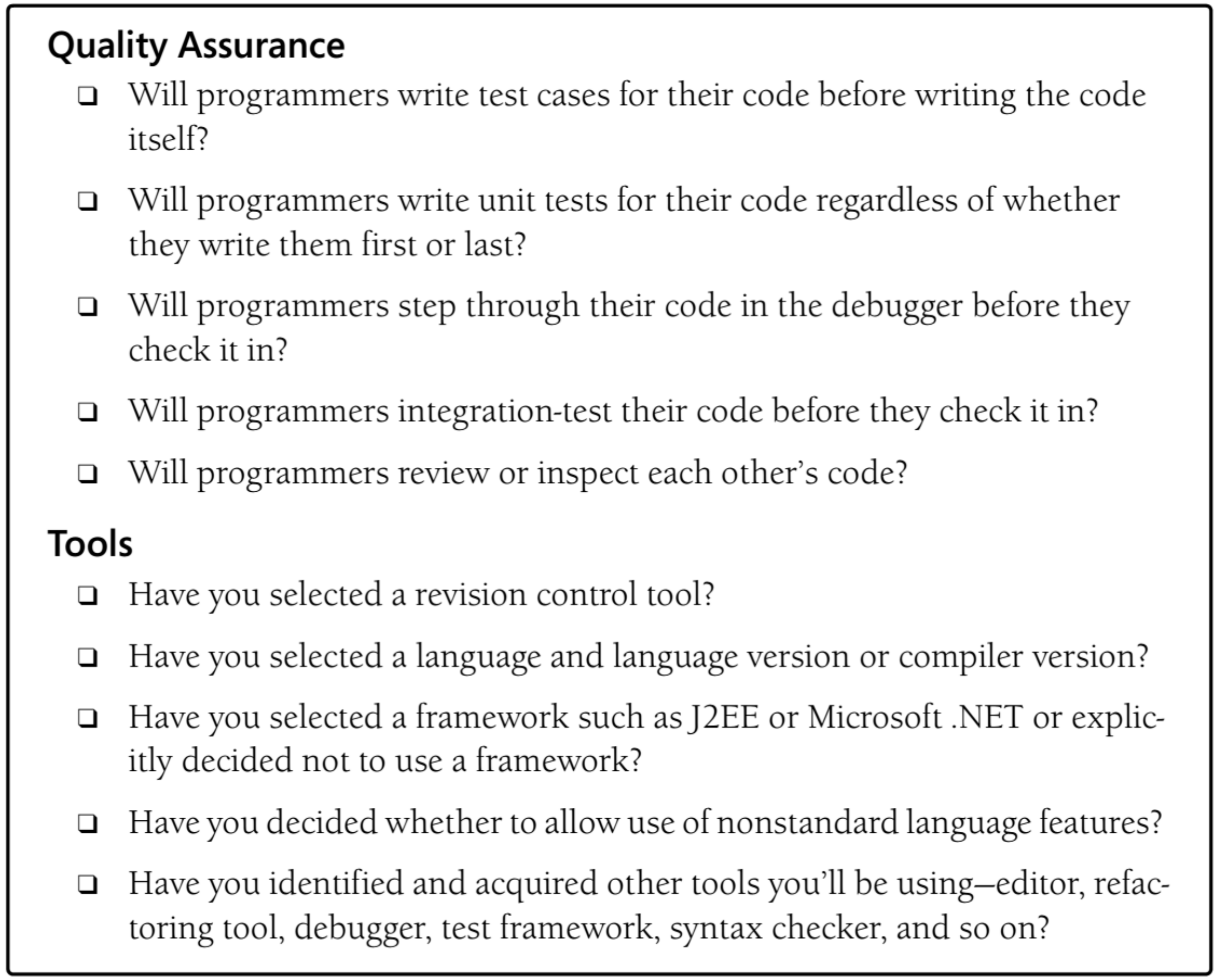
Checklist: Major Construction Practices

Coding

- ❑ Have you defined how much design will be done up front and how much will be done at the keyboard, while the code is being written?
- ❑ Have you defined coding conventions for names, comments, and layout?
- ❑ Have you defined specific coding practices that are implied by the architecture, such as how error conditions will be handled, how security will be addressed, what conventions will be used for class interfaces, what standards will apply to reused code, how much to consider performance while coding, and so on?
- ❑ Have you identified your location on the technology wave and adjusted your approach to match? If necessary, have you identified how you will program *into* the language rather than being limited by programming *in* it?

