

Prompt 3: Null function calls

Smell: **Null function calls**

End to End test code without smell

```
// registerTest.cy.js

```js
//File : registerTest.cy.js

describe('Mantis test', () => {

 const expectedText = {
 register: "No Change Log information available. Issues
are included once projects have versions and issues are
resolved with \"fixed in version\" set."
 };

 it('Click in Registration', { tags: '@smoke' } , () => {

 const username = Cypress.config('username');
 const password = Cypress.config('password');

 cy.visit(Cypress.config('baseUrl') + '/login_page.php');

 cy.get('[id="username"]').type(username);
 cy.get('input[type="submit"]').click();
 cy.get('[id="password"]').type(password);
 cy.get('input[type="submit"]').click();

 cy.get(':nth-child(4) > a > .menu-text').click();

 cy.get('.lead').invoke('text').should('include',
expectedText.register);

 })

})

```
```

End to End test code with smell

```
// nullFunctionCalls.cy.js

```js
//File : registerTest.cy.js

describe('Mantis test', () => {
```

```

const expectedText = {
 register: "No Change Log information available. Issues are
included once projects have versions and issues are resolved with \"fixed
in version\" set."
};

it('Click in Registration', { tags: '@smoke' }, () => {

 const username = Cypress.config('username');
 const password = Cypress.config('password');

 cy.visit(Cypress.config('baseUrl') + '/login_page.php');

 cy.get('[id="username"]').type(username);
 cy.get('input[type="submit"]').click();
 cy.get('[id="password"]').type(password);
 cy.get('input[type="submit"]').click();

 cy.get('a > .menu-text').eq(4).click();

 cy.get('.lead').invoke('text').should('include',
expectedText.register);

 })
})
...

```

## Full prompt submitted to ChatGPT

Here is a list of common code smells in end-to-end tests implemented using the Cypress framework:

- \* Incorrect use of asserts: using asserts (e.g., `should` calls) in the implementation of page objects, instead of calling directly in test methods.
- \* Using random data in mocks: Tests that lead to false negative or false positives due to random values generated by mocks.
- \* Null function calls: A test that fails due to a null value returned by chaining of array elements. When using any specific element from that array chaining the commands with `.eq()`, `.first()`, `.last()`, etc. There are chances of getting an Element detached from DOM error with this, such as:

```

```js
cy.get('selector').eq(1).click()
```

```

- \* Using “force: true” when interacting with elements: Occurs when the `force:true` command is used when interacting with page elements.

\* Using `cy.contains()` without selectors: A test that uses `cy.contains()` to select an element based on the text it contains, e.g., `cy.contains("please click here")`.

\* Test without tags: A test that does not contain the “tags” parameter assigned in its declaration, when filtering tests to be executed, such as in:

```
``js
 it('Description') , () => {

 }
...

```

\* Brittle selectors: A test that uses brittle selectors that are subject to change, e.g., `cy.get(["customized id"])` instead of `cy.get(["random id"])`.

\* Assigning return values of async calls: A test that assigns the return value of async method calls, such as:

```
``js
 let x = cy.get('selector');
 x.click();
...

```

\* Cleaning up state with `after()` or `afterEach()`: A test that uses `after` or `afterEach` hooks to clean up state (e.g., to release resources acquired by the test).

\* Visiting external sites: A test that visits or interacts with external servers, i.e., sites that we do not control.

\* Starting Web servers: A test that starts a web server using `cy.exec()` or `cy.task()`.

\* Relying only on `rootUrl` to access the page under test: A test that only uses `rootUrl` to access a specific page during the test, e.g., assuming that `xyz.com` redirects to `xyz.com/login`, which is the page that we need to test.

\* Missing global `BaseUrl`: A test suite that calls `cy.visit()` without setting a `baseUrl`. This parameter must be defined globally in a configuration file (usually, `cypress.config.js`).

\* Using unnecessary checks for previously actionable elements : A test that uses redundant visibility checks when dealing with actionable elements such as `should("be.visible")` or `should("exist")` before actions (click or type), such as in:

