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.

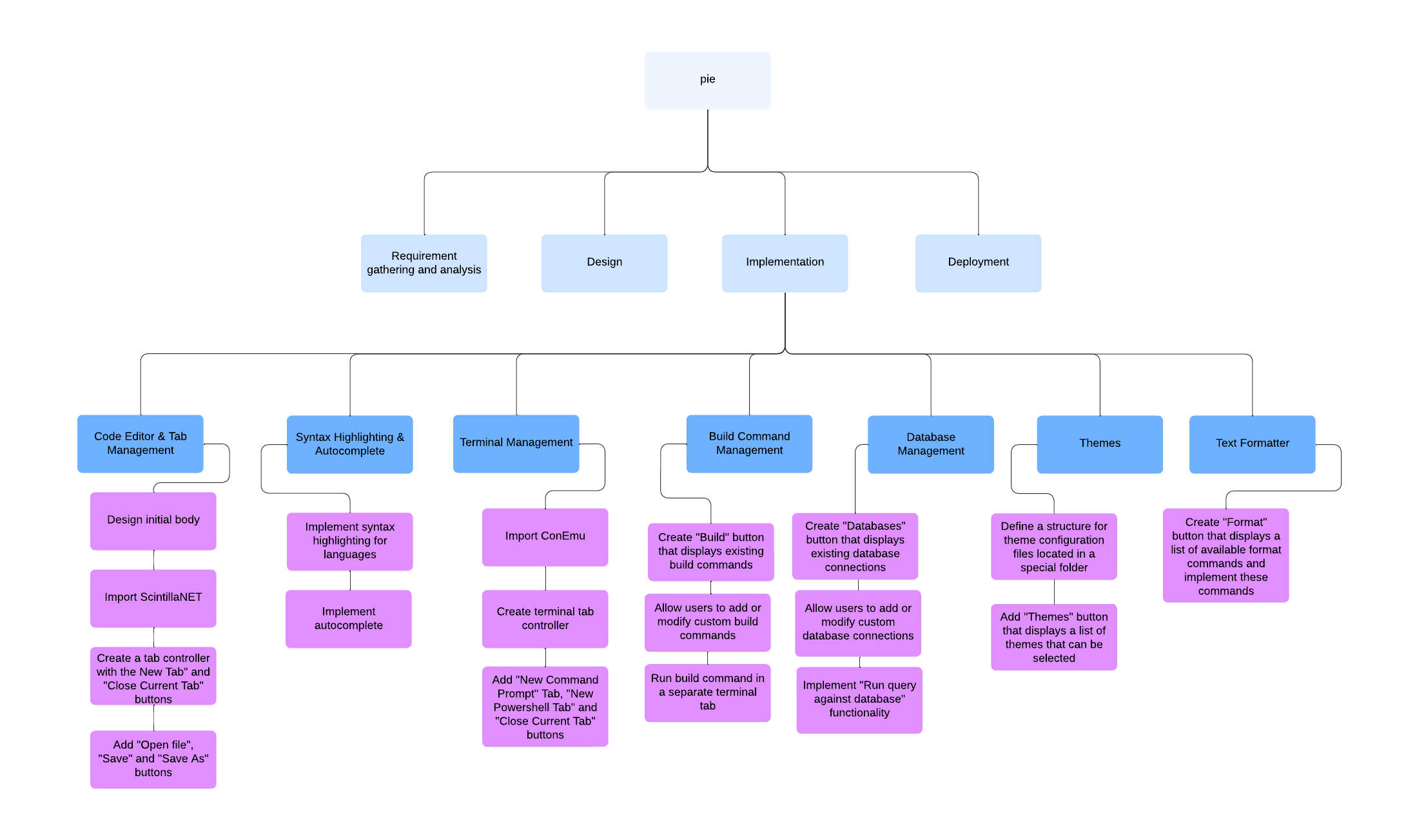
**2. Requirements**

* 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 files of different themes. Define a proper structure for such 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**

