DrillBit

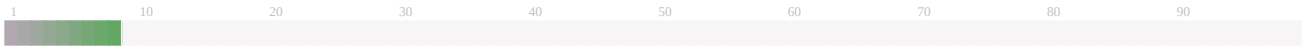
The Report is Generated by DrillBit Plagiarism Detection Software

Submission Information

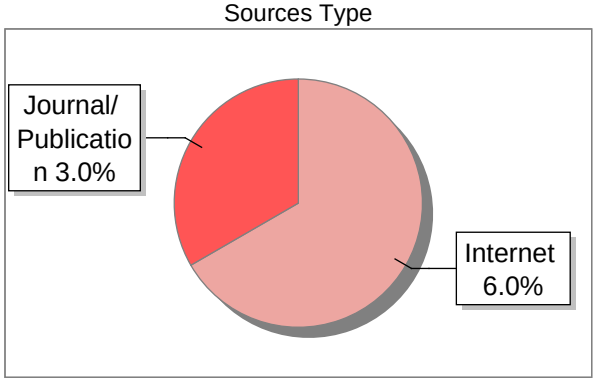
Author Name	Batch 2
Title	Batch 2
Paper/Submission ID	2218757
Submitted by	dr.kavithat.nhce@newhorizonindia.edu
Submission Date	2024-08-10 09:20:30
Total Pages, Total Words	33, 3872
Document type	Project Work

Result Information

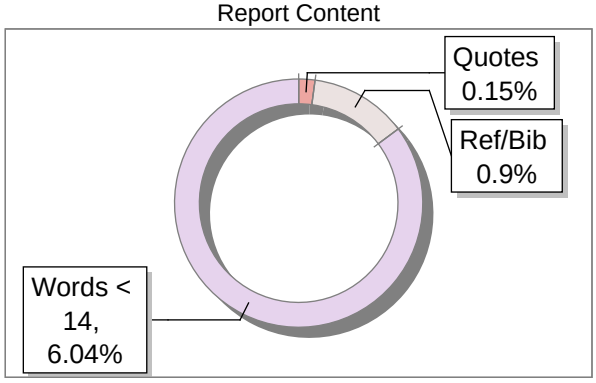
Similarity **9 %**



Sources Type



Report Content



Exclude Information

Quotes	Not Excluded
References/Bibliography	Not Excluded
Source: Excluded < 14 Words	Not Excluded
Excluded Source	**0 %**
Excluded Phrases	Not Excluded

Database Selection

Language	English
Student Papers	Yes
Journals & publishers	Yes
Internet or Web	Yes
Institution Repository	Yes

A Unique QR Code use to View/Download/Share Pdf File

DrillBit

DrillBit Similarity Report

9	26	A
SIMILARITY %	MATCHED SOURCES	GRADE

A-Satisfactory (0-10%)
B-Upgrade (11-40%)
C-Poor (41-60%)
D-Unacceptable (61-100%)

LOCATION	MATCHED DOMAIN	%	SOURCE TYPE
1	bmcbioinformatics.biomedcentral.com	1	Internet Data
2	www.smec.ac.in	1	Publication
3	adoc.pub	1	Internet Data
4	ir.canterbury.ac.nz	1	Publication
5	Thesis Submitted to Shodhganga Repository	<1	Publication
6	avxlive.icu	<1	Internet Data
7	www.ncbi.nlm.nih.gov	<1	Internet Data
8	www.lambdatest.com	<1	Internet Data
9	A computer approach toward automation of a chemical services laborator by E-1971	<1	Publication
10	digitalcommons.unl.edu	<1	Publication
11	mdpi.com	<1	Internet Data
12	www.aktek.io	<1	Internet Data
13	adoc.pub	<1	Internet Data

14	ACM Press the 22nd ACM SIGSAC Conference- Denver, Colorado, USA (20, by Chudnov, Andrey Na- 2015	<1	Publication
15	bg.copernicus.org	<1	Internet Data
16	docplayer.net	<1	Internet Data
17	docplayer.net	<1	Internet Data
18	docplayer.net	<1	Internet Data
19	docplayer.net	<1	Internet Data
20	MotorIA Automatic E-Learning Course Generation System by de-2020	<1	Publication
21	pdfcookie.com	<1	Internet Data
22	technodocbox.com	<1	Internet Data
23	vdocuments.mx	<1	Internet Data
24	writeanessay-forme.com	<1	Internet Data
25	www.investopedia.com	<1	Internet Data
26	www.readbag.com	<1	Internet Data

CHAPTER 1

PREAMBLE

1.1 INTRODUCTION

Password Generator Program-

Welcome to our Password Generator program! This application allows you to create secure, random passwords tailored to your specific needs. Whether you need a password for a new online account, a secure key for your Wi-Fi network, or any other purpose, this tool ensures that your passwords are both strong and customizable.

Creating a strong password is essential for safeguarding your personal and sensitive information. A robust password combines a mix of uppercase letters, lowercase letters, digits, and symbols. By using diverse character sets and avoiding predictable patterns, you enhance the security of your accounts and data. Secure Gen helps to streamline this process, offering customizable options to generate complex and secure passwords effortlessly.

