

Project Four

Hyperlink Integrity Checker for Web Documents



Presented by:

Lara Hossam Eldine Mostafa

ID: 6853

Nour Hesham Shaheen

ID: 7150

Ihab Sherif Mohamed

ID: 6738



Table of Contents:

Summary	2
Problem statement	2
Solution approach	2
Algorithm and implementation	4
Design	4
Sample Run	4
Data Structures and implementation	5
Classes	5
Main algorithm	8
Creating a thread	10
Chart plotting numbers of threads used vs time	11
User case UML diagram	11
Class UML diagram	12
Activity UML diagram	12
Sequence UML diagram	13


Summary

Problem statement

Validating Hyperlinks for web documents is a crucial task for servers' administrators, it helps in detecting broken links, corrupt files and tracking changes that happen to websites over time. This time cannot be done manually given the rapid growth of websites. This is why we're in need of an automatic hyperlink integrity checker.

Solution approach

Hyperlink integrity checker is a program that only takes two inputs, the main URL link, and the depth desired. The main URL of the document is the link at which the check starts, and the depth is to determine to which level we want to check our links, to prevent an indefinite execution and to be as efficient as possible.



Our program is **multithreaded** where the same resources may be accessed concurrently by several flows of control, in order to provide an efficient program with the fastest execution possible using the optimal number of threads.

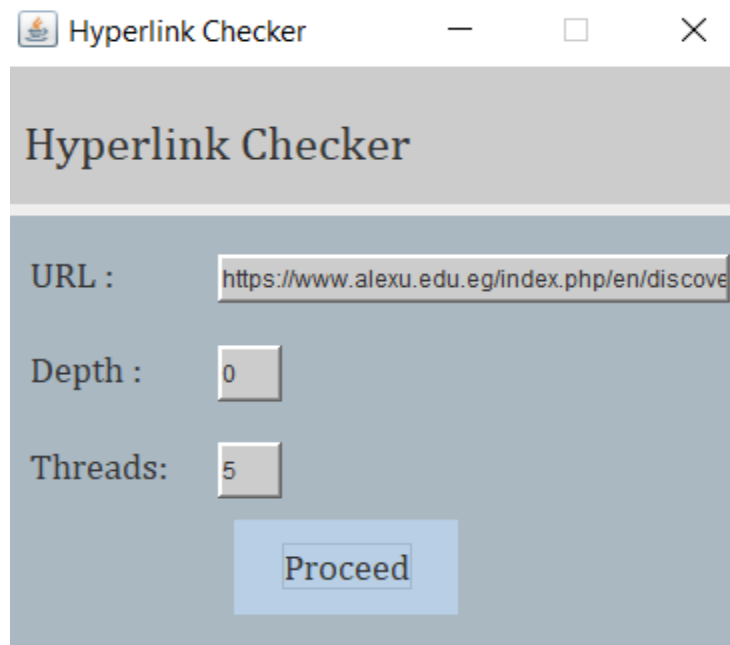
The program takes input from the user the number of threads he desires to work concurrently, this is a static way of getting the optimal number of threads for the best performance, as the time is printed to the user after the execution.

Using the **ExecutorService**, it creates a reusable pool of threads that execute submitted tasks. The **service** also manages a queue, which is used when there are more tasks than the number of threads in the pool and there is a need to queue up tasks until there is a free thread available to execute the task. **Thread pools** provide improved performance when working with a very large number of tasks. Fixed **thread pool executor** creates a **thread pool** that reuses a fixed number of **threads** to execute any number of tasks, the number of threads to provide maximum efficiency changes from a device to another.

Algorithm and implementation

Design

Sample Run



The screenshot shows a window titled "Hyperlink Checker" with a standard Windows title bar (minimize, maximize, close buttons). The window has a grey header bar with the title "Hyperlink Checker". Below the header, the interface is light blue. It contains three input fields: "URL :" with the value "https://www.alexu.edu.eg/index.php/en/discover", "Depth :" with the value "0", and "Threads:" with the value "5". Below these fields is a blue button labeled "Proceed".

