River James

river.l.james@sutdents.ittralee.ie

T00202161

Munster Technological University

5th December 2023

Source Code GitHub: <https://github.com/riverleejames/BerryFrame>

A blue and white logo

Description automatically generated

Berryframe

**A Proof-of-Concept Framework for Raspberry Pi-Based IoT Management**

Table of Contents

[1. Introduction 2](#_Toc152684005)

[1.1. Overview of the IoT Control Framework 2](#_Toc152684006)

[1.2. Significance of the Project 2](#_Toc152684007)

[1.3. Objectives and Scope of the Assignment 2](#_Toc152684008)

[2. Customised Scenario Selection / System Development 3](#_Toc152684009)

[2.1. Description of the IoT Control Framework 3](#_Toc152684010)

[2.2. Functional and Non-Functional Requirements 3](#_Toc152684011)

[2.2.1. Functional Requirements 3](#_Toc152684012)

[2.2.2. Non-Functional Requirements 4](#_Toc152684013)

[3. Design Pattern Analysis 5](#_Toc152684014)

[3.1. Exploration of Creational, Structural and Behavioural Patterns 5](#_Toc152684015)

[3.1.1. Creational Patterns 5](#_Toc152684016)

[3.1.2. Structural Patterns 7](#_Toc152684017)

[3.1.3. Behavioural Patterns 8](#_Toc152684018)

[3.2 Selected Design Patterns and Justification 11](#_Toc152684019)

[3.2.1 Singleton Pattern for ConnectionManager 11](#_Toc152684020)

[3.2.2 Factory Pattern for API Functions 11](#_Toc152684021)

[3.2.3 Observer Pattern for Connection Status Management 12](#_Toc152684022)

[3.2.4 Command Pattern for SSH Commands 13](#_Toc152684023)

[3.2. Selected Architectural Pattern and Justification 13](#_Toc152684024)

[3.3.1 Model-View-Controller (MVC) 14](#_Toc152684025)

[4. Implementation of Design and Architecture Patterns 15](#_Toc152684026)

[4.1 Integration of Design Patterns in the IoT Framework 15](#_Toc152684027)

[4.1.1 Singleton Pattern 15](#_Toc152684028)

[4.1.2 Factory Method 16](#_Toc152684029)

[4.1.3 Observer Pattern 18](#_Toc152684030)

[4.1.4 Command Pattern 19](#_Toc152684031)

[4.2 Architectural Pattern Integration 21](#_Toc152684032)

[4.2.1 Overview: 21](#_Toc152684033)

[4.2.2 Model - ConnectionManager: 21](#_Toc152684034)

[4.2.3. View - SSHView: 21](#_Toc152684035)

[4.2.4. Controller - SSHController: 21](#_Toc152684036)

[4.2.5. Integration of MVC Pattern: 22](#_Toc152684037)

[4.2.6 MVC Code Examples: 22](#_Toc152684038)

[5. Testing, Validation and Code Quality 24](#_Toc152684039)

[5.1. Testing Strategy and Approach 24](#_Toc152684040)

[5.2. Evidence of Successful Testing and Validation 24](#_Toc152684041)

[5.3. Code Smell Identification and Corrective Actions 25](#_Toc152684042)

[5.4. Documentation of Code Quality Improvements 25](#_Toc152684043)

[6. Conclusion 26](#_Toc152684044)

[6.1. Summary of Key Findings and Achievements 26](#_Toc152684045)

[6.2. Reflection on the Assignment and Objectives and Outcomes 26](#_Toc152684046)

[7. References 27](#_Toc152684047)

[8. Bibliography 27](#_Toc152684048)

# Introduction

## Overview of the IoT Control Framework

This document introduces a proof-of-concept IoT Control Framework primarily designed with Raspberry Pi. Currently, the framework offers essential functionality to control and monitor IoT devices, mainly managed through a command-line interface. It lays the foundational architecture for future integration of WiFi sensing (Vodovozov, 2022) capabilities. The framework utilises simple yet effective software design patterns and architectural strategies to ensure a user-friendly, secure, and scalable operation, setting the groundwork for more advanced features in subsequent development phases.

## Significance of the Project

The significance of this project lies in its potential to evolve into a comprehensive solution for WiFi Sensing applications using Raspberry Pi. Currently, as a proof-of-concept, it demonstrates the feasibility of a user-friendly and command-line-based control system for IoT devices. This project is crucial for understanding the essential requirements and challenges in developing such a framework and sets the stage for future enhancements, including integrating WiFi sensing capabilities.

## Objectives and Scope of the Assignment

The primary objective of this assignment is to establish a basic yet functional control framework for IoT devices, intending to expand this into a full-fledged WiFi Sensing solution. The current scope involves developing a simple, efficient, and user-friendly system that uses Raspberry Pi to manage and monitor IoT devices. The project focuses on demonstrating the core functionalities essential for any IoT control system, such as device connectivity and basic data handling. Emphasis is placed on creating a scalable and adaptable foundation that can be built upon to include more complex features, such as WiFi sensing, in future development stages. The assignment underscores the importance of a solid proof-of-concept in paving the way for advanced IoT solutions.

# Customised Scenario Selection / System Development

## Description of the IoT Control Framework

The IoT Control Framework, as conceptualised for this project, is tailored for Raspberry Pi and is currently at its proof-of-concept stage, focusing on foundational functionalities. The framework is designed to be intuitive and accessible, primarily managed through a command-line interface. It suits users who prefer a straightforward, no-frills approach to device control.