1.2 OBJECTIVES

1. Develop a Flexible Password Generator: Create a tool that allows users to specify password length and character types (e.g., uppercase letters, digits) to meet varying security needs.

2. Ensure Password Complexity: Implement features that guarantee the inclusion of different character types, thereby avoiding easily guessable passwords.

3. Promote User Customization: Provide options for users to tailor the password generation process according to their preferences and security requirements.

4. Enhance Security Standards: Incorporate best practices for password generation, such as randomness and unpredictability, to ensure passwords are robust against attacks.

5. Provide User-Friendly Interface: Design an intuitive interface that guides users through the process of setting password parameters and generating passwords seamlessly.

6. Handle Errors Gracefully: Implement error handling to manage invalid input, such as insufficient password length or lack of character type selection, and provide helpful feedback.

7. Ensure Reliable Performance: Optimize the password generation algorithm to produce secure passwords quickly and efficiently.

1.3 SCOPE OF THE PROJECT

This password generator program aims to provide users with a robust tool for creating secure, customizable passwords. It offers several functionalities that cater to different user needs and security requirements. The scope of the program encompasses the following areas:

1. User Interaction and Interface:

- **Command-Line Interface**: The program utilizes a command-line interface (CLI) to interact with users. It guides users through various options for customizing their passwords and provides clear instructions and feedback.
- **User Prompts**: The program includes prompts for user inputs, allowing users to specify password length and toggle the inclusion of uppercase letters and digits.

- **Input Validation**: The program ensures that user inputs are valid, such as checking for a minimum password length and handling invalid menu choices gracefully.

2. Password Customization Options:

- **Password Length**: Users can specify the length of the password, with a minimum requirement of 4 characters to ensure basic security. The default length is set to 12 characters.
- **Uppercase Letters**: Users can choose to include or exclude uppercase letters in the password. This option enhances the complexity of the password.
- **Digits**: Users can choose to include or exclude numerical digits in the password, further increasing the password's strength and complexity.

3. Password Generation Logic:

- **Character Sets**: The program uses predefined character sets for lowercase letters, uppercase letters, and digits. It dynamically constructs a pool of characters based on user preferences.
- **Random Selection**: The program fills the remaining length of the password with randomly selected characters from the constructed pool, ensuring randomness and unpredictability.
- **Shuffling**: To enhance security, the program shuffles the characters in the generated password to avoid predictable patterns.

4. Security Considerations:

- **Minimum Length Enforcement**: The program enforces a minimum password length of 4 characters to avoid weak passwords.
- **Character Type Requirements**: By ensuring the inclusion of at least one character from each selected type, the program avoids passwords that lack diversity in character composition.

5. Error Handling and User Feedback

- **Invalid Input Handling**: The program handles invalid inputs gracefully, providing appropriate feedback and prompting users to enter valid choices.
- **Error Messages**: Clear error messages are displayed when user inputs do not meet the requirements, such as specifying a password length less than 4 or not selecting any character types.
- **Informative Prompts**: The program keeps users informed about the current settings and any changes made, ensuring transparency and ease of use.

CHAPTER 2

LITREATURE SURVEY

2.1 BASE PAPERS

For a literature survey on generating strong and secure passwords, you can refer to foundational and influential papers in the field of cryptography and password security. Here are some key papers and resources that could serve as the basis for understanding password generation techniques and security principles:

1. **Password Hashing Competition :**
   - Authors: D. J. Bernstein, T. Lange, and P. Schwabe
   - Published in: 2016
   - Overview: This paper discusses the Password Hashing Competition (PHC) and the evolution of secure password hashing techniques. It highlights the importance of password hashing algorithms in protecting against brute-force attacks.
2. **The Risks of Key Recovery, Key Escrow, and Trusted Third-Party Encryption:**
   - Authors: L. S. P. Schneier
   - Published in: 1997
   - Overview: Bruce Schneier's work explores the risks and limitations of various encryption methods, including those relevant to password security and key management.
3. **Password Security: A Case History**
   - Authors: A. M. Davis and A. S. K. Bellovin
   - Published in: 1995
   - Overview: This paper examines historical cases of password breaches and analyzes factors contributing to password security weaknesses.

2.2 EXISTING SYSTEMS

For the "Secure Gen" project, reviewing existing systems for generating strong, secure passwords can provide valuable insights and help identify features that are important for creating a robust password generator. Here are some notable existing systems and tools in this area:

1. **Password Managers:**

- LastPass: Offers password generation and storage with customizable settings for length and complexity.

- 1Password: Provides a password generator with options for including various character types and complexity settings.

- Dashlane: Features a built-in password generator with security and customization options.

2. **Online Password Generators:**

- Strong Password Generator (passwordsgenerator.net): Allows users to specify length and character types, including uppercase letters, digits, and symbols.

- Random.org Password Generator: Uses true random number generation to produce secure passwords with user-defined length and character set options.

