CS1632:
Systems Testing the Web with Playwright

Wonsun Ahn

Background

- Systems testing: testing the entire system as a whole
  - We would like to automate systems testing, just like we did for unit testing
- So far, all of our testing has been text-based java programs
  - A.k.a. Command Line Interface (CLI) programs
- Automating testing for CLI programs is easy!
  - Just create an “input script” and redirect to stdin
  - Redirect stdout to file and compare to expected output

Automated Systems Test for CLI Programs Using `Bash`

```
#!/bin/bash


# Test case 1: Rent out cat #2 and check cat list
echo -e "2\n2\n1\n5\n" | java -jar rentacat.jar > testcase1.observed.out
diff testcase1.observed.out testcase1.expected.out


…
```

Automated Systems Test for GUI Programs

- Turns out that not every program is a CLI program
  - Web pages, mobile applications, windows applications, etc.
  - A.k.a. Graphical User Interface (GUI) programs
  - How do we deal with these?

- The theory behind testing remain the same
  - Compare **observed behavior vs. expected behavior**
  - Preconditions
  - Execution steps
  - Postconditions

- But we need different tools to automate testing GUI programs

Insight: GUI Apps ≈ Text-based Apps

- GUI apps also have a text representation for the output
  - It's just that the text is rendered into a graphical representation for end-user

- Example: Web applications with HTML (HyperText Markup Language)
  - HTML text is fetched from web server when a URL is requested
  - HTML text is rendered by web browser into graphical elements
- Example: Mobile applications with XML (Extensible Markup Language)
  - XML text is fetched from mobile app server or generated by app
  - XML text is rendered by Android / iOS into graphical elements

- So, in theory, GUI apps could be tested just like text-based apps
  - Using a simple string comparison of expected and observed output text
  - Assuming rendering is bug-free (safe to assume for modern browsers, Android / iOS)

Testing a Web App like a Text-based App

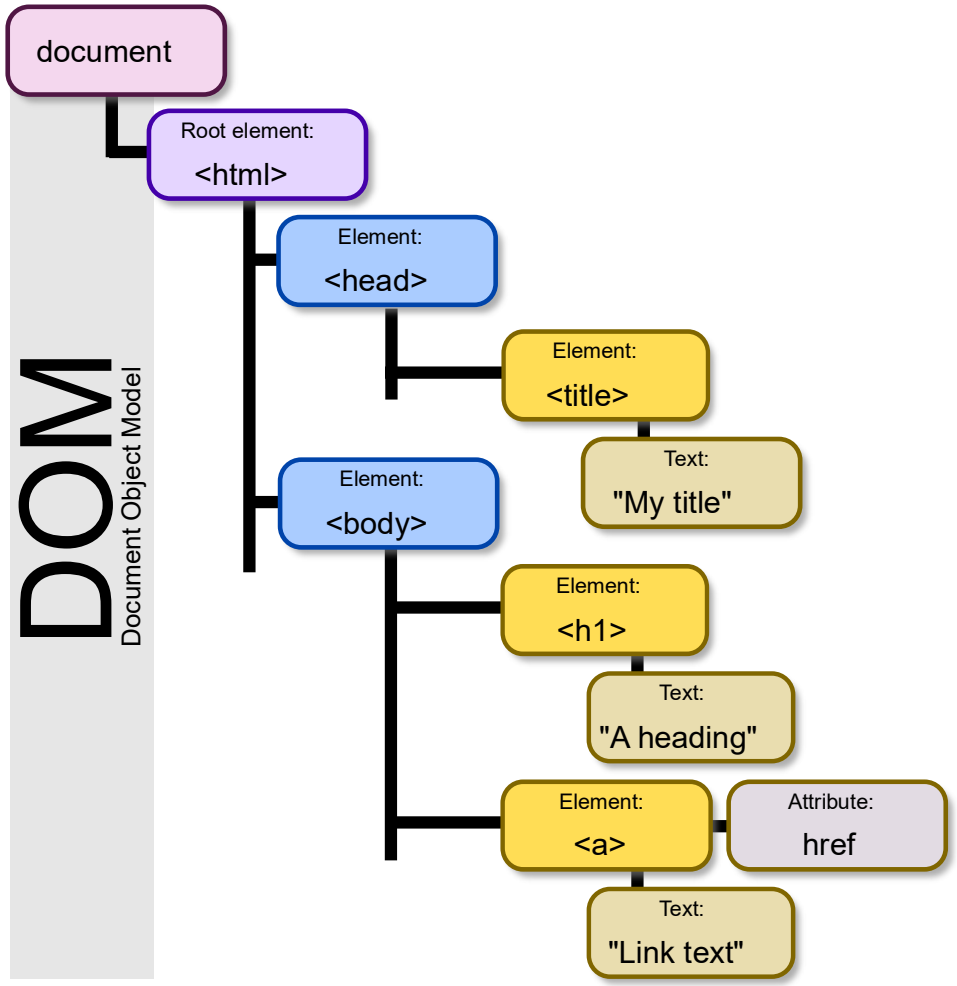
```
#!/bin/bash

# Test case 1: Fetch index.html and compare
wget example.com/index.html
diff index.html index.expected.html
…
```

