ILLINOIS TECH

Application Software Design

ECE 528

Homework 2

Abhilash Kashyap Balasubramanyam

abalasubramanyam@hawk.iit.edu

A20566944

Spring 2025

Illinois Institute of Technology

Date: March 6, 2025

Table of Contents

List of Figures

List of Tables

List of Codes

Questions and Solutions

1 Question 1

Q: Describe the purpose of using a Java Interface in Project 2. Provide your own answer, not from the lecture notes *(5 points)*

Solution

Java interfaces facilitate loose coupling and maintainability by establishing explicit agreements between system components. Project 2's RequestHandler interface, which is implemented by the HTTPCommands class, exemplifies how interfaces specify abstract behaviors without providing implementation specifics.

An interface's definition of the handleGet method signature creates a contract that HTTP request handlers have to abide by. Higher abstraction levels are made possible by this division of interface from implementation, which is consistent with Java's design ethos.

Interfaces allow for interchangeable implementations and polymorphic behavior. This is useful in Internet of Things scenarios where various parts must react to HTTP requests while adhering to a standard protocol. Regardless of implementation details, the RequestHandler interface establishes a standardized method for handling HTTP requests.

Interfaces also make it easier for components to loosely couple. Any PlugSim object can be accessed using HTTPCommands, regardless of implementation specifics. Because additional plug types or command handlers can be added without changing the current code, this increases the system's flexibility.

Interfaces are shown in Project 2 as blueprints that outline necessary activities. The RequestHandler interface guarantees that implementing classes have the required functionality by defining expected HTTP request handling behavior. This contractual structure maintains system design and makes component roles clear.

With components managing distinct functional aspects and interacting via clearly specified abstract interfaces, this method achieves a clear separation of responsibilities, improving maintainability, encouraging reusability, and facilitating system evolution.

2 Question 2

Q: Multiple users may control the same plug via the web page. If one of them requests to switch on the plug and another one requests to switch off the plug at the same time… *(5 points)*

2.1 Sub-question 1

Q: What will they see on their web pages as responses?

Solution

If two people use conflicting commands to control the same plug at the same time (one turning it on, the other turning it off), each user will see a response that reflects the current state once their particular request has been handled. An HTML response indicating that the plug is on will be sent to the user who sent the "on" command, and a response indicating that the plug is off will be sent to the user who sent the "off" command. This happens because, without being aware of other concurrent requests, the HTTPCommands class immediately generates answers depending on the plug's state after processing each individual request.

2.2 Sub-question 2

Q: In the end, is the plug switched on or off?

Solution

Whichever request the server processes last will determine whether the plug is turned on or off in the end. In a concurrent context, the device's final state will be determined by the last request to finish execution. The plug will stay on if the "switch on" request is the last to finish. On the other hand, the plug will stay off if the "switch off" request finishes last. As a result, there is a race condition where the outcome is unpredictable.

2.3 Sub-question 3

Q: What is the proper mitigation to overcome this issue?

Solution

Several mitigating techniques can be used to mitigate this concurrent issue:

- Serialization of Requests: Put in place synchronization techniques (such as locks or semaphores) that handle requests in a sequential manner as opposed to concurrently.
- Optimistic Concurrency Control: Use version tracking for device states to identify changes in states between a user's viewing and modification attempts.
- State Broadcasting: Real-time state updates can be broadcast to all connected clients via WebSocket or Server-Sent Events.
- Last-Writer-Wins Policy: Clearly state that the most recent command is executed first, with timestamps being used to establish order.
- User Permissions Hierarchy: Put in place a structure of priorities where commands from specific users are given priority over those from other users.

Specific application needs and the anticipated frequency of concurrent control attempts determine the best course of action.

3 Question 3

Q: Below is an HTTP request received from a browser *(10 points)* :

```
GET /~wyi3/ece448/system-setup-work-flow.html HTTP/1.1

Host: ece.iit.edu

Accept: text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8

Accept-Encoding: gzip, deflate

Accept-Language: en-US,en;q=0.9

Connection: keep-alive

User-Agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like
Gecko) Chrome/133.0.0.0 Safari/537.36

If-Modified-Since: Fri, 17 Jan 2025 08:42:50 GMT

If-None-Match: "48000f-2960b-62be2e54e9680"

Referer: http://ece.iit.edu/~wyi3/ece448/system-setup-work-flow.html
```

3.1 Sub-question 1

Q: Identify the HTTP method, the request resource (URI), and the HTTP version of this request.

