Software Design Guide

This document serves as both a tutorial and an example of designing system context and class diagrams in Visual Studio Code (VS Code). You will learn to use Markdown and PlantUML syntax to create well-formatted documents. These tools are commonly used in courses and are widely adopted by companies.

Please read this document carefully, it contains useful information for software engineering. After reading this document, pay special attention to the Markdown, PlantUML and HTML syntaxes in the plain text of this document.

1. Introduction

1.1 Markdown file

This Markdown file can be rendered into a well-formatted document. VS Code provides built-in support for Markdown, making it easy to use right out of the box. To render this file in VS Code, press **F1** and select **Markdown: Open Preview**. The formatted document opens in a new tab. If the diagrams are not visible, you may need to install the PlantUML extension in VS Code. Instructions for installing it are provided in the next section.

A Markdown file is a plain text file with a `.md` extension that uses simple syntax to format content, making it easy to read and write structured documents. Because of its lightweight nature and broad compatibility, Markdown is widely used for documentation, README files, web content, design documents, and notes.

Key Features:

- **Simple Formatting** – Uses symbols like # for headings, **bold** for bold text, and *italic* for italics.
- **Portable** – Can be opened and edited in any text editor.
- **Easy Conversion** – Can be converted to HTML, PDF, or other formats.
- **Popular in Development** – Commonly used in GitHub README files and technical documentation.
- **Version control** – Text format is lesser error-prone for merging changes in GIT.
- **Extensive formatting** – Supports HTML code for more extensive formatting.

This document contains a combination of Markdown, HTML, and PlantUML syntax in the plain text. Please read the formatted tutorial to learn about diagram design. The tutorial will instruct you to look at the syntaxes in the plain text file.

1.2 The PlantUML extension

Markdown supports text and images but does not render diagrams by default. To draw diagrams in a Markdown document, you need to install the PlantUML extension by Jebbs in VS Code. Installing it is simple: Open the Extensions tab in VS Code, type "plantuml" (without quotes) in the search box, find **PlantUML by Jebbs**, and install the extension.

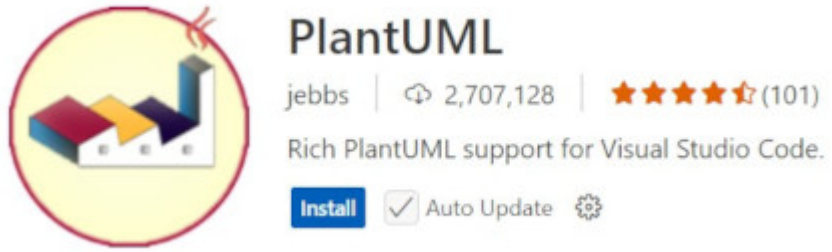

Figure 1. The PlantUML extension in VS Code.

After installing the extension, you must set the renderer in the settings of the extension. The extension has a settings icon, called "Manage". Go to the extension and click on the "Manage" icon and a dropdown menu appears. In the dropdown menu, select "Settings". This opens the settings and scroll down to "Plantuml: render" and select "PlantUMLServer" (without quotes). Then go down to "Planuml: server" and type in "https://www.plantuml.com/plantuml" (without quotes).

After installing the extension, you need to configure the renderer in its settings. To do this, go to the extension and click the **Manage** icon. A dropdown menu will appear and select **Settings** from the menu. In the settings, scroll down to **PlantUML: Render** and choose **PlantUMLServer**. Then, navigate to **PlantUML: Server** and enter "https://www.plantuml.com/plantuml" (without quotes)

After you have set the settings, you are ready to create diagrams in Markdown documents.

1.3 PlantUML Diagrams

PlantUML is a tool that allows you to create UML (Unified Modeling Language) diagrams using a simple text-based syntax. Instead of manually drawing diagrams, you describe diagrams in text and PlantUML generates the visual representation for you. The type of syntax you use defines the type of diagram it will render.