## Functional and Non-Functional Requirements

### Functional Requirements

|  |  |  |
| --- | --- | --- |
| **No.** | **Functional Requirement** | **Implementation Details** |
| 1 | SSH Connection Management | Implemented in connection\_manager.py. It uses the Paramiko library to handle SSH connections. Singleton’s design pattern ensures a single instance of the connection manager. Provides methods to establish, execute commands over, and close SSH connections. |
| 2 | Remote Command Execution | Facilitated by functions inapi\_functions.py. Leverages the ConnectionManager for executing commands on a remote server (Raspberry Pi). Offers flexibility to perform a wide range of commands for different operational needs. |
| 3 | System Statistics Retrieval | Part of api\_functions.py. Includes functions to retrieve system stats like CPU usage, memory usage, and disk space. Utilises SSH connection to fetch real-time data from Raspberry Pi. Essential for monitoring and maintaining system health. |
| 4 | Configuration Management | Configuration settings are stored in config.ini. Includes SSH connection details like host, port, username, and password. Allows for easy customisation and flexibility without altering the core codebase. Read by main.py to initialise settings for the application. |
| 5 | Unit Testing | It is implemented intest\_api\_functions.py and test\_connection\_manager.py. Uses pytest framework for testing. Ensures functionality and reliability of API functions and the ConnectionManager. Critical for validating the correctness of code and logic. |

### Non-Functional Requirements

|  |  |  |
| --- | --- | --- |
| **No.** | **Non-Functional Requirement** | **Implementation Details** |
| 1 | Performance and Efficiency | The system is optimized for resource efficiency, which is crucial in the limited-resource environment of a Raspberry Pi. The use of a singleton pattern in the connection management module reduces resource usage by ensuring only one instance of the connection manager is active. |
| 2 | Reliability and Stability | The design of the system prioritizes consistent performance. Unit tests are extensively used to ensure the system is free of frequent crashes and bugs, enhancing overall reliability and stability. |
| 3 | Maintainability | The code is organized into well-defined modules, facilitating easy updates and maintenance. The inclusion of unit testing supports quality assurance and simplifies ongoing system maintenance. |
| 4 | Security | Security is addressed through secure SSH communication protocols. The project also demonstrates a careful approach to handling configuration settings, especially those that are sensitive, to protect against potential security breaches. (config.ini) |
| 5 | Scalability | The modular design of the framework allows for scalability. It is built to easily adapt to different models of Raspberry Pi and varying project scales, offering flexibility for future enhancements. |
| 6 | Development Best Practices | The project adheres to best practices in Python development. It utilizes standard libraries and frameworks such as Paramiko for SSH communication and pytest for unit testing, ensuring a reliable and consistent development environment. |

# Design Pattern Analysis

## Exploration of Creational, Structural and Behavioural Patterns

### Creational Patterns

Creational design patterns are a fundamental part of object-oriented design and programming. They provide solutions for creating objects in a system. Here is an overview of the five critical creational patterns: Factory, Abstract Factory, Prototype, Singleton, and Builder (Gamma et al., 1994).

**Factory Method**: This pattern provides an interface for creating objects in a superclass but allows subclasses to alter the type of objects that will be made. It's advantageous when a class cannot anticipate the class of objects it needs to create or when a class wants its subclasses to specify the objects it makes (Gamma et al., 1994).

For example, a logistics management application might use a Factory Method to create different types of vehicles, like trucks or ships, depending on the delivery distance and cargo type.

**Abstract Factory**: This pattern provides an interface for creating families of related or dependent objects without specifying their concrete classes. It's useful when the system needs to be independent of how its products are made, composed, and represented (Gamma et al., 1994).

An example of the Abstract Factory pattern is a UI toolkit that provides an interface for creating windows, buttons, and scrollbars with different implementations for different operating systems.

**Prototype**: This pattern creates duplicate objects while keeping performance in mind. It involves copying existing objects to new objects rather than making them from scratch, especially when creating a new object is more expensive or complicated (Gamma et al., 1994).

For instance, in a game, the Prototype pattern can be used to duplicate complex terrains or characters with pre-loaded assets.

**Singleton**: The Singleton pattern ensures that a class has only one instance and provides a global point of access to it. This pattern is used for managing shared resources, such as a connection to a database (Gamma et al., 1994).

For example, a Singleton might manage the database connection in an application, ensuring that only one connection is active at any given time.

**Builder**: The Builder pattern is used to construct a complex object step by step, and the final step will return the object. The process of construction is separated from the absolute representation. This pattern is beneficial when an object needs to be created with many optional components or sub-objects (Gamma et al., 1994).

An example is constructing a complex meal from a fast-food restaurant menu consisting of items like a burger, a drink, chips, and a dessert.

### Structural Patterns

Structural patterns in software design focus on how classes and objects are composed to form larger structures. They simplify the structure by identifying the relationships. Here is an overview of the five critical structural patterns: Adapter, Bridge, Composite, Decorator, Facade, Flyweight, and Proxy (Gamma et al., 1994).

**Adapter**: This pattern allows interfaces of incompatible objects to work together. It involves a wrapper that translates calls to one interface into a compatible interface (Gamma et al., 1995).

For instance, an adapter can enable a modern USB-C device to charge through an older USB port.