- Generate Password (generatepassword.org): Provides customizable password generation with options for length and inclusion of various character types.

3. **Cryptographic Libraries and APIs:**

- Python secrets Module: A standard Python library designed for generating cryptographically secure random numbers suitable for password generation.

- Bcrypt Library: Implements the bcrypt hashing algorithm for secure password storage and can be used in conjunction with password generators to enhance security.

2.3 PROBLEM statement

For our **"Secure Gen"** project, there can be two main problem statements:

1. **Ensuring Adequate Complexity and Randomness:**

- Problem Statement: The challenge is to develop a password generator that produces passwords with high entropy and diverse character sets, ensuring robustness against various types of attacks. Many existing password generators lack sufficient complexity or randomness, leading to passwords that may still be vulnerable to brute-force or attacks.

2. **User-Friendly Customization and Error Handling:**

- Problem Statement: Current password generation tools often struggle with balancing advanced customization options with ease of use.

- The goal is to create a password generator that offers flexible customization options while maintaining a user-friendly interface and providing clear error handling and feedback. Users may face difficulties in setting parameters or understanding the implications of their choices.

2.4 PROPOSED SYSTEMS

For our **"Secure Gen"** project, several proposed systems can be considered for generating strong and secure passwords. These systems focus on addressing the challenges of complexity, randomness, customization, and user-friendliness.

Here are some notable proposed systems and approaches:

1. **Enhanced Randomization Techniques:**

- System: Cryptographically Secure Random Number Generators

- Description: Utilize cryptographically secure random number generators to ensure that password generation is highly unpredictable and resistant to attacks. These systems

enhance password security by producing random values that are less likely to be predicted or reproduced by attackers.

2. **Adaptive Complexity Algorithms:**

- System: Dynamic Complexity Adjustment

- Description: Implement algorithms that dynamically adjust password complexity based on user-defined parameters and security best practices. For example, the system could require at least one character from each selected character set (lowercase, uppercase, digits, symbols) and adjust the difficulty of password cracking accordingly.

3. **User-Friendly Interface with Real-Time Feedback:**

- System: Interactive Configuration Interface

- Description: The system should provide real-time feedback on password strength and compliance with security guidelines to help users make informed choices and to develop an intuitive, interactive user interface that allows users to easily configure password parameters such as length and character types.

2.5 METHODOLOGY

1. **Cryptographic Randomization:**

- Utilize cryptographic random number generators to produce unpredictable and secure passwords, ensuring high levels of randomness and resistance to security attacks.

2. **Complexity Enforcement:**

- Implement algorithms to enforce password complexity requirements, including mandatory inclusion of uppercase letters, digits, and symbols, to meet security standards.

3. **User Customization:**

- Provide an interactive interface for users to specify password length and character types, with real-time feedback to guide them in creating strong, customized passwords.

4. **Integration of Security Guidelines:**

- Integrate established security guidelines from organizations like NIST or OWASP to ensure generated passwords adhere to industry best practices.

5**. Error Handling and Validation:**

- Implement robust error handling to manage invalid inputs and provide clear feedback, helping users correct issues and generate valid, secure passwords.

CHAPTER 3

REQUIREMENT SPECIFICATIONS

3.1 FUNCTIONAL REQUIREMENTS

1. **Password Generation:**

- The system must generate passwords based on user-specified parameters such as length and inclusion of character types (uppercase letters, lowercase letters, digits, and special characters).

-The generated passwords must be random and meet the specified criteria.

2. **Customization Options:**

- Users must be able to customize the password generation settings.

- This includes setting the desired password length (with a minimum requirement), choosing which character types to include (uppercase, lowercase, digits, special characters), and optionally specifying any exclusions or patterns.

3. **Complexity Enforcement:**

- The system must ensure that the generated passwords contain at least one character from each selected character type.

-If the user specifies to include uppercase letters, digits, and special characters, the password must contain at least one of each.

4. **User Interface:**

- The system must provide an intuitive and user-friendly interface for configuring password generation options.

- This interface should guide the user through the process, providing clear instructions and real-time feedback on the strength and validity of the password settings.

5. **Error Handling and Validation:**

- The system must include robust error handling and input validation. If the user inputs invalid parameters (e.g., setting a password length less than the minimum requirement or not selecting any character types),the system should provide informative error messages and guidance to correct the input.

3.2 NON FUNCTIONAL REQUIREMENTS

1. **Security:**

- The system must use cryptographic random number generation to ensure that all passwords are highly unpredictable and resistant to brute-force and dictionary attacks.

- Sensitive operations, such as generating passwords, must be implemented using secure coding practices to prevent vulnerabilities.

2. **Performance:**

- The password generation process should be efficient, with passwords generated in real-time or near real-time, ensuring minimal latency.

- The system should handle multiple password generation requests without significant degradation in performance.

3.**Usability:**

- The user interface should be intuitive, easy to navigate, and accessible, ensuring that users can easily understand and configure password settings.

- Provide clear and concise feedback to users, including error messages and password strength indicators, to enhance the user experience.

