UFV

UFV.ES

Software

MODULE 2 / UNIT 5 / 0.8

MOISES M. MARTINEZ

FUNDAMENTALS OF COMPUTER ENGINEERING

2025/2026

What is software?

What is software?

Software is a set of instructions called programs used to operate computers executing specific tasks.

Software is a sequence of instructions that tell a computer what to do.

Software is a sequence of computer instructions, which also includes data, documentation and other intangible components used to execute specific tasks in a computer.

What is software?

Software is a set of instructions called programs used to operate computers executing specific tasks.

Software is a sequence of instructions that tell a computer what to do.

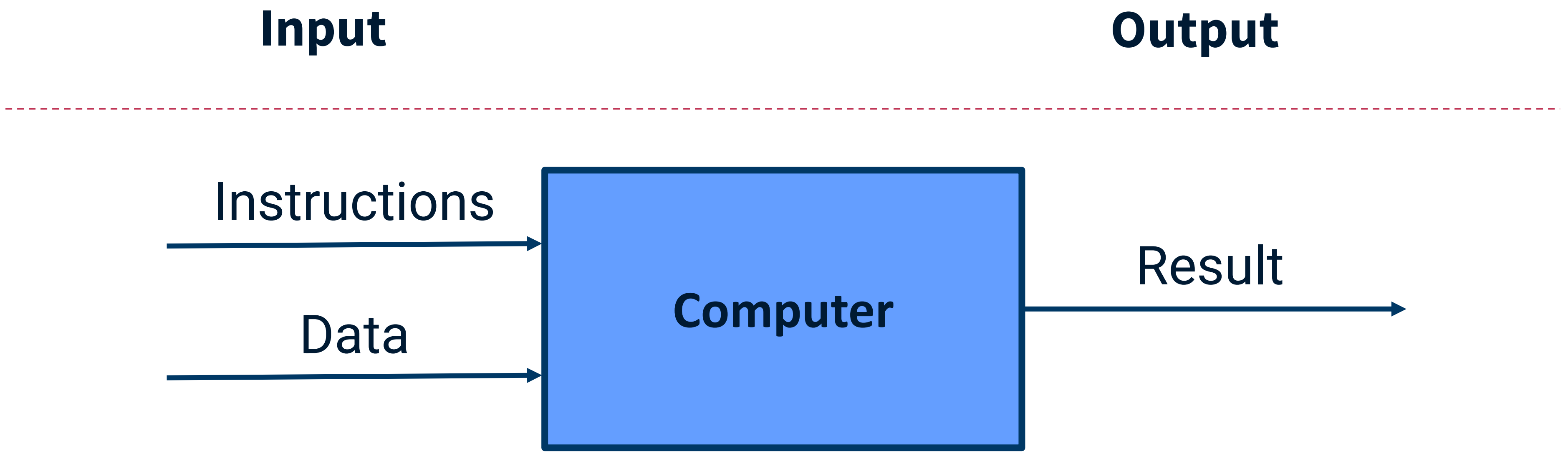
Software is a sequence of computer instructions, which also includes data, documentation and other intangible components used to execute specific tasks in a computer.

Basic concepts

01

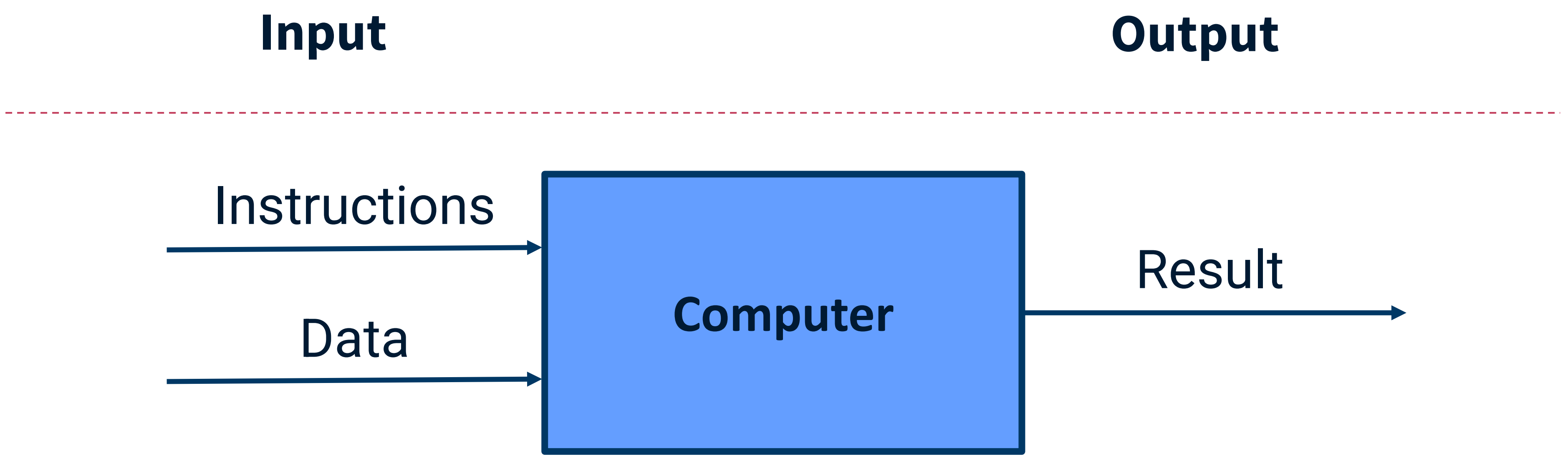
Fundamental Concepts

Software is a sequence of computer instructions, which also includes data, documentation and other intangible components used to execute specific tasks in a computer.



Fundamental Concepts

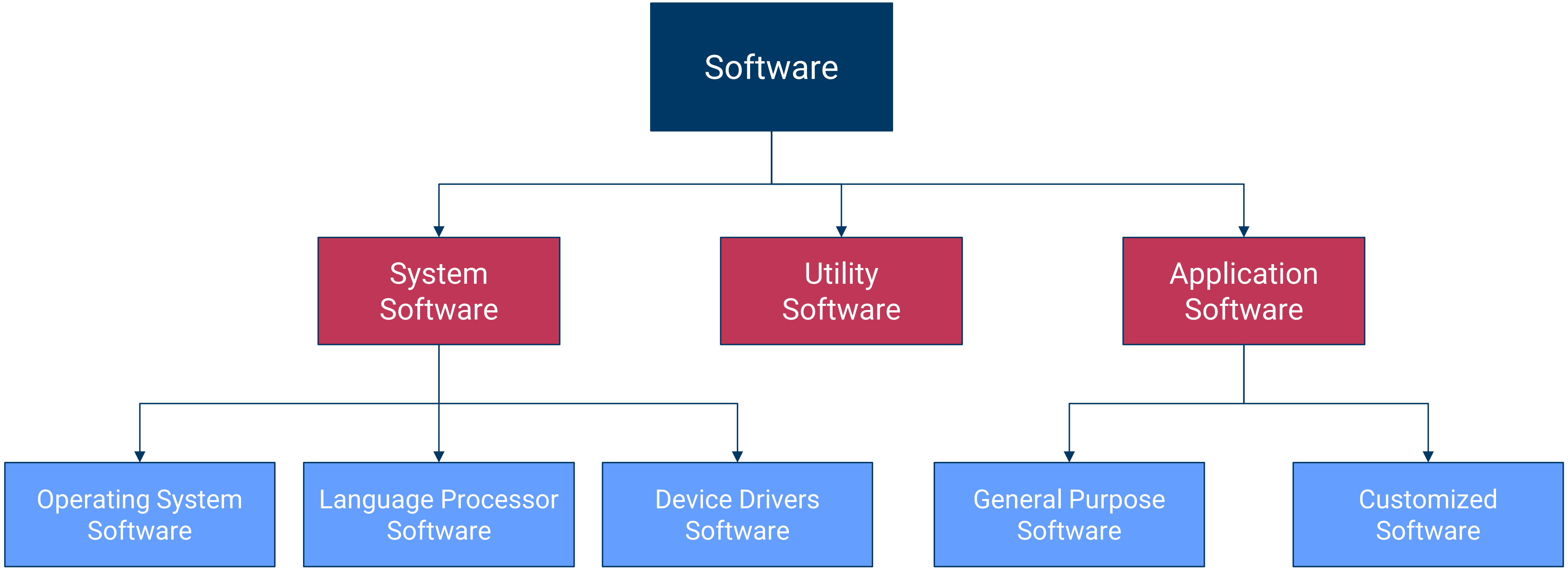
Software is a sequence of computer instructions, which also includes data, documentation and other intangible components used to execute specific tasks in a computer.



Software typically comprises computer programs, which are sequences of instructions written in a particular programming language intended for a computer.

Fundamental Concepts

Computer are compatible with various categories of software, including system software, utility software, and application software.