**Bridge**: The Bridge pattern separates an object’s abstraction from its implementation so that the two can vary independently (Gamma et al., 1995).

A practical example is a program that should run on multiple platforms, where the high-level code is separated from the platform-specific parts.

**Composite**: This pattern composes objects into tree structures representing part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly (Gamma et al., 1995).

For example, a graphic design program uses shapes (circles, squares) and groups of shapes as composites.

**Decorator**: The Decorator pattern adds new functionalities to objects by placing these objects inside particular wrapper objects that contain the behaviours (Gamma et al., 1995).

For example, when adding scrolling to a text view object in a user interface, wrapping it in a ScrollDecorator.

**Facade**: This pattern provides a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem more straightforward (Gamma et al., 1995).

For instance, a computer starting up encapsulates complex operations in a simple interface.

**Flyweight**: Flyweight reduces the cost of creating and manipulating many similar objects. It shares objects to support a large quantity without a comparable increase in memory.

An example is a text editor that uses characters as flyweight objects (Gamma et al., 1995).

**Proxy**: The Proxy pattern provides a surrogate or placeholder for another object to control access to it. This could be for security reasons, handling the cost of object creation, or managing remote resources (Gamma et al., 1995).

A practical application is a lazy loading of a large image file. A proxy object initially presents a placeholder image, and the actual image is loaded on demand.

### Behavioural Patterns

Behavioural design patterns are essential in software engineering, as they help manage algorithms, relationships, and responsibilities between objects. Here is an overview of the eleven critical behavioural patterns: Chain of Responsibility, Command, Interpreter, Iterator Pattern, Mediator Pattern, Memento Pattern, Observer Pattern, State, Strategy, Template Method, and Visitor (Gamma et al., 1994).

**Chain of Responsibility**: This pattern allows an object to send a command without knowing which object will handle it. The request travels along a chain of handlers until one of them takes it (Gamma et al., 1995).

For example, in a customer support system, a request might pass through several layers (e.g., junior, senior, manager) until it is resolved.

**Command**: This pattern turns a request into a stand-alone object containing all information about the request. This transformation lets you parameterise methods with different requests, delay or queue a request's execution, and support undoable operations (Gamma et al., 1995).

For instance, a text editor can use this pattern to perform and undo text changes through a series of command objects.

**Interpreter**: The Interpreter pattern provides a way to evaluate language grammar or expression (Gamma et al., 1995).

This pattern can be used in SQL parsing, symbol processing engines, or for turning strings into numeric expressions.

**Iterator**: This pattern provides a way to access the elements of an aggregate object sequentially without exposing its underlying representation (Gamma et al., 1995).

A real-world example is traversing a collection of objects, such as a list or tree, without understanding its internal structure.

**Mediator**: The mediator pattern reduces communication complexity between multiple objects or classes. This pattern provides a mediator class that usually handles all the communications between different classes and supports easy code maintenance by loose coupling (Gamma et al., 1995).

For instance, a control tower at an airport acts as a mediator between planes.

**Memento**: This pattern is used for restoring the state of an object to a previous state. It's beneficial in the case of error recovery or undo functionality in applications (Gamma et al., 1995).

For example, in a text editor, the memento pattern can save the state of the text at various points and restore it if needed.

**Observer**: This pattern defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically (Gamma et al., 1995).

A typical example is the Model-View-Controller (MVC) architecture, where the model (subject) notifies views (observers) of any state changes.

**State**: The State pattern allows an object to change its behaviour when its internal state changes (Gamma et al., 1995).

For example, a document may change its state from draft to moderation and published, with different behaviours at each state.

**Strategy**: This pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. Strategy lets the algorithm vary independently from its clients (Gamma et al., 1995).

For instance, a sorting algorithm can be selected for different data sets at runtime.

**Template Method**: This pattern defines the skeleton of an algorithm in operation, deferring some steps to subclasses. Template Method lets subclasses redefine specific steps of an algorithm without changing the algorithm's structure (Gamma et al., 1995).

An example is a general data processing framework that defines the steps but lets subclasses implement specific processing details.

**Visitor**: This pattern lets you define a new operation without changing the classes of the elements on which it operates. It is helpful for operations across a collection of different objects (Gamma et al., 1995).

Visitors can traverse a document object model (DOM) tree by implementing different operations.

## 3.2 Selected Design Patterns and Justification

### 3.2.1 Singleton Pattern for ConnectionManager

The singleton pattern used in the ConnectionManager class provides significant advantages: it guarantees a unique instance of the connection manager, thus preventing issues arising from multiple connections and conflicting configurations. This approach ensures global access to the connection manager from anywhere in the application, enabling efficient and centralised management of SSH connections. The pattern optimises resource usage by managing these connections through a singular instance. Moreover, it maintains a consistent state of the connection manager throughout the application, ensuring reliability and uniformity in connection handling.

The utilisation of the Singleton Pattern in this context provides several advantages:

1. **Consistent State Management:** Using a Singleton for SSH connection management, your application ensures that all components interact with a constant and shared state of the SSH connection.
2. **Resource Optimization:** This pattern is beneficial in scenarios like managing SSH connections where creating multiple instances could lead to resource wastage or conflicts.
3. **Centralised Control:** The ConnectionManager provides a centralised point to manage SSH connections, simplifying the architecture and making the system more manageable.