Solution

The following essential elements of the request line are discernible in the given HTTP request:

- HTTP Protocol: GET

Without changing any data, information can be retrieved from the server using the GET technique. One of the most widely used HTTP methods, it is exclusively intended for data retrieval tasks. The HTTP protocol states that GET requests must be idempotent, which means that several identical requests should function in the same way as one.

- Resource Request (URI): /~wyi3/ece448/system-setup-work-flow.html

The precise path to the resource on the server that the client is requesting is specified by this URI (Uniform Resource Identifier). Located in the ece448 directory under

the user wyi3's home directory on the server, it looks to be an HTML document pertaining to the system setup workflow in this instance.

- HTTP Version: HTTP 1.1

This shows the version of the protocol is being utilized in this exchange. Compared to its predecessor HTTP/1.0, HTTP/1.1, which was first released in 1997, is still in use today and has significant features like chunked transfers, persistent connections, and extra caching methods.

The request line, the first line of an HTTP request, is made up of these three components and describes the basic content of the client's request to the server.

3.2 Sub-question 2

Q: What roles do If-Modified-Since the and If-None-Match headers play in HTTP communications? You may need to look this up online.

Solution

In web caching and conditional queries, the If-Modified-Since and If-None-Match headers are essential for greatly enhancing network efficiency and resource usage:

- If-Modified-Since:

The timestamp (Friday, January 17, 2025, 08:42:50 GMT) in this header indicates when the client last received the resource. In essence, the browser is asking the server, "Has this resource changed since this specific time?" when it includes this header in a request. The resource's Last-Modified date and this date are compared by the server to see if a new version has to be provided. The server may reply with a 304 Not Modified response if the resource hasn't been altered in the allotted period, telling the browser to use its cached copy rather than downloading it again.

- If-None-Match:

An entity tag ("48000f-2960b-62be2e54e9680") in this header uniquely identifies a particular resource version. ETags are usually produced according to the resource's content, giving each version a distinct fingerprint. "Is the current version of the resource different from the version I already have?" is the question that a browser poses when it sends an If-None-Match header. The resource's current ETag and this value are compared by the server. The server can produce a 304 response if they match, which shows that the content hasn't changed.

These conditional headers work together to provide effective caching methods that:

- Cut down on pointless network data transfers.
- Prevent redundant processing to reduce server load.

- Boost user page loading speeds.
- Reduce the amount of bandwidth used by servers and clients.
- Permitting browsers to use locally cached resources, when necessary, will improve overall web speed.

Static assets, pictures, and style sheets are examples of regularly requested resources that change infrequently and benefit greatly from these approaches.

3.3 Sub-question 3

Q: If the resource is unchanged, which HTTP response code would be received from the server?

Solution

The server will reply with the HTTP 304 Not Modified response code if the resource remains unaltered because the ETag in the If-None-Match header and the time indicated in the If-Modified-Since header still match the current resource.
Because the client's cached copy of the requested resource is still valid, there is no need to retransmit it, according to this status code, which is especially made for conditional GET requests. Although the 304 answer is a member of the 3xx redirection class of status codes, it actually advises the client to use its own cached version of the resource rather than sending them to another one.

When a 304-status code is returned by a server:

- Usually, the answer has headers but no message body.
- Rather than downloading the resource again, the browser uses its local cache to access it.
- With a 200 OK status, the cached resource is handled as though it had just been sent.
- Using the updated response, the browser modifies any cache-related headers.
- Because there is no content transfer, the user perceives faster page loading.

This caching technique greatly lowers bandwidth use and speeds up load times, particularly for websites that receive a lot of traffic and have comparatively static content. This can result in significant reductions in processing time and data transfer for huge resources like JavaScript files, stylesheets, and photos. As web optimization techniques continue to advance to handle ever-more complicated online applications while preserving performance across a range of network situations, the 304-status code is especially pertinent today.

