Pie: A General-Purpose Code Editor  
Focused on Simplicity

**1. Brief Introduction**

The project called “pie” is a code editor I have been working on since March 2023. It will be also used for my diploma project and it is intended to be a competitor to other mid-range text/code editors such as Notepad++, but with a much more intuitive design, less confusing features and additional capabilities, essential for software developers.

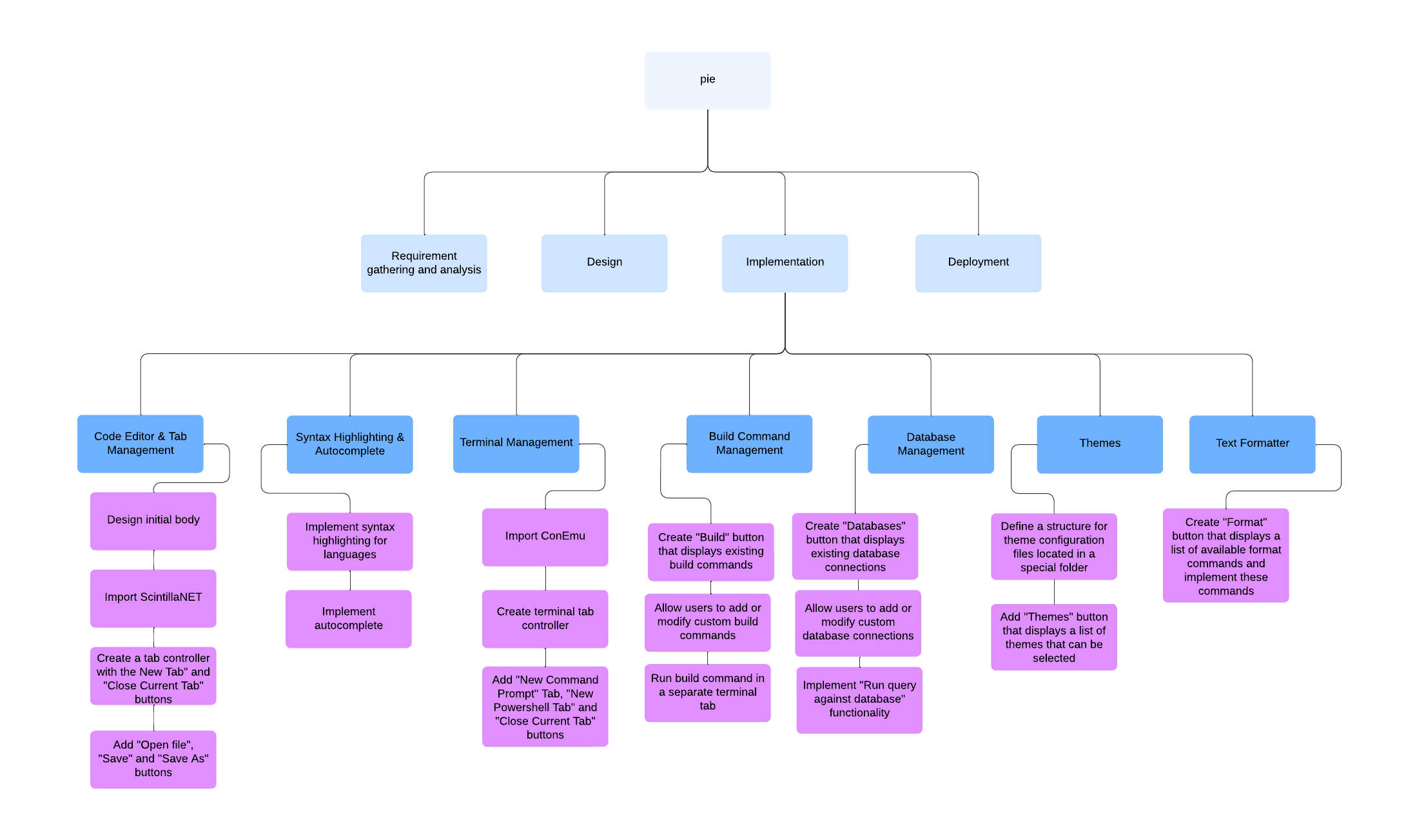
**2. Requirements**

* Pie should look simple and clean: make sure all of the settings are accessible and visible to the novice user, but also make sure they don’t interfere with the user’s primary area of work.
* Pie should allow creating a new file, or modifying an existing file.
* The code editor should allow handling of multiple files at the same time. The user should be able to navigate between them.
* Pie should provide syntax highlighting and autocomplete for the most used programming languages.
* The user should be able to open (multiple) terminal instances that are integrated inside the application. Thus, the user won’t need to open an external terminal window.
* Pie should be able to save and run custom user commands.
* Users should be able to connect to databases and run SQL query against them.
* The code editor should be customizable, have multiple pre-defined themes and also let users configure their own.
* We want Pie to also have several text formatting capabilities, in order to reach the same usability level (or above) with software from competitors, such as Notepad++.