### 3.2.2 Factory Pattern for API Functions

The APIFunctionFactory class, a pivotal element in the project's architecture, exemplifies applying the Factory Method design pattern. This design pattern, a cornerstone in object-oriented programming, facilitates object creation by defining an interface for creating objects but allows subclasses to alter the type of objects that will be made. The APIFunctionFactory serves as the abstraction layer for instantiating various API function objects, thereby encapsulating the object creation process and enhancing the modularity and maintainability of the code.

The utilisation of the Factory Method pattern in this context provides several advantages:

1. **Encapsulation of Object Creation Logic**: The centralisation of object creation logic within the factory method reduces complexities associated with direct object instantiation throughout the system.
2. **Enhanced Maintainability**: The system's maintainability is substantially improved by localising the changes required for introducing new API function types.
3. **Loose Coupling**: The factory pattern reduces dependencies within the system, as components that require API function objects rely on the factory rather than instantiating objects directly. This loose coupling paves the way for greater flexibility and ease of modification.

### 3.2.3 Observer Pattern for Connection Status Management

The Observer pattern is a fundamental design paradigm in software engineering, pivotal in scenarios demanding the maintenance of consistency between related objects without tight coupling. This behavioural pattern establishes a one-to-many dependency between objects, allowing an object, referred to as the subject, to notify a set of dependent objects, known as observers, about state changes. It is particularly beneficial in applications where a change in one object necessitates updates in others, and a direct object-to-object calling is not feasible or desired due to the need for decoupling.

In this project, the Observer pattern was chosen to manage state changes in SSH connections efficiently. The requirement was to enable various system components to remain updated about the connection status (connected or disconnected) without establishing rigid dependencies on the ConnectionManager (the subject). This pattern was an ideal choice due to its ability to:

The utilisation of the Observer Pattern in this context provides several advantages:

1. **Ensure Real-time Notifications**: Guarantee immediate and consistent notification to interested parties about the connection status.
2. **Promote Decoupling**: Allow various components to observe connection states without hard-coding dependencies, enhancing modularity.
3. **Simplify Extendibility**: Facilitate easy addition of new observers to respond to connection status changes.

### 3.2.4 Command Pattern for SSH Commands

The Command Pattern is a behavioural design pattern that turns a request into a stand-alone object containing all information about the request. This transformation allows command-based operations, like queuing, logging, and undoing, to be performed more efficiently and systematically. The pattern involves creating a central command interface with a method for executing the command and concrete command classes that implement this interface.

The command pattern is a strategic choice for the project, which deals with various SSH and file management operations. It provides a clear and flexible structure for handling different commands, enhances the maintainability of the code, and supports potential expansions or modifications with minimal impact on the existing system architecture. This approach aligns with best practices for creating scalable and robust software systems, particularly in high-command management contexts.

The utilisation of the Command Method pattern in this context provides several advantages:

1. **Decoupling Command Issuer and Executor:** By separating the invoker and receiver, your project benefits from reduced dependencies between system components. This enhances modularity and flexibility.
2. **Extension and Maintenance:** Adding new commands becomes simpler and doesn't affect existing code. This makes the system more extensible and easier to maintain.
3. **Reusability and Organized Code:** Commands encapsulate all information needed for an action. This makes commands reusable and leads to more organised and understandable code.
4. **Support for Undo Operations:** The pattern can easily be extended to support undo functionalities, which is invaluable in systems where actions might need to be rolled back.
5. **Queueing and Logging of Commands:** The pattern makes implementing command logging and queuing straightforward, which can be essential for tracking and managing executed operations, especially in complex systems like those involving SSH interactions.

## Selected Architectural Pattern and Justification

### 3.3.1 Model-View-Controller (MVC)

The Model-View-Controller (MVC) architectural pattern has been chosen for the BerryFrame project, a Raspberry Pi control framework. This decision is founded on several key considerations that align with the unique requirements and goals of the project.

Firstly, the MVC pattern offers a clear separation of concerns, which is crucial for maintaining the modularity and scalability of the BerryFrame system. This separation is achieved by dividing the application into three interconnected components: the Model, which encapsulates the application's data and associated logic; the View, responsible for presenting data to the user; and the Controller, which interprets user inputs and transforms them into actions for the Model or View. This structure enhances maintainability and facilitates easier debugging and testing, as each component can be developed and tested independently.

Secondly, the MVC pattern supports a high degree of flexibility and extensibility, which are vital for the evolving nature of the BerryFrame project. Given the Raspberry Pi's diverse range of applications and the ongoing advancements in software design patterns, the ability to adapt and extend the framework is paramount. The MVC architecture allows for easy modification or addition of new Views or Models without requiring significant changes to the existing codebase, accommodating future enhancements with minimal disruption.

Furthermore, the MVC pattern is well-suited for collaborative development environments like the BerryFrame project. With distinct roles and responsibilities assigned to each component, multiple developers can work concurrently on different aspects of the application without causing significant merge conflicts.

In addition, the MVC pattern aligns well with the user-centric design philosophy of the BerryFrame project. By separating the View from the Model, the framework ensures that the user interface can be modified or replaced without altering the underlying logic. This separation is crucial for developing user interfaces tailored to different user groups' specific needs and preferences, a key consideration given the diverse user base of Raspberry Pi enthusiasts and professionals.

Lastly, the widespread adoption and well-documented nature of the MVC pattern in software engineering provide an added advantage. The availability of extensive resources, community support, and best practices significantly reduces the learning curve for new developers joining the project and enhances the overall robustness of the application.