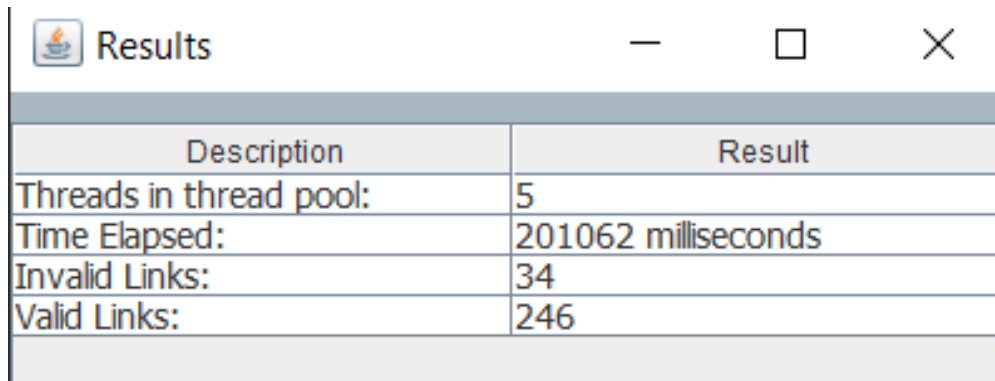
The user first inputs the URL he desires to check, and the depth he wants to reach. He also inputs the number of threads he wants to run concurrently.

The input is validated as follows:

- The user can't leave an empty field
- Depth must be an integer number (0,1,2,...)
- Threads number must be an integer number starting from 1 (1,2,3,...)

The user is prompted to try again if any of his inputs doesn't match the guidelines.

The output is a panel showing the details of the program:



A screenshot of a Java Swing window titled "Results". The window contains a table with two columns: "Description" and "Result". The table has four rows of data. The window has standard Mac OS X window controls (a red close button, a yellow maximize button, and a green minimize button) in the top right corner.

Description	Result
Threads in thread pool:	5
Time Elapsed:	201062 milliseconds
Invalid Links:	34
Valid Links:	246

Data Structures and implementation

Classes

- **Main Class**

Used for testing.

- **HyperlinkChecker class**

Contains the **depth** method which recursively creates threads and validates links, discussed later in this report.

Contains **myProgram** method, which basically monitors the program, by calculating the start and end time of execution to calculate the elapsed time, and to manage the threadpool executor with the maximum number of concurrent threads chosen by the user.

- **HyperlinkCheckerLinear class**

It functions in a special case scenario when the maximum number of the thread pool executor is 1.

- **Checker class (Utility class to check validity of a single link)**

```
import java.io.IOException;

import org.jsoup.HttpStatusException;
import org.jsoup.Jsoup;
import org.jsoup.nodes.Document;
```

```

public class Checker {

    public boolean checkSingleLink(String link) {
        boolean flag = false;
        try {
            Document mainPage = Jsoup.connect(link).get();
            flag = true;
        } catch (HttpException e) {
            flag = false;
        } catch (IOException ex) {
            flag = false;
        } catch (IllegalArgumentException e)
        {
            flag=false;
        }
        return flag;
    }
}

```

- Links class (Utility class to extract links from a main link)

```

package linkchecker;

import java.io.IOException;

import org.jsoup.Jsoup;
import org.jsoup.nodes.Document;
import org.jsoup.select.Elements;

public class Links {

    int valid;
    int invalid;
    int i;

    public int getValid() {
        return valid;
    }
}

```

```

    }

    public void setValid(int valid) {
        this.valid = valid;
    }

    public int getInvalid() {
        return invalid;
    }

    public void setInvalid(int invalid) {
        this.invalid = invalid;
    }

    public String[] getLinks(String link) throws IOException {
        Document mainPage = Jsoup.connect(link).get();
        Elements links = mainPage.select("a[href]");
        System.err.println("Links: " + links.size() + "\nMAIN PAGE: " +
link);
        String[] pages = new String[links.size()];

        links.forEach(page -> {
            pages[i] = page.attr("abs:href");

            i++;

        });
        return pages;
    }
}

```

- **Runny class**

This class implements runnable and its run method is overridden by the depth recursive method, it's used to create threads in our multithreaded program.

- **InputPanel and outputPanel classes**

Classes used for the GUI interface.

