Software Construction and Testing:

1. Software construction basics,
2. Object-oriented design principles,
3. Object-oriented programming languages (Java, C++, Python),
4. Software testing basics (unit testing, integration testing, system testing),
5. Test-driven development (TDD)

# INTRODUCTION

The term software construction refers to the detailed creation of working software through a combination of coding, verification, unit testing, integration testing, and debugging. The Software Construction knowledge area (KA) is linked to all the other KAs, but it is most strongly linked to Software Design and Software Testing because the software construction process involves significant software design and testing. The process uses the design output and provides an input to testing (“design” and “testing” in this case referring to the activities, not the KAs). Boundaries between design, construction, and testing (if any) will vary depending on 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 during the construction activity.

It is also related to project management, insofar as the management of construction can present considerable challenges.

The importance of software construction

1. Construction is a large part of software development. Depending on the size of the project, construction typically takes 30 to 80 percent of total time spent on a project.
2. Construction is the central activity in software development
   * Requirement and architecture
   * Construction
   * System Testing

# SOFTWARE CONSTRUCTION FUNDAMENTALS

Software Construction fundamentals includes:

* 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.

## 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. 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 testing of software constructions. 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, modular design, and numerous other specific techniques. It is also supported by construction-focused quality techniques.

## Anticipating Change

Most software will change over time, and the anticipation of change drives many aspects of software construction; changes in the environments in which software operates also affect software in diverse ways. Anticipating change helps software engineers build extensible software, which means they can enhance a software product without disrupting the underlying structure. Anticipating change is supported by many specific techniques .

## 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.

## 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. Reuse is best 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.

## Standards in Construction

Applying external or internal development standards during construction helps achieve a project’s objectives for 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++)\*coding standards (for example, standards for naming conventions, layout, and indentation)
    - platforms (for example, interface standards for operating system calls)
    - tools (for example, diagrammatic standards for notations like UML (Unified Modeling Language)).
* **Object Oriented Principles in OOAD**

Object-oriented principles are a set of guidelines for designing and implementing software systems that are based on the idea of objects. Objects are self-contained units of code that have both data and behavior. They can interact with each other to perform tasks.

Object-Oriented Analysis and Design (OOAD) is a software engineering methodology that uses object-oriented principles to design and implement software systems. OOAD involves a number of techniques and practices, including:

* **Object-Oriented Modelling:**This involves using visual diagrams to represent the different objects in a software system and their relationships to each other.
* **Use Cases:**This involves describing the different ways in which users will interact with a software system.
* **Design Patterns:**This involves using reusable solutions to common problems in software design.

**Important Topics for Object Oriented Principles in OOAD**

* [Abstraction](https://www.geeksforgeeks.org/object-oriented-principles-in-ooad/#abstraction)
* [Encapsulation](https://www.geeksforgeeks.org/object-oriented-principles-in-ooad/#encapsulation)
* [Modularity](https://www.geeksforgeeks.org/object-oriented-principles-in-ooad/#modularity)
* [Hierarchy](https://www.geeksforgeeks.org/object-oriented-principles-in-ooad/#hierarchy)
* [Typing](https://www.geeksforgeeks.org/object-oriented-principles-in-ooad/#typing)
* [Concurrency](https://www.geeksforgeeks.org/object-oriented-principles-in-ooad/#concurrency)

**Abstraction**

*Think of a TV remote control. It has buttons like power, volume up, volume down, and channel change. Now, let's use this as an example of Abstraction*

In OOP, abstraction is like using a TV remote without knowing how it works on the inside. You don't need to know about the wires, circuits, or tiny components inside the remote. All you care about are the buttons and what they do.

In this example:

* **Buttons** are like the functions or actions in a program, such as play, pause, or stop.
* **What the buttons do**is like the behavior of objects or classes in OOP. For example, when you press the volume up button, the volume goes up,, but you don't need to understand how it happens.

So, **Abstraction** in OOP is about using objects or classes (like our TV remote) without worrying about how they work internally. You only care about what they can do and how to use them, just like using a TV remote without needing to be an electrical engineer to make it work.

**Advantages of Abstraction**

* Abstraction makes things simpler. It helps us focus on what's important and ignore what's not, making it easier to understand.
* We can reuse the same template for different things, saving time and making our work more efficient.
* When something goes wrong or needs an update, we can fix just the part that's broken without messing up everything else.
* Abstraction helps us grow our projects without making them messy.