**3. Specifications**

1. Implement a code editor and a tab management system
   1. Design the initial body of the application.
   2. Import ScintillaNET code editor wrapper.
   3. Create a tab controller and add “New Tab” and “Close Current Tab” buttons
   4. Add the “Open file”, “Save” and “Save as” buttons.
2. Implement syntax highlighting and autocomplete in the code editor
   1. Implement syntax highlighting for the following languages: C, C#, Java, Python, Lua, SQL, XML, HTML, JavaScript, JSON and Markdown.
   2. Implement autocomplete functionality for the words defined in the syntax highlighting task.
3. Implement a terminal management system
   1. Import the open-source ConEmu wrapper module inside the project solution.
   2. Create a terminal tab controller that can be toggled from a button in the editor.
   3. Implement the “New Command Prompt Tab”, “New Powershell Tab”, and “Close Current Tab” available by right clicking any tab from the terminal tab controller.
4. Implement a build command management system
   1. Create a “Build” button displaying a list of the available custom build commands, read from a .json file.
   2. Allow the user to add custom commands or modify existing ones
   3. Run the selected build command in a separate terminal tab.
5. Implement a database management system
   1. Create a “Databases” button displaying a list of the available database connections read from a .json file.
   2. Allow the user to add custom database connections or modify existing ones.
   3. Implement the “Run query against database” button, available in tabs containing .sql files. The query should be run against the chosen database connection and the result (if any) should be displayed in a tabular structure, in a separate form.
6. Implement a theme system that also allows adding custom themes
   1. Create the “themes” folder containing multiple .json configuration of different themes. Define a proper structure for such configuration files.
   2. Implement the “Themes” button that displays all of the available themes (by reading the “themes” folder and provides a way of selecting between those themes).
7. Implement text formatting features
   1. Create the “Format” button and make use of C#’s string parsing methods in order to implement the most common and necessary text formatting features inside Pie. The formatting options should be displayed in a popup form containing a list. An option can be chosen on double-click.

**4. WBS (Work Breakdown Structure)**



Shareable edit link: <https://lucid.app/lucidchart/d5676293-db6e-47c2-a011-b3a235c10b01/edit?viewport_loc=-2420%2C-1178%2C3700%2C1794%2C0_0&invitationId=inv_8f37d911-a1e2-4416-bb3a-e96d1fe454b1>