Main algorithm

The input panel calls the HyperlinkChecker constructor with the main link to be checked, desired depth and numbers of threads to be called concurrently.

```
HyperlinkChecker(String link, int depth, int counter)
{
    this.link = link;
    this.depth = depth;
    this.counter = counter;
    e = Executors.newFixedThreadPool(counter);
}
```

The method (depth) is the main method that recursively works and is used to create threads to be executed. It takes 3 parameters, the main link to be checked, and the depth current state and maximum desired number to be reached.

```
public void depth(String link, int current, int max) throws IOException
{
    Checker c = new Checker();
    Links l = new Links();
    if (c.checkSingleLink(link) == false) {
        System.out.println("Invalid Link " + link);
        invalid++;
        return;
    } else if (c.checkSingleLink(link) == true) {
        System.out.println("Valid Link " + link);
        valid++;
        String[] data = l.getLinks(link);
        if (current > max) {
            return;
        } else {
            for (int i = 0; i < data.length; i++) {
                runny myrunny = new runny(data[i], current, max);
            }
        }
    }
}
```



```

        Thread mythread = new Thread(myrunny);
        e.execute(mythread);
        thread_counter++;
    }
}
}
}
}

```

This method creates instances of two classes, **Checker** and **Links**, so that every thread created only has access to its special instances. The **Checker** class has a method that checks if the main entered link is valid. If it's not the case, it's printed on the console that this link is invalid for documentation, and it returns after updating the counter. Else, if the link is valid, the counter gets updated and we call the method `getLinks` that returns an array of Strings with the names of every sublink of the main page, those links are used to complete the program according to the depth chosen by the user. If he chooses depth 0; the links are recursively validated by creating an instance of the **runny** class that implements **Runnable**, each thread is initialized and sent to the executor, according to the threadpool fixed number chosen by the user, this thread is to be executed. As it's a recursive function, the base case for returning is that the current depth is bigger than the maximum depth, which means that our program has come to an end, and the desired depth has been reached and checked.

myProgram method in the `HyperlinkChecker` class is the caller to the method `depth`.

- The time is calculated when the program starts and right before it ends
- Threadpool executor runs until its active count is equal to zero
- Executor is then shut down and an instance of the output panel is created to be shown to the user.

```

public void myProgram() throws InterruptedException {
    long startTime = System.nanoTime();
    try {

        depth(link, 0, depth);

    } catch (IOException ex) {

        Logger.getLogger(HyperlinkChecker.class.getName()).log(Level.SEVERE, null, ex);
    }
    boolean flag = false;
    while(((ThreadPoolExecutor)e).getActiveCount() != 0)

```

```

    {

    }
    e.shutdown();
    e.awaitTermination(10, TimeUnit.MINUTES);

    if(e.isShutdown())
    {
        flag = true;
    }
    if (flag) {
        System.err.println("\nProgram has terminated. CONGRATULATIONS!");
        if(valid== -1)
        {
            invalid = 0;
            JFrame f = new JFrame();
            JOptionPane.showMessageDialog(f, "Input URL is invalid!
(It should be a valid link starting with https://www. ....)");
        }
        else
        {
            System.out.println("Valid count: " + valid + "\nInvalid
count: " + invalid);
            long elapsedTime = System.nanoTime() - startTime;
            System.err.println("Time elapsed with " + counter + "
threads" + " is: " + elapsedTime / 1000000 + " milliseconds.\n");
            new outputPanel(valid, invalid, elapsedTime/ 1000000 ,
counter).setVisible(true);
        }
    }
}

```

Creating a thread

Threads are recursively created in the depth method in HyperlinkChecker class, the executor then takes the threads to be executed by calling their run method.

```

public runny(String link, int current, int max)
{