Fundamental Concepts

System software

System software is designed to manage computer hardware directly and provide essential functionality to both users and other software applications, ensuring smooth and efficient operation.

Key characteristics of system software include:

- Managing the internal operations of the computer.
- Facilitating communication between applications and hardware devices.
- Typically developed using low-level programming languages.
- Challenging in terms of design and understanding.
- Exhibiting faster execution speeds compared to other software types.
- Often implemented in low-level languages, such as assembly or machine code.

Fundamental Concepts

System software

System software is further categorized into three distinct subtypes:

- Operating System (OS) Software serves as the core program of a computer, managing all resources, including memory, the CPU, and storage devices. It also provides a user interface, enabling interaction with the computer system.
- Language Processor Software translates human-readable programming languages into machine language and vice versa. It converts programs written in high-level languages like Java, C, C++, and Python into machine code or object code.
- Device Driver Software is a essential software components that control the operations of various hardware devices, ensuring their proper functioning. Devices such as keyboards, GPUs, routers, and other peripherals require drivers to communicate effectively with the computer system.

Fundamental Concepts

Utility software

Utility software, a specialized subset of system software, is designed to perform specific functions that help maintain a computer's optimal performance.

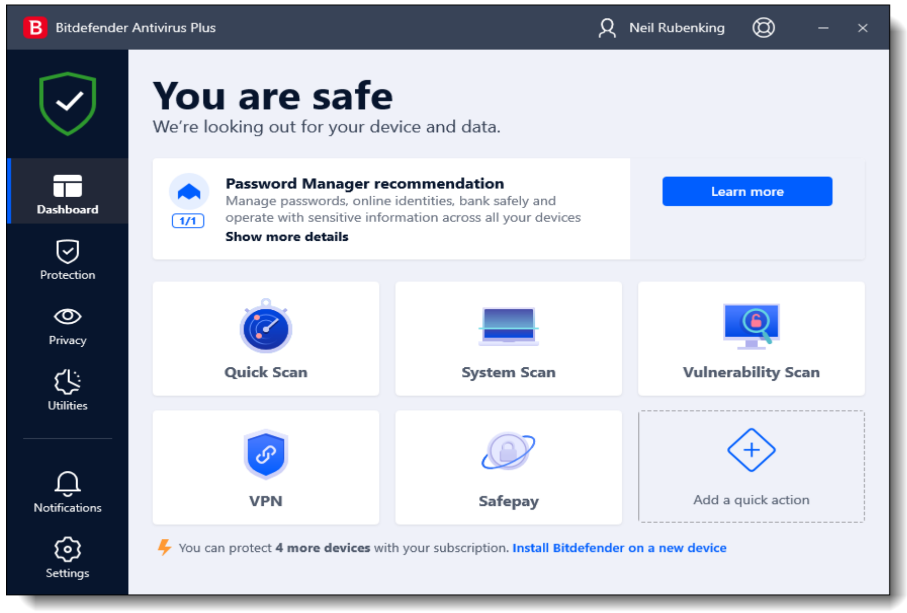
Key characteristics of utility software include:

- This is tailored to perform dedicated functions that ensure the safe and efficient operation of the operating system.
- It runs seamlessly in the background, executing its tasks without interrupting the user's activities.
- It relies on certain operating system components to perform its functions effectively.
- It is generally developed using high-level programming languages such as Python, C, or C#.
- It can be controlled and configured to suit their specific needs and preferences.

Fundamental Concepts

Utility software

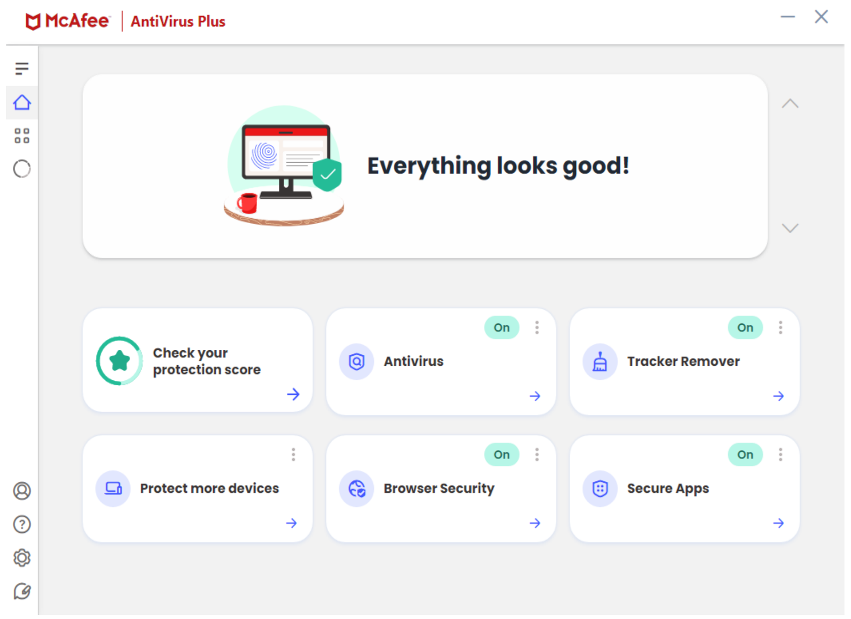
Examples of utility software include security programs such as antivirus software, which scans for and removes viruses from the operating system. Additionally, optimization tools encompass system clean-up utilities, disk defragmentation programs, and file compression software.



You are safe
We're looking out for your device and data.

Password Manager recommendation
Manage passwords, online identities, bank safely and operate with sensitive information across all your devices
Show more details

You can protect 4 more devices with your subscription. Install Bitdefender on a new device



Fundamental Concepts

UFV

Application software

Application software is developed to perform specialized tasks and provide capabilities that go beyond the basic functions of a computer. This category includes a wide range of end-user applications, such as word processors, spreadsheets, database management systems, inventory management software, payroll programs, web browsers, video games, media players, and more.

Key characteristics of application software include:

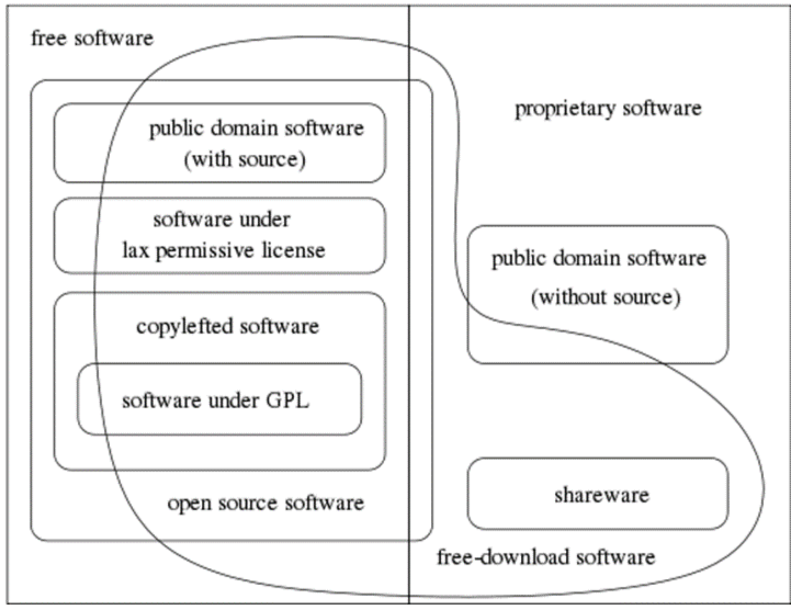
- It is designed to perform specific tasks, such as word processing, spreadsheet management, email communication, and video gaming, among others.
- It generally requires more storage space compared to system software.

Fundamental Concepts

Software licenses

A software license constitutes a legally binding instrument, typically governed by contract law, that regulates the utilization or distribution of software. As per the GNU Project:

- Freeware: Free, copyrighted software.
- Shareware: It can be used with limitations.
- Free software: It may be copied, modified and distributed.
- Open source software: It shares intellectual property.
- Proprietary software: It is not free, it belongs to a company.
- Commercial software: Its purpose is to generate economic profit.



