INDIVIDUAL ASSIGNMENT

TECHNOLOGY PARK MALAYSIA

CT087-3-3-RTS

REAL TIME SYSTEMS

UC3F2205CS(DA)

HAND OUT DATE : 18 MAY 2022

HAND IN DATE : 04 SEPTEMBER 2022

WEIGHTAGE : 50%

NAME : ONG CHENG KEI

TP NUMBER : TP055620

LECTURER NAME : ASSOC. PROF. DR. IMRAN MEDI

Implications of Program Design on Real-Time System Performance: Mediator Design Pattern vs Observer Design Pattern

ONG CHENG KEI TP055620

Abstract

The computing power of computers has increased tremendously in the last decade, and this growth is followed by the demand for much more capable computing systems such as real-time systems. Real-time systems are systems that must satisfy time constraints in addition to their functional requirements; these systems also must be efficient and predictable. Software design patterns are blueprints intended for designing efficient software. Thus, this research aims to study the implications of software design patterns on real-time systems performance. Particularly, the observer and mediator design patterns are compared by implementing a proposed simulation of a real-time system in both design patterns. The proposed simulation simulates an automated passport control system used by multiple passengers. The performance of both designs will be evaluated via micro-benchmark and CPU and memory profiling. According to the results obtained, the implemented observer design pattern is slightly more efficient than the implemented mediator design pattern.

Key Terms: Real-time systems, Software design patterns, Observer design pattern, Mediator design pattern, Micro-benchmarking, CPU and memory profiling, Java

1.0 Introduction

The increasing need for automation and real-time processing of data implies that real-time systems (RTS) have gradually become more relevant and important in this day and age. According to IDC, an estimated 79.4 zettabytes of data will be generated by 2025 and 30% of it will require real-time processing by RTS (Intel, n.d.). Besides, it can be observed that RTS has many applications in different fields such as process control systems for manufacturing, machine vision systems, and healthcare monitoring systems (Intel, n.d.). In essence, RTS can be defined as information technology systems that are designed to respond to events within a specified and predictable period, otherwise, the systems might incur unintended consequences to the users (Intel, n.d.). In addition, RTS has a few distinct characteristics such as determinism, high performance, priority-based scheduling and safety to ensure that the system can deliver its intended value without any negative side effects (Wind River Systems, n.d.). Under the hood, RTS is run by a real-time operating system (RTOS) instead of a general-purpose operating system (GPOS) found in most computers. Unlike GPOS, RTOS comes in a smaller footprint with fewer features as it is meant to execute a specific set of tasks with a consistent and predictable time frame, whereas GPOS is built to handle multiple usage purposes and is optimized to execute different tasks within a variable time frame (NI, 2022). Another difference that can be noticed is that GPOS does not strictly follow the priorities set by developers, because GPOS is built to ensure that all processes receive some CPU time, thus a low priority thread will still get to execute in the presence of high priority threads (NI, 2022). Moving on, RTS can be divided into two types which are hard RTS and soft RTS. Hard RTS must strictly follow the time

constraints set by the developers otherwise the systems will fail and cause catastrophic consequences even missing a single deadline (Intel, n.d.). Soft RTS can still function even after missing a deadline, but the usefulness of the response will slowly degrade as time passes the deadline (Intel, n.d.).

Software design patterns are high-level descriptions of solutions to software design problems (Shvets, 2021). Typically, a software design solution will incorporate a few design patterns to solve some of the design problems faced in the project. Currently, all software design patterns can be classified into three types which are creational patterns, structural patterns, and behavioural patterns (Rahman, 2019). According to Rahman (2019), creational patterns dictate how objects in an object-oriented development system are instantiated or created. Structural patterns dictate the structure and composition of the objects so that they are efficient and flexible (Shvets, 2021). Lastly, behavioural patterns dictate the interaction and communication between classes. In this paper, only behavioural patterns will be investigated as it closely reflects a typical problem faced in RTS designing which is how to coordinate the communications between different threads. Specifically, the mediator design pattern and observer design pattern are chosen to be investigated in this study. The mediator design pattern restricts classes to only communicate via a mediator object whereas the observer design pattern allows classes to have a subscription mechanism that notifies other classes whenever an event has occurred (Shvets, 2021). Having a well-planned software design blueprint is crucial for RTS development as it helps to eliminate any potential design problems that might creep in or go unnoticed and cause a system failure later on. Therefore, a thorough investigation of different design patterns in RTS should be carried out to determine their implications in RTS.

Thus, this paper attempts to investigate the performance implications of adopting a mediator software design pattern and an observer software design pattern in an RTS. To do this, two Java concurrent programs are proposed to simulate a busy automated passport control system (APCS). Both simulation program behaves identically where there will be passengers performing actions such as placing their passport on a scanner and the APCS will have to respond to those actions promptly. A detailed explanation of the simulation will be provided in the methodology section of this paper. To eliminate bias, both simulations will have the same number of threads running for the APCS. The difference between the two simulations is that the first simulation adopts the mediator design pattern in which there will be a mediator thread that facilitates the overall execution of APCS by sending instructions and data to the targeted threads for execution. In this simulation, aside from the mediator threads, other threads would not know the current state of the system but blindly follow the instructions provided by the mediator thread. On the other hand, the second simulation adopts the observer design pattern. In this simulation, all the threads will be responsible for determining which instructions to execute whenever they get notified about the occurrence of events concerned. In short, each thread can listen for events that are emitted by other threads to determine their next action. The performance of these two simulations proposed will be benchmarked and profiled to compare their performance in terms of resource usage and executed time.

The following of this paper is divided into several sections which are the literature review, methodology, results and discussion, and

conclusions. The literature review section uncovers all the necessary domain knowledge in real-time system considerations and micro-benchmarking in Java. This section also reviews and analyses similar research carried out. After that, the methodology section provides an in-depth explanation of the scenario and implementation details for each of the simulations. The section also details the overall approach taken to benchmark and profile the two simulations proposed. The results and discussion section highlights the results obtained from benchmarking and profiling and provide interpretations of the results obtained for both simulations. Finally, this paper ends with a conclusion and some suggestions for future work.

2.0 Literature Review

2.1 Real-time Systems Considerations

The logical correctness of real-time systems is defined by their ability to compute an expected response and output the response on time (Laplante, 2004). Failure to achieve either one of the criteria will cause catastrophic events in hard real-time systems, while significantly degrading the value of response generated by soft real-time systems. Therefore, it is important to identify and understand all the important considerations and characteristics that an RTS should fulfil so that it is guaranteed no failure will occur.

