UNIVERSITAT POLITÈCNICA DE VALÈNCIA

ACG - Auditoría Calidad y Gestión de Sistemas

2024/2025

Francesco Lorenzoni
PCA25403GU

Practica 8 y 9

TESTAR - Selenium

Contents

Chapter 1

Práctica 8 y 9

1.1 Introduction

The implementation follows the core principles of the TESTAR approach:

1. Automatically detect interactive elements on a web page
2. Derive meaningful actions that can be performed on these elements
3. Execute these actions in sequences to explore the application
4. Check for application correctness using oracles after each action
5. Collect and report statistics about the test execution

The framework was tested on three different web applications:

- LambdaTest Todo App: A simple todo application
- SauceDemo: An e-commerce test application with authentication
- Practice Software Testing: A comprehensive testing playground

Note that for simplicity and readability only partial code snippets have been reported here

1.1.1 Chromedriver flags

Optimization flags were used to slightly improve performance and reliability of the Chrome WebDriver.

Listing 1.1: Chrome WebDriver Configuration

```
# Performance optimization options
chrome_options.add_argument("--disable-search-engine-choice-screen")
chrome_options.add_argument("--disable-infobars")
chrome_options.add_argument("--disable-extensions")  # Disable extensions for speed
chrome_options.add_argument("--disable-dev-shm-usage")  # Overcome limited resource problems
chrome_options.add_argument("--no-sandbox")  # Bypass OS security model
```

Additionally, preferences were set to disable password manager leak detection and other potential popup interruptions:

Listing 1.2: Chrome WebDriver Preferences

```
chrome_options.add_experimental_option("prefs", {
    "profile.password_manager_leak_detection": False,
    "credentials_enable_service": False,
    "profile.password_manager_enabled": False
})
```

The WebDriver was also configured with timeouts to prevent test runs from hanging:

Listing 1.3: WebDriver Timeouts

```
driver.set_page_load_timeout(15)  # 15 seconds max to load a page
driver.implicitly_wait(2)  # 2 seconds implicit wait instead of default 10
```

1.2 CSS Selectors

The framework uses CSS selectors to identify interactive elements on the web page. A comprehensive set of selectors was implemented to cover various types of interactive elements:

Listing 1.4: CSS Selectors for Interactive Elements

```
selectors = [
    "a", "button", "select",  # Most common interactive elements
    "input[type='submit']", "input[type='button']",
    "input[type='text']", "input[type='password']",
    "input[type='checkbox']", "input[type='radio']",
    "textarea",
]
```

These selectors enable the framework to identify and interact with:

- Links and anchors
- Form elements (text fields, buttons, checkboxes, radio buttons)
- Dropdown menus
- Text areas

Additionally, the framework includes filters to exclude problematic elements:

- Links with `target="_blank"` that would open in new tabs
- Links to external domains outside the application under test
- Application-specific elements that should be ignored

1.3 Actions derived

Once interactive elements are identified, the framework derives possible actions that can be performed on each element. The action derivation is element-type aware, meaning different actions are derived based on the type of element:

Listing 1.5: Action Derivation Logic

```
# Basic click action for all elements
actions.append(lambda el=element: (log("click", el), el.click()))

# Actions for text input fields
if tag_name == "input" and element_type in ["text", "password", "email", "search"]:
    random_text = generate_random_text(10)
    actions.append(lambda el=element, txt=random_text: (
        log(f"input text: {txt}", el),
        el.clear(),
        el.send_keys(txt)
    ))

# Actions for select (dropdown) elements
elif tag_name == "select":
    select = Select(element)
    if len(select.options) > 0:
        random_index = random.randint(0, len(select.options)-1)
        actions.append(lambda el=element, idx=random_index: (
            log(f"select option at index: {idx}", el),
            Select(el).select_by_index(idx)
        ))
```

The action selection logic also includes a weighted selection approach to prioritize certain types of actions:

- Text input actions have higher weight (3x)
- Dropdown selection actions have medium weight (2x)
- Click actions have standard weight (1x)

This weighting ensures more thorough testing of form inputs, which were present in the applications tested.

1.4 Oracles

The framework implements oracles to check for application correctness after each action:

1. **JavaScript Error Detection:** Checks if any JavaScript errors have been detected in the console.

Listing 1.6: JavaScript Error Oracle

```
    js_errors = driver.execute_script("""
        return window.jsErrors || [];
```

```
    """)
    if js_errors:
        print(f"Oracle violation: JavaScript errors detected: {js_errors}")
```

2. **Error Page Detection:** Checks if the current page title contains error codes or messages.