```

```

        this.link = link;
        this.current = current;
        this.max = max;
    }
    @Override
    public void run()
    {
        try {
            HyperlinkChecker.depth(link, current+1, max);
        } catch (IOException e) {
            // TODO Auto-generated catch block
            System.out.println("ERROR.");
            e.printStackTrace();
        }
    }
}

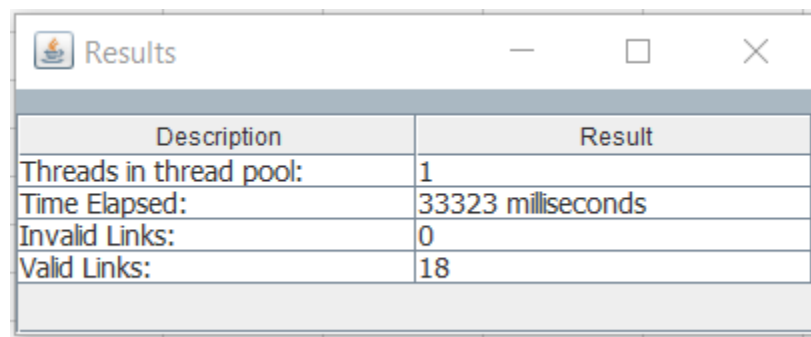
```

Multithreading

Everytime the program gets a link, a thread is created. This thread is an instance of the runny class, its main role is to check the link, dive into it if the depth says it's necessary. The threads are created and simultaneously sent to the executor, which doesn't execute them unless the queue is empty, or the current tasks are done. This is multi-threading, if an exception occurred in one, the other threads are not affected.

Chart plotting numbers of threads used vs time

As our program is a static program to calculate the time elapsed with a certain number of threads, here are the results and a chart plotting them using <https://www.google.com> as a main code, with depth 0;



Description	Result
Threads in thread pool:	1
Time Elapsed:	33323 milliseconds
Invalid Links:	0
Valid Links:	18

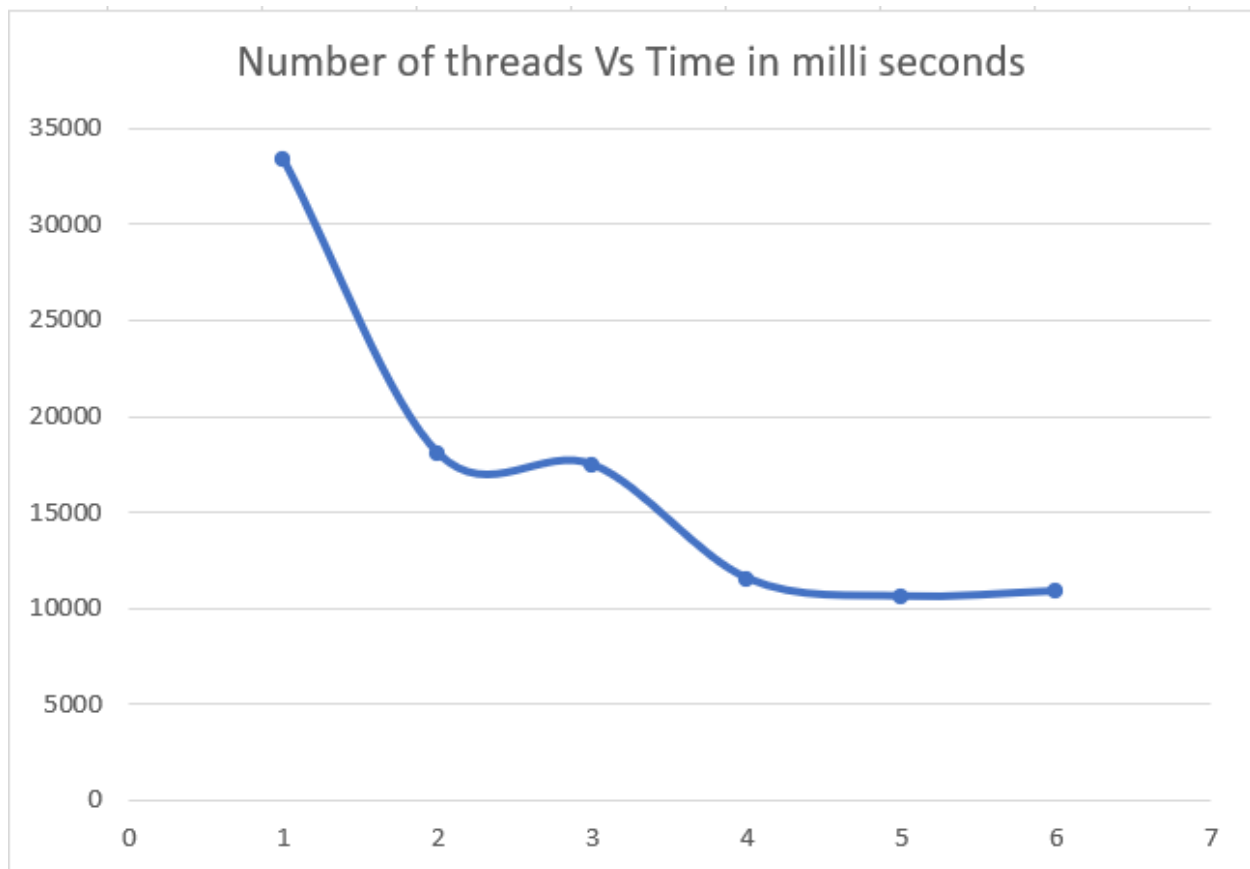
Results	
Description	Result
Threads in thread pool:	2
Time Elapsed:	18037 milliseconds
Invalid Links:	0
Valid Links:	18