Common Types of PlantUML Diagrams:

- **System Context Diagrams** – Represent I/O interfacing between the software and hardware.
- **Class Diagrams** – Represent relationships between classes in object-oriented programming.
- **Sequence Diagrams** – Show interactions between objects over time.
- **Use Case Diagrams** – Illustrate user interactions with a system.
- **Activity Diagrams** – Represent workflows and processes.
- **Component Diagrams** – Show system architecture and dependencies.
- **State Transition Diagrams** – Show behaviour of I/O interfacing.

I/O interfacing = input and output interfacing

2. System Context Diagrams

Usually, you start a project with a set of requirements, objectives and expectations. This can be written in a project plan. Still, lots of things are be unknown. Starting with coding is a bad idea especially if you don't understand the overall system for which the software needs to be developed. Understanding the system is top priority. In fact, system analysis is required. But how?

In this document, System Context Diagrams are utilized within the domain of embedded systems. The design procedure outlined in this chapter is a concise version, focusing on only a limited number of System Context

Diagrams. Typically, all entities are described and specified, but certain elements have been omitted.

When starting a project, you typically begin with a set of requirements, objectives, and expectations, which can be documented in a project plan. However, many aspects remain unknown at this stage. Jumping into coding too soon—especially without a clear understanding of the overall system—can lead to complications. Before development begins, *understanding the system* is the top priority, requiring thorough system analysis. But how should this be approached?

In this document, System Context Diagrams are applied within the domain of embedded systems. The design procedure presented in this chapter offers a streamlined approach, focusing on a limited number of System Context Diagrams. While entities are usually described and specified in detail, their descriptions and specifications have been intentionally omitted to keep this guide concise

2.1 Purpose

The System Context design is a good approach to get a better understanding of the system. It helps you to see the bigger picture with simple drawings.

A system context diagram is a high-level visual representation of a system and its interactions with peripherals and devices at the edges of the system, or external entities, such as users, other systems, or organizations. It helps defining the system's boundaries and illustrates the flow of information between the system and its environment.

Its primary purpose is to:

- **Define System Boundaries** – It helps stakeholders understand what is inside the system and what interacts with it externally.
- **Clarify Interactions** – It visually represents the flow of data or interactions between the system and external entities.
- **Aid in Requirement Gathering** – By showing how the system fits into its environment, it helps identify functional and non-functional requirements.
- **Facilitate Communication** – It serves as a bridge between technical and non-technical stakeholders, ensuring everyone has a shared understanding of the system's scope.
- **Identify Risks and Dependencies** – It highlights external factors that may impact the system's performance, security, or reliability.
- **Defining low-level functions** – It defines the low-level functions that the software uses to interact with the external system, helping to shape the software from the bottom up, as opposed to the top-down approach.

System Context Diagrams are an essential tool in *system analysis*, particularly when little is known about a system. As developers, the system's context is often the first aspect we encounter—it defines what exists, what matters, and how components interact. These diagrams focus on the flow of data or messages (represented by labels on arrows) between the core system (to be developed) and its terminals (existing components). By using a System Context Diagram, you can identify interactions between the system core and the boundaries of its context (terminals). These diagrams clarify how a system integrates into its environment, facilitating communication between technical and non-technical stakeholders. They also aid in risk identification by highlighting dependencies and external influences

If you're working on system design or analysis, a System Context Diagram can be an invaluable tool for structuring your approach. You can read more about System Context Diagrams on geeksforgeeks and

Wikipedia.

2.2 Example – The Elevator system

Let's take a closer look at the elevator system in a building. To understand its functionality, it is useful to analyse the system by designing System Context Diagrams.

We begin by examining the fundamental physical components, including the **cabin**, **shaft**, and **multiple floors**. Consider the broader context of the elevator, as illustrated in Figure 2.