4 Question 4

Q: Consider three actions “toggle”, “on”, and “off” that one may apply to a plug. If the actions need to be delivered via messages, discuss among the three delivery guarantees – at least one, exactly once, and at most once – which may or may not cause problems and why. *(5 points)*

Solution

The delivery assurances of messages containing "toggle," "on," and "off" operations have a big impact on the behavior and dependability of IoT plug control systems. When applied to these acts, each delivery semantic—at-least-once, exactly-once, and at-most-once—presents distinct difficulties, which are discussed separately below.

At least once delivery

The method ensures that messages will never be lost with at-least-once delivery, but they may be transmitted more than once. This method causes significant issues with plug control:

This assurance poses special challenges for the "toggle" action. The desired operation will be essentially cancelled if a toggle message is provided twice since the plug will change states twice. For instance, if a user wants to switch on an off light, a duplicate toggle message would do the exact opposite, turning the light on and then off again right after.
It is more advantageous for "on" and "off" actions. Applying these commands more many once yields the same outcome because they are idempotent. When a plug receives two "on" messages, it stays in the desired state by staying on after both deliveries.

Exactly once delivery

The ideal situation is exactly once delivery, in which every message is delivered exactly once. True exactly once delivery is typically seen as unattainable in distributed systems because of basic limits outlined by the FLP impossibility result and the Two Generals Problem, even if it is theoretically perfect for all three operations. What is frequently advertised as "exactly-once" is really realized as "effectively exactly-once" in practice thanks to idempotent operations or deduplication mechanisms4. This would guarantee that toggle operations for plug control function as planned and prevent inadvertent state reversals.

At most once delivery

Messages may be lost with at-most-once delivery, but they will never be delivered more than once. This strategy poses a distinct set of difficulties:

Message loss means that the plug may not react to user requests for the "toggle," "on," and "off" actions. Even if a user presses "on" in their control interface, the plug stays off if the message is lost. The user's expectations and the actual status of the system become inconsistent as a result.
In crucial applications like security systems or medical equipment, where dependable control is crucial, this assurance is especially difficult.

The most sensible method for IoT plug control systems is to:

- For operations that are intrinsically idempotent ("on" and "off"), use at-least-once delivery.
- Use sequence numbers or application-level deduplication for non-idempotent operations ("toggle").
- Instead of giving feedback about the requested state, the user interface should be designed to give feedback about the actual plug state.

Developers may design more dependable and predictable IoT control systems that skillfully manage the inherent difficulties of distributed messaging by comprehending these delivery semantics and their ramifications.

5 Question 5

Q: In the Project code and/or in the Lecture code, HashMap, Map, TreeMap are used. Explain their differences in your own words. *(5 points)*

Solution

The Map interface and its implementations, particularly HashMap and TreeMap, are fundamental data structures for key-value association management in the context of Java collections. The strategic implementation of various structures, each chosen for unique qualities that correspond with distinct functional requirements, is best demonstrated by the IoT simulator project.

Interface for Maps: The Conceptual Basis

The Map interface is an example of an abstract collection paradigm in which there is a one-to-one mapping relationship between each key and exactly one value. As demonstrated in the HTTPCommands class, where plug names are linked to their respective PlugSim instances, this abstraction makes data organization easier through meaningful identification. Key-value pair storage, retrieval, removal, and existence verification are among the fundamental operations defined by the interface.

Prioritizing Performance using HashMap

The most common Map implementation in both project and lecture code are HashMap, which uses a hash table structure that provides notable performance benefits. Under ideal conditions, the implementation provides constant-time complexity for basic operations by storing elements according to the hash codes of keys. Because HashMap is unordered, it cannot guarantee the iteration sequence, which may change when elements are added or removed. It also provides versatility in data format by supporting multiple null values and one null key. When processing numerous independent requests or other situations that call for quick access to plugs without regard for their sequential placement, this solution is quite helpful.

TreeMap: Guaranteeing Structured Order

For instance, the HTTPCommands class purposefully uses TreeMap to store plugs:

```
private final TreeMap<String, PlugSim> plugs = new TreeMap<>();
```

Code 1: Example of Tree map in HTTPCommands.

The unique features of TreeMap that cater to particular needs are reflected in this implementation choice. TreeMap facilitates predictable iteration patterns by keeping keys in sorted order (by default, ascending). Although operations have logarithmic complexity, which makes them marginally less effective than hash maps, the speed is still sufficient for the majority of uses.