Results	
Description	Result
Threads in thread pool:	3
Time Elapsed:	17464 milliseconds
Invalid Links:	0
Valid Links:	18

Results	
Description	Result
Threads in thread pool:	4
Time Elapsed:	11528 milliseconds
Invalid Links:	0
Valid Links:	18

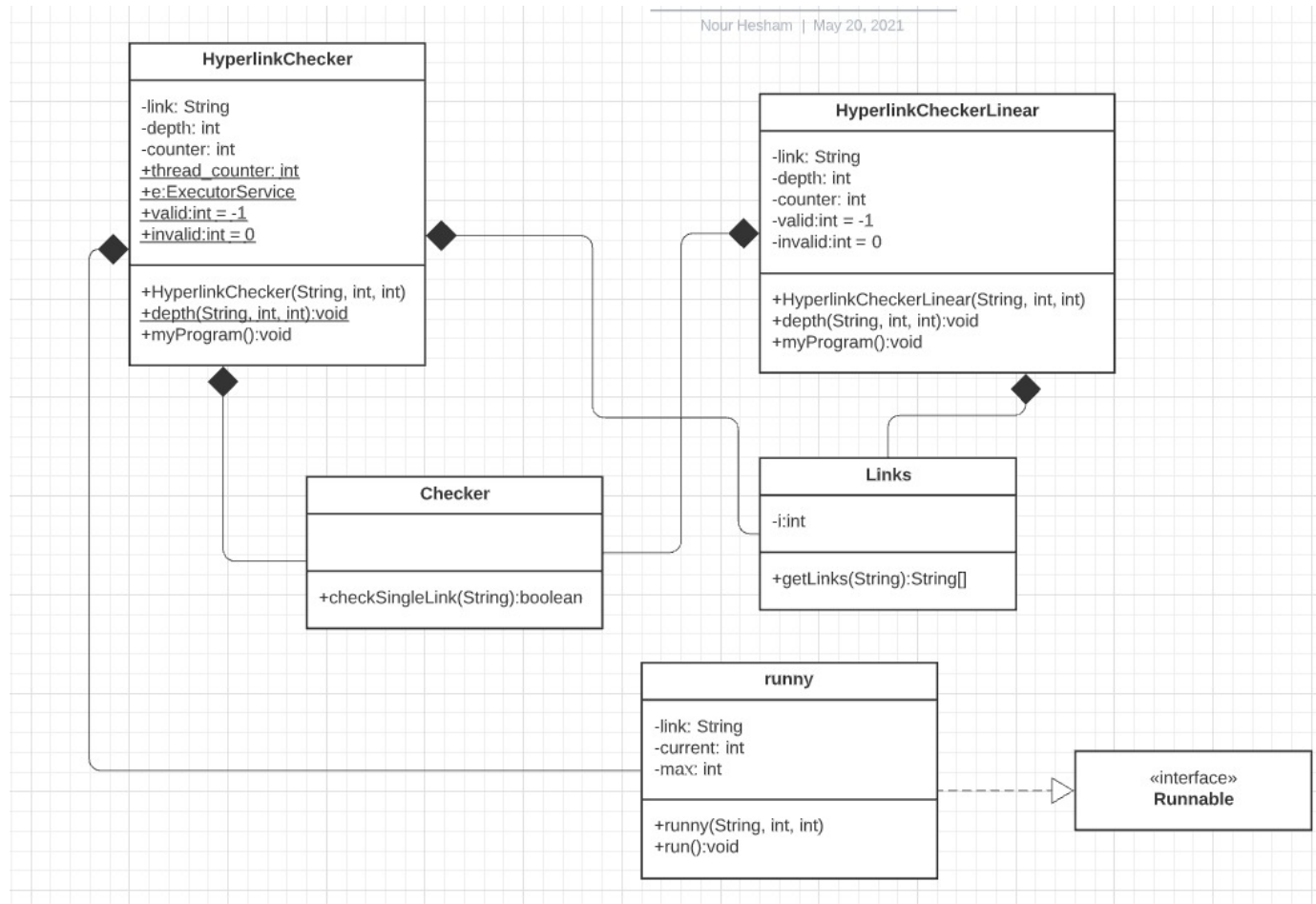
Results	
Description	Result
Threads in thread pool:	5
Time Elapsed:	10690 milliseconds
Invalid Links:	0
Valid Links:	18

