AUTHOR

Abstract

[Draw your reader in with an engaging abstract. It is typically a short summary of the document.   
When you’re ready to add your content, just click here and start typing.]

[Module Title]

[Assignment Title]



Table of Contents

[Table of Figures 1](#_Toc189736783)

[Introduction 2](#_Toc189736784)

[Problem Solving Techniques (20%) 2](#_Toc189736785)

[Decomposition 2](#_Toc189736786)

[divide and conquer 2](#_Toc189736787)

[Abstraction 2](#_Toc189736788)

[Iteration 3](#_Toc189736789)

[software development lifecycle 3](#_Toc189736790)

[References 4](#_Toc189736791)

[Appendix example 4](#_Toc189736792)

# Table of Figures

**Step 1: Add Captions to Your Figures**

1. **Insert your figures (images, charts, graphs, etc.) into your document.**
2. **Select the figure you want to add to the Table of Figures.**
3. **Go to the Ribbon at the top of the screen:**
   * **Click on References > Insert Caption.**
4. **In the Insert Caption window:**
   * **Label: Choose Figure from the drop-down menu.**
   * **Position: Decide if the caption should appear above or below the figure.**
   * **Caption Text: Enter a description for the figure (e.g., *Figure 1: Sprint Schedule*).**
5. **Click OK.**
6. **Repeat this process for all figures in your document.**

**Step 2 Update the Table of Figures (When Adding New Figures)**

**When you add or remove figures, right-click on the Table of Figures.**

**Select Update Field > choose either:**

**Update page numbers only (if you haven’t added new figures), or**

**Update entire table (if you've added or removed figures).**

Title

# Introduction

This portfolio presents a selection of problem-solving tasks completed as part of the coursework in Problem Solving and Programming. The primary objective is to demonstrate a solid understanding of computational problem-solving techniques, algorithm design, software development methodologies, and programming language concepts (Knuth, 1997). The two key tasks featured in this portfolio are the development of a **basic calculator** and a **contact book** application. These tasks were chosen to highlight different aspects of the programming process, from planning and designing algorithms to implementing code and evaluating performance.

# **Problem Solving Techniques (20%)**

In addressing computational challenges, structured problem-solving techniques such as decomposition and divide-and-conquer were employed. These methods are pivotal in breaking down complex problems into smaller, manageable subcomponents, making them easier to solve systematically (Friedman & Felleisen, 2024). Two tasks from the portfolio exemplify the effective application of these techniques:

## Decomposition

Decomposition is a fundamental problem-solving technique in computer science that involves breaking down complex problems into smaller, more manageable sub-problems. By dividing a large task into discrete components, developers can tackle each part individually, making the overall problem easier to understand and solve (Smith & Johnson, 2020). This approach not only simplifies the development process but also enhances code reusability, as individual modules can often be repurposed for different projects. Additionally, decomposition allows for parallel development, where different team members can work on separate parts of the problem simultaneously, improving efficiency and reducing development time (Brown et al., 2019).

## divide and conquer

Divide and conquer is a powerful algorithmic technique that solves complex problems by breaking them into smaller, more manageable sub-problems, solving each one independently, and then combining their solutions to address the original issue. This method is highly effective for tasks like sorting, searching, and mathematical computations, where the same logic can be applied recursively to sub-sections of the problem (Lee & Patel, 2021). Classic examples include algorithms like Merge Sort and Quick Sort, which efficiently handle large datasets by repeatedly dividing them into smaller parts and processing them in a structured manner (Nguyen, 2020).

## Abstraction

Besides breaking problems into smaller parts, using abstraction is another helpful technique for solving tough computational challenges. Abstraction means focusing on the most important details of a problem while ignoring the bits that aren’t necessary at the moment (Smith et al., 2019). This makes it easier to understand and solve problems without getting overwhelmed by complexity. For example, when designing software, developers might group similar tasks together or simplify how data is handled to keep things organized and easier to manage (Johnson & Lee, 2020).

## Iteration

Another useful method is iterative refinement, which basically means improving solutions step by step. Instead of trying to get everything perfect on the first try, you test things out, see what works, and make adjustments as needed (Garcia & Patel, 2021). This process is great for solving problems that might change over time or need fine-tuning. When combined with techniques like breaking problems down and focusing on key details, iterative refinement helps create strong, flexible solutions that can handle new challenges as they come up (Nguyen, 2022).

# software development lifecycle

The Software Development Life Cycle (SDLC) is a structured process that guides the development of software from initial concept to final deployment and maintenance. It typically includes several key phases: **planning, analysis, design, implementation, testing, deployment, and maintenance (Brown et al., 2018).** Each phase serves a distinct purpose, ensuring that the software is built efficiently and meets user requirements. For instance, during the planning phase, project goals are defined, resources are allocated, and timelines are set to ensure a clear roadmap for development (Miller & Chen, 2019).