Feature	HashMap	TreeMap
Implementation Structure	Hash Table	Red-Black Tree
Time Complexity (average)	O(1)	O(log n)
Ordering	Unordered	Sorted by keys
Null Support	One null key, multiple null values	No null keys
Memory Usage	Higher	Lower
Use Case	Fast access without ordering requirements	When ordered iteration is needed

Table 1: HashMap and TreeMap Comparison.

In contrast to HashMap, TreeMap forbids null keys and guarantees balanced structure with its internal red-black tree implementation. In the case of our project, TreeMap ensures that all plugs are listed using the listPlugs() method in a consistent, alphabetical order, improving the user experience while engaging with the web interface.

The following are the IoT Simulator's practical implications:

- TreeMap's choice in HTTPCommands is an intentional design choice with observable advantages. The alphabetical organization makes it easier for users to navigate and locate accessible plugs on the website. Additionally, when working with ranges of plug names, TreeMap's sorted nature may allow for more effective operations.
- Due to the system's generally small number of plugs, the little performance difference over HashMap becomes insignificant. This demonstrates how developers may combine performance considerations with user experience needs by having a sophisticated understanding of Map implementations, which leads to the creation of more efficient and user-friendly systems.
- This deliberate choice of data structures highlights how crucial it is to comprehend the fundamental workings of Java collections when developing software systems, especially ones that have interactive elements where usability and performance must be carefully managed.

6 Question 6

Q: Modify the 'MLB Scoreboard' codes introduced in the lecture by using the model introduced in "lec08-simple" from the lecture code package, where the output will be updated and printed to the screen every 2 seconds. As proof of your application is running as expected, you should include some sample inputs that would change the scores, and record a short demonstration video with your personal explanation of the code and displaying that your program is updating every 2 seconds. Submit your code and your recorded video (upload to Canvas or link it to your OneDrive) *(20 points)*.

Solution

```
kashyap@Kashyaps-MacBook-Air java % javac *.java
kashyap@Kashyaps-MacBook-Air java % java GetScores
New Observer created ID: 1
Observer ID 1
Cubs: 5 vs Sox: 0
Yankees: 0 vs Mets: 0

Observer ID 1
Cubs: 5 vs Sox: 1
Yankees: 0 vs Mets: 0

Observer ID 1
Cubs: 5 vs Sox: 1
Yankees: 4 vs Mets: 0

Observer ID 1
Cubs: 5 vs Sox: 1
Yankees: 4 vs Mets: 4

New Observer created ID: 2
Observer ID 1
Cubs: 7 vs Sox: 1
Yankees: 4 vs Mets: 4

Observer ID 2
Cubs: 7 vs Sox: 1
Yankees: 4 vs Mets: 4

Observer ID: 1 has been deleted from the list

Observer ID 2
Cubs: 7 vs Sox: 1
Yankees: 4 vs Mets: 8

kashyap@Kashyaps-MacBook-Air java %
```

Figure 1: Code output prior to the edits.

The terminal output prior to the edits made to the code is as shown in the above Figure 1.