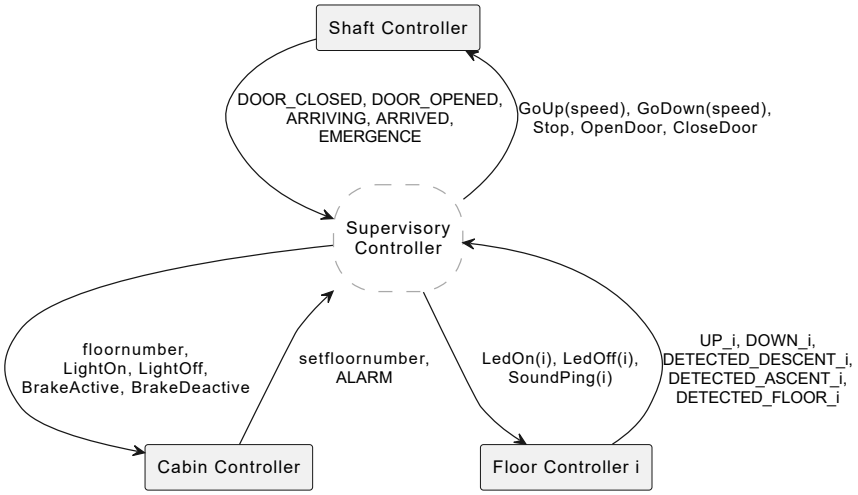

Figure 2. The Main context diagram.

In the figure, the dashed circle in the center represents the core of this context—the part for which the software must be developed. This circle encapsulates the internal processes. The **Supervisory Controller** entity manages top-level processes, ensuring safety and efficiency.

The **Shaft Controller**, **Cabin Controller**, and **Floor Controller *i***—collectively known as terminals—define the boundary of this scope. Terminals can be peripherals, devices, subsystems or users. Peripherals include sensors, actuators, and storage components, while devices refer to equipment such as printers, mice, keyboards, or machines. The arrows illustrate the flow of data or information between the core system and its terminals. In embedded systems, terminals are hardware-related components, so users are not included in the system context.

The System Context Diagram depicted in Figure 2. illustrates distinct types of data flow interactions:

1. **values** – setfloornumber, floornumber
2. **states** – DOOR_CLOSED, DOOR_OPENED, ARRIVING, ARRIVED, EMERGENCY, ALARM, UP_ *i*, DOWN_ *i*, DETECTED_DESCENT_ *i*, DETECTED_ASCENT_ *i*, DETECTED_FLOOR_ *i*
3. **actions** – Stop, OpenDoor, CloseDoor, LightOn, LightOff, BrakeActive, BrakeDeactive
4. **actions with values** – GoUp(speed), GoDown(speed), LedOn(*i*), LedOff(*i*), SoundPing(*i*)

Here, *i* is the floor number.

The actions `GoUp(speed)` and `GoDown(speed)` require the parameter `speed`. This parameter enables smooth acceleration and deceleration, preventing harse starts and stops of the cabin, which could be uncomfortable or even harmful for passengers. As demonstrated in the System Context Diagram, this approach allows for a detailed examination of specific behaviors related to system interactions.

By carefully analyzing these messages, you can translate them into variables, enumerations, and functions within your code. We will explore this in more detail later. When designing System Context Diagrams, it's essential to step back from coding and focus purely on the design process until you are fully satisfied with the entities and the structure.

The terminals in Figure 2 represent subsystems, which can also be designed using System Context Diagrams. Figure 3 illustrates the context of the **Cabin Controller**, where the **Cabin Controller** becomes the dashed circle, signifying the core system. In this context, the **Supervisory Controller** shifts to the role of a terminal.

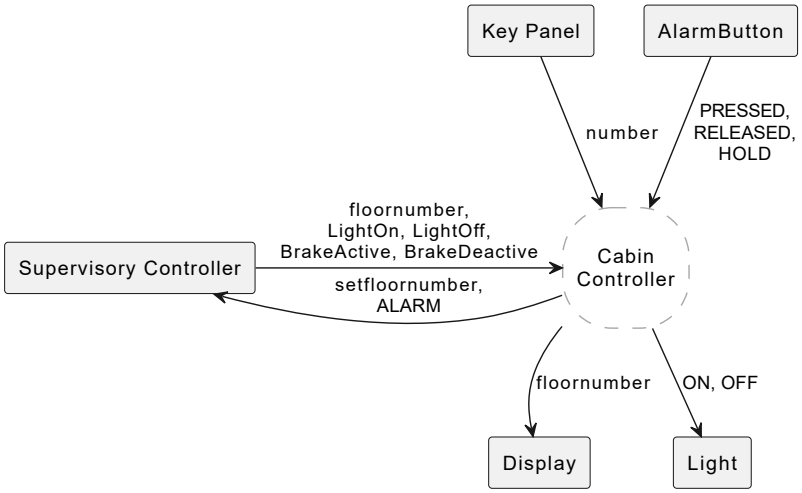

Figure 3. The Cabin Controller context diagram.