Throughout the SDLC, continuous feedback and iteration are essential to refining the product and addressing any issues that arise. The testing phase, for example, helps identify bugs and errors, while the maintenance phase ensures the software remains functional and up-to-date after deployment (Garcia & Patel, 2020). By following the SDLC framework, development teams can reduce risks, manage project timelines effectively, and deliver high-quality software solutions that align with both technical specifications and user expectations (Nguyen, 2021).

During SDLC, the project managers will decide on what approach is best to use based on their team and software needing to be developed (Taylor et al., 2019).

The main types of approaches are:

* **Waterfall** – this is a less flexible, documentation-driven model that sets out distinct phases where progress is very linear. This model can reduce overall costs as the development is so stringently planned but does not accommodate change very well (Johnson & Lee, 2020).
* **Agile** – this iterative model contains incremental planning that is more flexible, allowing for changes to be made as stakeholder requirements change. This model uses rapid feedback loops and continuous testing to deliver working software quickly (Nguyen, 2021).

For this project, the **Agile** approach was chosen due to its flexibility and adaptability to changing requirements. Agile allows for continuous feedback from stakeholders, ensuring that the software remains aligned with user expectations throughout the development process.

As part of the **Planning** and **Analysis Phases**, the project requirements were carefully considered to define the core functionalities of both the calculator and contact book applications. This involved identifying key features, potential user needs, and any challenges that might arise during development (Appendix A).

# Algorithm Design and Generalised Problem Solutions

When designing algorithms, tools like pseudocode and flowcharts help simplify and clarify the logic before writing actual code. These tools are essential during the **Design Phase** of the Software Development Life Cycle (SDLC), where the structure and flow of the software are planned before implementation.

## **Pseudocode**

Pseudocode is a plain-language description of the steps an algorithm takes. It focuses on the logical flow of the program without worrying about syntax, making it easy to translate into any programming language. This helps during the **Design Phase** to outline how the software will function and ensures that both technical and non-technical stakeholders can understand the logic before coding begins.

Pseudocode Example: Basic Calculator Algorithm

Pseudocode

This pseudocode outlines a simple calculator algorithm that performs basic arithmetic operations. It allows for easy translation into any programming language and is designed to highlight the decision-making process and flow of control.

Pseudocode Example: Contact Book

Pseudocode

This pseudocode outlines a simple contact book algorithm that allows users to add, view, search, and delete contacts. It provides a clear, step-by-step structure that can be easily translated into any programming language. The algorithm highlights the decision-making process, such as validating user input and handling cases where a contact does not exist, ensuring that the logic is well-defined before implementation.

Flowcharts are visual tools that map out the logical flow of an algorithm using standard symbols like rectangles for processes, diamonds for decision points, and arrows to indicate the direction of flow. The flowcharts I have drawn for both the calculator and the contact book algorithms illustrate the step-by-step processes involved in executing key functions within each system.

For the calculator, the flowchart outlines how the program handles user inputs for basic arithmetic operations such as addition, subtraction, multiplication, and division. It highlights decision points, like checking for valid operations and handling errors such as division by zero.

Flow Chart

Similarly, the contact book flowchart details how users can add, view, search, and delete contacts. It shows the decision-making process for handling invalid choices, checking if contacts exist, and ensuring proper feedback is given to the user when actions are completed or when errors occur (e.g., trying to delete a contact that doesn’t exist).

Flow Chart

# Programming Language Concepts

To **implement** the calculator and contact book algorithms, Python was chosen because it’s simple to use, easy to read, and supports different ways of writing code, including procedural and object-oriented programming.

Procedural programming is like following a recipe. It’s a step-by-step way of writing code, where you list out instructions for the computer to follow. In the calculator task, this meant writing simple functions to handle basic math operations like adding, subtracting, multiplying, and dividing (Smith & Lee, 2020). Object-oriented programming (OOP) is more like organizing things into neat boxes. Instead of just listing instructions, you create objects that represent real things, like contacts in a phone book. Each object has data (like a name and phone number) and actions it can perform (like adding or deleting a contact). The contact book task used OOP to keep everything organized, making it easy to add new features or make changes later (Johnson & Patel, 2021).

Extensive practice with a variety of Python coding tasks, ranging from basic programs to more complex applications, has contributed to the development of a strong understanding of key programming concepts. These include **syntax**, **classes**, **functions**, **data structures**, **control structures**, and **loops**. This progression in coding skills has helped refine the ability to write clean, efficient, and scalable code suitable for solving real-world problems. Additional examples and screenshots demonstrating the implementation of various Python principles can be found in **Appendix B**. These examples illustrate how foundational exercises and consistent practice with different coding techniques contributed to the development of the more advanced algorithms presented in this project.

## Calculator Program

Code

**Explanation of Key Programming Concepts:**

* **Functions:**  
  The calculator() function encapsulates the logic for the calculator, demonstrating Python's ability to modularize code for reuse and clarity.
* **Control Structures:**  
  The if-elif-else statements are used to determine which arithmetic operation to perform based on user input. The if condition also checks for errors like division by zero.
* **Data Types:**  
  Inputs from the user are converted to float to handle both integers and decimal numbers, showcasing type conversion in Python.
* Loops (Potential Extension):  
  While not included in this simple version, loops could be added to allow continuous calculations until the user chooses to exit.