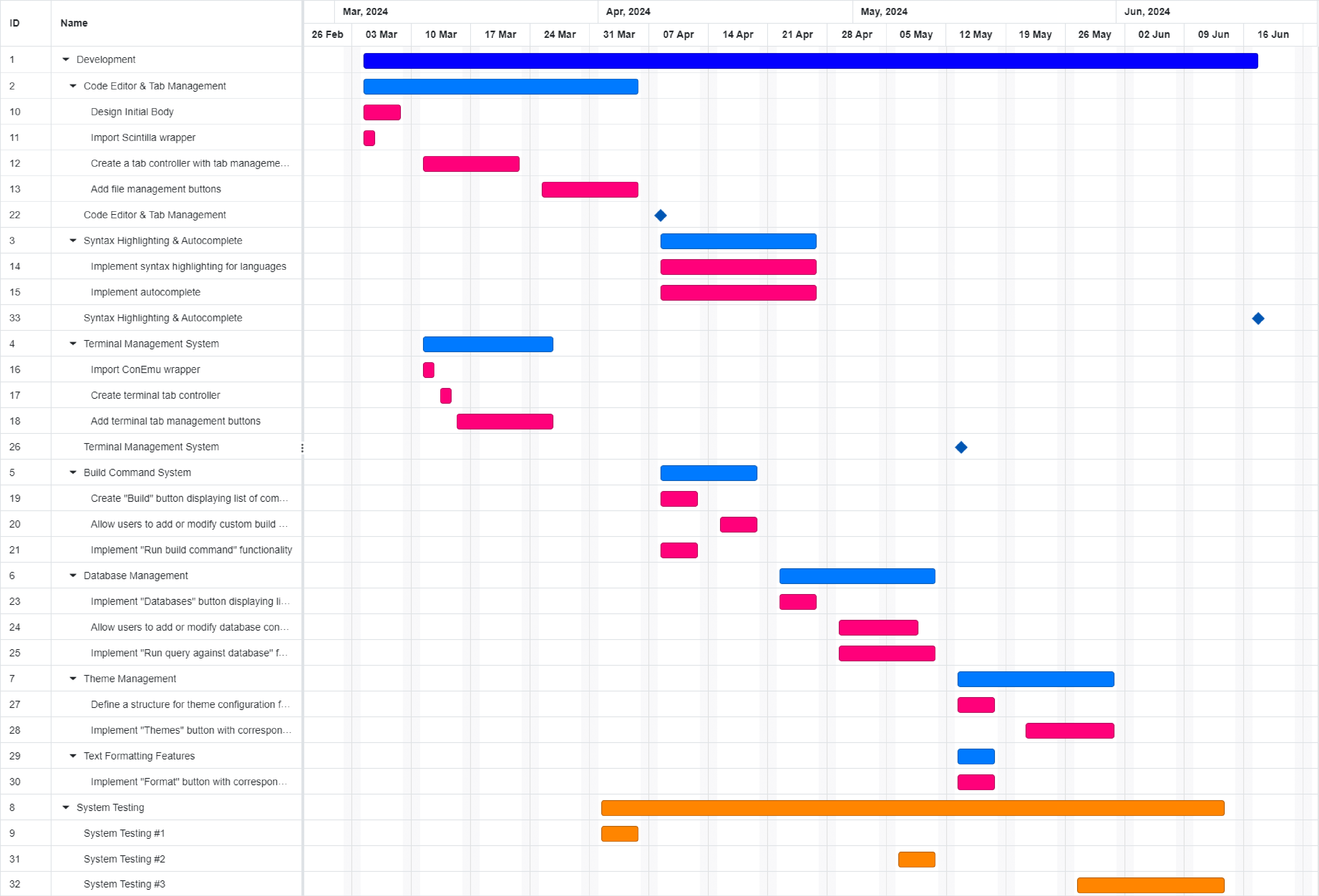
We will develop our software project in an Agile manner. The project development process will consist of 3 sprints, coming from 5 to 5 weeks.