**5. Gantt Diagram**

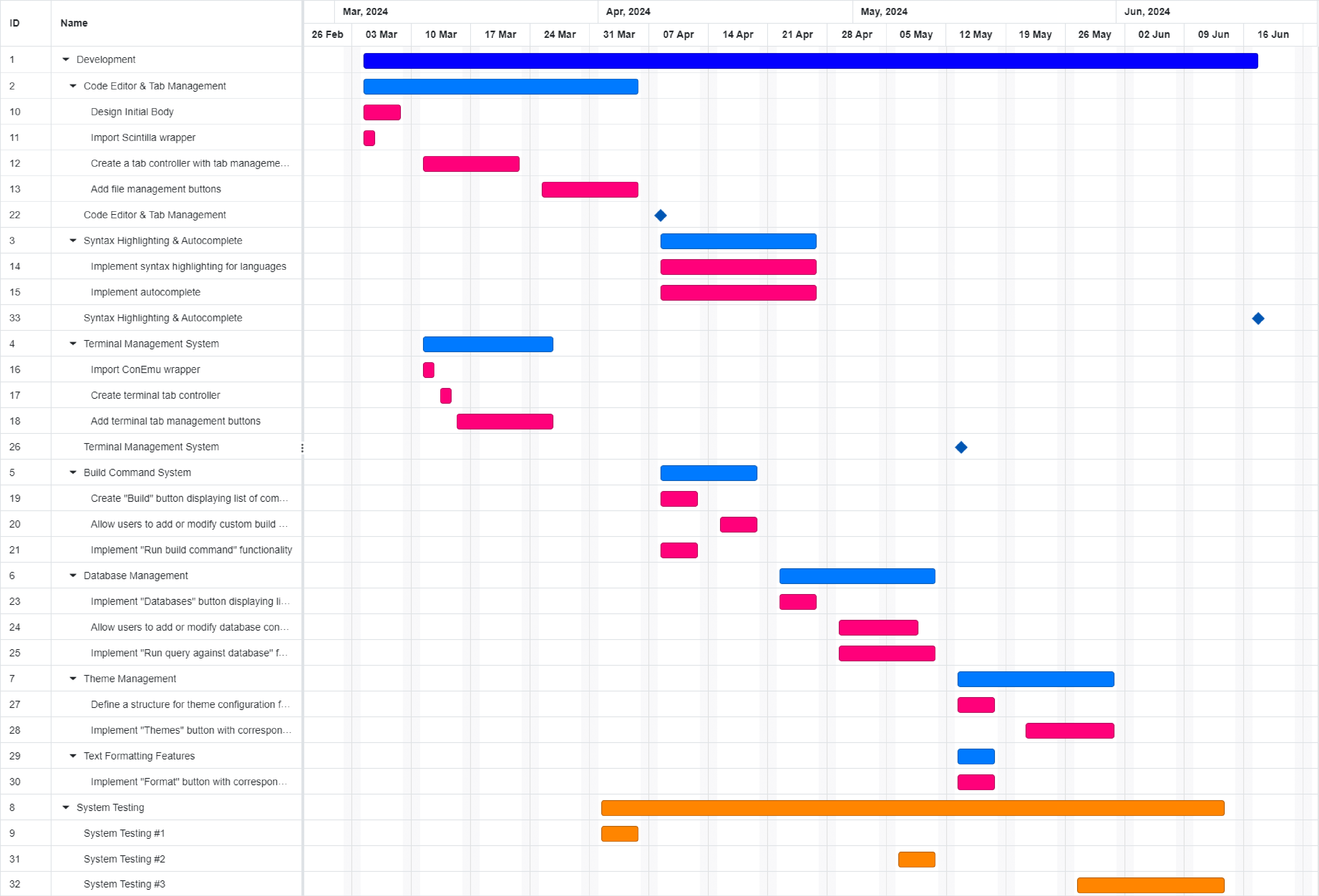
We will develop our software project in an Agile manner. The project development process will consist of 3 milestones, coming from 5 to 5 weeks.

During the first 5 weeks, we will start by developing the essential aspects of our product: the **code editor and tab management** system, and the **terminal** system.

In the next 5 weeks, we will work on developing the **syntax highlighting & autocomplete** features, together with the **build command management** and the **database management** system.

The **formatting** and **customization** logic will be left for the last 5 weeks of the development period.

Testing will be done after **every** feature added to Pie. System testing (testing all of Pie’s functionalities together) will be done every last week of each milestone, as all of the elements are tightly coupled and modification of one component can affect other components.



**6.1. Software Testing: Introduction**

*Software Testing* represents the process of verifying and validating a software solution. It makes sure that the software contains no (or a minimal amount of) bugs and errors, while it also ensures that the system is reliable and it meets the client’s requirements. Software Testing represents one of the most important steps in the Software Development Lifecycle (SDLC), and it should be prioritized as much as the development process itself.

As stated in the first paragraph, Software Testing is a combination of *verification* and *validation*. *Verification* ensures that the product is built right, meaning that the software doesn’t have any implementation errors and it functions as intended. *Validation* ensures that the right product is developed. It makes sure that the solution does what it was intended to do, in a business manner.

There are two methods that need to be taken into consideration while testing a software product:

1. Black-box testing
2. White-box testing

*Black-box testing* is done without knowing or looking at the actual implementation of the project. The only aspects that need to be taken into consideration are the inputs and the outputs are the module. There are several techniques of black-box testing, including equivalence partitioning testing, boundary testing, state transition testing, performance testing and smoke testing.

*White-box testing*, in opposition to black-box, assumes that the implementation of the software system is known. Thus, all of the tests are done while analyzing the code, instead of ­just looking at the output of the module. White-box testing types include static analysis, dynamic analysis, and unit testing – the only white-box testing technique we are going to use for our project.