Results	
Description	Result
Threads in thread pool:	6
Time Elapsed:	10881 milliseconds
Invalid Links:	0
Valid Links:	18

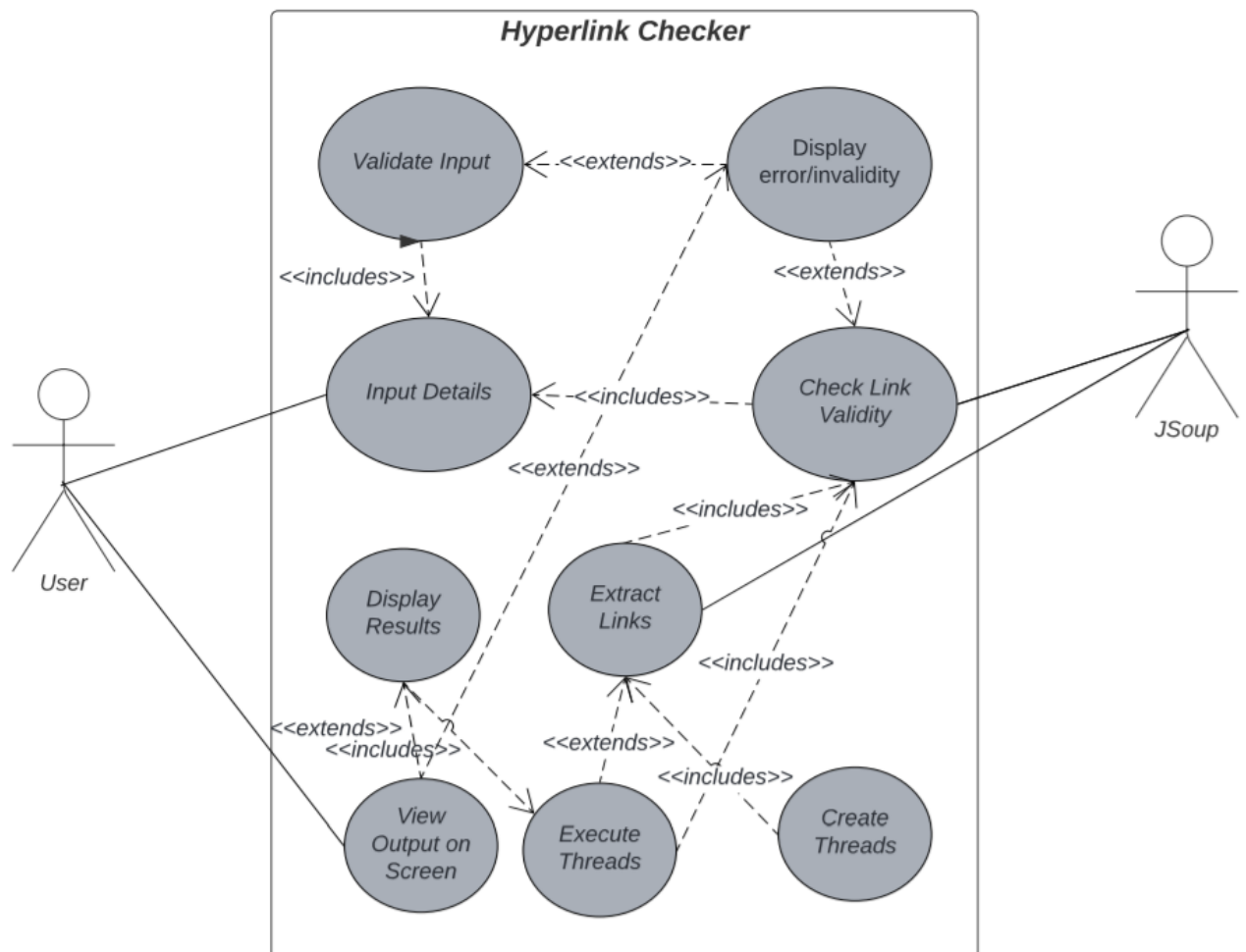


We can clearly notice that by increasing the number of threads (multi-threading) the total time of execution decreases significantly, until