Take note that the data exchanged between the **Supervisory Controller** and the **Cabin Controller** in Figure 2 and Figure 3 remains consistent. Maintaining consistency across different System Context Diagrams is crucial, as it ensures a coherent and accurate representation of system interactions.

The System Context Diagram of the **Shaft Controller** is shown in Figure 4. and the System Context Diagram of the **Floor Controller *i*** is shown in Figure 5.

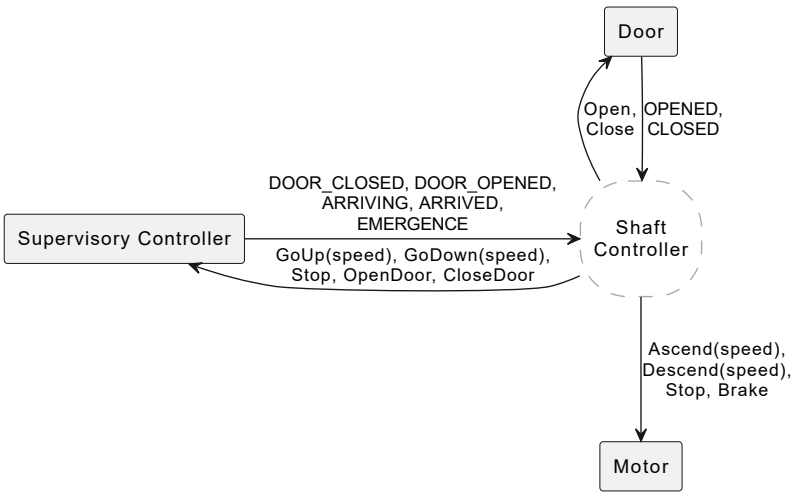

Figure 4. The Shaft Controller context diagram.

Figure 5. shows 3 detectors at each floor *i*. The "Detector Cabin Ascent *i*" is the lower sensor that detects the cabin coming from below. This detector is used to slow down the motor a graceful. The "Detector Cabin Descent *i*" is the upper sensor that detects the cabin coming from above. This detector is used to slow down the motor to a graceful stop. The "Detector Cabin Floor *i*" detects if the cabin is at the floor so that the door can be opened and passengers can step in or out.

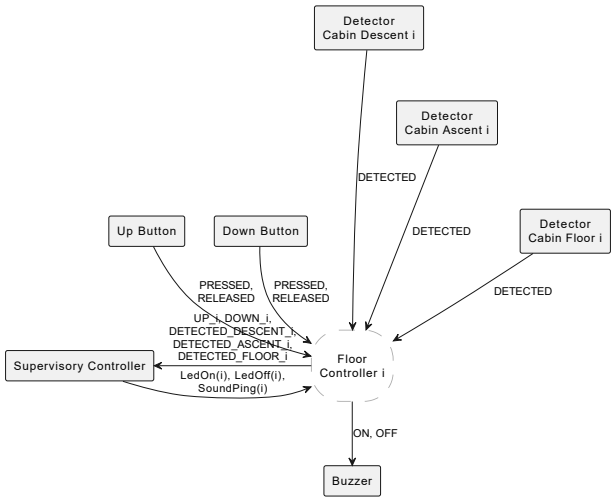

Figure 5. The Floor Controller context diagram.

This System Context design was developed through multiple iterations, utilizing four System Context Diagrams to refine structure and interactions. Through the design process, the designer gained a much deeper understanding of the contexts that define the Elevator System. This iterative approach was essential in shaping a well-structured and accurate representation.