One important consideration is the execution time of RTS (Cedeño & Laplante, 2007; Osama, 2021; Sharma et al, 2015). Typically, developers measured the performance of RTS in worst-case execution time (WCET) which calculates the maximum possible time required for an RTS to process a given task (Sharma et al., 2015). WCET can assure the developers that the RTS will still perform as expected if an unexpected worst scenario occurs. In addition, Osama (2021) cited an RTOS should be deterministic with a guaranteed worst-case interrupt latency and guaranteed worst-case context switch times. Determinism in RTOS will cause the system to be predictable in terms of the execution steps in each process, this allows the execution time to be measured extensively and achieve an understanding of the performance of RTS with its underlying hardware (Osama, 2021). Next, interrupt latency is another key characteristic that measures the time taken for an RTOS to process interrupt events, and a low interrupt latency implies that the RTOS is very responsive to events (Osama, 2021). Usually, RTOS will receive interrupts in their current workflow when there exists a more urgent process that needs to be executed; this causes the RTOS to pause the current processes so that the appropriate interrupt service routine can be called to handle the interrupt event. Ideally, short interrupt latency is preferable as it minimizes the risk of RTS missing any deadlines. Lastly, Osama cited that a guaranteed context switching time is also important to determine the performance of RTS as it represents the overhead needed in handling interrupts. A high context switching time implies that the RTS requires a longer time in preparing the system to handle the interrupt before processing it. Similarly, Cedeño and Laplante (2007) listed some of the basic requirements for an RTS. The authors cited that low overhead, deterministic synchronization, and predefined latencies are important time considerations in developing an RTS. Low overhead allows the RTS to quickly perform context switch to handle interrupts. A deterministic synchronization implies that the threads are able to initiate communication with each other within a predictable timeframe. Predefined latency is crucial for RTS as it must always produce a response on time, and having a predefined latency means that the RTS will be able to generate a response after a certain amount of time receiving an event. Another aspect of RTS

that needs to be considered is preemptive and priority levels (Cedeño & Laplante, 2007). According to the authors, processes in RTS must be able to be pre-empted whenever there is a more urgent process to be executed. The urgency of a process is calculated based on its priorities. Therefore, RTOS must be able to assign correct priorities to each event. These two aspects are related to each other to enable RTOS to prioritize urgent events so that urgent and important deadlines are not missed. In short, an RTS must be predictable and efficient in prioritizing and handling interrupts.

2.2 Micro-benchmarking in Java

Micro-benchmarking is a type of performance testing with a similar granularity to unit testing; microbenchmark aims to test the performance metrics such as throughput and execution time of a specific piece of code (Costa et al., 2021; Samoaa & Leitner, 2021). Unfortunately, similar to other high-level programming languages, benchmarking in Java is difficult due to many factors that might affect the results of performance testing, such as different Java Virtual Machine implementations, different hardware platforms, and unpredictable garbage collection runs (Georges et al., 2007). Thus, performance testing should be carefully written to model the actual usage patterns (Leitner & Bezemer, 2017). According to Leitner & Bezemer (2017), Caliper by Google and Java Microbenchmark Harness (JMH) by OpenJDK are examples of low-level benchmarking frameworks available in the Java ecosystem that provides an interface to writing performance testing code. The authors cited that JMH is much more preferable and widely used by the community as it has better low-level control on the Java Virtual Machine such as garbage collections and just-in-time (JIT) compilation. Diving deeper into JMH, the execution of JMH can be summarized into four steps; the first step involves invoking benchmark fixtures, which are processes that are necessary to prepare the benchmark method to be tested such as set up and tear down methods. The second step is running warmup iterations for the benchmark method followed by running the actual benchmark iterations. Lastly, JMH will repeat the execution of the steps above in a new process if more than one fork is specified; each iteration of warmup and benchmark iterations represents one fork (Costa et al., 2021). On a high-level view, JMH is a framework that offers developers the flexibility to write microbenchmarks via Java annotations on top of the methods to be benchmarked and has a variety of parameters that can be configured (Samoaa & Leitner, 2021). However, one downside of configuring parameters is it comes with a longer execution time according to Samoaa and Leitner (2021). The authors cited that the execution model of JMH will instantiate a benchmark instance for all possible combinations of parameters value, thus a JMH benchmark with m parameters and n values will result in n^m benchmarks which are computationally expensive if the parameter values are large.

Although JMH is a stable microbenchmark framework, there are still some issues that can affect the performance results obtained. A study conducted by Costa et al (2021) identified the top 5 most common bad practices in writing JMH tests that might significantly affect the results. These bad practices include not using a value returned in the benchmark method, running a benchmark with zero forks, and using a final primitive for benchmark input. All of these practices listed out by the authors can potentially introduce unpredictable optimization techniques such as deadcode elimination and constant folding used by the JVM. Thus, this results in inconsistency and unreliable benchmark results (Costa et al., 2021). To validate their findings, the authors submitted 7 pull requests to open-source software that fixed bad JMH practices that have a

significant impact on the benchmark results; 6 out of 7 pull requests were acknowledged, but one is rejected due to the software is migrating away from JMH (Costa et al., 2021). This shows that the arguments made by the authors are valid.

2.3 Similar Work

Similar work was conducted by Rana and Wanabrahman (2017), the authors investigated the impact of applying software design patterns on real-time software efficiency. The authors proposed to measure the effects of three design patterns which are state, observer, and strategy by implementing a scenario that simulates an instant messaging system. To determine whether there are any efficiency gains, the authors implemented the same simulation scenario but without any design patterns applied. The authors measured the performance of the simulation by getting the timestamps of outputs printed on the console. Then, the authors subtract the timestamp from the timestamp in the previous output to obtain the response time needed to process one request. The authors then find the average response time for each of the simulation implementations. Based on the results, simulations implemented with the state and observer design patterns perform slightly better than their alternative implementations without design patterns. To be exact, the state pattern improves the performance by an average of 0.012 seconds, while the observer pattern improves the performance by an average of 0.005 seconds. However, the strategy pattern observed a minimal decrease in performance by an average of 0.001 seconds. The authors backed this finding by citing the strategy pattern is meant for improving code readability instead of the overall performance of the application. Then, the authors summed up their work by urging for more research effort to be put into investigating other design patterns and their effects on real-time applications.