1. HTML page is fetched from web server using `wget` utility
2. Fetched HTML page is compared against the expected HTML page

- Do you see any problems with this testing methodology?

HTML is organized into a hierarchy of elements

- Element: tag + content + attributes
- Tag: specifies type of element
  - <title>, <table>, <button>, <img>, …
  - <script>: JavaScript code that runs on browser
- Content: text to be displayed
- Attribute:
  - Specifies layout of element
  - Specifies action performed on interaction
- This model of web page is called DOM
  - DOM: Document Object Model



Problems with Naïve HTML Comparison

1. Tests are fragile
   - Trivial changes in HTML that don't impact final display can break test

2. Tests are untargeted
   - Changing HTML elements unrelated to test target will break test

3. JavaScript code is not functionally tested
   - JS code is compared verbatim to expected JS code, instead of executed
   - Changes in JS code that don't change functionality (e.g. commenting) breaks test

→ All of these lead to false positive defects while testing!

1. Tests are Fragile

- Are the two really different when displayed? (Hint: No)

[Expected HTML]

```
<html>
<head>
    <title>Example</title>
</head>
<body>
    …
</body>
</html>
```

[Observed HTML]

```
<HTML>
<HEAD>
    <TITLE>Example</TITLE>
</HEAD>
<BODY>
    …
</BODY>
</HTML>
```

2. Tests are Untargeted

- Which HTML element are we testing?

```
<html>
<head>
    <title>Example Domain</title>
    <style type="text/css">
    a:link, a:visited {
        color: #38488f;
    }
    </style>
</head>
<body>
    <a href="https://www.iana.org/domains/example">More info</a>
</body>
</html>
```

Title text?

CSS URL link color?

URL link?

Body text?

3. JavaScript code is not functionally tested

- Are the two really different when displayed? (Hint: No)

[Expected HTML]