2.3 Implementation

The data flows are labeled and represent values, states and actions. They define variables, enumerations, structs, functions and data types. The functions are boundary functions. They are low-level functions that are sometimes called *device driver functions*. These functions are not the lowest-level functions which are I/O functions. Device driver functions typically encapsulate the I/O functions responsible for hardware interfacing. Other boundary functions are dedicated to interfacing subsystems to establish a communication protocol; they are low-level in the protocol. The system context diagrams helps with resoning about these boundary functions and they contribute in shaping the embedded software from bottom-up.

The labeled data flows represent values, states, and actions, defining variables, datatypes, enumerations, structs, and functions. The functions within this system are boundary functions—low-level functions often referred to as device driver functions. These functions encapsulate the I/O functions responsible for hardware interfacing but are not the lowest-level functions themselves. Additionally, boundary functions can also facilitate subsystem communication by establishing protocols at a low level. They are essential in defining the canonical low-level functions that directly control the hardware, ensuring structured and efficient system integration.

In the next section on Class Diagrams, the classes and their functions are derived from the System Context Diagrams, ensuring a structured approach to system design.

3. Class Diagrams

In object-oriented programming languages, such as C++, classes serve as the foundation of object-oriented programming (OOP). They provide a structured way to define and organize data and behavior within a program.

Key Purposes of Classes in C++

- **Encapsulation** – Classes bundle data (attributes) and functions (methods) together, restricting direct access to data and ensuring controlled interaction.
- **Reusability** – Once defined, a class can be reused to create multiple objects, reducing redundancy in code.
- **Abstraction** – They allow programmers to hide complex implementation details and expose only necessary functionalities.
- **Inheritance** – Classes enable the creation of new classes based on existing ones, promoting code reuse and hierarchical relationships.
- **Polymorphism** – They support dynamic behavior, allowing objects to be treated as instances of their parent class, enhancing flexibility

The code structure is built around classes and their relationships, a topic that will be explored in next year's lectures. While this section does not cover class relationships in detail, their presence will still be noticeable, potentially sparking curiosity and motivation to learn more about them.

Class diagrams are used in software engineering to visually represent the structure and relationships of classes within a system. They help developers and designers understand how a system is organized and how its components interact. Classes are represented as rectangles labeled with their class names, containing their **public functions**. These public functions define the class's interface, specifying how other components interact with it.

Typically, classes represent entities within the system. In System Context Diagrams, terminals can be described as classes and later structured within a Class Diagram. Looking at Figure 3, these terminals can be modeled as classes, with their data flow translated into member functions that define their interactions. The actions defined by the data flow can be directly translated into functions, ensuring a structured and logical implementation within the system. These functions encapsulate low-level implementations, bridging high-level design with concrete functionality. Variables must be carefully translated into functions to ensure they serve the intended purpose within the system. Functions that represent variables require prefixes, such as `Read..`, `Write..`, `Get..`, `Set..`, `Is..`, and `Has..`. These functions can have parameters, return values, or both, depending on their role within the system. Parameters allow functions to provide outputting data from the core to the terminal, while return values provide inputting from the terminal to the core.

The classes shown in Figure 6. are derived from the System Context Diagram, but they can also be identified from different perspectives.

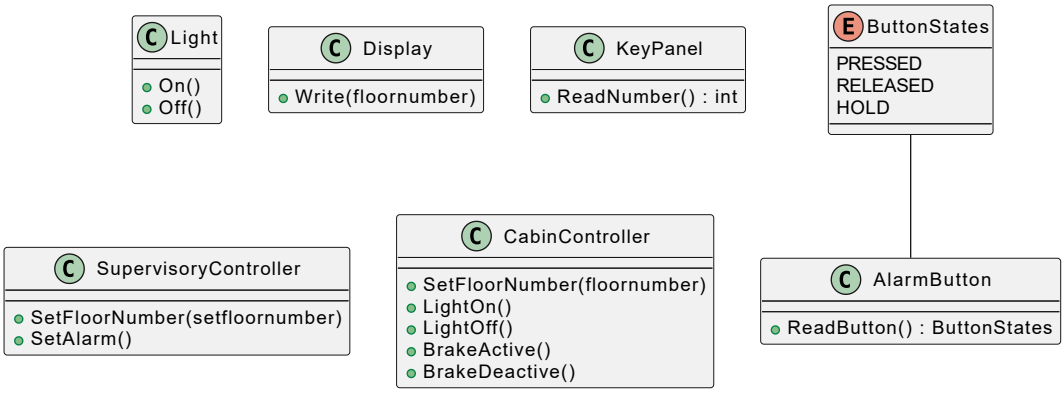

Figure 6. Class diagram.