4. **Portability:**

- The software should be cross-platform compatible, running smoothly on Windows, macOS, and Linux operating systems.

- The system should have minimal dependencies, making it easy to install and use in various environments.

5. **Reliability:**

- The system should be robust and reliable, with a high degree of fault tolerance to handle unexpected inputs or errors gracefully.

- Ensure that generated passwords are consistently secure and meet the specified criteria without fail.

3.3 SOFTWARE AND HARDWARE REQUIREMENTS

SOFTWARE REQUIREMENTS-

1. **Programming Language:**

- Python 3.6+ : The core language for developing the password generator.

2. **Libraries and Dependencies:**

- Python secrets Module: For generating cryptographically secure random numbers.

- Python string Module: For handling character sets.

- Python random Module: For additional randomization (if needed).

3. **Development Environment:**

- Integrated Development Environment (IDE): Such as PyCharm, VS Code, or any preferred IDE.

- Version Control System: Git and a GitHub repository for version control and collaboration.

4. **Operating System:**

- Cross-Platform Compatibility: Ensure the tool works on Windows, macOS, and Linux.

5. **Documentation Tools:**

- Markdown/README.md: For project documentation.

- Sphinx or MkDocs (optional): For generating detailed project documentation.

HARDWARE REQUIREMENTS-

1. **Development Machine:**

- Processor: Intel i3 or equivalent and above.

- RAM: Minimum 4 GB, recommended 8 GB.

- Storage: Minimum 1 GB free space for development and version control.

2. **Execution Environment:**

- Any modern computer capable of executing programs on Python 3.6+ efficiently.

Additional Requirements

1. **Backup and Version Control**:

- Regularly back up code and documentation to a cloud service or external drive.

- Use Git for version control and collaboration, with remote repository hosting (e.g., GitHub, GitLab).

2. **a good and strong Internet Connection:**

- For downloading dependencies and libraries.

- For accessing online resources and documentation.

CHAPTER 4

ANALYSIS AND DESIGN

4.1 DESIGN GOALS

- **Security:**

✓ Strong Password Generation: Ensure passwords generated are highly secure, incorporating a mix of uppercase letters, lowercase letters, digits, and potentially special characters.

✓ Randomness and Unpredictability: Utilize secure randomization techniques to guarantee the uniqueness and unpredictability of each password.

✓ Input Validation: Implement thorough input validation to prevent invalid configurations and ensure that generated passwords meet the required security standards.

- **Usability:**

➢ User-Friendly Interface: Design an intuitive interface that allows users to easily specify their password requirements and generate passwords without confusion.

➢ Clear Feedback: Provide immediate and clear feedback to users about their choices and the generated passwords, including any errors or validation issues.

➢ Customization Options: Allow users to customize password length and character inclusion (uppercase, digits) to meet their specific needs.

- **Performance:**

➢ Efficiency: Optimize the algorithm to ensure quick password generation, even for complex passwords.

➢ Resource Management: Ensure the application uses system resources efficiently to maintain performance across different environments.

- **Flexibility:**

➢ Customization: Provide flexible options for password length and character types, allowing users to generate passwords that meet various security requirements.

➢ Extensibility: Design the codebase to be easily extensible, enabling the addition of new features such as special character inclusion or integration with password management tools.

- **Maintainability:**

➢ Modular Code Structure: Organize the code into clear, modular functions to facilitate easy maintenance and future updates.

➢ Documentation: Provide comprehensive documentation and comments within the code to aid understanding and maintenance by other developers.

- **Compatibility:**

➢ Cross-Platform Support: Ensure the program runs seamlessly on various operating systems (Windows, macOS, Linux) without requiring significant modifications.

➢ Dependency Management: Minimize dependencies on external libraries to enhance compatibility and ease of installation.

- **Scalability:**

➢ Adaptable Design: Structure the application to handle increasing user demands and potential future enhancements without significant rework.

➢ Error Handling: Implement robust error handling to gracefully manage unexpected situations and ensure the application remains stable under various conditions.

- **Compliance:**

➢ Adherence to Best Practices: Follow industry best practices for password security and generation, ensuring compliance with relevant security standards and guidelines.

4.2 SYSTEM ARCHITECTURE

1. User Interface Layer

This layer handles all user interactions, providing a user-friendly interface to specify password generation requirements and view the generated passwords.

- Command-Line Interface (CLI) Module:
  - Provides a text-based interface for user interaction.
  - Displays options for setting password length, toggling character inclusion, and generating passwords.
  - Handles user input and displays generated passwords and error messages.

2. Core Logic Layer

This layer contains the core logic for generating secure passwords based on user-specified requirements.

- Password Generator Module:
  - generate_password(): The main function responsible for generating passwords. Takes parameters such as length, inclusion of uppercase letters, and digits.
  - Utilizes Python's random and string libraries to generate random characters and assemble the password.
  - Ensures at least one character from each selected type (uppercase, digits) is included.
  - Validates input parameters and raises appropriate errors for invalid configurations.