Teamwork

- ❑ Have you defined an integration procedure—that is, have you defined the specific steps a programmer must go through before checking code into the master sources?
- ❑ Will programmers program in pairs, or individually, or some combination of the two?



Quality Assurance

- ❑ Will programmers write test cases for their code before writing the code itself?
- ❑ Will programmers write unit tests for their code regardless of whether they write them first or last?
- ❑ Will programmers step through their code in the debugger before they check it in?
- ❑ Will programmers integration-test their code before they check it in?
- ❑ Will programmers review or inspect each other's code?

Tools

- ❑ Have you selected a revision control tool?
- ❑ Have you selected a language and language version or compiler version?
- ❑ Have you selected a framework such as J2EE or Microsoft .NET or explicitly decided not to use a framework?
- ❑ Have you decided whether to allow use of nonstandard language features?
- ❑ Have you identified and acquired other tools you'll be using—editor, refactoring tool, debugger, test framework, syntax checker, and so on?

MANAGING CONSTRUCTION

2.3 Construction Measurement

The measurements can be useful for purposes of managing construction, ensuring quality during construction, and improving the construction process.

1. **Code Metrics:**

Lines of Code: Measures the size of a software program by counting the number of lines in the source code.

Cyclomatic Complexity: Represents the number of linearly independent paths through a program's source code, offering insights into the code's complexity.

Code Churn: Measures the amount of code added, modified, or deleted from a software project over time, indicating the stability of the codebase.

2. **Quality Metrics:**

Defect Density: Calculates the number of defects per size of the code, e.g., per KLOC (Thousand Lines of Code).

Unit Test Coverage: Represents the percentage of source code covered by unit tests, indicating the extent of testing.

Code Review Feedback: Measures the number and severity of issues found during code reviews, indicating the code quality and adherence to standards.

3. **Productivity Metrics:**

Function Points per Person-Month: Measures the functionality delivered per person-month, providing insights into team productivity.

Defects Fixed per Hour: Represents the number of defects resolved in an hour, offering insights into the team's efficiency in debugging.

Code Commits per Day: Counts the number of code commits made per day, indicating the pace of development.

4. **Efficiency Metrics:**

Build Time: Measures the time taken to compile and build the software, indicating the efficiency of the build process.

Integration Time: Represents the time taken to integrate new code changes into the existing codebase, offering insights into the efficiency of the integration process.

Debugging Time: Measures the amount of time spent on identifying and fixing bugs, indicating the efficiency of the debugging process.

PRACTICAL CONSIDERATIONS

Construction is an activity in which the software engineer has to deal with sometimes chaotic and changing real-world constraints, and he or she must do so precisely.

3.1 Construction Design

In some projects, software design is handled in a separate, formal phase before coding, while in others, design continues into the construction stage.

Regardless some **detailed design** work will occur at the construction level, because developers must make practical choices that are not fully defined at the architectural level.

These choices are often dictated by real-world constraints such as system performance, memory limitations, integration with existing systems and business or regulatory rules.

Developers refine high-level architectural blueprints into specific details, including selecting algorithms that meet performance goals, choosing data structures that balance speed and memory, and defining clear interfaces for modules to communicate effectively.

PRACTICAL CONSIDERATIONS

3.2 Construction Languages

Construction languages are the tools we use to describe a solution in a form that a computer can execute, including programming languages, scripting languages, configuration languages, and toolkit languages.

The choice of construction language is directly influences key software quality attributes such as performance, reliability, maintainability, security, and portability across different platforms.