Unit testing usually relies on independently testing a single module (or functionality) of our project, in order to make sure it does what it was intended to do (logically). After carefully testing the module, we can move further to integration testing, in order to check if the module is properly going to be integrated with its dependencies.

Other types of testing that are going to be used in the *testing phase* of our product (that overlaps at the beginning with the development phase) will include *alpha* and *beta* testing, which can be considered internal and external subtypes of user acceptance testing (UAT).

1. Alpha testing involves testing the software product from the user’s perspective, but the users are still part of the organization that worked on developing the product.
2. Beta testing is similar to alpha testing, but it is conducted by real users, in a less-controlled environment.

**6.2. Unit Testing and Load Testing of User Interface Components**

We are developing Pie in several sprints. The initial development will last 15 weeks (5 weeks for each sprint), with the developers prioritizing the essential features of the application. The **code editor** and the **terminal system** will be the first features to be implemented, because the upcoming features rely heavily on them.

As it can be seen from the Gantt diagram, we will split each of the functionalities into several subtasks. For each subtask, there will be several test scenarios created. After each implementation of a task, the developer needs to ensure that the component works as intended.

In order to minimize the number of errors present in our solution, the developers will be required to write **unit tests** after each finished task (the two tasks “Import Scintilla wrapper” and “Import ConEmu wrapper” can be ignored in this case). The unit tests will ensure that the logic executed within a module is correct. The developer should cover positive cases in their unit tests, but should also think about exceptional cases that can occur while executing the logic. Such cases should also be tested.

Pie’s code consists mostly of GUI element handling. It also features several services that process data, but these will be presented later. Microsoft provides a way of testing Windows Forms (the framework used for building Pie) user interface elements by writing code (<https://github.com/microsoft/WinAppDriver>), meaning that such tests do not need to be executed manually by looking at the test case specification. They can be defined inside specialized test classes and integrated inside the build process of the application.

Taking as an example our first feature, “Code Editor & Tab Management”, the whole logic of the modules integrated here will heavily rely on .NET’s user interface API, meaning that most of the unit tests will be automated clicks on the tab control buttons (“New Tab”, “Close Current Tab”) and key presses inside the Scintilla code editor (while asserting that the user input is actually present inside the textbox).

Load testing can (and should) also be done using UI automation. Load testing examines how the system behaves during high loads. In our case, also taking into consideration the first feature, high loads mean multiple tabs having multiple files opened. We can assume that in a worst case scenario, a user has to work on 20-30 files at the same time. We can create files of different sizes in our development environment and automate clicking the “Open file” button inside Pie, in order to check the reliability of our software solution. While having such a huge number of files open, we should check the response times of other UI elements that can be toggled, such as the Find & Replace dialog (accessible through CTRL + F), the terminal tab manager (accessible through CTRL + B), and the Git user interface (accessible via the tool strip menu). We will also set a maximum allowed response time for these GUI elements. After each feature implemented, the user should make sure that these response times are kept inside the defined boundaries. If not, some code needs to be rewritten.

**6.3. Unit Testing of Services**

Pie has certain services that do not rely on its user interface elements. Such services focus on handling and processing data, instead of operating on UI elements. The services present inside Pie are:

* File handling service (for handling .json configuration files)
* Formatting service (for formatting the content present in the code editor textbox)

We can write unit tests for these services, without needing to automate any user interface interaction. Starting with the **file handling service**, we can programmatically create a .json file containing general configuration of pie during the setup phase of our unit test class. This file should be automatically deleted after the execution of all test cases present inside that class. The test cases for the file handling service should run all of the service’s public methods, giving the file reference as a parameter. Those methods should return an object containing the configuration parameters that will be used by our application. We should write several assertions, in order to check that the file has been parsed properly and that the object has set the correct values

The **formatting service** simply accepts the content of the Scintilla code editor as an input (a string), and returns the formatted output of that value. Formatting tasks can include: removing duplicate lines, removing lines consisting of only whitespaces, inverting the order of the lines, sorting the lines alphabetically in an ascending order, and so on. These formatting options have been integrated inside Pie in order to provide the user an easier and faster way of manipulating text. Because the formatting service doesn’t directly rely on any UI elements, we can write several unit tests where we define strings containing certain values, that are given as inputs to our methods. We can then check if the service returned the proper output.

**6.4. System Testing**