The study by Rana and Wanabrahman (2017) provides a good starting ground for future research, but it has some limitations that can be addressed. Firstly, the study conducted did not perform a direct comparison between the design patterns. A comparison between different design patterns can provide more insights into the optimal software design pattern for real-time applications. In addition, the manual calculation of the response time of each response produced in the console during the simulation can be unreliable and prone to error compared to the usage of a sophisticated benchmarking framework.

3.0 Methodology

3.1 Overview of Simulation

Figure 1 shows a UML Activity Diagram that depicts the entire execution flow of one iteration of the simulation proposed. In short, the simulation can be viewed with two entities, which are the passenger and the automated passport control system (APCS). At one iteration, the passenger will perform a series of steps in order to pass through APCS, while APCS will have to respond to the actions of the passenger. To have enough data load, the entire simulation will comprise multiple iterations of passengers using the APCS. Since the simulation proposed will be implemented in an object-oriented language, Figure 2 shows a UML Class Diagram that presents how the simulation will be implemented in Java. Both mediator and observer design patterns will follow the class structure designed in Figure 2. Based on Figure 2, it can be observed that the APCS is made up of six smaller components, as their name suggests, these six smaller components are threads that are responsible for the operation of parts of the APCS. For instance, the GateControl is responsible for controlling the entry gate and exit gate. Since this is a simulation, some simpler operations such as opening gates and scanning passports will be

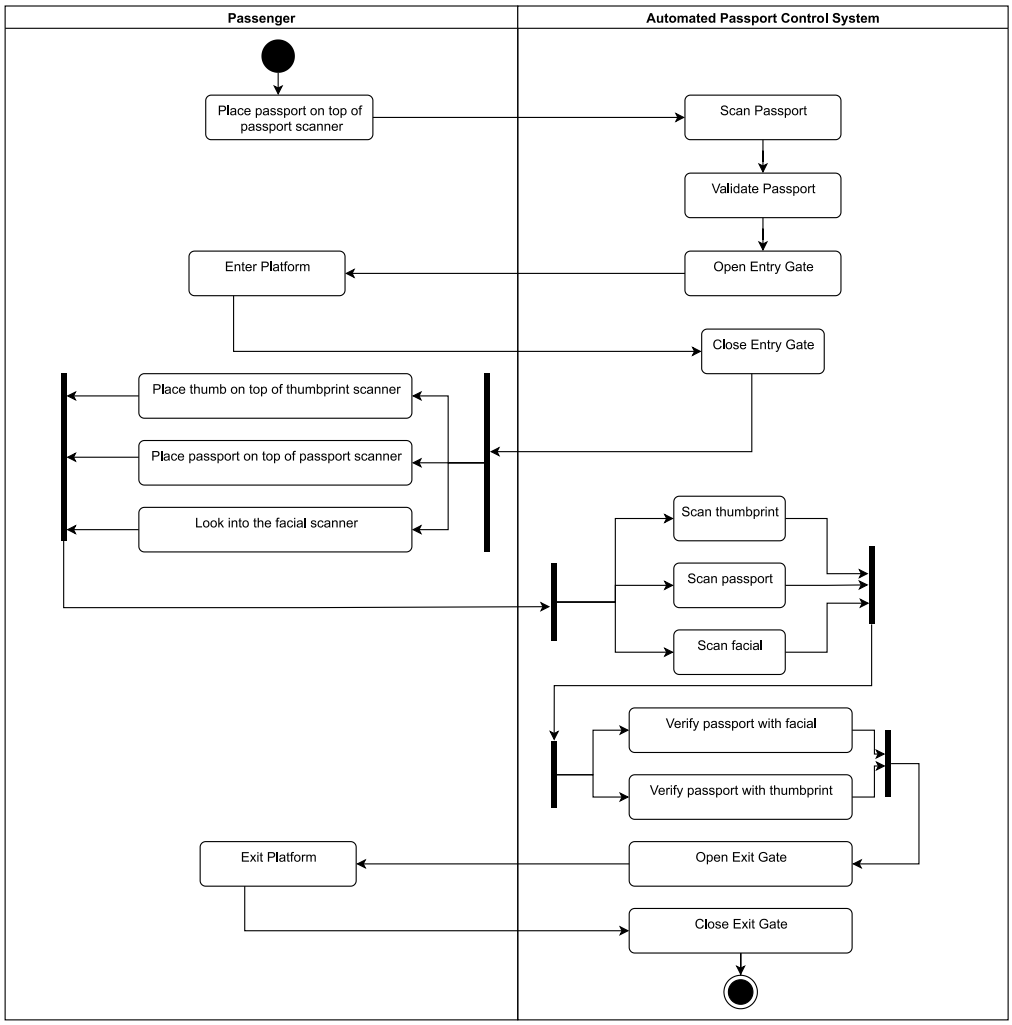

Figure 1 UML Activity Diagram for the flow of proposed simulation

simulated by simply instructing the responsible thread to sleep for *x* seconds. On the other hand, the action of validating and verifying the passport with the passenger's facial and thumbprint are sophisticated and involves cryptography operations. In this simulation, passport validation is where the passport's issuer is checked, while passport verification is where the passport data is verified with the passenger's facial and thumbprint. To simulate the validation, passport objects are generated with a signature engine that signs on the object which can be verified later. For the verification, the passport object contains an encrypted field which stores the encrypted passenger's name while the passenger's thumbprint and facial field is a string that contains the passenger's name. Thus, the verification process involves decrypting the encrypted field in the passport and comparing the decrypted passenger name with the passenger's name stored in both the thumbprint and facial field of the passenger. All this logic is abstracted away from the DataProcessing thread, and it is found in the Security and Utility class (refer to Figure 2).