3. Validation and Error Handling Layer

This layer ensures that the inputs provided by the user are valid and handles any errors that may arise during password generation.

- Validation Module:
    - Validates password length to ensure it meets the minimum requirement.
    - Ensures that at least one character type is selected if uppercase or digits are to be included.
    - Provides meaningful error messages to guide users in correcting their inputs.

4. Configuration and State Management Layer

This layer manages the configuration options set by the user and maintains the current state of these options.

- Configuration Module:
    - Stores user preferences such as password length, inclusion of uppercase letters, and digits.
    - Provides methods to update and retrieve these preferences.
    - Ensures that changes to preferences are immediately reflected in the password generation process.

5. Utility and Helper Functions Layer

This layer contains utility functions that support the main functionality of the application.

- Utility Module:
    - display_password_options(): Displays the available options for password generation.
    - main(): The main function that drives the application, handling the user interaction loop and invoking the password generation logic.

6. Entry Point

The entry point initializes the application and starts the user interaction loop.

- Main Script:
    - Contains the if __name__ == "__main__": block to start the application.
    - Calls the main() function to begin the user interaction loop.

4.3 DATA FLOW DIAGRAM / USECASE DIAGRAM ETC

Context diagram-:

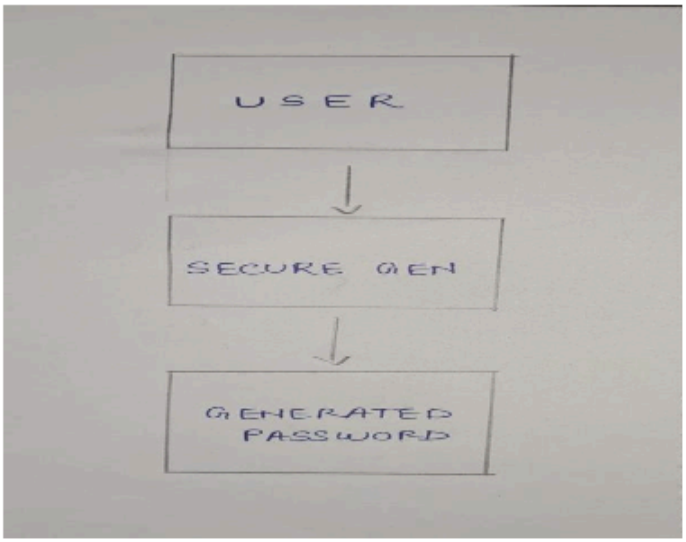


Fig. 4.1

Data Flow Description-:

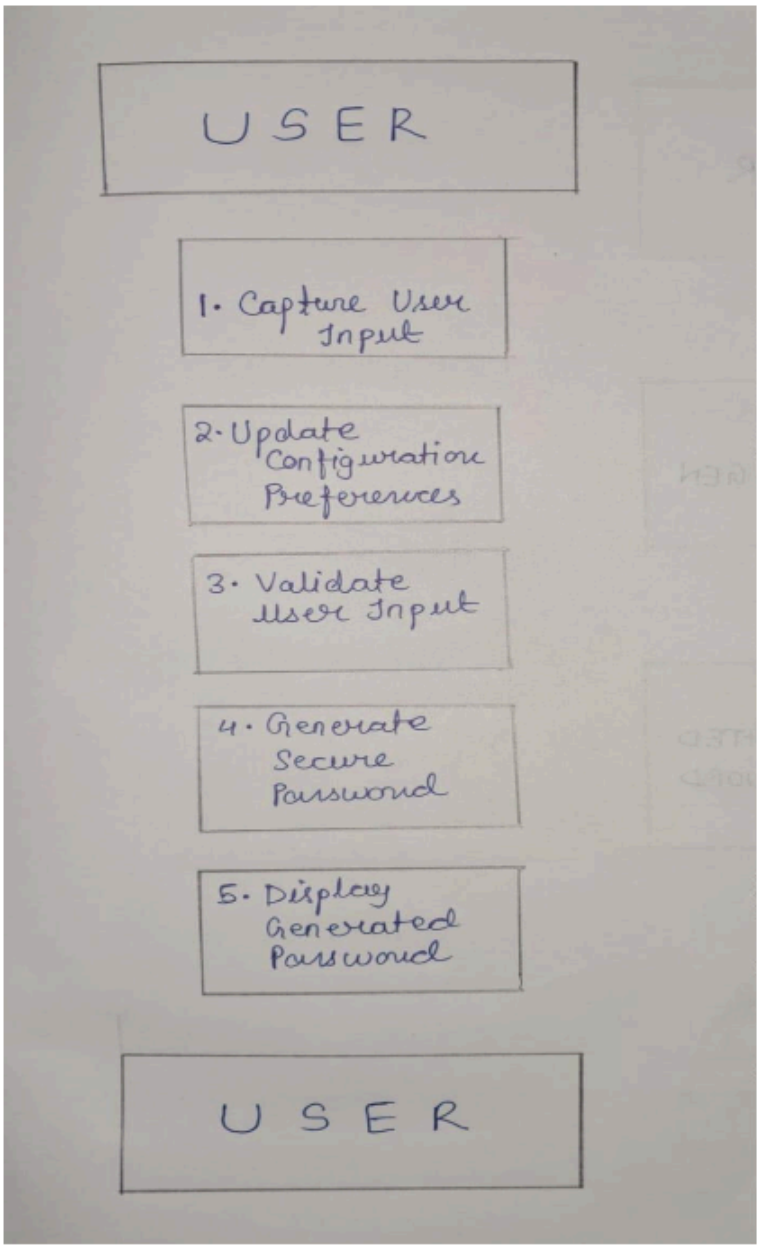


Fig. 4.2

CHAPTER 5

IMPLEMENTATION

1. User Interface Layer

This module handles all user interactions.

```python
def display_password_options():
    """
    Display the available options for password generation.
    """
    print("\nPassword Generator Options:")
    print("1. Set length of password")
    print("2. Toggle inclusion of uppercase letters")
    print("3. Toggle inclusion of digits")
    print("4. Generate password")
    print("5. Exit")
```

2. Core Logic Layer

This module contains the core logic for generating secure passwords.

```python
import random
import string

def generate_password(length=12, use_upper=True, use_digits=True):
    """
    Generate a secure password with the specified parameters.

    :param length: Length of the password (default is 12)
    :param use_upper: Include uppercase letters (default is True)
    :param use_digits: Include digits (default is True)
    :return: Generated password as a string
    """
    if length < 4:
        raise ValueError("Password length should be at least 4 characters")

    # Character sets
    lower_case = string.ascii_lowercase
    upper_case = string.ascii_uppercase
    digits = string.digits

    # Build the pool of characters to choose from
    char_pool = lower_case
    if use_upper:
        char_pool += upper_case
    if use_digits:
        char_pool += digits
```

```
    # Check if at least one character type is selected
    if not (use_upper or use_digits):
        raise ValueError("At least one character type must be selected")

    # Ensure at least one character from each selected type is included
    password = []
    if use_upper:
        password.append(random.choice(upper_case))
    if use_digits:
        password.append(random.choice(digits))

    # Fill the rest of the password length with random choices from the full pool
    while len(password) < length:
        password.append(random.choice(char_pool))

    # Shuffle to ensure randomness
    random.shuffle(password)

    return ''.join(password)
```

3. Validation and Error Handling Layer

This module ensures that inputs are valid and handles errors.

```
def validate_password_parameters(length, use_upper, use_digits):
    """
    Validate password parameters.

    :param length: Length of the password
    :param use_upper: Include uppercase letters
    :param use_digits: Include digits
    :return: Boolean indicating if the parameters are valid, and an error message if not
    """
    if length < 4:
        return False, "Password length should be at least 4 characters"
    if not (use_upper or use_digits):
        return False, "At least one character type must be selected"
    return True, ""
```

4. Configuration and State Management Layer

This module manages user preferences for password generation.

```python
class Configuration:
    def __init__(self, length=12, use_upper=True, use_digits=True):
        self.length = length
        self.use_upper = use_upper
        self.use_digits = use_digits

    def update_length(self, length):
        if length < 4:
            print("Length should be at least 4. Setting to default 12.")
            length = 12
        self.length = length
        print(f"Password length set to {self.length}.")

    def toggle_use_upper(self):
        self.use_upper = not self.use_upper
        print(f"Include uppercase letters: {'Yes' if self.use_upper else 'No'}")

    def toggle_use_digits(self):
        self.use_digits = not self.use_digits
        print(f"Include digits: {'Yes' if self.use_digits else 'No'}")
```

5. Utility and Helper Functions Layer

This module contains utility functions to support the main application flow.

```python
def main():
    """
    Main function to interact with the user and generate passwords.
    """
    print("Welcome to the Password Generator!")

    config = Configuration()

    while True:
        display_password_options()
        choice = input("Enter your choice (1-5): ")

        if choice == '1':
            try:
                length = int(input("Enter the desired length of the password:
                "))
                config.update_length(length)
            except ValueError:
                print("Invalid input. Please enter a number.")

        elif choice == '2':
            config.toggle_use_upper()

        elif choice == '3':
            config.toggle_use_digits()

        elif choice == '4':
```

```
27              valid, error = validate_password_parameters(config.length, config
                    .use_upper, config.use_digits)
28              if valid:
29                  try:
30                      password = generate_password(config.length, config
                            .use_upper, config.use_digits)
31                      print(f"Generated Password: {password}")
32                  except ValueError as e:
33                      print(f"ERROR! {e}")
34              else:
35                  print(f"ERROR! {error}")
36
37          elif choice == '5':
38              print("Exiting the Password Generator. Goodbye!")
39              break
40
41          else:
42              print("Invalid choice. Please enter a number between 1 and 5.")
43
44  if __name__ == "__main__":
45      main()
```

CHAPTER 6

TESTING

6.1 UNIT TESTING

Comprehensive unit tests validate the core functionalities, ensuring that each function and class method works correctly in isolation.