The suffices `: ButtonStates` and `: int` denote return types. It's different notation than in C++. A function without suffix is similar to `: void`, meaning that there is no return type.

Figure 7 provides an overview of class relationships. While you may not yet be familiar with these notations, this overview serves to illustrate the types of relationships that can be defined between classes. You may find some of them useful when designing your own Class Diagrams.

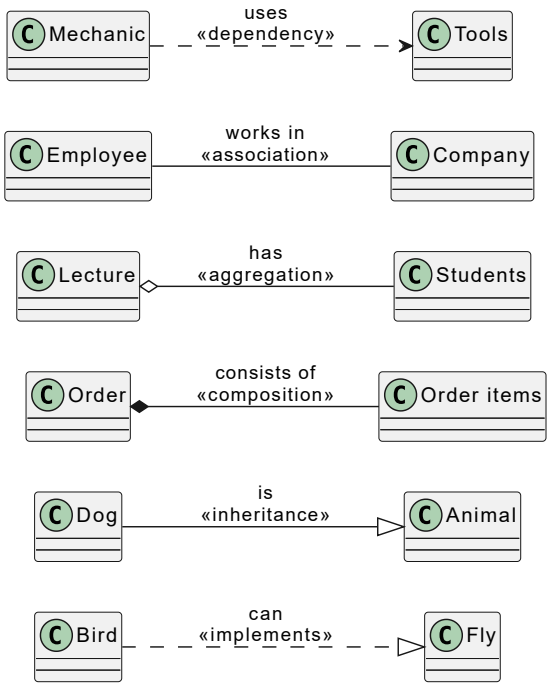

Figure 7. Class diagram with relationships.

4. Examine the formatting

After introducing System Context Diagrams and Class Diagrams, we will now explore the Markdown, HTML, and PlantUML syntaxes in this text file. Markdown provides a more efficient way to format text compared to HTML, and it is also easier to use than LaTeX. PlantUML, which relies on GraphViz, is a powerful graphics rendering tool available as an extension in VS Code. GraphViz automatically generates an optimal topology of shapes and arrows. HTML is used where Markdown and PlantUML fall short.

You have reviewed the formatted document and are aware of the locations of its diagrams. Now, switch to the Markdown file tab to view the plain text. Scroll from top to bottom and examine the PlantUML descriptions. Take a moment to study them—presumably, reading the code should not be too difficult.

Tip: You can drag the Preview tab into a separate window alongside the editor. This allows you to edit Markdown content while simultaneously viewing the rendered output. Any changes made in the Markdown file will automatically update in the preview, making it a highly efficient workflow.

The PlantUML descriptions start with:

```plantuml
```

and ends with

```
```

The ` key on your keyboard is below the ESC key on a US keyboard layout. This character is different from the ' key.

Take a moment to examine the Class Diagram and review its PlantUML description. In PlantUML syntax, the prefix `+` denotes public members, while `-` represents private members. The `+` symbol is used to specify functions that define the class interface, ensuring structured interaction within the system.

More information about the syntax can be found on https://plantuml.com/, or search the internet. A PlantUML description starts in Markdown with

In VS Code, single diagrams can be created by adding files with the `.plantuml` extension instead of using Markdown. Each PlantUML file must begin with

```
@startuml
```

and end with

```
@startuml
```

Remarks

Keep in mind that design is never purely top-down. It is typically a dynamic combination of top-down, bottom-up, and middle-out approaches, all working together to achieve a complete design. Abstraction and the ability to map abstraction to realization are essential. Focusing solely on implementation can be inefficient —time-consuming, costly, and complex.

The System Context Diagram serves as an intuitive paradigm, acting as glue logic between abstraction and realization through a simple graphical language. Class Diagrams play a crucial role in defining classes and their interfaces, where the public functions establish how objects (instances of classes) can a interacts within each other.