3.2 Methodology Implementation

The simulation proposed in 3.1 for both design patterns will be implemented in the Java Programming Language and executed by a local Java Virtual Machine (JVM), version 17.0.1. In addition, both implementations will import the same third-party packages from the Maven repository

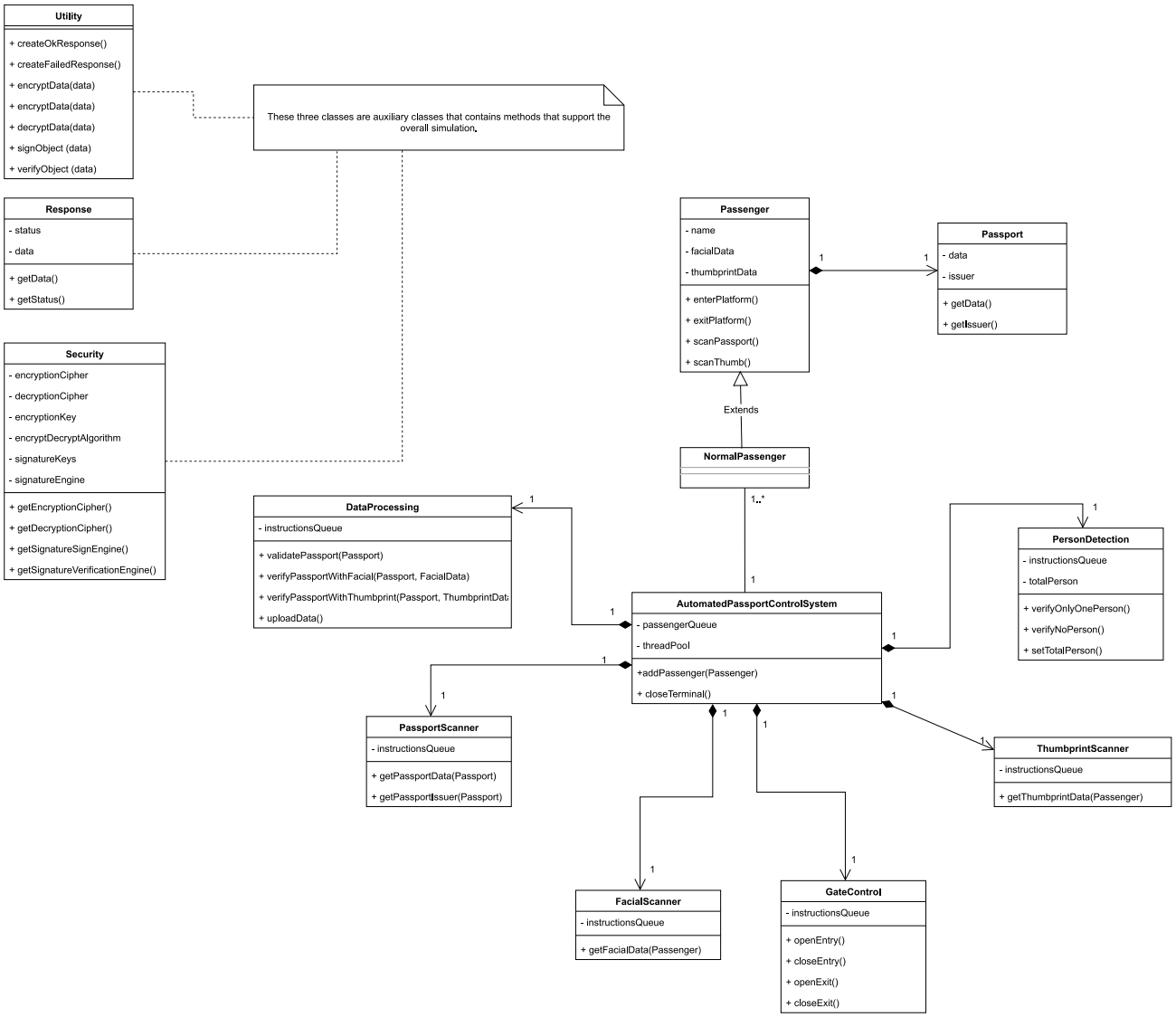

Figure 2 UML Class Diagram for Simulation Proposed

using the pom.xml file shown in Figure 3. Two main dependencies are imported which are com.github.javafaker and org.openjdk.jmh. The former is a library that helps to generate fake name for each passenger, while the latter is Java Microbenchmark Harness (JMH), a framework used to micro-benchmark the simulation implemented.

```
<dependencies>
    <!-- https://mvnrepository.com/artifact/com.github.javafaker/javafaker -->
    <dependency>
        <groupId>com.github.javafaker</groupId>
        <artifactId>javafaker</artifactId>
        <version>1.0.2</version>
    </dependency>
    <!-- https://mvnrepository.com/artifact/org.openjdk.jmh/jmh-core -->
    <dependency>
        <groupId>org.openjdk.jmh</groupId>
        <artifactId>jmh-core</artifactId>
        <version>1.35</version>
    </dependency>
    <!-- https://mvnrepository.com/artifact/org.openjdk.jmh/jmh-generator-annprocess -->
    <dependency>
        <groupId>org.openjdk.jmh</groupId>
        <artifactId>jmh-generator-annprocess</artifactId>
        <version>1.35</version>
    </dependency>
```

Figure 3 Dependencies from pom.xml

3.3 Mediator Design Implementation

Following the structure shown in Figure 2, the mediator design pattern is implemented by making the AutomatedPassportControlSystem behave as the mediator thread that controls the execution of other threads under it. This is achieved by programming the mediator thread to send instructions and receive outputs from each of the smaller component threads (refer to Figure 4). Each of the smaller component

threads will have two dedicated queues that receive instruction from the mediator thread and send output to the mediator thread as shown in Figure 5. The smaller component threads are scheduled to run at a fixed delay of 10ms by using a ScheduledExecutorService (refer to Figure 7). For each run, these threads will wait for instructions from the mediator thread; the instructions are in the form of a Callable, thus the threads will execute the callable that is passed into their instructions queue (refer to Figure 6). The results returned by the instructions are then added to the results output queue where they will be acknowledged and checked by the mediator thread as shown in Figure 4.

```
instructionsToDataProcessing.transfer(() → dataProcessingThread.validatePassport(passportIssuer));
if (outputFromDataProcessing.take().getStatus() ≠ Response.STATUS.OK) {
    throw new RuntimeException("Unable to validate passenger passport issuer");
}
```

Figure 4 Code sample from AutomatedPassportControlSystem class

```
public class PersonDetection implements Runnable {
    2 usages
    private final TransferQueue<Callable<Response>> instructionsQueue;
    2 usages
    private final TransferQueue<Response> resultOutput;
```