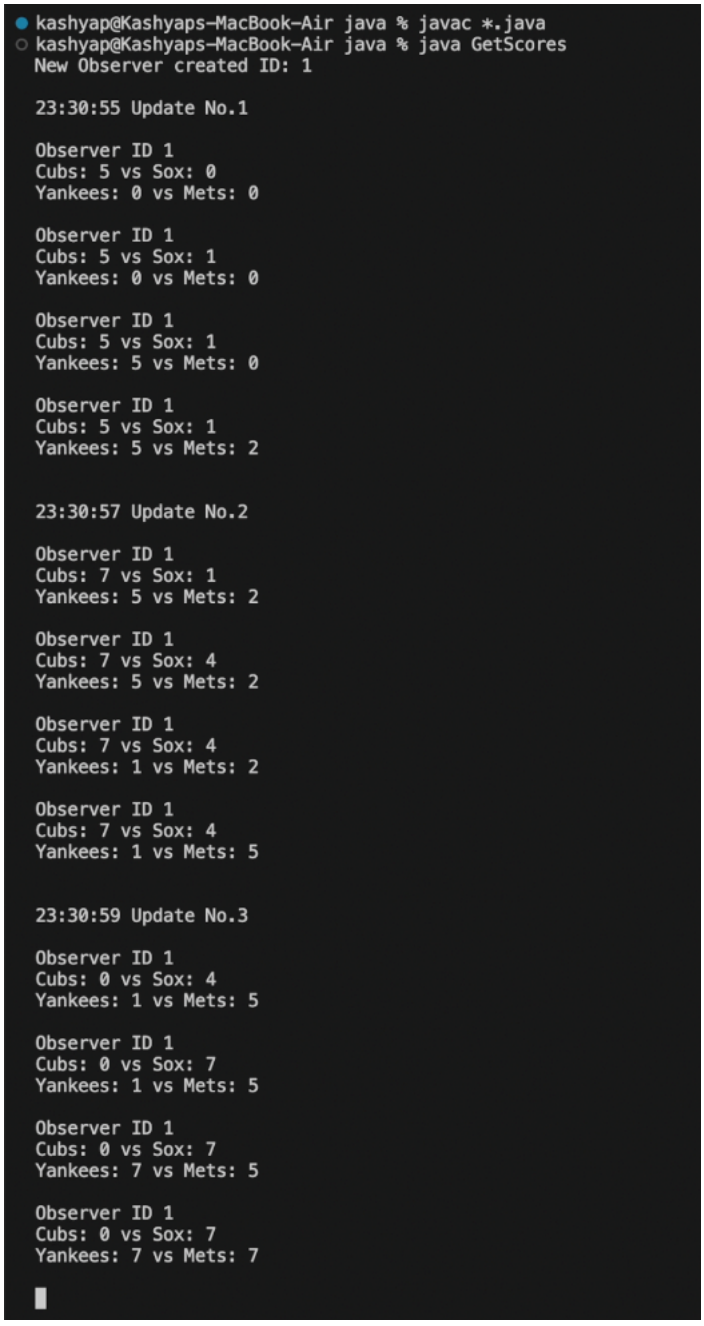

Figure 2: Terminal output post edits made to the code base.

Similarly, the above Figure 2 shows the requirements fulfilled with updates by the program in the terminal being posted every 2 seconds and also to substantiate the changes actually being made, there is an update counter, and a timestamp included. The updated scores are random values fulfilled used a random number generator. The code file updated is only to the GetScores.java program which is attached below.

```java
// GetScores.java

//Lines after modification {
import java.util.Random;
import java.util.concurrent.Executors;
import java.util.concurrent.ScheduledExecutorService;
import java.util.concurrent.TimeUnit;
import java.time.LocalDateTime;
import java.time.format.DateTimeFormatter;
//Lines after modification }


public class GetScores {

    private static final Random random = new Random();
    private static int updateCount = 0;

    public static void main(String args[])
    {
        ScoreGetter scoreGetter = new ScoreGetter();

        ScoreObserver observer1 = new ScoreObserver(scoreGetter);

        //Lines after modification {
        ScheduledExecutorService executor = Executors.newSingleThreadScheduledExecutor();
        executor.scheduleAtFixedRate(() -> updateScores(scoreGetter), 0, 2, TimeUnit.SECONDS); }

        private static void updateScores(ScoreGetter scoreGetter)
        {
            updateCount++;
            String timestamp = LocalDateTime.now().format(DateTimeFormatter.ofPattern("HH:mm:ss"));
            System.out.println("\n" + timestamp + " Update No." + updateCount + "\n");
            scoreGetter.setCubsScore(random.nextInt(10));
            scoreGetter.setSoxScore(random.nextInt(10));
            scoreGetter.setYankeesScore(random.nextInt(10));
            scoreGetter.setMetsScore(random.nextInt(10));
        }
        //Lines after modification }

        /*scoreGetter.setCubsScore(5);
        scoreGetter.setSoxScore(1);
        scoreGetter.setYankeesScore(4);
        scoreGetter.setMetsScore(4);

        ScoreObserver observer2 = new ScoreObserver(scoreGetter);
        scoreGetter.setCubsScore(7);

        scoreGetter.unRegister(observer1);

        scoreGetter.setMetsScore(8);*/
    }
```

Code 2: Updated codebase to accommodate the requirements of the Question 6.

The video demonstration of the requirement is attached along with this report in Canvas.