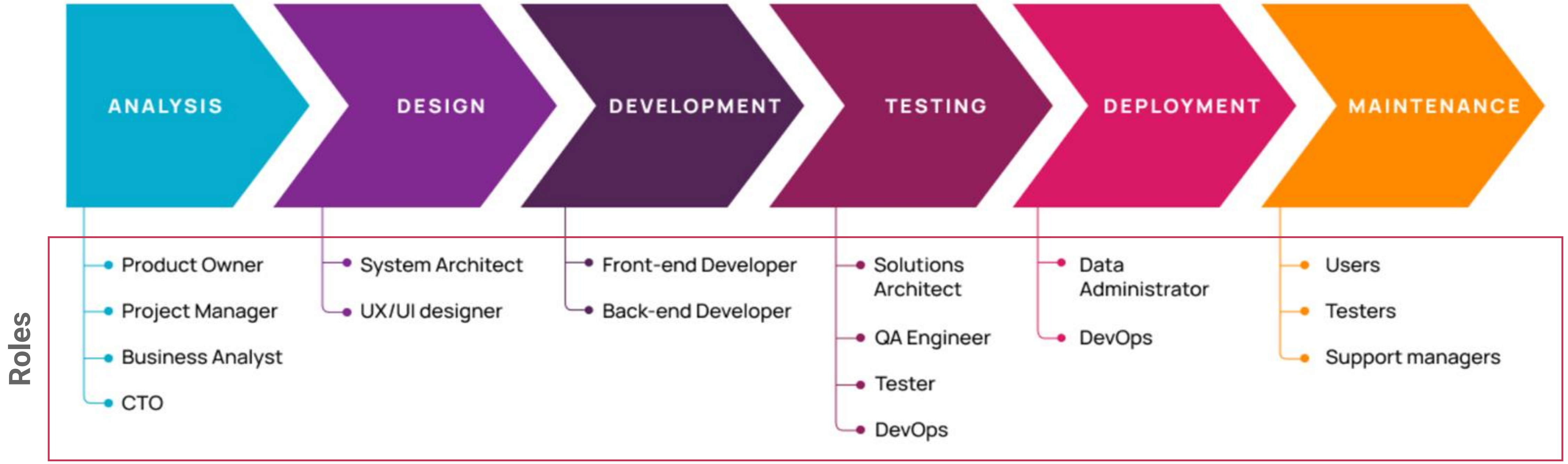
UFV

Software development life cycle

02

Software development life cycle

The Software Development Life Cycle (SDLC) comprises a set of standardized phases adhered to by the software development team throughout the development process.



The quantity of phases within a software development life cycle may fluctuate, contingent upon the chosen software development methodology and framework.

Software development life cycle

Why is the software development life cycle important?

The Software Development Life Cycle (SDLC) is a structured process that typically results in higher software product quality and shorter development and deployment times, which in turn leads to reduced overall costs.

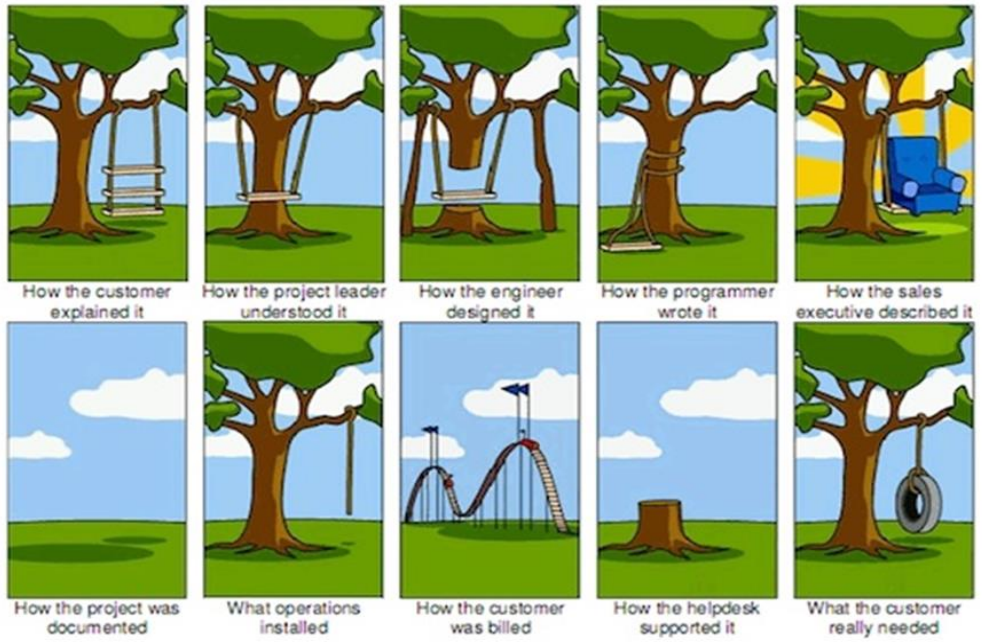
- The SDLC enhances management oversight and control, enabling effective supervision of each project phase.
- The SCLS generates comprehensive documentation covering both the software and the development process.
- Software objectives are clearly defined at the outset by all stakeholders, allowing for the creation of an actionable roadmap.
- Stakeholder input is systematically incorporated at the most appropriate stages of the development process.
- The SDLC improves risk management by identifying and mitigating potential risks early in the project.
- The SDLC ensures alignment with regulatory and quality standards, enhancing software reliability and compliance.

Software development life cycle

The Analysis phase

The specific **needs and expectations of the customer and end-users** are thoroughly defined, analyzed, and documented, ensuring a clear understanding of the project's objectives and requirements.

- What problem does the software aim to address?
- What is the potential value of solving this problem?
- What is the estimated size of the user base that could benefit from the software?
- Are there existing solutions in the market that address the same problem, and how does this software intend to offer improvements?
- What is the projected development cost for effectively solving the problem with this software?
- Is the potential value generated by solving the problem sufficient to provide an acceptable return on investment for the development cost?
- Do we possess the necessary resources to support the associated development expenses?


How the customer explained it
How the project leader understood it
How the engineer designed it
How the programmer wrote it
How the sales executive described it
How the project was documented
What operations installed
How the customer was billed
How the helpdesk supported it
What the customer really needed

Software development life cycle

The Design phase

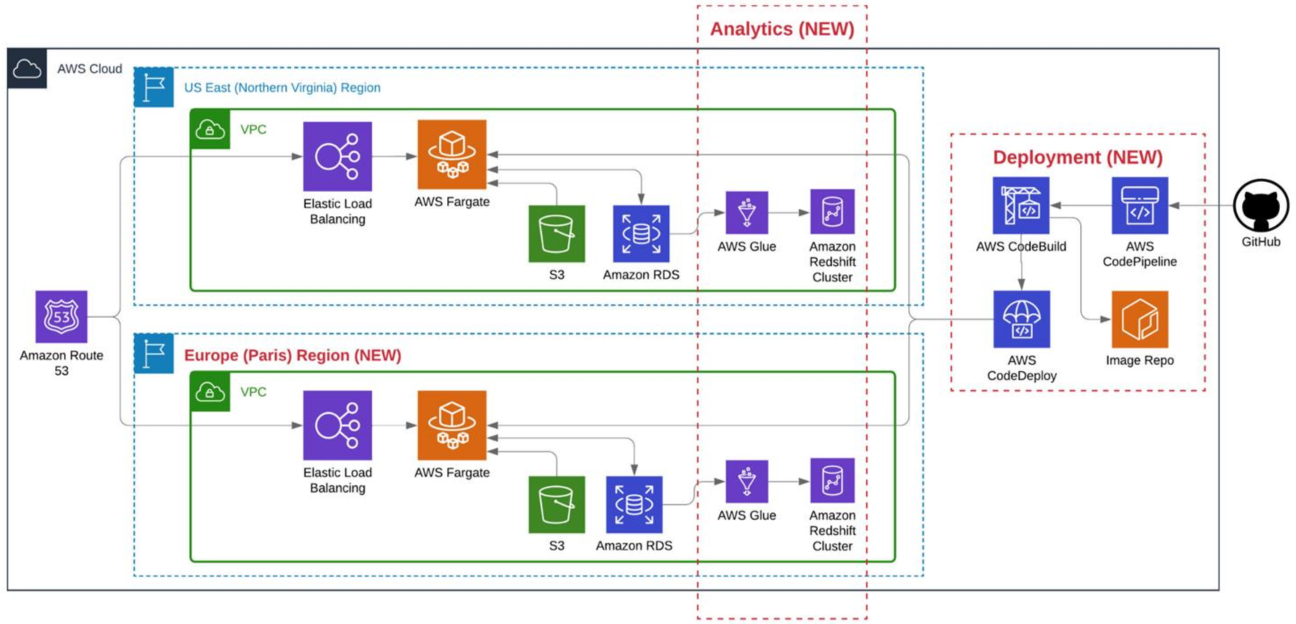
The software's architecture and structure are carefully outlined, based on the insights gained from the preceding analysis phase.

The primary deliverable of this stage is the Software Design Document (SDD), a detailed blueprint of the system's components, where each element is meticulously defined. The SDD is often organized into two distinct documents:

- The Architectural Design Document (ADD) defines the solution's overall structure, building on the problem description from the analysis phase. It identifies the major modules—sets of related functions—and outlines their interconnections and interactions.
- The Detailed Design Document (DDD) offers an in-depth view of the components and algorithms, providing a precise blueprint of the code's structure and the implementation details for each module.