Figure 5 Code sample from PersonDetection class

```
@Override
public void run() {
    try {
        Callable<Response> instructions;
        Response response;
        instructions = instructionsQueue.take();
        response = instructions.call();
        resultOutput.put(response);
    } catch (InterruptedException e) {
        System.err.println("WARNING : " + this + " has been interrupted and terminated.");
    } catch (Throwable e) {
        System.err.println("ERROR : ");
        e.printStackTrace();
    }
}
```

Figure 6 Code sample for the execution of each smaller component threads

```
threadPool.scheduleWithFixedDelay(gateControlThread, initialDelay: 0, delay: 10, TimeUnit.MILLISECONDS);
threadPool.scheduleWithFixedDelay(personDetectionThread, initialDelay: 0, delay: 10, TimeUnit.MILLISECONDS);
threadPool.scheduleWithFixedDelay(facialScannerThread, initialDelay: 0, delay: 10, TimeUnit.MILLISECONDS);
threadPool.scheduleWithFixedDelay(thumbprintScannerThread, initialDelay: 0, delay: 10, TimeUnit.MILLISECONDS);
threadPool.scheduleWithFixedDelay(dataProcessingThread, initialDelay: 0, delay: 10, TimeUnit.MILLISECONDS);
threadPool.scheduleWithFixedDelay(passportScannerThread, initialDelay: 0, delay: 10, TimeUnit.MILLISECONDS);
```

Figure 7 Code sample from AutomatedPassportControlSystem

3.4 Observer Design Implementation

Unlike the mediator design implementation, the observer design pattern does not require a mediator thread but a subscription mechanism where each thread can listen to events emitted by other threads to determine the action to be executed. Similarly, the observer design implementation will follow the structure in Figure 2 but with an additional two interfaces to introduce a subscription mechanism to the simulation. The two interfaces are observable, and observer (refer to Figure 8). The observable interface contains the addObserver() method which allows the threads that implement this interface to define a method that registers other threads as an observer which will listen to the events emitted. The observer interface contains the sendEvent() method which allows the threads to define a method that allows other threads to send events to this thread. As a result, all observable threads will have a HashMap that keeps track of the subscribers for different types of events, while all threads that are observers will have a queue that receives events from observed threads via the sendEvent() method (refer to Figure 10). Both implementations can be found in Figure 9. All events are sent as a string wrapped inside a Response object.

```
public interface Observable {
    public void addObserver(String event, Observer observer);
}

interface Observer {
    public void sendEvent(Response res);
}
```

Figure 8 Observable and Observer Interface

```java
@Override
public void addObserver(String event, Observer observer) {
    if (!subscribers.containsKey(event))
        throw new IllegalArgumentException(observer + " trying to subscribe to an unknown event of " + this);
    subscribers.get(event).add(observer);
}

@Override
public void sendEvent(Response res) {
    try {
        if (!eventsReceived.tryTransfer(res, timeout: 60, TimeUnit.SECONDS)) {
            throw new RuntimeException("Message is not acknowledged by " + this + " after waiting for 60 second");
        }
    } catch (InterruptedException e) {
        throw new RuntimeException(e);
    }
}
```

Figure 9 addObserver() and sendEvent() implementation

By implementing these two interfaces on all the threads in APCS, the subscription mechanism is achieved. Each thread is an observable and observer to all other threads in APCS and can emit different types of events (refer to Figure 10). Thus, each observer of a thread would have to specify which specific events to listen to (refer to Figure 11). Figure 11 shows the code sample in initializing all the observers for the entire simulation flow. Since the AutomatedPassportControlSystem class is no longer a mediator thread, this thread only functions as an entry point for the passengers to the APCS.

```java
public class GateControl implements Runnable, Observable, Observer {

    private final HashMap<String, List<Observer>> subscribers;
    private final HashMap<String, Callable<Consumer<Response>>> eventsResponses;

    private final TransferQueue<Response> eventsReceived;

    // Events Emitted
    public static final String openedExit = "OpenedExit.GateControl";
    public static final String openedEntry = "OpenedEntry.GateControl";
    public static final String closedExit = "ClosedExit.GateControl";
    public static final String closedEntry = "ClosedEntry.GateControl";
```

Figure 10 GateControl with defined events, list of observers in hashmap, and eventsReceived queue

```java
// Event flow
this.addObserver(AutomatedPassportControlSystem.addedNewPassenger, passportScannerThread);
passportScannerThread.addObserver(PassportScanner.getIssuer, dataProcessingThread);
dataProcessingThread.addObserver(DataProcessing.validatedPassport, gateControlThread);
gateControlThread.addObserver(GateControl.openedEntry, personDetectionThread);
personDetectionThread.addObserver(PersonDetection.onePersonDetected, gateControlThread);
gateControlThread.addObserver(GateControl.closedEntry, passportScannerThread);
gateControlThread.addObserver(GateControl.closedEntry, thumbprintScannerThread);
gateControlThread.addObserver(GateControl.closedEntry, facialScannerThread);
facialScannerThread.addObserver(FacialScanner.getFacial, dataProcessingThread);
thumbprintScannerThread.addObserver(ThumbprintScanner.getThumbprint, dataProcessingThread);
passportScannerThread.addObserver(PassportScanner.getData, dataProcessingThread);
dataProcessingThread.addObserver(DataProcessing.verifiedPassenger, gateControlThread);
gateControlThread.addObserver(GateControl.openedExit, personDetectionThread);
personDetectionThread.addObserver(PersonDetection.nonePersonDetected, gateControlThread);
gateControlThread.addObserver(GateControl.closedExit, observer: this);
```

Figure 11 Code sample for initializing the observers for each thread in AutomatedPassportControlSystem class

Similar to the mediator design pattern, each thread including the AutomatedPassportControlSystem class will be scheduled to run with a fixed 10ms delay. At each run, each thread will wait for events emitted by its observed threads. Once the thread gets notified of an event, the thread will fetch and execute the correct event response stored in a HashMap (refer to Figure 12). The event responses are stored as a Callable that returns a Consumer function which can be called by passing the response *(events received)* as the argument. Upon completion, the thread will emit an event notifying its observer.

```java
@Override
public void run() {
    // wait for events
    Response receivedEvent;
    try {
        receivedEvent = eventsReceived.take();
        Callable<Consumer<Response>> responseToEvent = eventsResponses.get(receivedEvent.getContent());
        if (responseToEvent == null)
            throw new RuntimeException(this + " received an event that does not know how to handle..");
        responseToEvent.call().accept(receivedEvent);
    } catch (InterruptedException e) {
        System.err.println("WARNING : " + this + " has been interrupted and terminated.");
    } catch (Throwable e) {
        System.err.print("ERROR :");
        e.printStackTrace();
    }
```