In conclusion, selecting the MVC architectural pattern for the BerryFrame project is a strategic decision that leverages the strengths of this well-established design approach. It ensures modularity, flexibility, collaborative development, user-centric design, and accessibility, all of which are critical for the success of the BerryFrame project in both academic and practical domains.

# Implementation of Design and Architecture Patterns

## Integration of Design Patterns in the IoT Framework

### 4.1.1 Singleton Pattern

The ConnectionManager class, implemented as a singleton, ensures a single instance throughout the application using a private static attribute \_instance and a private constructor, coupled with a get\_instance method for global access. Upon the first invocation of get\_instance, it instantiates ConnectionManager, and subsequent calls to this method return the same instance. The class initialises a paramiko.SSHClient for SSH connections and automatically adds missing host keys. It provides essential methods like connect, disconnect, and execute\_command for connection management, facilitating streamlined and consistent SSH connection handling across the application.

**Critical Aspects of the Singleton Implementation in ConnectionManager:**

1. **Single Instance Guarantee:**

The Singleton pattern is enforced by ensuring that only a single instance of the ConnectionManager can exist. This is achieved by maintaining a private class attribute \_instance, which refers to the single created instance of the class.

class ConnectionManager:

\_instance = None

# ...

1. **Controlled Instance Creation:**

The ConnectionManager class defines a static method get\_instance, which is responsible for managing the instantiation of the class. If no instance exists, this method creates and returns a new instance. If an instance already exists, it simply returns the existing instance.

@staticmethod

def get\_instance():

if ConnectionManager.\_instance is None:

ConnectionManager.\_instance = ConnectionManager()

return ConnectionManager.\_instance

1. **Prevention of Multiple Instantiations:**

The constructor (\_\_init\_\_) of the ConnectionManager class is designed to prevent the creation of multiple instances. If an attempt is made to create a second instance, a RuntimeError is raised, thus enforcing the Singleton property.

def \_\_init\_\_(self):

if ConnectionManager.\_instance is not None:

raise RuntimeError("Singleton class, use get\_instance() method")

self.ssh\_client = paramiko.SSHClient()

self.ssh\_client.set\_missing\_host\_key\_policy(paramiko.AutoAddPolicy())

### 4.1.2 Factory Method

1. **APIFunctionFactory (Factory Method Pattern)**

The APIFunctionFactory class implements the Factory Method pattern. It creates instances of different API functions (ExecuteRemoteCommand, GetSystemStats, UploadFile, DownloadFile) based on the provided function\_type.

This design allows quickly adding new API function types without altering the factory's code structure.

class APIFunctionFactory:

def create\_api\_function(self, function\_type):

if function\_type == "ExecuteRemoteCommand":

return ExecuteRemoteCommand()

elif function\_type == "GetSystemStats":

return GetSystemStats()

elif function\_type == "UploadFile":

return UploadFile()

elif function\_type == "DownloadFile":

return DownloadFile()

else:

raise ValueError("Invalid function type")

1. **APIFunction (Base Class)**

APIFunction serves as a base class for all API functions. It defines a standard interface by declaring the execute method, which all subclasses must implement.

This approach ensures that all API function subclasses conform to a standard structure, promoting code reusability and consistency.

class APIFunction:

"""

Base class for API functions.

This class serves as a template for all API functions, requiring subclasses

to implement the execute method.

"""

def execute(self, \*args, \*\*kwargs):

raise NotImplementedError("Subclasses should implement this!")

1. **ExecuteRemoteCommand (Concrete Implementation and Singleton Usage)**

ExecuteRemoteCommand is a subclass of APIFunction and provides a specific implementation of the execute method for executing commands on a remote server.

Inside its execute method, it utilises the Singleton ConnectionManager instance. By calling ConnectionManager.get\_instance(), the same SSH connection is used throughout the application, thus maintaining a consistent connection state and optimising resource usage.

class ExecuteRemoteCommand(APIFunction):

"""

API function for executing a remote command.

This class allows execution of a command on a remote server using the ConnectionManager.

It handles specific exceptions related to SSH connections and command execution.

"""

def execute(self, command):

try:

# Get the singleton instance of ConnectionManager

connection\_manager = ConnectionManager.get\_instance()

# Execute the command and get the output

output = connection\_manager.execute\_command(command)

# Trim the output to remove unnecessary whitespaces and newlines

return output.strip()

except (SSHException, AuthenticationException, ConnectionResetError) as e:

# Handle specific exceptions (e.g., connection issues, command errors)

return f"Error executing command '{command}': {e}"

The factory encapsulates the logic of instantiating API function objects, isolating the instantiation process from the rest of the system. This isolation simplifies the object creation process and gives the system significant flexibility. For instance, introducing a new API function type involves minimal modifications to the existing codebase, primarily confined to the create\_api\_function method.

### 4.1.3 Observer Pattern

The ConnectionManager class, acting as the subject, includes functionalities to manage observers (other components or classes interested in the connection status) and notify them about the changes in the SSH connection state.  
  
**Key Implementation Aspects:**

**Observer Management:** The ConnectionManager class includes methods to attach or detach observers. Each observer is an object that implements a specific interface, generally containing an update method.