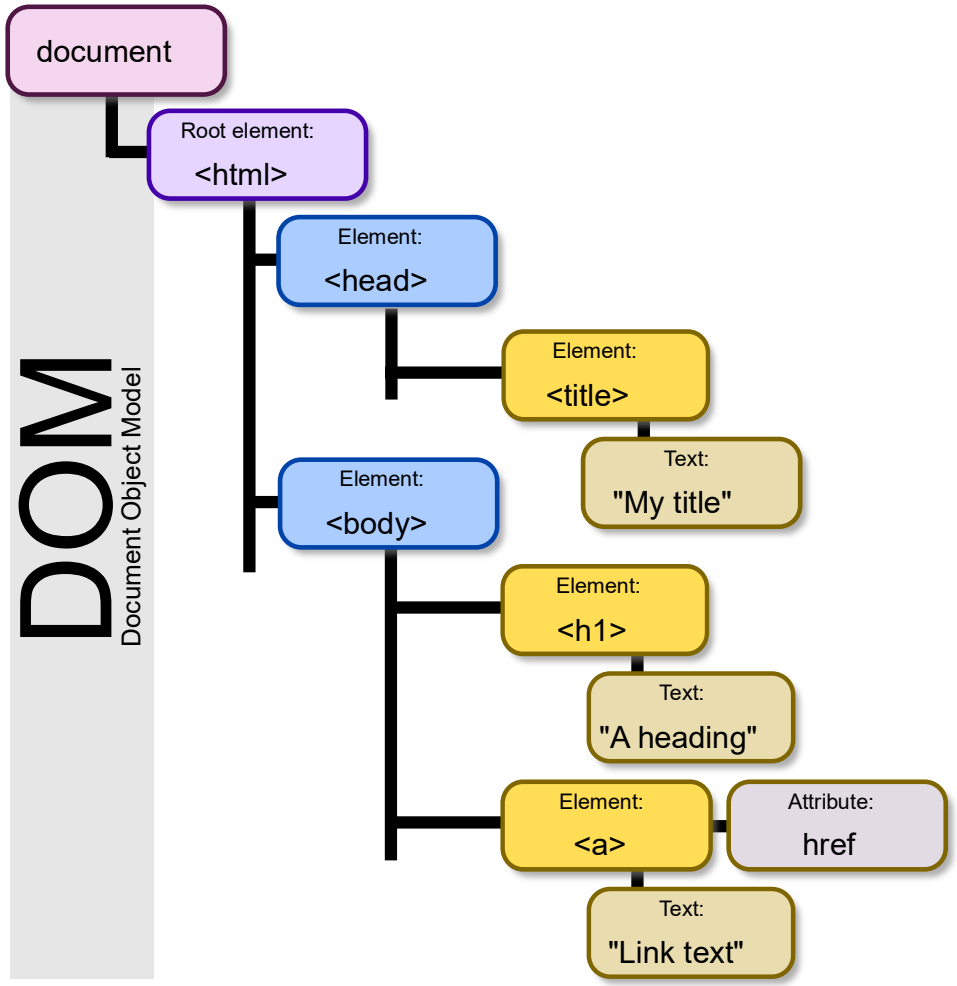
```
<html>
    <head>
        <script>
        function doAlert() {
            var msg = 'hello';
            alert(msg);
        }
        </script>
    </head>
    <body onload="doAlert();"></body>
</html>
```

[Observed HTML]

```
<html>
    <head>
        <script>
        function doAlert() {
            var message = 'hello';
            alert(message);
        }
        </script>
    </head>
    <body onload="doAlert();"></body>
</html>
```