Figure 12 Run method for each thread in observer design

3.5 Benchmarking and Profiling

Both implementations proposed above will be micro-benchmarked using JMH to obtain the throughput and average execution time of both simulations. JMH is configured to benchmark both simulations with 5 warmup iterations and 50

```java
public class JMHBenchmark {
    @Benchmark
    @Fork(1)
    @BenchmarkMode({Mode.Throughput, Mode.AverageTime})
    @Warmup(iterations = 5)
    @Measurement(iterations = 50)
    @OutputTimeUnit(TimeUnit.MINUTES)
    public void startAPCSMediator() {
        AutomatedPassportControlSystem apcs = new AutomatedPassportControlSystem();
        Thread automatedPassportControlSystem = new Thread(apcs);
        Thread passengerGenerator = new Thread(new PassengerFactory(apcs, numberOfPassenger: 1));
        automatedPassportControlSystem.start();
        passengerGenerator.start();
        try {
            passengerGenerator.join();
            automatedPassportControlSystem.join();
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
        System.out.println("\n\n ---------- Simulation Program Terminated ---------- ");

    }
```

Figure 13 JMH Benchmark Configuration in Mediator Design

```
public class JMHBenchmark {
    @Benchmark
    @Fork(1)
    @BenchmarkMode({Mode.Throughput, Mode.AverageTime})
    @Warmup(iterations = 5)
    @Measurement(iterations = 50)
    @OutputTimeUnit(TimeUnit.MINUTES)
    public void startAPCSObserver() {
        AutomatedPassportControlSystem apcs = new AutomatedPassportControlSystem();
        Thread passengerGenerator = new Thread(new PassengerFactory(apcs,  numberOfPassenger: 1));
        passengerGenerator.start();
        try {
            apcs.start();
            passengerGenerator.join();
            apcs.threadPool.awaitTermination( timeout: 20, TimeUnit.MINUTES);
        } catch (InterruptedException e) {
            throw new RuntimeException(e);
        }
        System.out.println("\n\n ------------------ Simulation Program Terminated -------------------------- ");
    }
```

Figure 14 JMH Benchmark Configuration for Observer Design

measurement iterations in a single fork. To avoid any potential optimization by the JVM, each iteration will execute the simulation with only one passenger. Therefore, the APCS will have processed a total of 50 passengers. Figure 13 and Figure 14 show the JMH configurations written in annotations.

To support the results obtained from the JMH benchmark, CPU profiling and memory profiling are performed with the same load of 50 passengers. Since the simulation proposed does not have heavy heap memory usage, the allocated heap size for both implementations is set to 15Mb by sending *-Xms15M* and *-Xmx15M* commands to the Java Virtual Machine (JVM). This is done so that the garbage collection process can occur and be analyzed. The garbage collector used for both simulations will be the Garbage-First (G1) Garbage Collector, which is the default garbage collector for JVM version 17.0.1 (Oracle, n.d.). Finally, the main tool used for CPU and memory profiling is YourKit Java Profiler.

4.0 Results and Discussion

4.1 JMH Benchmark

Table 1 and Table 2 show the results obtained for throughput and average execution time for the mediator and observer design pattern. According to both tables, a difference of 0.010 ops/min for throughput and a difference of 0.001 min/op for average execution time can be observed. Therefore, the implemented observer design pattern is slightly more efficient than the implemented mediator design pattern. This could be due to the observer design pattern having less overhead for a thread to send messages to its receiver threads. For instance, a thread in the mediator design pattern would have to pass a message to the mediator thread before the message gets sent to the receiver threads, while a thread in the observer design pattern can directly send the message to its receivers via publishing events. In this simulation, the mediator design pattern might have introduced extra overhead in the inter-threads communication process, thus resulting in a slightly lesser throughput and higher average execution time than the observer design pattern.

4.2 CPU Profiling

Figure 15 and Figure 16 show the CPU Usage Graph for both mediator and observer design implementations of the proposed simulation. Based on the graphs, it can be observed that there is only a small spike in the CPU usage at the beginning of the simulation, this can be explained by the overhead in initializing all the 50 passengers and classes needed for the simulation. Since the simulation proposed did not involve any operations that require heavy CPU usage, both implementations have low CPU usage. The mediator design used a total CPU time of 31 seconds (refer to Figure 15), while the observer design used a total CPU time of 29 seconds (refer to Figure 16). This can be explained by the fact that most operations of the APCS are simulated by simply putting a thread to sleep for *x* seconds. This resulted in threads being put into an idle state more often instead of using the CPU to perform computations. Nevertheless, the CPU time usage suggests that the observer design pattern is slightly more efficient than the mediator design pattern.

Furthermore, Table 3 shows the time spent in milliseconds (ms) in the run method of all key threads of APCS extracted from YourKit Java Profiler. It can be observed that the observer design pattern spent lesser time in all of the threads than the mediator design pattern. The difference in time spent between both implemented design range from 979ms to 1631ms. This finding supports the previous findings that the implemented observer design pattern is more efficient than the implemented mediator design pattern as lesser time is spent to accomplish the same amount of work.

Benchmark	Count	Throughput Score (ops/min)	Error (ops/min)
Implemented Mediator Design	50	4.391	± 0.002
Implemented Observer Design	50	4.401	± 0.003

Table 1 JMH Results for Throughput

Benchmark	Count	Average Execution Time (min/op)	Error (min/op)
Implemented Mediator Design	50	0.228	± 0.001
Implemented Observer Design	50	0.227	± 0.001

Table 2 JMH Results for Average Execution Time

Methods	Time Diff (ms)	Time Spent by Mediator Design (ms)	Time Spent by Observer Design (ms)
com.apcs.Profiler.main(String[])	1,000	680,623	679,623
com.apcs.AutomatedPassportControlSystem.run()	1,631	679,843	678,212
com.apcs.DataProcessing.run()	979	677,516	676,536
com.apcs.FacialScanner.run()	1,031	679,243	678,212
com.apcs.GateControl.run()	1,035	677,551	676,515
com.apcs.PersonDetection.run()	1,042	678,681	677,639
com.apcs.ThumbprintScanner.run()	1,042	679,228	678,186
com.apcs.PassportScanner.run()	1,050	678,695	677,645