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** - and **database management system**.

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

Unit and integration testing will be done after **every** new feature added to Pie. System testing (testing all of Pie’s functionalities together) will be done at the end of each sprint (during the 5th week), 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 implementation 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.

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 over the actual implementation of the project. The only aspects that need to be taken into consideration are the inputs and the outputs of 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 created in relation to the code. White-box testing types include static analysis, dynamic analysis, and unit testing – the testing technique we are going to use the most in our project.

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

Other types of testing that are going to be used during the testing phaseof our product (that overlaps at the beginning with the implementation phase) 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 owns the product. In a preferable way, the testers should not have contributed to the development of 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 during several sprints. The initial development process 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 other 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 make sure that the component works as intended.

In order to minimize the number of errors present in our application, the developer will write **unit tests** after each implemented task (the 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 business logic. Such cases should also be verified.

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 to be implemented, “Code Editor & Tab Management”, the entire logic of the modules that are part of this feature will rely on .NET’s user interface API, meaning that most of the unit tests will consist of 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 files opened at the same time inside Pie. We can assume that in the worst-case scenario, a user has to work with 20-30 files at the same time. We can create files of different sizes in our development environment and automate clicks on the “Open file” button, in order to check the reliability of our application. 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 new implemented feature, the developer 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 specifically rely on the user interface. Such services focus on handling and processing of data, instead of operating on UI components. The services available inside Pie are:

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

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 a reference to the created file 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 returned object has the correct values set.

The **formatting service** simply receives the content of the Scintilla code editor as an input, as a string, and returns the formatted output of that value. Formatting tasks include: removing duplicate lines, removing lines consisting of only whitespaces, inverting the order of the lines or sorting the lines alphabetically in an ascending order. 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 that use pre-defined strings, given as inputs to the methods of the service. We can then check if the service returned the proper output for each string.

**6.4. Integration testing**

Integration testing can use a combination of the UI handling logic and the services defined above, and it can also be automated. We can think of several integration test cases for our product:

* Create a configuration file containing several parameter definitions and check if those configuration parameters have been read properly by Pie (for example, set Pie to “dark” mode through the theme configuration file and check if the color of the title bar is, indeed, dark).
* Check if the output of the formatting service is properly applied to the Scintilla code editor present in the selected tab

These tests should be run after making sure that all of the unit tests pass.

**6.5. System Testing**

System testing should be done during the last week of each sprint. System testing means testing the software product as a whole, not testing individual components per se. We will define several test scenarios for our product after each sprint, making sure that we are not trying to test components that have not yet been integrated. Such test scenarios include:

* We will open a new tab and write a Python script that asks for user input (the name of the user) and greets the user. We will then save the file in a temporary location and click the “Run as Python script” button, available in the context menu of the Scintilla code editor. We will check if a terminal window has been opened and we will write “Admin” in the terminal. We will then check if the text “Hello, Admin!” is present in the terminal’s output.
* We will add several custom build commands inside Pie, then press the “Build” button and check if the defined names are available in the dropdown menu. We will then check if these build commands have been saved in Pie’s configuration file.
* We will add a new database connection inside Pie. We will start the database server and click the “Test connection” button. We will wait 5 seconds and check if a popup form has been displayed, containing the “Database connection established successfully.” text. We will then stop the database server and click on the “Test connection” button again. We will wait 10 seconds (as Pie will try multiple times to connect to the database) and check if a popup form has been displayed, containing the “Could not establish connection to the database.” text.

If the automated tests pass, the development team, together with the testers, will start doing manual exploratory testing, by interacting with the GUI elements and trying to find bottlenecks or exceptional cases in the logic of the application.

**6.6. Alpha and Beta Testing**

After the 15 weeks of development, we need to leave the pre-defined test cases behind and start testing Pie in more realistic ways.