Software development life cycle

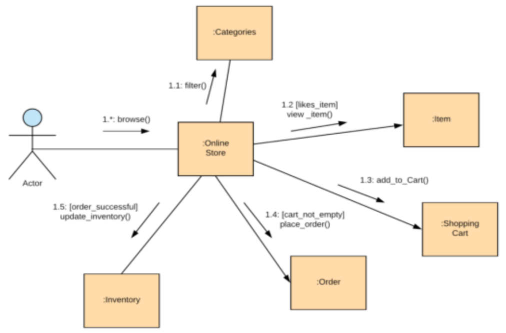
The Design phase



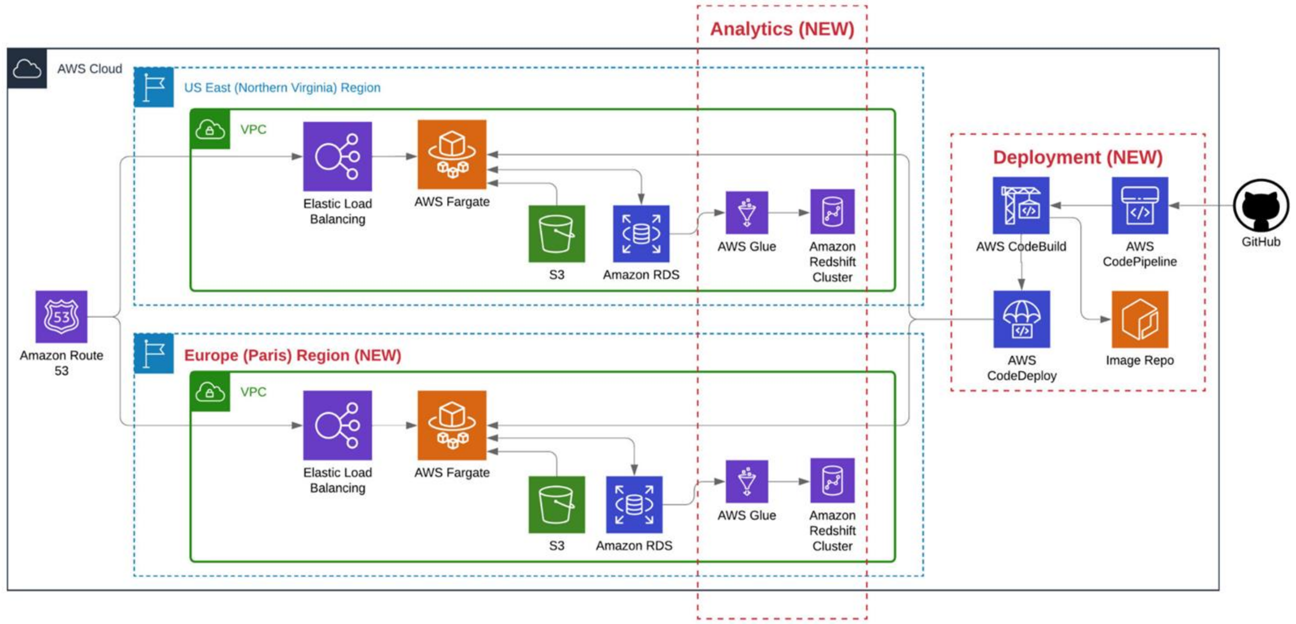
Cloud Architecture Diagram

Software development life cycle

The Design phase



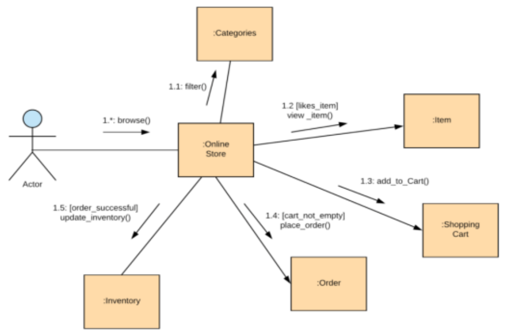
User story Diagram



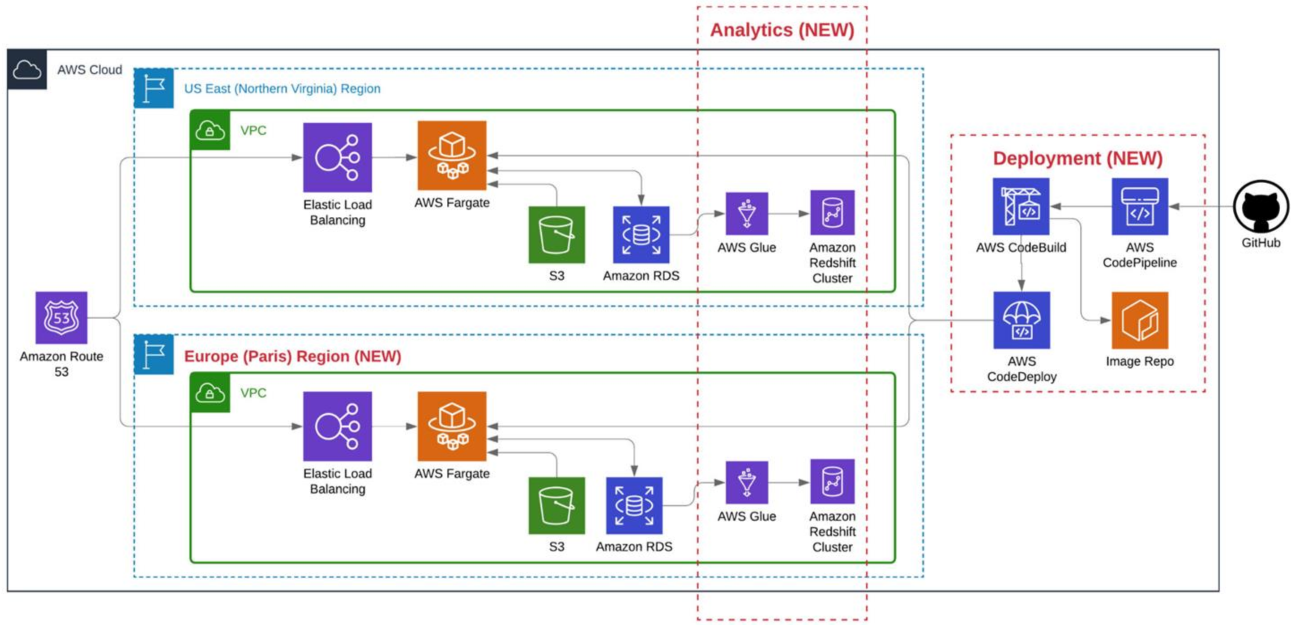
Cloud Architecture Diagram

Software development life cycle

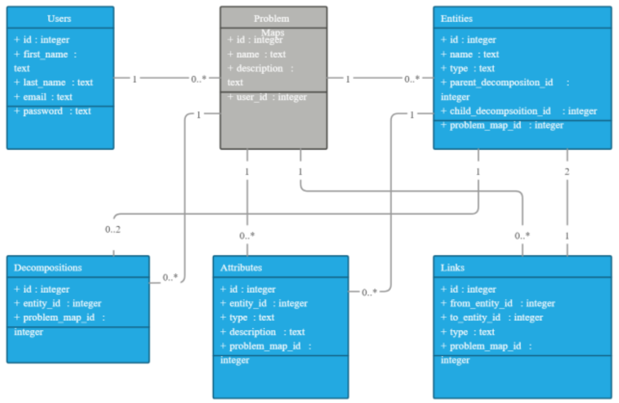
The Design phase



User story Diagram



Cloud Architecture Diagram



Database Diagram

Software development life cycle

The Design phase

During this phase, it is essential to thoroughly identify all software elements and their interactions with other components, including:

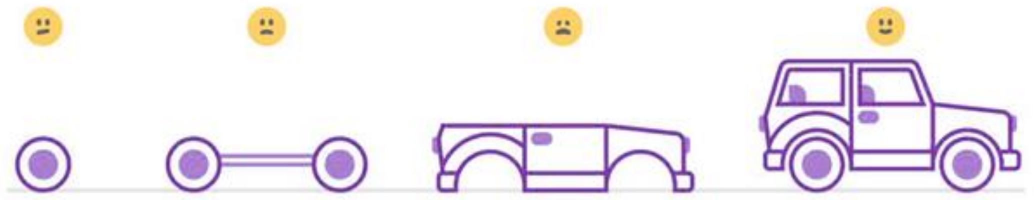
- Graphical User Interfaces (GUIs).
- Storage systems.
- Security and access systems.
- Algorithms.
- Core components.