**Disadvantages of Abstraction**

* Sometimes, too much abstraction can make things more hard to understand.
* Abstraction can add extra work and code that might slow down our program a little bit.
* Using abstraction tools can be tricky for beginners.

**Encapsulation**

Let us take an example of a water bottle to explain encapsulation:

* **The Bottle:**In OOP, a class is like the bottle, with visible features (attributes and methods) and hidden contents (data and functions).
* **The Cap:**The cap is like encapsulation. It protects what's inside the bottle (the object) and keeps it safe from outside interference.
* **The Water:**Inside the bottle is data, like water. You can use the bottle (object) to access and modify the data, without needing to know how it's stored or processed inside.

*So, encapsulation in OOP is like a cap on water bottle, keeping the inner workings hidden and secure. letting you use the object without worrying about its internal details.*

**Advantages of Encapsulation**

* Encapsulation keeps data safe and prevents unauthorized access or modification.
* It allows controlled access to data through methods.
* You can change the internal workings of an object without affecting the code that uses it.
* Encapsulation makes code easier to understand and maintain because you only need to focus on what a class does, not how it does it.

**Disadvantages of Encapsulation**

* It can add extra layers to your code, making it a bit more complex.
* Encapsulation can sometimes make your code a bit slower because of the extra control it imposes.

[Modularity](https://www.geeksforgeeks.org/modularity-and-its-properties/)

**Modularity** in OOAD is like organizing your kitchen. Just as you keep pots in one cabinet and dishes in another for easier access and maintenance, in OOAD, you group similar functions and data into organized module or classes. This makes it simpler to understand and change specific parts of your software without affecting the entire program, similar to how you can upgrade one appliance in your kitchen without redoing the whole room.

**Advantages of Modularity**

* Modularity makes it easier to fix or update one part of a software system with messing up the rest.
* You can reuse modules in different parts of your software, saving time and effort.
* Multiple programmers can work on different modules simultaneously.

**Disadvantages of Modularity**

* Overdoing modularity can make your software too complex with many small parts, making it harder to grasp.
* Breaking a program into modules may add some extra work and slow down the software slightly.

**Hierarchy**

Lets us take an example of a family. In a family there are grandparents, parents, and children:

* **Classes as Family Members:**In OOAD, think of your family members as classes or objects. Each class has a specific job, like each family members has a role in the family.
* **Family Hierarchy:**Just like your family tree has a hierarchy with grandparents at the top, parents in the middle and children at the bottom, in OOAD classes can can be organized in a hierarchy. Some classes are more general (like parents) and others are more specific (like children).
* **Inheritance:**Imagine your grandparents passing down family traditions to your parents, who then pass then on to you. This is similar in OOAD where classes can inherit features from other classes higher up in the hierarchy.
* **Specialization:**You and your siblings have more specialized roles compared to your parents. This is like specialization in OOAD, where subclasses have specific features compared to their parent classes.

*So, hierarchy in OOAD is like arranging classes in an organized way, just as your family tree helps you understand your family's structure. It helps in managing and understanding the relationships between different classes in a software system.*

**Typing**

Typing involves categorizing objects based on their data types (e.g., integers, strings, custon objects) to ensure they are used appropriately.

**Example:**

*Think about sorting your belongings. You wouldn't mix up your books, clothes, and kitchen utensils in the same box. Similarly, in programming, you categorize data based on their data types (e.g., numbers, text, dates) to perform operations correctly. This helps to prevent errors and make code more readable and maintainable.*

**Concurrency**

Concurrency in Object Oriented Analysis and Design (OOAD) is like managing multiple tasks at at the same time, just as people multitask in every day life.

*Imagine you're a chef in a restaurant. You have several orders to prepare, and each order consists of different dishes. You can't cook one disk at a time and move to the next dish because customers are hungry and waiting for their food. So, you need to work on multiple dishes simultaneously*

Now, let's relate this to OOAD:

* **Tasks as Objects:**In OOAD, think of each dish you're cooking as an object or a task. Each dish has its recipe ad cooking instructions, just like objects have their method and properties.
* **Concurrency in Kitchen:**You are working concurrently in kitchen, managing multiple dishes simultaneously. While one dish is simmering, you might be chopping ingredients for another or seasoning a third. You switch between tasks efficiently to serve all orders.

In OOAD, concurrency is about managing multiple tasks or processes within a software system simultaneously. It's like juggling different tasks efficiently to make the most of your time.