# Example method in ConnectionManager for attaching an observer

def attach(self, observer):

self.observers.append(observer)

# Example method in ConnectionManager for detaching an observer

 def detach(self, observer):

self.observers.remove(observer)

# Example method in ConnectionManager for notifying observers

def notify\_observers(self, message):

for observer in self.observers:

observer.update(message)

**Notification Mechanism:** Upon a change in the connection state, such as establishing or terminating an SSH connection, the ConnectionManager calls the update method of each registered observer, passing along relevant information about the state change.

def notify\_observers(self, message):

for observer in self.observers:

observer.update(message)

# When a connection is established

def connect(self, ...):

# Connection logic

self.notify\_observers("Connected")

# When a connection is terminated

def disconnect(self):

# Disconnection logic

self.notify\_observers("Disconnected")

(BerryFrame) river@Rivers-MBP BerryFrame % python main.py

[Logger] SSH Connection status changed: Connected

[Alert] Attention: SSH Connection is now Connected

Connected to 100.81.129.125

`.::///+:/-. --///+//-:`` pi@pihole

`+oooooooooooo: `+oooooooooooo: ---------

/oooo++//ooooo: ooooo+//+ooooo. OS: Raspbian GNU/Linux 11 (bullseye) armv7l

`+ooooooo:-:oo- +o+::/ooooooo: Host: Raspberry Pi 3 Model B Rev 1.2

`:oooooooo+`` `.oooooooo+- Kernel: 6.1.58-v7+

`:++ooo/. :+ooo+/.` Uptime: 9 mins

...` `.----.` ``.. Packages: 1451 (dpkg)

.::::-``:::::::::.`-:::-` Shell: bash 5.1.4

-:::-` .:::::::-` `-:::- CPU: BCM2835 (4) @ 1.200GHz

`::. `.--.` `` `.---.``.::` Memory: 191MiB / 921MiB

.::::::::` -::::::::` `

.::` .:::::::::- `::::::::::``::.

-:::` ::::::::::. ::::::::::.`:::-

:::: -::::::::. `-:::::::: ::::

-::- .-:::-.``....``.-::-. -::-

.. `` .::::::::. `..`..

-:::-` -::::::::::` .:::::`

:::::::` -::::::::::` :::::::.

.::::::: -::::::::. ::::::::

`-:::::` ..--.` ::::::.

`...` `...--..` `...`

.::::::::::

`.-::::-`

[Logger] SSH Connection status changed: Disconnected

[Alert] Attention: SSH Connection is now Disconnected

Disconnected

### 4.1.4 Command Pattern

The command pattern fundamentally separates the object that invokes the operation from the one that knows how to perform it. It involves five components:

1. Command: Declares an interface for executing an operation.
2. ConcreteCommand: Defines a binding between a Receiver object and an action.
3. Client: Creates a ConcreteCommand object and sets its receiver.
4. Invoker: Asks the command to carry out the request.
5. Receiver: Knows how to perform the operations.

**Implementation in Your System**

This pattern is used in several components, notably in ssh\_command.py, remove\_file\_command.py, list\_files\_command.py, neofetch\_command.py, and command\_invoker.py.

SSHCommand Class (ConcreteCommand)

In ssh\_command.py, the SSHCommand class is a perfect example of a ConcreteCommand. It encapsulates the execution logic for SSH commands.

class SSHCommand:

def \_\_init\_\_(self, ssh\_client, command):

self.ssh\_client = ssh\_client

self.command = command

def execute(self):

# Logic to execute SSH command

The SSHCommand binds a ssh\_client (Receiver) with a **command** (action).

RemoveFileCommand Class (ConcreteCommand)

Similarly, in remove\_file\_command.py, the RemoveFileCommand class represents another ConcreteCommand, specifically for removing files.

class RemoveFileCommand:

def \_\_init\_\_(self, ssh\_client, file\_path):

self.ssh\_client = ssh\_client

self.file\_path = file\_path

def execute(self):

# Logic to remove a file

CommandInvoker Class (Invoker)

In command\_invoker.py, the CommandInvoker class acts as the Invoker. It's responsible for initiating the commands.

class CommandInvoker:

def \_\_init\_\_(self):

self.commands = []

def store\_command(self, command):

self.commands.append(command)

def execute\_commands(self):

for cmd in self.commands:

cmd.execute()

The command pattern implementation effectively demonstrates its strengths in managing operations, particularly with SSH interactions. By encapsulating command execution in distinct classes and delegating the execution to an invoker, your design achieves a high level of modularity, flexibility, and maintainability.

## 4.2 Architectural Pattern Integration

### 4.2.1 Overview:

The BerryFrame application is structured around the MVC architectural pattern, which separates the application's concerns into three interconnected components: the Model, the View, and the Controller.

### 4.2.2 Model - ConnectionManager:

Purpose: Manages SSH connections and encapsulates the business logic related to connection state and execution of commands.

Implemented as a singleton, it uses the Paramiko library to handle SSH connections. It also acts as a subject in the Observer pattern, notifying observers about connection status changes.

### 4.2.3. View - SSHView:

Purpose: Responsible for presenting data to the user, specifically messages related to SSH operations like connection status and errors.

Implementation: It provides a user-friendly interface for displaying messages, abstracting the complexities of user interaction. The primary method, `show\_message`, uses standard output for message display, keeping the class versatile for command-line applications.

### 4.2.4. Controller - SSHController:

Purpose: Acts as an intermediary between the Model and the View. It manages user input, processes it, and utilises the Model to retrieve or manipulate data.

Implementation: It leverages the `ConnectionManager` for SSH connections and employs the `SSHView` for displaying relevant messages to the user. This class demonstrates how different components of the MVC pattern interact.