Software development life cycle

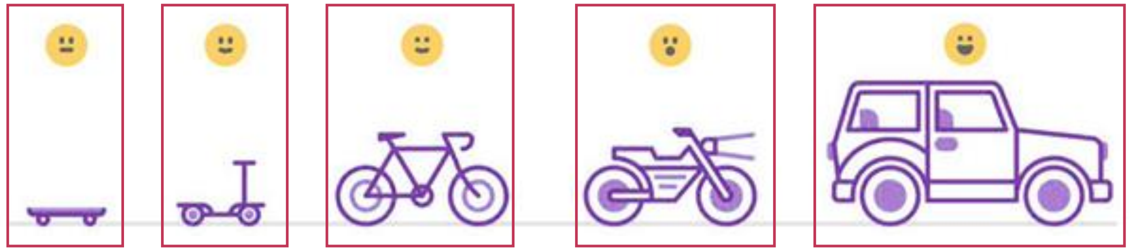
The Development phase

The development phase encompasses both coding and testing the software.

In the Waterfall methodology, the development process is thoroughly defined and documented from the outset, with the software being built sequentially from start to finish.



In contrast, the Agile methodology breaks development into multiple iterative cycles within the SDLC. During each iteration, a **Minimum Viable Product (MVP)** is created, allowing for continuous refinement and adaptation based on feedback.

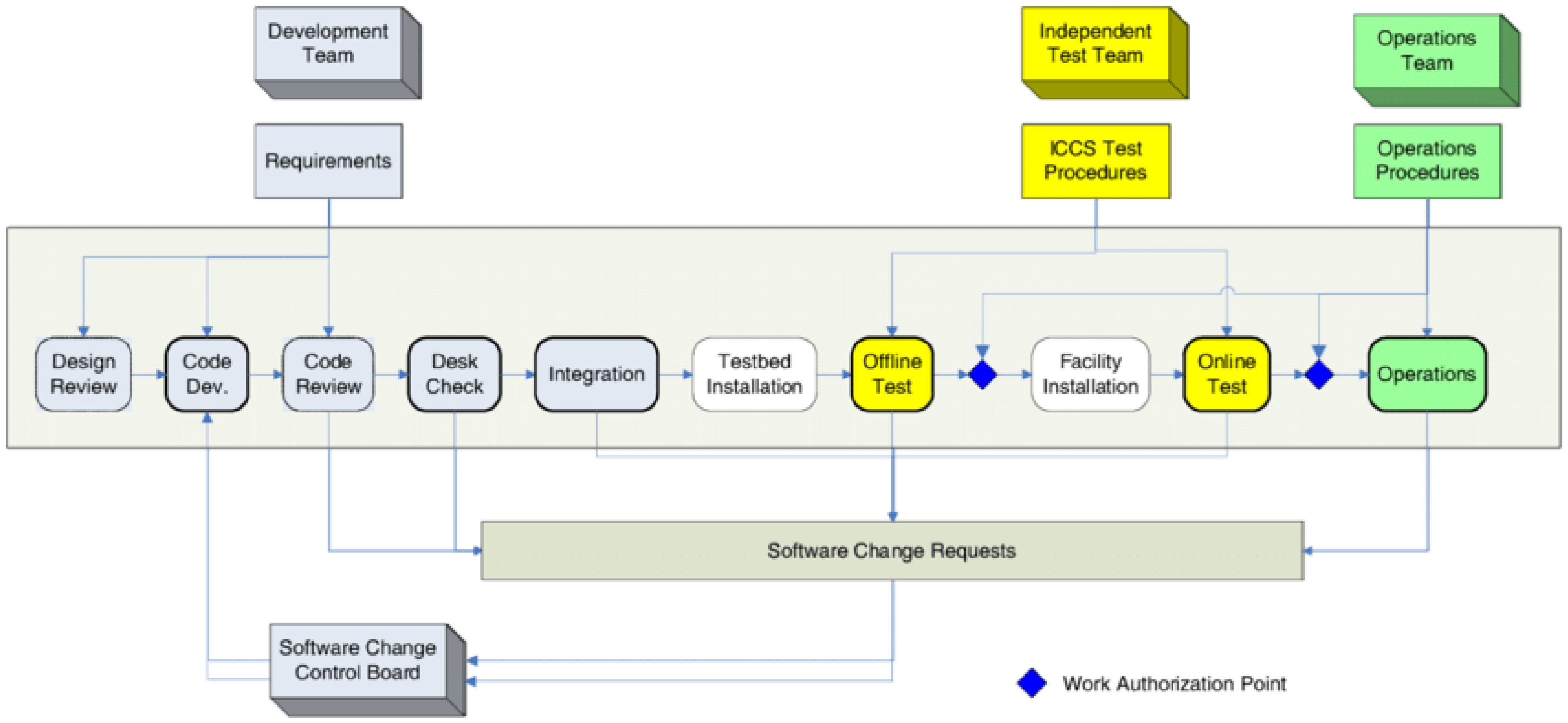


Each one is a MVP.

Software development life cycle

The Deployment phase

This phase involves moving the software application from the development and testing environments to the production environment, where it becomes accessible to end-users. The complexity of this process can vary greatly, depending on the specific features and requirements of the software.



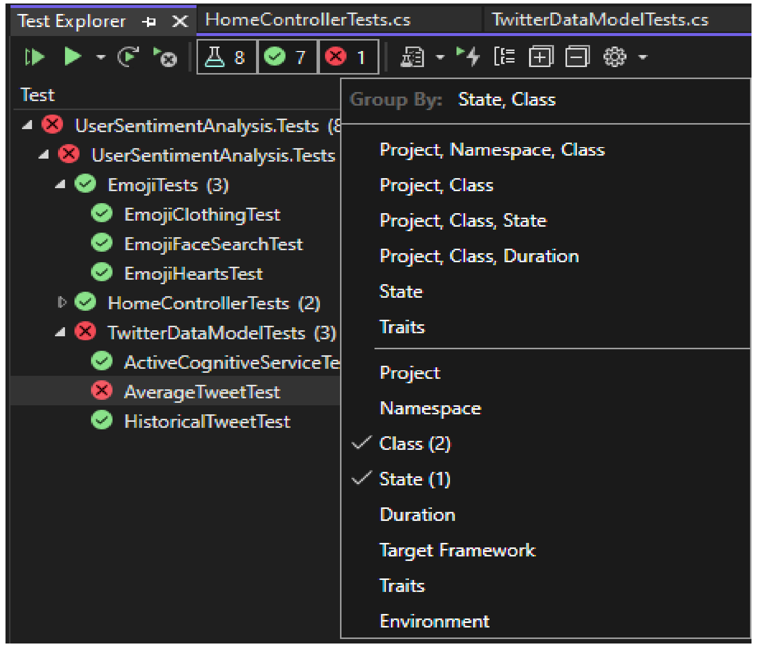
Software development life cycle

The Testing phase

In this phase, the software developed during the preceding stages is subjected to thorough testing to ensure it aligns with the requirements defined in the planning and analysis phases. This includes rigorous validation against functional specifications, as well as performance and bug testing.

The primary goal of this phase is to confirm that the software has been developed according to the established specifications verifying that it meets user functionality requirements, operates efficiently with the necessary resources, and adheres to availability and security standards.

- Manual testing of user workflows to ensure usability and functionality.
- Automated testing using a variety of tools and scripts to systematically identify and address issues.



```
Test
◢ ✖ UserSentimentAnalysis.Tests (8
  ◢ ✖ UserSentimentAnalysis.Tests
    ◢ ✓ EmojiTests (3)
        ✓ EmojiClothingTest
        ✓ EmojiFaceSearchTest
        ✓ EmojiHeartsTest
    ▷ ✓ HomeControllerTests (2)
    ◢ ✖ TwitterDataModelTests (3)
        ✓ ActiveCognitiveServiceTe
        ✖ AverageTweetTest
        ✓ HistoricalTweetTest
```

The testing phase is arguably the most critical stage in the software development process.

Software development life cycle

The Maintenance phase

The maintenance phase involves activities dedicated to ensuring the software application's continued proper functioning, focusing on its functionality, performance, and availability.

This phase consists of ongoing tasks that persist throughout the product's lifecycle, well beyond the development phase.

- Monitoring software performance to ensure optimal operation.
- Conducting regular evaluations to assess the system's effectiveness.
- Identifying and resolving errors and bugs that may arise.
- Implementing enhancements or system updates to address evolving needs.