Listing 1.7: Error Page Oracle

```
if "404" in driver.title or "500" in driver.title or "Error" in driver.title:
    print(f"Oracle violation: Error page detected: {driver.title}")
```

3. **Error Message Detection:** Checks for visible error messages on the page.

Listing 1.8: Error Message Oracle

```
error_elements = driver.find_elements(By.CSS_SELECTOR, ".error, .alert, [role='alert
    ']")
for error in error_elements:
    if error.is_displayed():
        print(f"Oracle violation: Error message displayed: {error.text}")
```

4. **Broken Image Detection:** Checks for images that failed to load properly using a JS script.

Listing 1.9: Detecting broken images

```
  broken_images = driver.execute_script("""
  const images = document.getElementsByTagName('img');
      const brokenImages = [];

      for (let i = 0; i < images.length; i++) {
          const img = images[i];
          if (img.complete &&
              typeof img.naturalWidth !== 'undefined' &&
              img.naturalWidth === 0 &&
              !img.classList.contains('icon') &&
              !img.classList.contains('svg')) {

              brokenImages.push({
                  src: img.src || 'no-src',
                  alt: img.alt || 'no-alt'
              });
          }
      }
      return brokenImages;
  """)
```

5. **Accessibility Check:** Ensures interactive elements have accessible names, which is important for users relying on assistive technologies.

Listing 1.10: Testing accessibility

```
accessible_name = (
    el.get_attribute("aria-label") or
    el.get_attribute("title") or
    el.get_attribute("alt") or  # For images in links
    el.text or
    el.get_attribute("value")  # For buttons with value
)

# Also check child content for links
if not accessible_name and el.tag_name.lower() == "a":
    # Check if there's an image with alt text
    img_children = el.find_elements(By.TAG_NAME, "img")
    for img in img_children:
        alt_text = img.get_attribute("alt")
        if alt_text:
            accessible_name = alt_text
            break

# For inputs, also consider the placeholder as accessible name
if el.tag_name.lower() == "input":
    accessible_name = accessible_name or el.get_attribute("placeholder")
```

Some oracle checks can be selectively enabled or disabled to balance thoroughness with testing speed.
This helped mostly during development for debugging purposes, since the initial implementation was prone to false positives.

1.5 Testing summary

The implementation includes a simple test statistics tracking system, collecting various metrics during test execution and providing a summary report at the end of the testing session.

1.5.1 Statistics Collection

The framework tracks the following metrics during test execution:

- **Actions Executed:** Total number of actions performed during the test session.
- **Errors Encountered:** Number of exceptions or errors that occurred during testing.
- **Unique URLs Visited:** Set of distinct URLs the framework navigated to, providing a measure of navigation coverage.
- **Element Types Interacted:** Distribution of interactions across different element types (clicks, text inputs, selections).
- **Test Duration:** Total time elapsed from the beginning to the end of the test session.

At the conclusion of testing, a formatted summary report is displayed.

1.5.2 Example Output

A typical summary report might look like:

```
================================================
TEST EXECUTION SUMMARY
================================================
Total actions executed: 150
Errors encountered: 3
Unique URLs visited: 12
URLs: https://example.com/home, https://example.com/products, ...
Element types interacted with:
  - click: 78 interactions
  - text_input: 42 interactions
  - select: 30 interactions
Total test duration: 0:05:23.456789
================================================
```

The summary could be further extended to include additional metrics such as unique states visited, form submission counts, or specific interface components tested, depending on the application's nature and testing objectives.

1.6 Conclusions

The implementation was tested using the `testingTESTAR.py` file provided in Poliformat.

Key issues encountered during the implementation included:

- **External Links Disruption:** Links to external domains or with `target="_blank"` would open new websites or tabs, disrupting the testing flow.
  - *Solution:* Implemented automatic filtering of external links based on domain comparison and target attribute detection.
- **Browser Interruptions:** Chrome's password manager and other browser features would display popups that interrupted automated testing.
  - *Solution:* Chromedriver was configured with options to disable password manager prompts and other popups that could interfere with test execution.
- **Accessibility Testing False Positives:** Many elements were initially flagged as accessibility violations that were actually acceptable in context.
  - *Solution:* Improved accessible name detection to check for more attributes and child elements, implemented tracking to avoid reporting the same violation multiple times.
- **Broken Image Detection:** Initial implementation had many false positives when checking for broken images.
  - *Solution:* Improved the JavaScript detection logic and added special cases for SVG and icon images.

- **Performance Concerns:** Testing the implementation was initially slow due to network operations and unnecessary checks.
  - *Solution:* Added options to selectively enable/disable certain oracles and optimized browser settings.