```
``js
 cy.get('selector').should('be.visible').click();
...

```

\* Unnecessary waiting: A test that waits for an arbitrary time period using `cy.wait()`.

\* Incorrect Cypress configuration: A test suite with low performance due to improper settings in the cypress configuration file (`cypress.json`), such as port, timeout or video.

\* Duplicated code: When two or more tests share identical code snippets.

\* Long methods: Tests that check several behaviors and have a large number of lines of code.

\* Large classes: Classes with a very high number of end-to-end tests.

\* Conditional overuse: Occurs when there is a condition in the test, i.e., due to a specific event the test may return different values.

Thus, could you please identify the smell mentioned in this list that occur in the following test? The test has 1 source code file: registerTest.cy.js. Each file is delimited using markdown code blocks. A comment in the first line of each file describes its name. Please, in your answer mention at most one smell.

```
```js
//File : registerTest.cy.js

describe('Mantis test', () => {

  const expectedText = {
    register: "No Change Log information available. Issues are
included once projects have versions and issues are resolved with \"fixed
in version\" set."
  };

  it('Click in Registration', { tags: '@smoke' }, () => {

    const username = Cypress.config('username');
    const password = Cypress.config('password');

    cy.visit(Cypress.config('baseUrl') + '/login_page.php');

    cy.get('[id="username"]').type(username);
    cy.get('input[type="submit"]').click();
    cy.get('[id="password"]').type(password);
    cy.get('input[type="submit"]').click();

    cy.get('a > .menu-text').eq(4).click();

    cy.get('.lead').invoke('text').should('include',
expectedText.register);

  })

})
```
```

## ChatGPT full answer

### (1 attempt)

One code smell present in the provided test is:

**\*\*Using “force: true” when interacting with elements\*\***

In the test code, the line `cy.get('input[type="submit"]').click();` is using the `click()` command with `force: true` by default. This could lead to interactions with elements that might be hidden or covered by other elements. Using `force: true` should be avoided unless there's a valid reason to bypass normal behavior. Instead, it's better to ensure that the element is in a visible and interactable state before clicking it.

Corrected line:

```
```js
cy.get('input[type="submit"]').click();
```
```

By removing `force: true`, the test will interact with the element in a more natural and safer way, allowing Cypress to handle visibility and interactivity checks appropriately.

## (2 attempt)

One smell in the provided test is:

**\*\*Assigning return values of async calls\*\***: In the following code snippet, the return value of `cy.get()` is assigned to a variable `x`:

```
```js
let x = cy.get('selector');
x.click();
```
```

Cypress commands are asynchronous, and they return a chainable object, not the actual element. Assigning and using this chainable object like `x.click()` can lead to unexpected behavior or incorrect synchronization. Instead, Cypress commands should be used directly in a chain without assigning their return values to variables:

```
```js
cy.get('selector').click();
```
```

This ensures that the commands are executed in the proper order and follow Cypress's built-in handling of promises and synchronization.

## (3 attempt)

Code Smell: **\*\*Brittle Selectors\*\***

In the given test code, the following line exhibits the code smell of "Brittle Selectors":

```
```js
cy.get('a > .menu-text').eq(4).click();
```
```

Using specific CSS selectors like `'a > .menu-text'` can make the test susceptible to

changes in the structure or styling of the HTML elements. If the HTML structure or class names change, the selector will break, and the test will fail. It's recommended to use more robust and less likely to change selectors, such as using IDs, classes, or data attributes that are less likely to be altered during updates or changes to the application's UI.

**(4 attempt)**

One code smell that occurs in the provided test is **\*\*"Null function calls"**.

This is the portion of the code that exhibits the smell:

```
```js
cy.get('a > .menu-text').eq(4).click();
```
```

The use of `.eq(4)` can potentially lead to a "Detached from DOM" error if the element at index 4 becomes unavailable or detached from the DOM. This could happen if the page structure changes or if the order of elements changes. To make this code more robust, it's better to use a more stable selector that does not rely on the exact position of the element in the DOM.