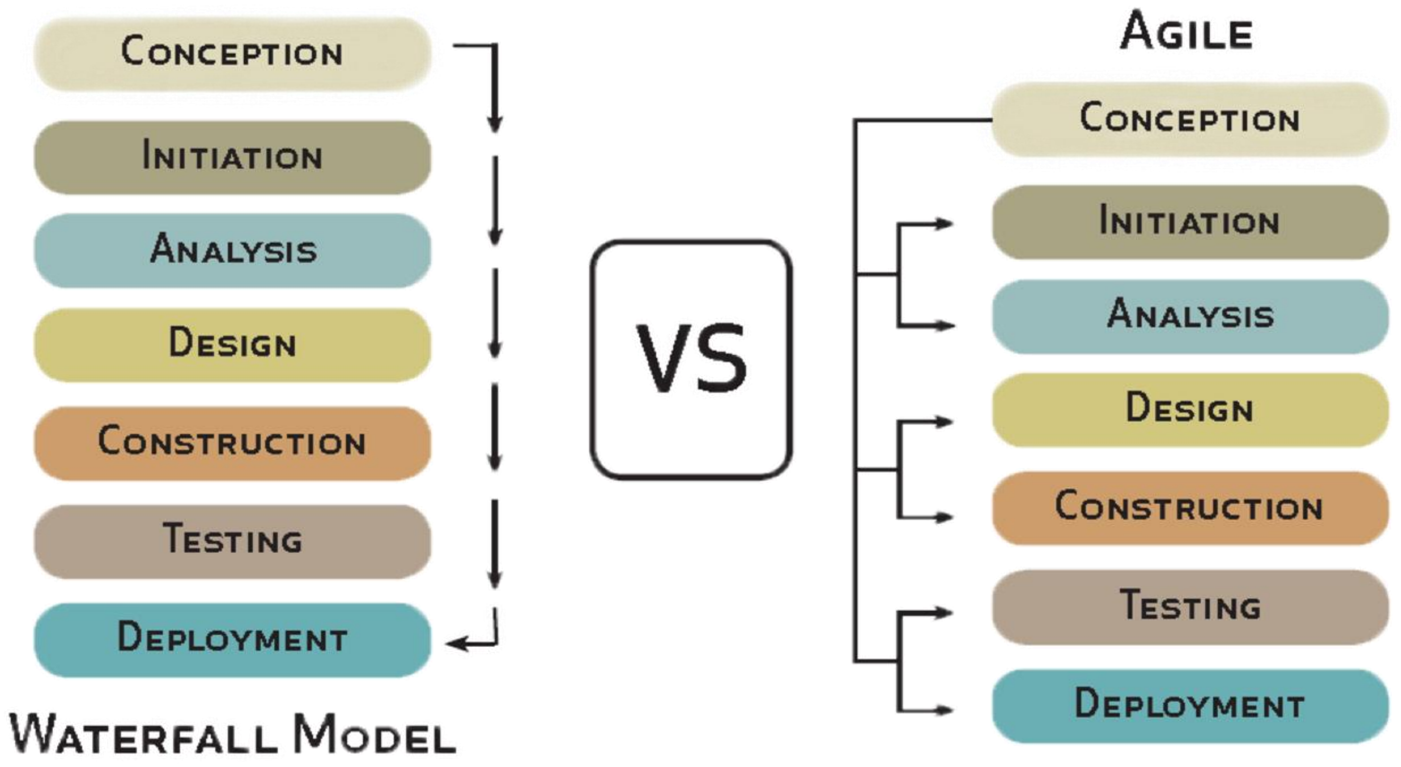
Software development methodologies

03

Software development methodologies

A software development methodology is a series of processes employed in the software development process.

- Waterfall (Big projects).
- Feature-Driven Development.
- Incremental.
- Rapid Application Design (RAD).
- Extreme Programming (XP).
- Scrum.
- Kanban.
- Lean.



Software development methodologies

Waterfall methodology

The **waterfall methodology** is a linear project management approach in which stakeholder and customer requirements are gathered at the outset of the project, followed by the development of a sequential project plan designed to address these requirements.

- It follows a linear, **sequential approach**, where each phase must be completed before the next begins.
- Extensive documentation is created at each stage, providing detailed guidelines and specifications.
- User requirements are gathered upfront and remain fixed throughout the project, making it difficult to adapt to changes.
- Testing occurs after development is fully completed, which can delay the identification of issues.
- It is best suited for **large projects with well-defined**, stable requirements and little anticipated change.
- Time-to-market is typically longer due to the rigid structure and lack of flexibility in the process.

Software development methodologies

Feature-Driven Development methodology

The **Feature-Driven Development (FDD) methodology** is a linear project management approach in which stakeholder and customer requirements are gathered at the outset of the project, followed by the development of a sequential project plan designed to address these requirements.

- It uses an **iterative and incremental approach**, focusing on delivering small, functional features.
- Documentation is **moderate**, focusing on the specific features being developed rather than the entire project.
- Features are developed in short cycles, allowing for regular updates and continuous improvement.
- The user feedback is integrated into the process, ensuring that the delivered features meet stakeholder expectations.
- Team collaboration is key, with feature teams working together to deliver each component effectively.
- Time-to-market is medium, with regular delivery of functional features that contribute to the overall product.

Software development methodologies

Incremental methodology

The **incremental methodology** is a linear project management approach that divides project requirements into distinct, self-contained modules within the software development cycle. Each module progresses through phases such as requirements gathering, design, implementation, and testing. Successive releases of the modules build upon the functionality of the previous releases.

- It involves breaking down the project into **smaller, manageable increments** that are developed **sequentially**.
- Documentation is **moderate**, focusing on the development and integration of each module.
- Each increment undergoes phases such as requirements gathering, design, implementation, and testing, allowing for focused development.
- Feedback is gathered after each increment, enabling adjustments and improvements in subsequent phases.
- Modules are integrated gradually, building upon the functionality of previous increments.
- Time-to-market is medium, as the project is delivered in stages, with each increment adding value to the final product.

Software development methodologies

Rapid Application Design methodology

The **Rapid Application Development (RAD) methodology** is an iterative project management approach that prioritizes design and prototyping to quickly gather user feedback. Continuous cycles of user input and rapid incremental updates are used to refine and improve the final project outcome.

- It emphasizes **rapid prototyping** and iterative design, allowing for quick adjustments based on user feedback.
- Documentation is **less emphasized**, with more focus on building and refining prototypes.
- User involvement is high throughout the development process, ensuring that the final product meets their needs.
- Development cycles are short, with a **focus on delivering functional prototypes quickly**.
- Flexibility is high, with the ability to adapt to changing requirements and refine the product continually.
- Time-to-market is short, making it ideal for projects that require quick delivery and frequent updates.

Software development methodologies

Extreme Programming methodology

The **Extreme Programming (XP) methodology** is an agile software development framework that emphasizes flexibility, collaboration, continuous feedback, and **rapid releases**. It focuses on improving software quality and responsiveness to changing customer requirements through practices like pair programming, test-driven development (TDD), continuous integration, and refactoring.

- It is based on **continuous integration and frequent releases**, promoting adaptability to changing requirements.
- Pair programming and **close collaboration are core practices**, ensuring high-quality code and shared knowledge.
- User feedback is constant, with frequent iterations that incorporate stakeholder input.
- Testing is rigorous and ongoing, with practices like Test-Driven Development (TDD) ensuring that code is reliable and functional.
- Flexibility is high, allowing the project to evolve based on continuous feedback and testing.
- Time-to-market is short, as the methodology emphasizes frequent releases and rapid iteration.

Software development methodologies

Scrum methodology

The **Scrum methodology** is an agile project management approach that focuses on delivering products in small, incremental cycles called **sprints**, typically lasting 2-4 weeks. It emphasizes collaboration, adaptability, and continuous improvement, with clearly defined roles such as Scrum Master, Product Owner, and Development Team.

- It organizes work **into time-boxed Sprints**, typically lasting 2-4 weeks, with each Sprint producing a potentially deliverable increment.
- The team is **cross-functional**, including roles such as Product Owner, Scrum Master, and Development Team, each with specific responsibilities.
- Daily stand-up meetings are held to ensure that the team is aligned and any obstacles are quickly addressed. At the end of each Sprint, a Sprint Review is conducted to gather feedback and a Sprint Retrospective to improve processes.
- Flexibility is high, with the ability to adjust the project backlog and priorities at the start of each Sprint.
- Time-to-market is medium, with regular delivery of functional increments that build towards the final product.

Kanban methodology

The **Kanban methodology** is an agile project management approach that emphasizes iterative development through a series of sprints. **Sprints** are short, time-boxed development cycles that involve structured meetings and focused development activities. Each Sprint yields a tangible, functional increment of the final product, which is potentially deliverable to the client.

- It based on a **visual workflow management system**, often represented by a board with columns for different stages of work.
- Tasks are pulled into the workflow as capacity allows, maintaining a continuous flow of work without fixed iterations.
- WIP (Work In Progress) limits are set for each stage to prevent bottlenecks and ensure that work progresses smoothly.
- User feedback is ongoing, integrated throughout the process as tasks move through the workflow.
- Flexibility is high, allowing for dynamic adjustments based on priorities and capacity.
- Time-to-market is short, with continuous delivery of completed tasks that add value incrementally.

Software development methodologies

Lean methodology

The **lean methodology** is an agile project management approach that emphasizes iterative development through a series of Sprints. **Sprints** are short, time-boxed development cycles that involve structured meetings and focused development activities. Each Sprint yields a tangible, functional increment of the final product, which is potentially deliverable to the client.