The first type of user acceptance testing (UAT) conducted during Pie’s testing phase is **alpha testing**. We will choose a team of quality assurance (QA) testers from our organization, who have not directly contributed to the development (or testing) of our product, in order to simulate real end-users in a more efficient manner. During a four-week period, the testers will interact with the application and use it on a daily basis, while working on personal projects or on other projects that belong to our organization. This will ensure that most features of Pie will be used in a practical manner. We are going to use an issue tracking system (Jira) in order to keep track of the problems found while testing Pie during the allocated month. Whenever a problem is found, the tester in charge should create a new issue with a certain priority (low, medium, high, or blocker) that will be added to the backlog of our project and will be scheduled for the upcoming sprints. Exceptions should **not** be the only issues reported. Slow response times and application design flaws should be equally prioritized.

After the alpha testing phase, if any bugs are found, we will analyze their severity. Depending on the severity of the identified errors, we will add them to the upcoming sprints and allocate a couple of weeks for fixing them. If there are multiple issues labeled as “high priority” or “blocker”, after fixing them, we should conduct another 2-week phase of alpha testing. If we have a large number of bugs with “medium” priority, even if we encounter little to no blockers, we should still re-launch the alpha testing phase. Having multiple “low” priority bugs should not influence at all the decision regarding restarting the alpha testing process.

Depending on the number and severity of issues opened during the next iterations of our alpha testing phase, we should decide whether to fix these bugs and start a new 2-week period, or simply move on to **beta testing**.

Beta testing is similar to alpha testing, but it doesn’t rely on the organizations’ environment and employees. It is performed by real users, not by testers that simulate real use cases. An efficient way of thinking about how to organize beta testing for our product would be by understanding the two types of beta testing:

1. Closed beta
2. Open beta

Closed beta is restricted to a small group of people, chosen after a certain criterion. Open beta means that the software is publicly available and can be used (and tested) by anyone.

The best way would be to start several weeks of closed beta testing. We will announce our intention one week before, and allow applications through our online portal. We will then take a look over the applications, and choose our users carefully. Those users will have a direct way of communicating with our testers and developers, either by mail, phone number, or social media. Issues will still be reported through Jira. The closed beta phase will last 2 weeks.

We will then allocate a 2-week period of fixing the issues reported during the closed beta testing phase of Pie. We will decide **not to** reschedule the closed beta testing phase, depending on the number of issues and their severity. After fixing the issues, we will simply move on to **open beta**.

We will continue with announcing that our software will be released in a beta version, and issues can be reported by anyone through our Jira page. Pie will then be tested by all types of users during a 6-week phase. All of the issues encountered will be fixed in the upcoming sprints. The beta version will **not** be restarted, if a certain number of issues has been reported.

**6.7. Continuous Integration. Pair Programming.**

We will also implement several Continuous Integration (CI) tasks that will be automatically executed, whenever someone pushes their code on the main branch (using a versioning control system – GitHub, in our case). The CI system will build Pie on multiple architectures, in order to make sure that the application is runnable from both 32-bit and 64-bit processors. Then, it will start the automated unit and integration tests on machines having different specifications, in order to ensure that Pie is runnable on multiple types of devices. We will use Atlassian Bamboo in order to automate these tasks and keep track of the builds that passed or failed.

Directly pushing the code on main will not be allowed. Developers should work on their own branches, and then initiate a pull request (PR) that needs approval of another developer from the team. The second developer will then analyze the written code and start a conversation, if something in the code doesn’t feel right, or simply approve the request.

**7. Bibliography**

<https://stackify.com/what-is-sdlc/>

<https://www.knowledgehut.com/blog/software-testing/sdlc-in-software-testing#the-software-development-life-cycle-process%C2%A0>

<https://study.com/academy/lesson/testing-phase-in-sdlc.html>

<https://testsigma.com/blog/test-documentation/#What_documentation_do_QA_teams_use>

<https://logiciel.io/software-testing-documentation>

<https://www.browserstack.com/guide/white-box-testing>

<https://www.indeed.com/career-advice/starting-new-job/alpha-vs-beta-testing>

<https://github.com/microsoft/WinAppDriver>