Table 3 Time spent in the run method of each thread of APCS for Mediator and Observer design

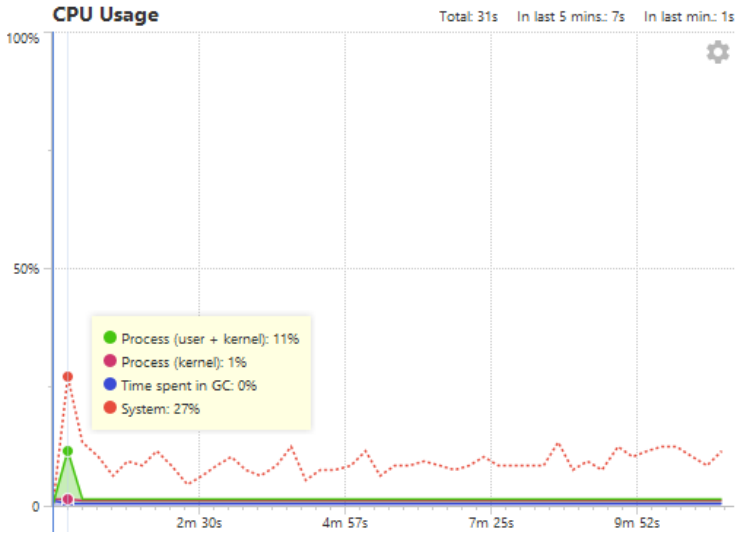

Figure 15 CPU Usage Graph for Mediator Design Pattern

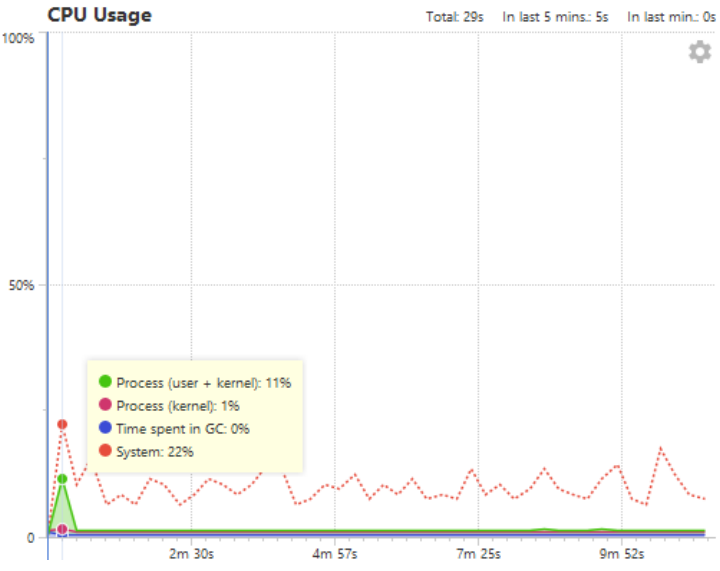

Figure 16 CPU Usage Graph for Observer Design Pattern

In addition, the high time difference in the AutomatedPassportControlSystem.run() could be an effect of the mediator thread in the implemented mediator design. This is because the mediator thread sends instructions and coordinates all the sub-components of APCS in the implemented mediator design, thus more time will be spent managing the overall communication compared to the observer design. In short, it can be concluded that the implemented observer design pattern is more efficient than the implemented mediator design pattern in this experiment.

4.3 Memory Profiling

Figure 17 and Figure 18 show the heap memory usage graph for the implemented mediator design and observer design. Based on both figures, it can be observed that they have a similar trend in the rise and fall of heap memory usage. Since the observer and mediator design pattern falls under the behavioural patterns category and is not meant for optimizing memory usage in an application, it is no surprise to get a similar heap memory usage for both implementations. However, a minor difference of 1MB can be observed in the total used heap memory between the implemented mediator and observer design. The excess heap memory used in the mediator design could be explained by the overhead in inter-threads communication and additional objects required to implement the mediator design. However, it is hard to determine the memory usage efficiency since the difference noticed is insignificant and both design patterns are not meant for memory usage optimization.

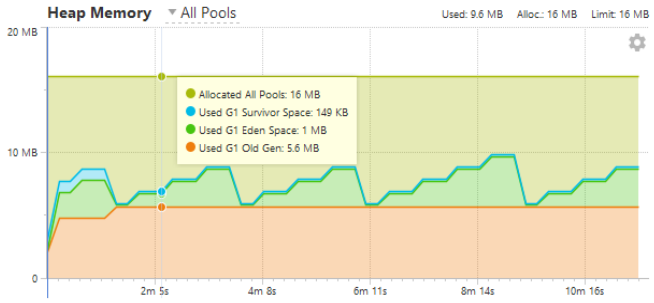

Figure 17 Heap Memory Usage Graph for Mediator Design Pattern

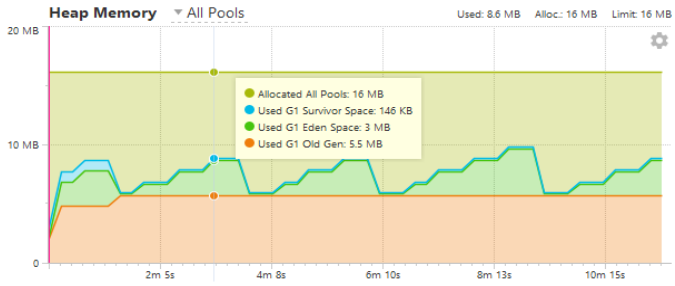

Figure 18 Heap Memory Usage Group for Observer Design Pattern

Furthermore, Figure 19 and Figure 20 show the garbage collection processes for the implemented mediator design and observer design. It can be easily observed that both implemented designs have similar garbage collections and garbage pauses. Both implemented designs have a total of 9 garbage collections and an average of 3ms garbage collection pauses. The results are not surprising as both design patterns originate from the behavioural patterns category that does not emphasize objects management and creation which may have a higher impact on garbage collections and memory usage.