- It focuses on eliminating waste and maximizing value, ensuring that resources are used efficiently.
- Continuous improvement (**Kaizen**) is a core principle, **encouraging teams to regularly assess and refine their processes**.
- Work is done in **small batches**, reducing lead time and improving efficiency.
- User feedback is integral, with regular input used to refine and improve the product.
- **Collaboration** across teams is **essential**, promoting shared responsibility and streamlined communication.
- Time-to-market is short, with a focus on delivering value quickly and continuously improving the process.

Software development methodologies

Which methodoly should I choose?

Name	Approach	Flexibility	User feedback	Documentation	Time to market	Team collaboration
Waterfall	Linear	Low	Minimal feedback until the end of project.	Extensive	Long	Structured
FDD	Iterative (feature-based)	Medium	Regular feedback on features.	Moderate	Medium	Collaborative
Incremental	Iterative (module-based)	Medium	Regular feedback after each increment.	Moderate	Medium	Collaborative
RAD	Iterative	High	Frequent feedback during prototyping.	On prototypes	Short	Highly collaborative
XP	Continuous feedback	High	Constant feedback integrated into development.	Moderate	Short	Highly collaborative
Scrum	Iterative (sprint-based)	High	Regular feedback at the end of each Sprint.	Moderate	Medium	Collaborative (roles)
Kanban	Continuous	High	Ongoing feedback from stakeholders.	Minimal	Short	Collaborative (roles)
Lean	Continuous	High	Continuous feedback integrated throughout.	Minimal	Short	Collaborative

From code to CPU

04

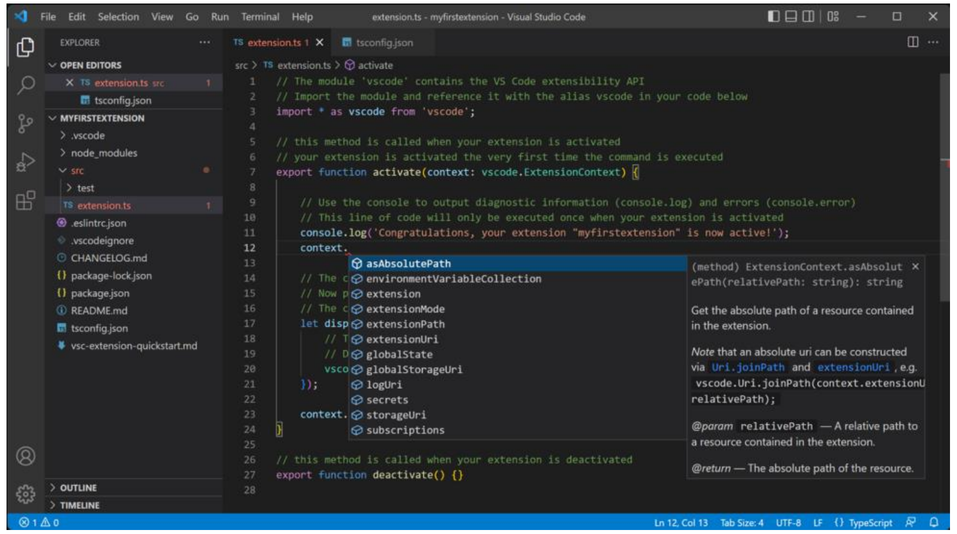
From code to CPU

Software development framework

A software development framework is an abstract structure designed to streamline the software development process.

- Frameworks provide a standardized approach to building and deploying software applications, serving as versatile, reusable environments within a larger software platform.
- Frameworks offer predefined functionalities to simplify and accelerate the development of software applications, enabling developers to focus on the unique aspects of their projects.

An Integrated Development Environment (IDE) is a software application that offers various tools for software development, including a source code editor, build automation tools, and a debugger. IDEs also facilitate the integration of different frameworks.



```
// The module 'vscode' contains the VS Code extensibility API
// Import the module and reference it with the alias vscode in your code below
import * as vscode from 'vscode';

// this method is called when your extension is activated
// your extension is activated the very first time the command is executed
export function activate(context: vscode.ExtensionContext) {

    // Use the console to output diagnostic information (console.log) and errors (console.error)
    // This line of code will only be executed once when your extension is activated
    console.log('Congratulations, your extension "myfirstextension" is now active!');
    context.
             asAbsolutePath
    // The c environmentVariableCollection
    // Now p extension
    // The c extensionMode
    let disp extensionPath
        // T extensionUri
        // D globalState
        vsco globalStorageUri
    });      logUri
             secrets
    context. storageUri
}            subscriptions

// this method is called when your extension is deactivated
export function deactivate() {}
```

(method) ExtensionContext.asAbsolutePath(relativePath: string): string

Get the absolute path of a resource contained in the extension.

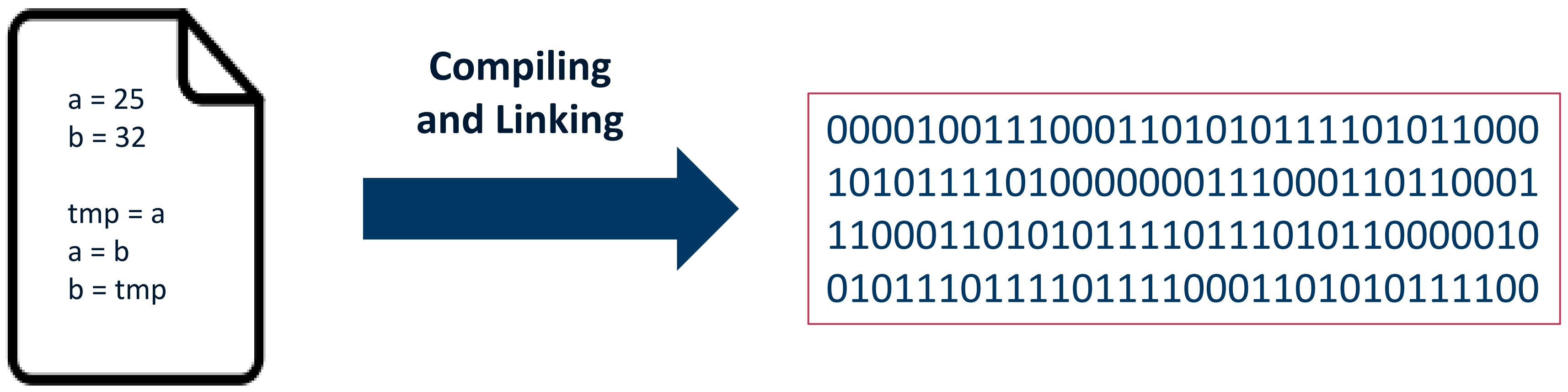
Note that an absolute uri can be constructed via `Uri.joinPath` and `extensionUri`, e.g. `vscode.Uri.joinPath(context.extensionUri, relativePath);`

@param `relativePath` — A relative path to a resource contained in the extension.

@return — The absolute path of the resource.

From code to CPU

Software is typically developed using high-level and/or low-level programming languages. To run this software on a CPU, it must first be compiled into machine code (a set of instructions that the CPU can directly execute).



```
a = 25
b = 32

tmp = a
a = b
b = tmp
```

```
00001001110001101010111101011000
10101111010000000111000110110001
11000110101011110111010110000010
01011101111011110001101010111100
```

High level code

Machine code

Each machine instruction corresponds to a basic operation that the CPU can execute directly.

From code to CPU

Computer languages - Machine

Machine language, also known as machine code or object code, consists of binary digits — 0s and 1s — that are universally recognized and interpreted by computer systems. As the native language of a computer, machine language is directly understood by the central processing unit (CPU), enabling it to execute instructions without any further translation.