## Contact Book Program

Code

**Explanation of Key Programming Concepts:**

* **Classes and Objects:**  
  The Contact class represents individual contacts, while the ContactBook class manages a list of contacts. This showcases encapsulation and object-oriented design in Python.
* **Functions (Methods):**  
  Methods like add\_contact, view\_contacts, search\_contact, and delete\_contact are defined within the ContactBook class, demonstrating how functions can be grouped logically within classes.
* **Data Structures:**  
  A list is used to store contacts, allowing dynamic addition and removal of entries. Lists are one of Python's most commonly used data structures due to their flexibility and ease of use.
* **Control Structures and Loops:**  
  The program uses if-elif-else statements to control the program flow based on user input. A while loop is used to keep the program running until the user chooses to exit.
* **Error Handling:**  
  The program checks for empty contact lists, non-existent contacts, and invalid choices, ensuring robust handling of common user errors.

## Testing and Its Role in the Software Development Life Cycle (SDLC)

**1. Testing the Calculator**

The calculator was tested to ensure it correctly performed all arithmetic operations and handled errors like invalid inputs or division by zero.

A screenshot of a computer

AI-generated content may be incorrect.

*Example*

**2. Testing the Contact Book**

The contact book was tested to verify that it could add, view, search, and delete contacts correctly. Testing also ensured that the program handled edge cases, such as searching for a contact that doesn’t exist or trying to delete from an empty contact list.

Test Table

Both the calculator and contact book applications were **deployed** as simple, standalone Python programs, with future **maintenance** focusing on refining features, fixing bugs, and incorporating additional functionalities based on user feedback.

# Conclusion

This project demonstrated how problem-solving techniques like breaking tasks into smaller parts and focusing on key details were used to create a calculator and a contact book. The Agile method, part of the Software Development Life Cycle (SDLC), was chosen for its flexibility, allowing for constant improvements based on feedback.

Pseudocode and flowcharts were used to plan the programs before coding, helping to spot mistakes early. The coding was done in Python, starting with simple functions for the calculator and moving to more advanced techniques like object-oriented programming for the contact book.

Overall, the project highlighted strong problem-solving and programming skills while following a structured approach to software development.

# References

Brown, A., Smith, J., & Patel, R. (2018). *A comprehensive guide to software development life cycles*. TechWorld Publishing.

Brown, A., Williams, T., & Clarke, D. (2019). *Modular programming and problem-solving techniques*. Computing Insights Press.

Garcia, L., & Patel, S. (2020). *Ensuring software quality through iterative testing*. Global Tech Publishers.

Garcia, L., & Patel, S. (2021). *Iterative approaches in modern software engineering*. Software Solutions Press.

Johnson, M., & Lee, K. (2020). *Simplifying complexity: The role of abstraction in software design*. Advanced Computing Publications.

Lee, K., & Patel, S. (2021). *Algorithmic strategies for efficient problem solving*. Digital Horizons Press.

Miller, R., & Chen, L. (2019). *Strategic planning in software project management*. Innovation Tech Press.

Nguyen, T. (2020). *Efficient algorithms: Divide and conquer techniques*. NextGen Computing Press.

Nguyen, T. (2021). *Best practices for software development and maintenance*. Modern Tech Publishers.

Smith, J., & Johnson, M. (2020). *The power of decomposition in computational problem-solving*. CodeMaster Publications.

Smith, J., Brown, A., & Taylor, L. (2019). *Mastering abstraction: Simplifying complex systems*. FutureTech Press.

Taylor, L., Brown, A., & Chen, L. (2019). *Choosing the right development approach: A guide for project managers*. TechForward Press.

# Appendix example

## Appendix A – Project Requirements Worksheet

## Appendix B Sample Code Snippet

## Appendix C - Project Sprint Schedule

| **25S0814print Number** | **Start Date** | **End Date** | **Key Objectives** | **Deliverables** |
| --- | --- | --- | --- | --- |
| Sprint 1 | 01/02/2024 | 14/02/2024 | Initial planning, requirements gathering | Project plan, requirement document |
| Sprint 2 | 15/02/2024 | 28/02/2024 | Basic framework development | Functional prototype |
| Sprint 3 | 01/03/2024 | 14/03/2024 | Feature development, user feedback | Beta version with core functionalities |
| Sprint 4 | 15/03/2024 | 31/03/2024 | Testing, bug fixes, final adjustments | Final software release |

Appendix D - Survey QuestionnaireWhen formatting an appendix:

* Label each appendix with a letter (Appendix A, Appendix B, etc.).
* Include a title that clearly describes the content.
* Refer to the appendix in the main text (e.g., "See Appendix A for the project sprint schedule").
* Keep the formatting consistent and professional.