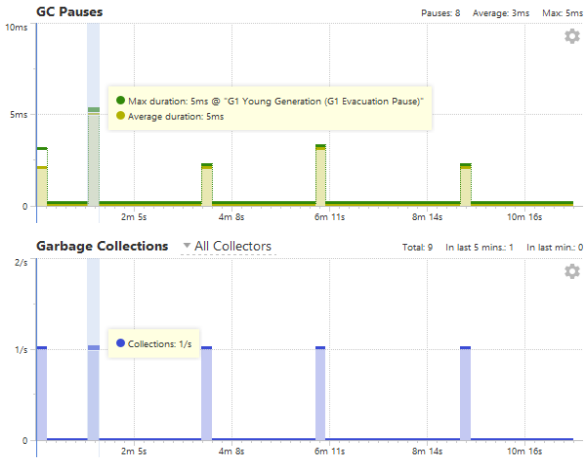

Figure 19 Garbage Collection Graphs for Mediator Design Pattern

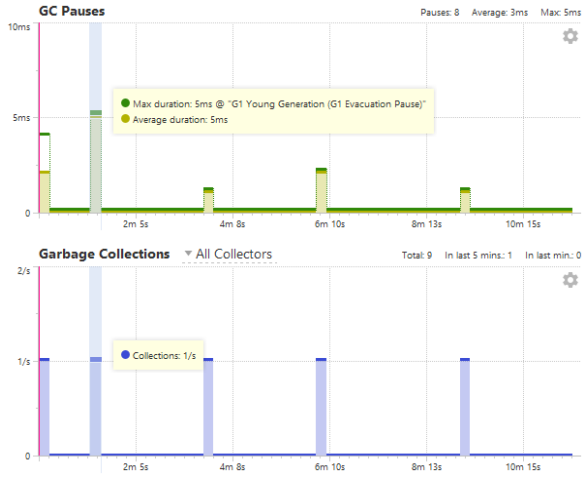

Figure 20 Garbage Collection Graphs for Observer Design Pattern

5.0 Conclusion

In summary, this research aims to investigate the implications of software design patterns on real-time applications. Particularly, this research investigates two behavioural design patterns which are the mediator design pattern and the observer design pattern. A scenario of passengers using an automated passport control system is proposed to simulate a real-time environment where there will be many inter-thread communication processes. The proposed simulation is implemented in the two designs proposed with Java Programming Language. To evaluate the performance of each design, micro-benchmarking and profiling are performed. Based on the results obtained, it can be concluded that the implemented observer design is more efficient than the implemented mediator design. The reason suggested could be the implemented mediator design has more overhead in the inter-thread communication process since all communications will have to pass through a mediator thread instead of directly going to the receiver threads.

There are some limitations present in this study that should be addressed in future works. Firstly, the implemented simulation lacks CPU-intensive operations which cause difficulty in profiling accurate CPU usage in both designs since the threads are in an idle state most of the time. Secondly, this research only investigated two software design patterns under the same category, future works can look into investigating software design patterns from different categories to measure the differences in CPU and memory usage. Lastly, this research still requires more testing on alternative scenarios and implementations so that much more reliable results can be obtained. All the limitations suggested above can be a starting point for future research in this domain.

6.0 References

Cedeño, W., & Laplante, P. A. (2007). An Overview of Real-Time Operating Systems. *JALA: Journal of the Association for Laboratory Automation*, *12*(1), 40–45. https://doi.org/10.1016/j.jala.2006.10.016

Costa, D., Bezemer, C. P., Leitner, P., & Andrzejak, A. (2021). What's Wrong with My Benchmark Results? Studying Bad Practices in JMH Benchmarks. *IEEE Transactions on Software Engineering*, *47*(7), 1452–1467. https://doi.org/10.1109/tse.2019.2925345

Georges, A., Buytaert, D., & Eeckhout, L. (2007). Statistically rigorous java performance evaluation. *Proceedings of the 22nd Annual ACM SIGPLAN Conference on Object Oriented Programming Systems and Applications (OOPSLA)*, 57–76. https://doi.org/10.1145/1297027.1297033

Intel. (n.d.). *Real-Time Systems Overview and Examples*. Retrieved August 22, 2022, from https://www.intel.com/content/www/us/en/robotics/real-time-systems.html

Laplante, P. A. (2004). *Real-Time Systems Design and Analysis*. https://ieeexplore.ieee.org/servlet/opac?bknumber=5237056

Leitner, P., & Bezemer, C. P. (2017). An Exploratory Study of the State of Practice of Performance Testing in Java-Based Open Source Projects. *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*, 373–384. https://doi.org/10.1145/3030207.3030213

NI. (2022, July 19). *What is a Real-Time Operating System (RTOS)?* Retrieved August 22, 2022, from https://www.ni.com/en-my/innovations/white-papers/07/what-is-a-real-time-operating-system--rtos--.html

Oracle. (n.d.). *Garbage-First (G1) Garbage Collector*. Oracle Help Center. Retrieved August 27, 2022, from https://docs.oracle.com/en/java/javase/17/gctuning/garbage-first-g1-garbage-collector1.html#GUID-ED3AB6D3-FD9B-4447-9EDF-983ED2F7A573

Osama, M. (2021, December 23). *Real-Time OS Characteristics*. LinkedIn. Retrieved August 24, 2022, from https://www.linkedin.com/pulse/real-time-os-characteristics-mohamed-osama

Rana, M. E., & Wanabrahman, W. N. (2017). *The Effect of Applying Software Design Patterns on Real Time Software Efficiency*. https://www.researchgate.net/publication/332556324_The_Effect_of_Applying_Software_Design_Patterns_on_Real_Time_Software_Efficiency

Rahman, S. (2019, June 24). *The 3 Types of Design Patterns All Developers Should Know (with code examples of each)*. freeCodeCamp.Org. Retrieved August 22, 2022, from https://www.freecodecamp.org/news/the-basic-design-patterns-all-developers-need-to-know/

Samoaa, H., & Leitner, P. (2021). An Exploratory Study of the Impact of Parameterization on JMH Measurement Results in Open-Source Projects. *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, 373–384. https://doi.org/10.1145/3427921.3450243

Sharma, M., Elmiligi, H., & Gebali, F. (2015). Performance Evaluation of Real-Time Systems. *International Journal of Computing and Digital Systems*, *4*(1), 43–52. https://doi.org/10.12785/ijcds/040105

Shvets, A. (2021). *Dive Into Design Patterns* [E-book]. Retrieved August 22, 2022, from https://refactoring.guru/design-patterns/book

Wind River Systems. (n.d.). *Intro to Real-Time Operating Systems (RTOS)*. Wind River. Retrieved August 22, 2022, from

https://www.windriver.com/solutions/learning/rtos