Byte

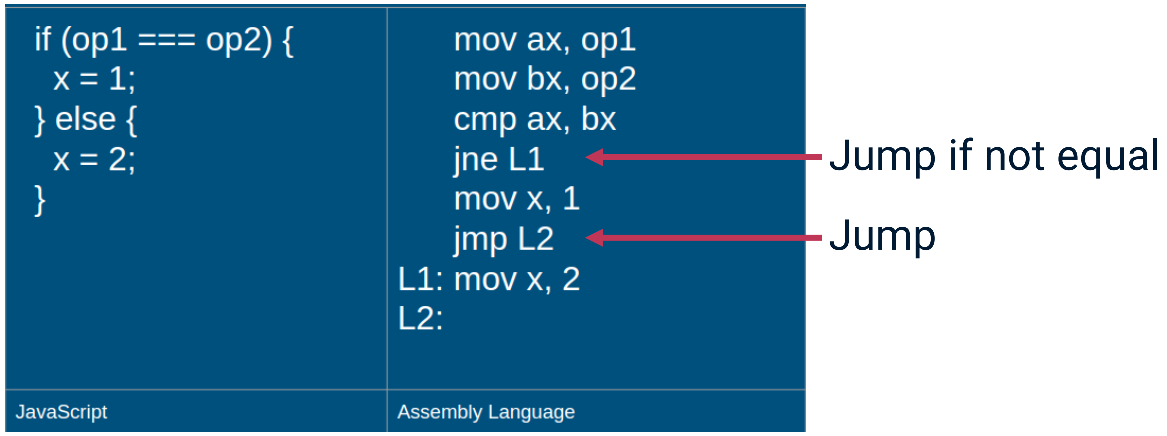
```
10100001 10111100 10010011 00000100
00001000 00000011 00000101 11000000
10010011 00000100 00001000 10100011
11000000 10010100 00000100 00001000
```

This code means: z = x + y;

From code to CPU

Computer languages - Assembly

Assembly language, often referred to as a low-level language, operates at a higher level of abstraction compared to machine language but remains closer to the hardware. It is used as a programming language for microprocessors and other programmable devices. Assembly language involves coding with the direct manipulation of registers, memory addresses, and call stacks, providing greater control over the hardware compared to higher-level languages.



```javascript
if (op1 === op2) {
  x = 1;
} else {
  x = 2;
}
```

JavaScript

```asm
    mov ax, op1
    mov bx, op2
    cmp ax, bx
    jne L1
    mov x, 1
    jmp L2
L1: mov x, 2
L2:
```

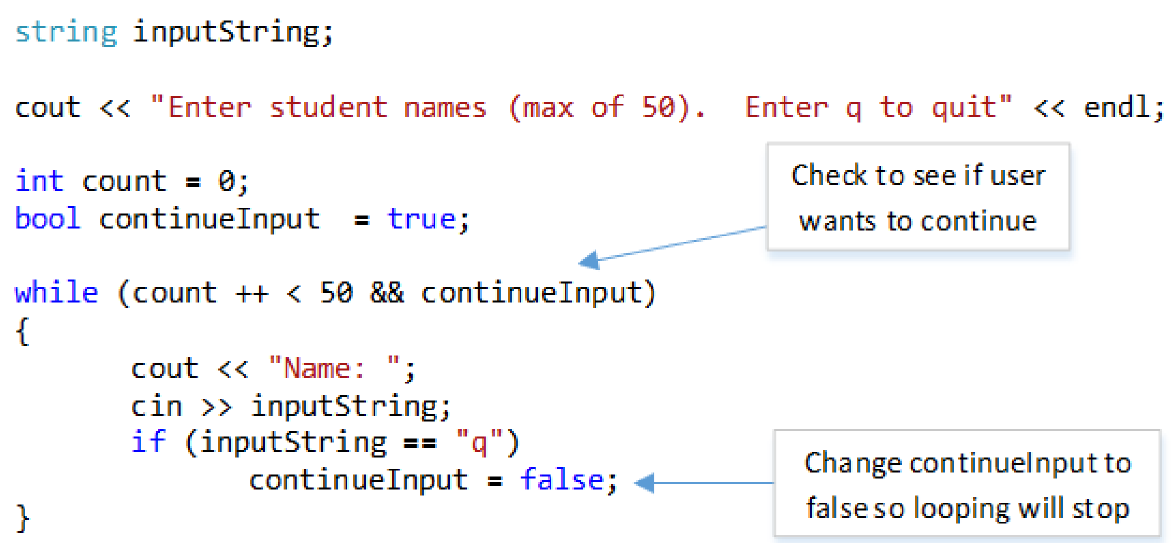
Assembly Language

These languages vary depending on the processor architecture they are designed for, such as Z80, x86, x86-64, IA-64, or AMD64. Each architecture has its own unique assembly language tailored to its specific instruction set and hardware characteristics.

From code to CPU

Computer languages - High level

- High-level languages operate at a greater level of abstraction than machine language, offering programmers a more efficient and user-friendly environment for writing and testing code. These languages include built-in safeguards to prevent the execution of commands that could potentially damage computer systems. However, while they prioritize ease of use and safety, they offer less direct control over hardware compared to low-level languages.
  - C
  - C++
  - C#
  - Pascal
  - Java
  - Python



```
string inputString;

cout << "Enter student names (max of 50).  Enter q to quit" << endl;

int count = 0;
bool continueInput  = true;

while (count ++ < 50 && continueInput)
{
        cout << "Name: ";
        cin >> inputString;
        if (inputString == "q")
                continueInput = false;
}
```

Check to see if user wants to continue
Change continueInput to false so looping will stop

From code to CPU

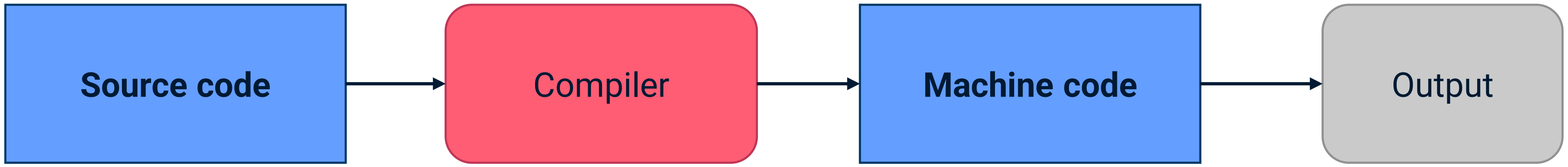
Main differences between computer languages.

High Level Language	Low Level Language
Programmer friendly	Machine friendly
Less memory efficient	Highly memory efficient
Easy to understand for programmers	Difficult to understand for programmers
Simple to debug	Complex to debug comparatively
Simple to maintain	Complex to maintain comparatively
Portable	Non-portable
Machine-independent. They can run on any platform	Machine-dependent
Needs compiler or interpreter for translation	Needs assembler for translation
Widely used for programming	Not commonly used in programming

From code to CPU

Compilers and interpreters

A compiler is a software program that translates code written in a high-level programming language into machine code. Essentially, it serves as a tool that converts human-readable instructions into a binary format — 1s and 0s — that can be directly understood and executed by a computer processor.

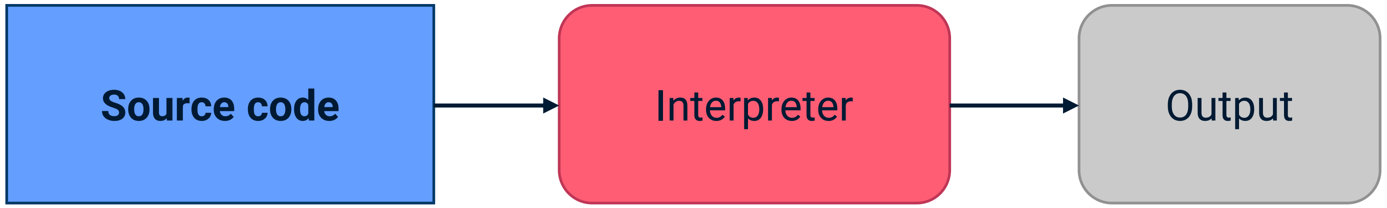


Additionally, compilers convert the code into machine code, generating an executable file (e.g., .exe) that the computer can run. **The compilation process must be completed before the program can be executed**.

From code to CPU

Compilers and interpreters

An interpreter is a software program that translates individual high-level program statements (source code, pre-compiled code, and scripts) into machine code. Like a compiler, an interpreter converts human-readable code into a binary format — 1s and 0s — that a computer processor can understand. However, unlike compilers, interpreters perform this translation on a statement-by-statement basis, executing each line of code as the program runs.



Interpreters translate code into machine code in real-time as the program is being executed.

From code to CPU

UFV

Compilers and interpreters

Dimension	Compilers	Interpreters
Advantage	The program code is already translated into machine code. Thus, it code execution time is less.	Interpreters are easier to use, especially for beginners.
Disadvantage	You cannot change the program without going back to the source code.	Interpreted programs can run on computers that have the corresponding interpreter.
Machine code	Store machine language as machine code on the disk	Not saving machine code at all.
Running time	Compiled code run faster	Interpreted code run slower
Model	It is based on language translation linking-loading model.	It is based on Interpretation Method.
Program generation	It generates output program (in the form of exe) which can be run independently from the original program.	It does not generate output program. So they evaluate the source program at every time during execution.

From code to CPU

UFV

Compilers and interpreters

Dimension	Compilers	Interpreters
Execution	Program execution is separate from the compilation. It performed only after the entire output program is compiled.	Program Execution is a part of Interpretation process, so it is performed line by line.
Memory requirement	Target program execute independently and do not require the compiler in the memory.	The interpreter exists in the memory during interpretation.
Input	It takes an entire program	It usually takes a single line of code.
Output	Compliers generates intermediate machine code.	Interpreter never generate any intermediate machine code.
Errors	Display all errors after, compilation, all at the same time.	Displays all errors of each line one by one.
Pertaining Programming languages	C, C++, C#, Scala, Java use a complier.	PHP, Perl, Ruby use an interpreter.

UFV