In Python, unit testing is commonly performed using the `unittest` framework, which provides a structured way to define test cases, execute them, and report results. Test cases are written as methods within a class that inherits from `unittest.TestCase`. Each test method checks a specific aspect of the function or class being tested, using assertions to compare actual outcomes with expected results. This process helps ensure that code changes or new features do not introduce regressions or break existing functionality, contributing to more reliable and maintainable software.

1. **Positive Testing**

- Purpose: To verify that the function works correctly with valid inputs.
- Example: Testing the generate_password function with valid length, uppercase, and digit settings to ensure it generates a password of the expected length and character set.

I. **Test Default Password Generation**

```
def test_default_password_generation():
    password = generate_password()
    assert len(password) == 12
    assert any(c.isupper() for c in password)
    assert any(c.isdigit() for c in password)
```

```
=== Code Execution Successful ===
```

II. **Test Password Length**

```
def test_password_length():
    password = generate_password(length=16)
    assert len(password) == 16

```

```
=== Code Execution Successful ===
```

III. Test Inclusion of Uppercase Letters

```
def test_include_uppercase():
    password = generate_password(use_upper=True)
    assert any(c.isupper() for c in password)

=== Code Execution Successful ===
```

IV. Test Inclusion of Digits

```
def test_include_digits():
    password = generate_password(use_digits=True)
    assert any(c.isdigit() for c in password)

=== Code Execution Successful ===
```

V. Test Password Without Uppercase Letters

```
def test_exclude_uppercase():
    password = generate_password(use_upper=False)
    assert not any(c.isupper() for c in password)

=== Code Execution Successful ===
```

VI. Test Password Without Digits

```
def test_exclude_digits():
    password = generate_password(use_digits=False)
    assert not any(c.isdigit() for c in password)

=== Code Execution Successful ===
```

2. Security Testing

- Purpose: To ensure that the function handles security-related aspects correctly, such as not generating predictable passwords.
- Example: Testing the generate_password function to ensure it generates random and non-repeating passwords, checking that the implementation does not have vulnerabilities that could lead to predictable output.

I. **Test for Randomness and Entropy**

```
def test_password_randomness():
    passwords = {generate_password() for _ in range(100)}
    assert len(passwords) > 1
```

```
=== Code Execution Successful ===
```

II. **Test for Minimum Password Length**

```
def test_minimum_password_length():
    with pytest.raises(ValueError):
        generate_password(length=3)
```

```
=== Code Execution Successful ===
```

III. **Test for Correct Exception Handling**

```
def test_no_character_type():
    with pytest.raises(ValueError):
        generate_password(use_upper=False, use_digits=False)
```

```
=== Code Execution Successful ===
```

IV. **Test for Password Complexity**

```
def test_password_complexity():
    password = generate_password(length=12, use_upper=True, use_digits=True)
    assert any(c.islower() for c in password)
    assert any(c.isupper() for c in password)
    assert any(c.isdigit() for c in password)
```

```
=== Code Execution Successful ===
```

6.2 INTEGRATION TESTING

Integration testing ensures that different components of the system work together as expected. For Secure Gen, integration tests will focus on the interaction between the user interface, configuration management, validation, and password generation modules.

Unlike unit testing, which isolates individual functions, integration testing looks at the larger picture of how components function together within the application. This often involves testing interfaces, APIs, and communication between modules to ensure that the system as a whole meets the required specifications and performs correctly under various conditions. By identifying issues at the integration level, developers can address problems

that may not be apparent during unit testing, leading to a more robust and reliable final product.

1. Big Bang Integration Testing

- Purpose: To integrate all components or modules simultaneously and test them as a complete system.
- Characteristics: All parts of the system are integrated at once, and testing is conducted on the entire system to detect any issues.