it reaches a point of relative stability, that's where the optimal number of threads lie. In this case, the optimal number of threads is 5 threads at a time.

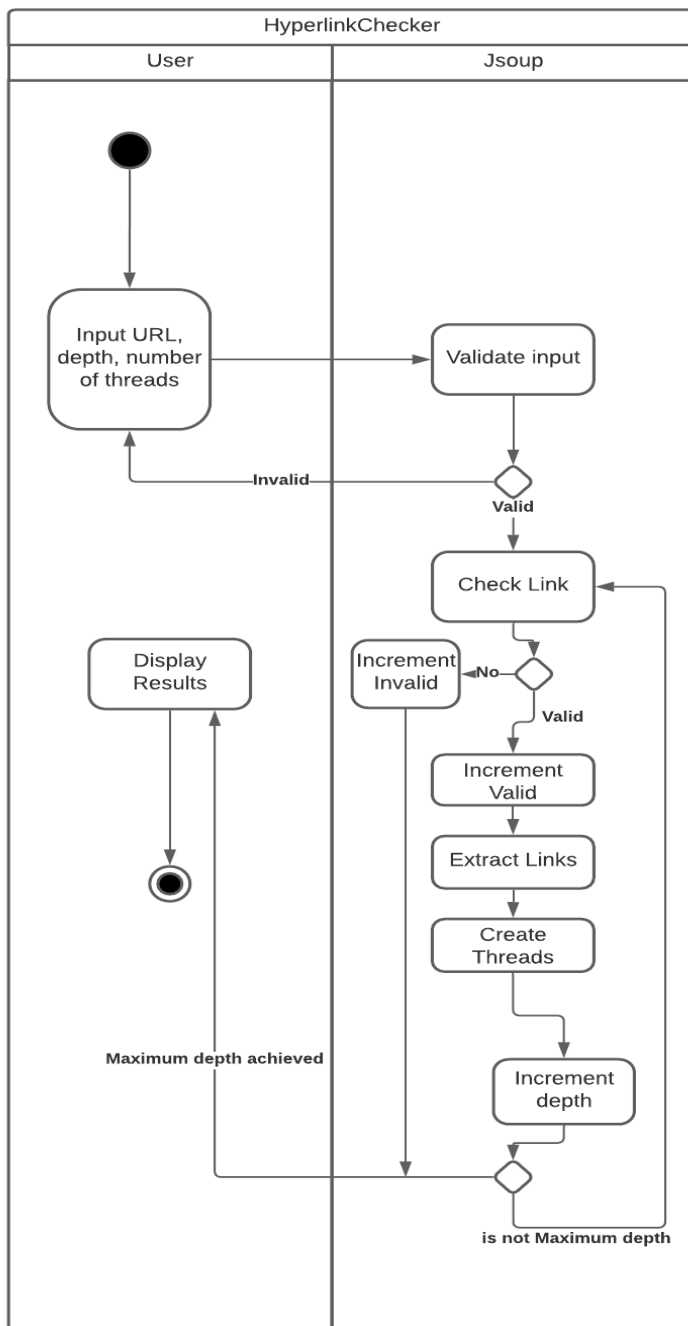
Class UML diagram



Use case UML diagram



Activity UML diagram



© 2006 The Authors