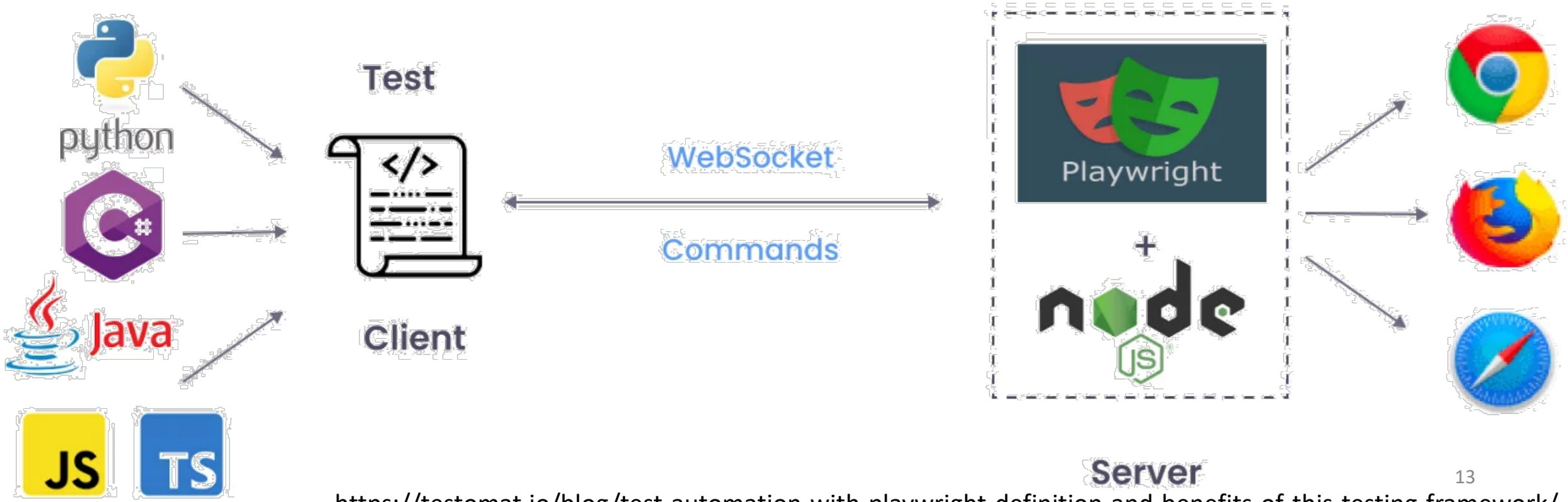
Solution: Web Testing Frameworks

- **Web testing framework**: Framework for testing web apps at a semantic level
  - **Robust**: Works at DOM level after parsing HTML into a DOM tree
  - **Targeted**: Provides APIs to target individual HTML elements
  - **Functionally tested**:
    - Provides APIs to call JavaScript code
    - Can emulate events like clicking or typing, which in turn invokes JavaScript code
- Examples: Selenium, **Playwright**, Cypress



Playwright is **cross-language / cross-browser**

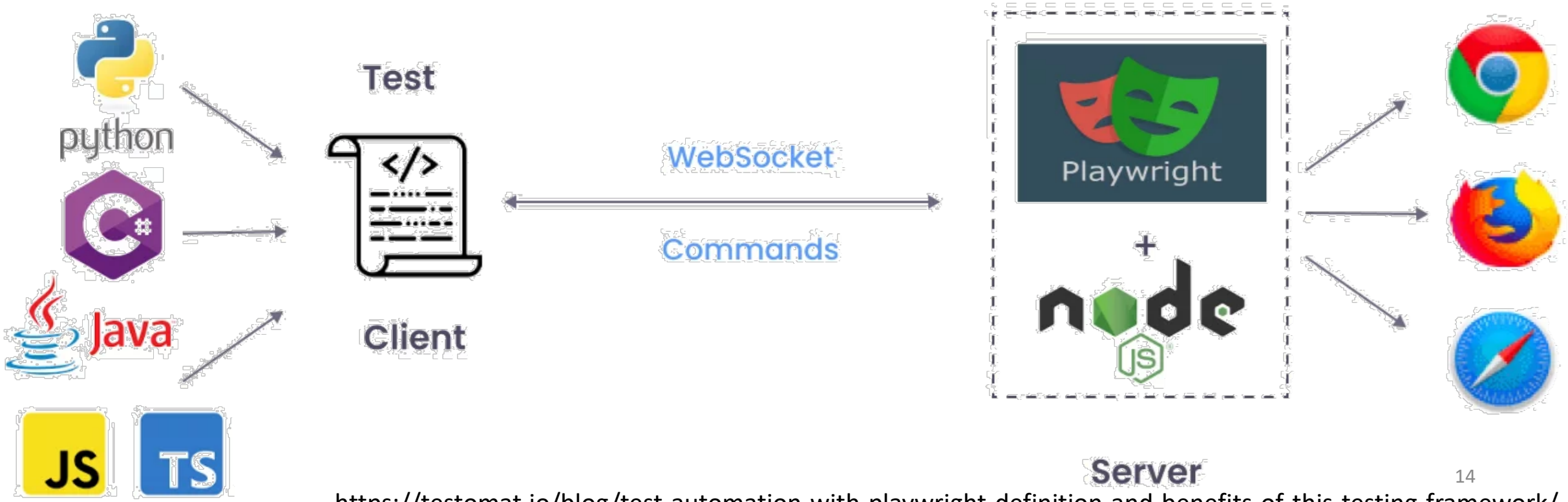
- Node.js server maintains websocket connections with client and browsers
  1. Test client sends JSON commands to Node.js server
  2. Node.js server translates commands to native protocol messages for browsers



https://testomat.io/blog/test-automation-with-playwright-definition-and-benefits-of-this-testing-framework/

Playwright works **asynchronously**

- Calling a test API returns a **promise** (a placeholder for a future result)
  - It takes an indefinite amount of time between command and response
  - Test client must be careful of running ahead of browser and causing race conditions



https://testomat.io/blog/test-automation-with-playwright-definition-and-benefits-of-this-testing-framework/

Test client must use `await` for synchronization

- Consider the following test code:

```
test('TEST-2-LOGO-EXISTS', async ({ page }) => {
  // Loads URL https://www.pitt.edu
  await page.goto('https://www.pitt.edu/');
  // Verifies an element with alt-text “Pitt” is visible
  await expect(page.getByAltText(‘Pitt')).toBeVisible();
});
```

- The first `await` allows page to load before verifying element.