### 4.2.5. Integration of MVC Pattern:

The MVC pattern is effectively demonstrated in the BerryFrame application with a clear separation of concerns. The Model (`ConnectionManager`) handles the data and logic; the View (`SSHView`) manages user interface and presentation; and the Controller (`SSHController`) bridges the Model and View, orchestrating user input and system output.

### 4.2.6 MVC Code Examples:

#### 4.2.6.1 Model - ConnectionManager:

This snippet shows how the ConnectionManager class manages an SSH connection. It's a part of the Model, handling the application's data logic.

class ConnectionManager:

# Singleton implementation details...

def connect(self, hostname, username, password):

# Initialize SSH client and connect to the server

self.ssh\_client = paramiko.SSHClient()

self.ssh\_client.set\_missing\_host\_key\_policy(paramiko.AutoAddPolicy())

self.ssh\_client.connect(hostname, username=username, password=password)

# Notify observers about the connection status

self.notify\_observers("SSH connection established.")

def execute\_command(self, command):

# Executing a command on the SSH server

stdin, stdout, stderr = self.ssh\_client.exec\_command(command)

return stdout.read()

#### 4.2.6.2 View - SSHView:

Here, the SSHView class provides a simple interface for displaying messages, a key aspect of the View in the MVC pattern.

class SSHView:

def show\_message(self, message):

print(message)

# Example usage:

ssh\_view = SSHView()

ssh\_view.show\_message("SSH connection established.")

#### 4.2.6.3 Controller - SSHController:

The SSHController class acts as the Controller, mediating between the Model (ConnectionManager) and the View (SSHView). It processes user input, uses the Model to perform actions, and updates the View.

class SSHController:

def \_\_init\_\_(self, connection\_manager, ssh\_view):

self.connection\_manager = connection\_manager

self.ssh\_view = ssh\_view

def establish\_connection(self, hostname, username, password):

try:

self.connection\_manager.connect(hostname, username, password)

self.ssh\_view.show\_message("Connected to SSH server.")

except Exception as e:

self.ssh\_view.show\_message(f"Connection failed: {e}")

def execute\_ssh\_command(self, command):

output = self.connection\_manager.execute\_command(command)

self.ssh\_view.show\_message(f"Command output: {output}")

These code snippets effectively illustrate the distinct roles and interactions of the Model, View, and Controller in the BerryFrame application, adhering to the principles of the MVC pattern. The ConnectionManager encapsulates the business logic, the SSHView handles data presentation, and the SSHController coordinates between them, managing user input and system output.

# Testing, Validation and Code Quality

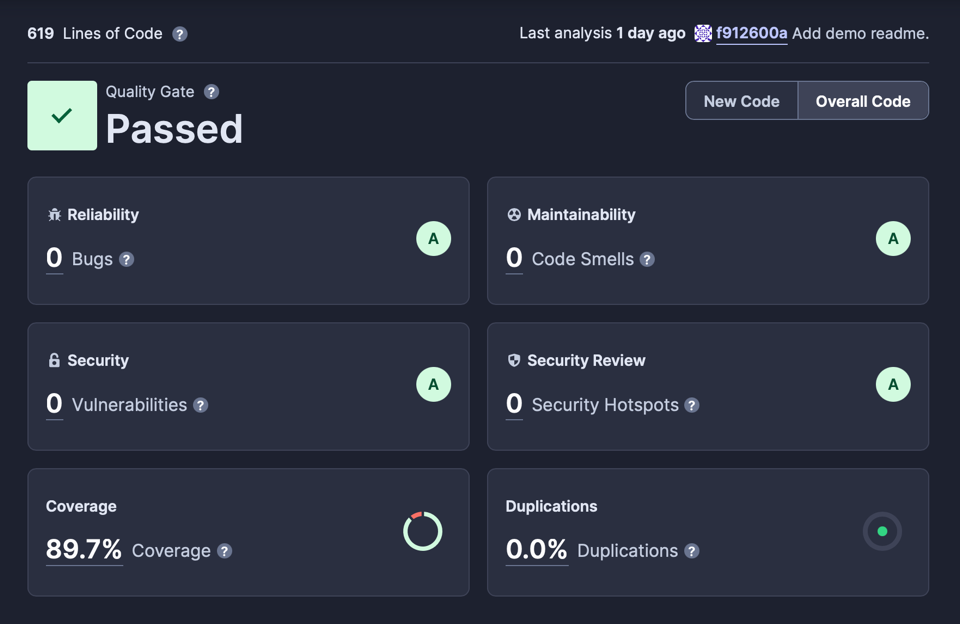
## Testing Strategy and Approach

I implemented a rigorous testing strategy in my project by integrating JetBrains Qodana (JetBrains, 2023), Sonarcloud (Sonar, 2023), and GitHub Actions (GitHub, 2023b) into my workflow. This strategy was designed to execute tests on each pull request, ensuring that new changes did not introduce regressions or decrease code quality. My approach was to establish a quality gate, setting a target of 80% code coverage, which I surpassed by achieving 89.7% coverage. This rigorous testing methodology guaranteed that my code was thoroughly tested and validated at every stage of development.

Using Tox (Too, 2023) and pytest (pytest, 2023) with coverage.xml, I conducted detailed test cases covering various scenarios. These tools provided accurate code coverage results, ensuring my test cases were comprehensive and practical. The test data was carefully chosen to simulate real-world scenarios, and I meticulously verified that the outcomes matched the expected results. This approach helped me ensure the robustness and reliability of my code.