Configuration languages are primarily used for configuring software and systems. They aren't designed for writing full-fledged applications but are crucial for setting up and configuring software applications, frameworks, or systems. Example: YAML and JSON

Toolkit languages are used to build applications by assembling predefined, reusable components that come from a toolkit or framework, they are more complex than configuration languages. E.g. LabVIEW

Scripting languages are kinds of application programming languages that supports scripts which are often interpreted rather than compiled.

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 therefore, they require the most training and skill to use effectively.

PRACTICAL CONSIDERATIONS

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

linguistic (e.g., C/C++, Java), formal (e.g., Event-B), visual (e.g., MatLab Simulink).

Linguistic programming languages use text-based syntax and are what most people typically think of when referring to programming languages. They have a set of grammatical rules, and developers write instructions in human-readable text.

Formal notation languages are based on mathematical models and are designed to describe a system rigorously with mathematical precision. They are mainly used to specify, develop, and verify systems in a way that eliminates ambiguity and enhances correctness. Specify and verify critical systems in domains such as aviation, nuclear power plants, and medical devices.

Visual notations rely on 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 the arrangement of icons on a display. Visual programming languages use graphical elements to represent actions, variables, controls flow, etc., allowing developers to design and implement software using visual metaphors.

PRACTICAL CONSIDERATIONS

3.3 Coding

The following considerations apply to the software construction coding activity:

- Use clear naming conventions and consistent code layout to improve readability and maintenance.
- Apply classes, enumerated types, constants, and variables to model the problem domain effectively.
- Keep control structures (if, loops, switches) simple, structured, and easy to follow.
- Handle errors and exceptions carefully, validating inputs and logging failures for diagnosis.
- Prevent security issues by sanitizing inputs, checking array bounds, and avoiding injection attacks.
- Manage resources (files, threads, database locks) safely using proper synchronization and cleanup.
- Organize source code into modules, routines, classes, and packages for clarity and reuse.
- Provide useful documentation, focusing on the intent and rationale behind code decisions.
- Perform code tuning only after profiling and identifying performance bottlenecks.

PRACTICAL CONSIDERATIONS

3.4 Construction Testing

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 when faults are inserted into the code and the time when those faults are detected, thereby reducing the cost incurred to fix them.

PRACTICAL CONSIDERATIONS

3.5 Construction for Reuse

Construction for reuse refers to the practice of developing software components with the intention of reusing them in different parts of the current project or in different projects altogether. This practice emphasizes creating modular, adaptable, and well-documented code components or libraries.

To avoid the problem of code clones, it is desired to encapsulate reusable code fragments into well-structured libraries or components.

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

- Variability analysis: Identify what is common and what may change across different uses.
- Encapsulation: Hide internal details, provide clean and stable interfaces.
- Testing for reuse: Verify components under different configurations and scenarios.

Result: Saves time, reduces bugs, and creates flexible, reliable software.

PRACTICAL CONSIDERATIONS

3.6 Construction with Reuse

Construction with reuse means to create new software with the **reuse of existing software assets.**

The most popular method of reuse is to reuse code from the **libraries** provided by the language, platform, tools being used, an organizational repository or many open-source libraries.

Benefit: Saves time and cost while using trusted, tested, and often higher-quality code.

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

- Select reusable assets (libraries, databases, test data).
- Evaluate the suitability and quality of reusable code or tests.
- Integrate chosen assets smoothly into the new system.
- Report and document which parts of the system rely on reused components.

PRACTICAL CONSIDERATIONS

3.7 Construction Quality

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

Numerous techniques exist to ensure the quality of code as it is constructed.

- unit testing and integration testing
- test-first development
- use of assertions and defensive programming
- debugging
- static analysis
- inspections

PRACTICAL CONSIDERATIONS

3.8 Integration

A key activity during construction is the integration of individually 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,
- identifying what hardware is needed,
- determining the degree of testing and quality work performed on components before they are integrated.

Programs can be integrated by means of either the phased or the incremental approach.

Phased integration, also called "big bang" integration, entails delaying the integration of component software parts until all parts intended for release in a version are complete.

In **incremental integration**, the developers write and test a program in small pieces and then combine the pieces one by one. 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.