2. Sandwich Integration Testing

- Purpose: Combines both top-down and bottom-up approaches, integrating both high-level and low-level components simultaneously.
- Characteristics: Testing is performed on both ends of the system and the integration is handled in a layered fashion

```
import unittest
import random
import string
from io import StringIO
from unittest.mock import patch
import password_generator  # Assuming the code is saved in a file named password_generator.py

class TestPasswordGeneratorIntegration(unittest.TestCase):

    @patch('sys.stdout', new_callable=StringIO)
    @patch('builtins.input', side_effect=['1', '16', '4', '5'])
    def test_generate_password_with_custom_length(self, mock_input, mock_stdout):
        # Setup the initial length and generate password
        password_generator.main()
        output = mock_stdout.getvalue().strip().split('\n')
        password = output[-1]  # Get the last line which should be the generated password

        self.assertTrue(len(password) == 16, "Password length should be 16.")
        self.assertTrue(all(c in string.ascii_letters + string.digits for c in password), "Password should contain only valid characters.")

    @patch('sys.stdout', new_callable=StringIO)
    @patch('builtins.input', side_effect=['2', '4', '5'])
    def test_generate_password_without_uppercase(self, mock_input, mock_stdout
```

```
        # Disable uppercase and generate password
        password_generator.main()
        output = mock_stdout.getvalue().strip().split('\n')
        password = output[-1]  # Get the last line which should be the
            generated password

        self.assertTrue(len(password) == 12, "Password length should be 12.")
        self.assertTrue(all(c in string.ascii_lowercase + string.digits for c
            in password), "Password should not contain uppercase letters.")

    @patch('sys.stdout', new_callable=StringIO)
    @patch('builtins.input', side_effect=['3', '4', '5'])
    def test_generate_password_without_digits(self, mock_input, mock_stdout):
        # Disable digits and generate password
        password_generator.main()
        output = mock_stdout.getvalue().strip().split('\n')
        password = output[-1]  # Get the last line which should be the
            generated password

        self.assertTrue(len(password) == 12, "Password length should be 12.")
        self.assertTrue(all(c in string.ascii_letters for c in password),
            "Password should not contain digits.")

    @patch('sys.stdout', new_callable=StringIO)
    @patch('builtins.input', side_effect=['2', '3', '5'])
    def test_generate_password_without_any_character_type(self, mock_input,
        mock_stdout):
```

```
        # Disable both uppercase and digits, which should raise an error
        with self.assertRaises(ValueError):
            password_generator.main()

    @patch('sys.stdout', new_callable=StringIO)
    @patch('builtins.input', side_effect=['1', '2', '4', '5'])
    def test_invalid_password_length(self, mock_input, mock_stdout):
        # Set an invalid length and generate password
        password_generator.main()
        output = mock_stdout.getvalue().strip().split('\n')
        password = output[-1]  # Get the last line which should be the
            generated password

        self.assertTrue(len(password) == 12, "Password length should revert to
            12 if an invalid length is set.")
        self.assertTrue(all(c in string.ascii_letters + string.digits for c in
            password), "Password should contain valid characters.")

if __name__ == '__main__':
    unittest.main()
python -m unittest test_password_generator.py
```

Running the Tests:

- To run the tests, execute the following command in your terminal

```
python -m unittest test_password_generator.py
```

CHAPTER 7

RESULTS AND SNAPSHOTS

```
Welcome to the Password Generator
Password Generator Options:
1. Set length of password
2. Toggle inclusion of uppercase letters
3. Toggle inclusion of digits
4. Toggle inclusion of special characters
5. Generate password
6. Exit
Enter your choice (1-6): 1
Enter the desired length of the password: 8
Password length set to 8.
Password Generator Options:
1. Set length of password
2. Toggle inclusion of uppercase letters
3. Toggle inclusion of digits
4. Toggle inclusion of special characters
5. Generate password
6. Exit
Enter your choice (1-6): 2
Include uppercase letters: No
Password Generator Options:
1. Set length of password
2. Toggle inclusion of uppercase letters
3. Toggle inclusion of digits
4. Toggle inclusion of special characters
5. Generate password
6. Exit
```

```
Enter your choice (1-6): 3
Include digits: No
Password Generator Options:
1. Set length of password
2. Toggle inclusion of uppercase letters
3. Toggle inclusion of digits
4. Toggle inclusion of special characters
5. Generate password
6. Exit
Enter your choice (1-6): 4
Include special characters: No
Password Generator Options:
1. Set length of password
2. Toggle inclusion of uppercase letters
3. Toggle inclusion of digits
4. Toggle inclusion of special characters
5. Generate password
6. Exit
Enter your choice (1-6): 4
Include special characters: Yes
Password Generator Options:
1. Set length of password
2. Toggle inclusion of uppercase letters
3. Toggle inclusion of digits
4. Toggle inclusion of special characters
5. Generate password
6. Exit
```

```
Enter your choice (1-6): 5
Generated Password: }"xl/+]d
Password Generator Options:
1. Set length of password
2. Toggle inclusion of uppercase letters
3. Toggle inclusion of digits
4. Toggle inclusion of special characters
5. Generate password
6. Exit
Enter your choice (1-6): 6
Exiting the Password Generator. Goodbye!

=== Code Execution Successful ===
```

CONCLUSION

Our project, “SecureGen”, implemented in Python, effectively integrates modular design principles with a user-friendly interface to deliver a robust password generation tool. By compartmentalizing user interaction, core logic, and configuration management, the system ensures both ease of maintenance and extensibility. This design approach guarantees that the password generation process is reliable and customizable, addressing various user needs through a clear and manageable codebase.

Thorough unit and integration testing confirm the application's reliability and functionality. The tests ensure that SecureGen generates secure passwords according to user specifications, handles different inputs gracefully, and maintains a smooth user experience. Overall, SecureGen stands as a dependable solution for creating strong passwords, with potential for further enhancements to improve its features and usability.

REFERENCES

1. **random module**: Python Random Module Documentation
2. **string module**: Python String Module Documentation
3. **unittest framework**: Python Unittest Documentation
4. **NIST Guidelines on Passwords**: NIST Digital Identity Guidelines