## Evidence of Successful Testing and Validation

The evidence of my successful testing and validation efforts is reflected in the metrics I achieved: 89.7% code coverage, 0 bugs, 0 vulnerabilities, 0 code smells, 0 security hotspots, and 0.0% duplications. These figures were not just targets but were indicative of the high standard of code quality I maintained throughout the project.



## Code Smell Identification and Corrective Actions

I utilised the static code analyser Pylint (Pylint, 2023) for code smell identification, which is pivotal in maintaining code quality. It provided valuable insights and refactoring suggestions, which I actively incorporated into my development process. This tool was instrumental in helping me identify and rectify issues early, thereby preventing potential problems in later stages of development. My commitment to clean code practices was evident in how I structured and maintained my codebase. I adhered to principles such as modularity, readability, and simplicity. The use of pylint further reinforced these practices by providing guidance on refactoring techniques, which I applied to improve the maintainability and clarity of my code.

## Documentation of Code Quality Improvements

The improvements in code quality were not just qualitatively felt but also quantitatively measured and documented. Tools like Qodana (JetBrains, 2023) and SonarCloud (Sonar, 2023) provided detailed reports and insights into code quality, which were invaluable for tracking progress. I especially appreciated Qodana's user-friendly UI and deep integration with JetBrains (JetBrains, n.d.) IDEs, making identifying and addressing issues more efficient. Qodana's capability to conduct 88 code quality inspections was particularly noteworthy, as it identified issues that SonarCloud had missed.

A screenshot of a computer

Description automatically generated

In addition to these tools, I integrated GitHub Dependabot (GitHub, 2023a) into my workflow. Dependabot scanned my dependencies for security issues, and it can be further configured to automatically open pull requests for updating dependencies as new versions become available. This automation would ensure that my project consistently used the most secure and up-to-date dependencies, further bolstering the overall security and integrity of the software.

# Conclusion

## Summary of Key Findings and Achievements

1. **Development of a Basic Control Framework:** A proof-of-concept has been created that offers essential functionalities for controlling and monitoring IoT devices via a command-line interface. This forms the groundwork for more sophisticated applications in the future.
2. **Implementation of Efficient Software Design Patterns:** The framework effectively incorporates design patterns such as Singleton, Factory Method, Command Pattern, and Observer, enhancing its maintainability, scalability, and robustness.
3. **Focus on Fundamental Functionalities:** Core functionalities like SSH Connection Management, Remote Command Execution, and System Statistics Retrieval have been successfully implemented, ensuring the framework's utility and reliability.
4. **Adherence to Non-Functional Requirements:** The project demonstrates a strong commitment to performance, efficiency, reliability, security, and development best practices crucial for IoT applications.
5. **Establishment of a Scalable and Adaptable System:** The framework's modular design and architecture allow for future expansions, particularly towards integrating WiFi sensing capabilities.

## Reflection on the Assignment and Objectives and Outcomes

Reflecting on the project, it's evident that the primary objectives have been met. The assignment's goal was to develop a basic yet functional control framework for IoT devices, emphasising Raspberry Pi, and this has been achieved. The framework serves as a solid proof-of-concept, demonstrating the viability of a user-friendly and command-line-based system for IoT control.

The outcomes of this project have laid a strong foundation for future enhancements. While the current version focuses on basic functionalities, it sets the stage for incorporating more advanced features, such as WiFi sensing. The modular design and adherence to software development best practices ensure that the framework can evolve and adapt.

# References

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. M. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software* (1st ed.). Addison-Wesley Professional. <http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612/ref=ntt_at_ep_dpi_1>

GitHub. (2023a). *Dependabot*. <https://github.com/dependabot>

GitHub. (2023b). *Features • GitHub Actions*. <https://github.com/features/actions>

JetBrains. (n.d.). *JetBrains: Essential tools for software developers and teams*. Retrieved December 5, 2023, from <https://www.jetbrains.com/>

JetBrains. (2023). *Qodana: Static Code Analysis Tool by JetBrains*. <https://www.jetbrains.com/qodana/>

pytest. (2023). *pytest: helps you write better programs — pytest documentation*. <https://docs.pytest.org/en/7.4.x/>

Sonar. (2023). *SonarCloud Features | Sonar*. <https://www.sonarsource.com/products/sonarcloud/>

Too. (2023). *tox*. <https://tox.wiki/en/4.11.4/>

Vodovozov, G. (2022, December 20). *What Is Wi-Fi Sensing? - Definition, Applications, Benefits*. <https://nami.ai/blog/what-is-wi-fi-sensing/>

# Bibliography

Freeman, E., Robson, E., Sierra, K., & Bates, B. (2020). *Head First Design Patterns, 2nd Edition*. <https://www.oreilly.com/library/view/head-first-design/9781492077992/>

Martin, R. C. (2017). *Clean Architecture: A Craftsman’s Guide to Software Structure and Design* (1st ed.). Prentice Hall Press.

Martin, R. C. (2008). *Clean Code: A Handbook of Agile Software Craftsmanship* (1st ed.). Prentice Hall PTR.

Gamma, E., Helm, R., Johnson, R., & Vlissides, J. M. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software* (1st ed.). Addison-Wesley Professional.

Fowler, M. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.