

07/20/2021

Report for code reusability in automation testing at Pure access team

1 Introduction

2 Approaching

3 Comparison tables

4 Conclusion

Introduction

Why we need the automation test:

As long as the software becomes more complicated, automation test has been a crucial part during the development. Especially in CI, CD, and Agile development, automation tests can help developers quickly find where the bugs are and solve them. And comparing with the manual test, automation tests can avoid test omissions caused by the test engineers' inertial thinking. And also, it can reduce the human errors caused by complicated and repetitive work in manual testing.

Why we need to make code reusable:

The cost of difficult-to-maintain software is extremely high. When the scale of the software continues to expand, no one can afford the comprehensive cost of such software, and it is difficult to obtain reliable products even with high capital investment, and the idea of code reuse is the fundamental way to solve this problem.[1]

Code reuse, design, and testing are all indispensable assets within the software development organization. To measure the code reusability, the characteristics are modularity, low coupling, high cohesion, data encapsulation, and SOC (Separation of concerns).[1]

Types of reusable components and the outcome for it:

Normally, we can make four types of reusable components, code block, algorithm, pattern, and abstract data type. But in the automation test at ISONAS, the code block is the main part. Some repeating code blocks have the same functionality, if we make them reusable, they can be clearer and easier to use and maintain. In the following section, we will discuss the methods to make the code reusable, and how those methods compare with each other.

Approaching

1 Wrap the repeating code block as a function

Since there is some repeating code block in automation test, we can try to wrap those code block which has the same functionality as one function. When defining those functions, the function should expose some extendable parameters to allow other programmers to easily call them in different scenarios.

To demonstrate this method, I am going to take confirm widget validation as an example.

E.g.

```
//confirm widget creation

cy.wait('@newWidget').then( fn: (newWidget : Interception ) => {
  newWidgets = newWidget
  let dynamicId = `${DASHBOARD_WIDGET_ID}${newWidget.response.body.id}`

  cy.get(`[id=${dynamicId}]`);
})
```

screenshot for original widget creation

This is one of the functionalities which appears 21 times only in the create_widget_spec.js file. And it takes 106 lines. Instead of writing each time when creating, editing, and deleting the different type widgets, we can call it from the outside in another file and wrap it as a function, this step can make it just 21 lines for this test file.

Also as I mentioned early, to make it suitable for more cases, the implementation for this helper function needs to expose extendable parameters to allow it to handle more situations.

So for this example, to make one function compatible with creating, editing, and deletion, I define createNot, editNot, and deleteNot as boolean parameters. When programmers code on the automation side, they can have options to select the testing behavior about the certain widget.

E.g.

```
* @param {Boolean} createNot True when it validates the create-widgets
* @param {Boolean} editNot True when it validates the edit-widgets
* @param {Boolean} deleteNot True when it validates the delete-widgets
*/
export function confirmWidgets(alias :string, prefixWidgetID :string, newWidgets :temp_var, createNot :Boolean, editNot :Boolean, deleteNot :Boolean ){
  return new Promise( {executor: function(resolve, reject){
    cy.wait(alias).then( fn: (newWidget :Interception ) => {
      if(createNot){
        newWidgets = newWidget;
        resolve(newWidgets);
      }
      let dynamicId = `${prefixWidgetID}${newWidget.response.body.id}`;

      if(editNot || createNot){
        cy.get(`[id=${dynamicId}]`);
      } else if(deleteNot) {
        cy.get(`[id=${dynamicId}]`).should( 'chainers', 'not.exist');
      } else{
        // make it extendable when needs more validations for widgets
        throw new Error("The confirm Widgets qa failed. Please check the widgets operation for auto-test");
      }
    })
  })
}
```

screenshot for confirmWidgets function

1.1 Document the function

Documentation is also one of the most important parts of this process. Since we wrap the code block as a function in another file, it will be a little bit difficult to find the comment about the certain function. To handle this, I use the comment block in IntelliJ IDE. So we can define the parameters' type, comments. And those hints will automatically pop up when programmers try to use confirmWidgets function.

E.g.

```
1
2 // confirm if widget successfully create, edit, or delete by checking the ID
3
4 /**
5  * @param {string} alias The alias
6  * @param {string} prefixWidgetID WidgetId for diff purpose
7  * @param {temp_var} newWidgets Variable to store the newWidgets ID
8  * @param {Boolean} createNot True when it validates the create-widgets
9  * @param {Boolean} editNot True when it validates the edit-widgets
10 * @param {Boolean} deleteNot True when it validates the delete-widgets
11 */
```

screenshot for documentation

1.2 Advantage and disadvantage:

There are two main advantages and one disadvantage of this approach.

First, wrapping the code block as a function can make the view of the code clear. Instead of repeating the same code multiple times, one function call can reduce the line to a “linear space”.

Second, collecting all the helper functions in one file can help programmers to save time to build their own “gears”. Sometimes, when the test file includes hundreds of line, it’s hard for a new programmer trying to find and use functionality which has already implemented. Building a collection of the helper functions can solve this problem. Before the programs, they can simply go through this file to know which code block has already been wrapped as a callable function. This can save a bunch of time.

The only disadvantage of this method is the “hidden” definition. Because the function definitions are commented outside of the testing file. It may be difficult when the developer wants to know what this function is doing. And if the programmers who define these helper functions don’t comment on it well, that also may cause an incomprehensible problem.

1.3 Enhance the robust

Robust is an important part of when developers design the function. The main idea is to make it easy to extend without rewriting them. Take the confirmWidgets function as an example again. In this case, the function can throw the error when it has other cases except for creation, editing, and deletion. This functionality makes this case can handle more cases and alarm correctly.

E.g.

```
if(editNot || createNot){
  cy.get(`[id=${dynamicId}]`);
} else if(deleteNot) {
  cy.get(`[id=${dynamicId}]`).should('not.exist');
} else{
  // make it extendable when needs more validations for widgets
  throw new Error("The confirm Widgets qa failed. Please check the widgets operation for auto-test");
}
```

screenshot for handling corner case

2 Wrap the repeating code block as a command

The idea of repeating code block as a command came out when I read the function inside the `create_widget_spec.js`. Test engineers used `useSearchableSelect` to select specific options like filters on the toggles. They defined `useSearchableSelect` as a command instead of writing several `cy.get` and `cy.click` operations.

In this section, I will continue to demonstrate how to turn the standard action into the commands in the confirm widgets editing and deleting.

E.g.

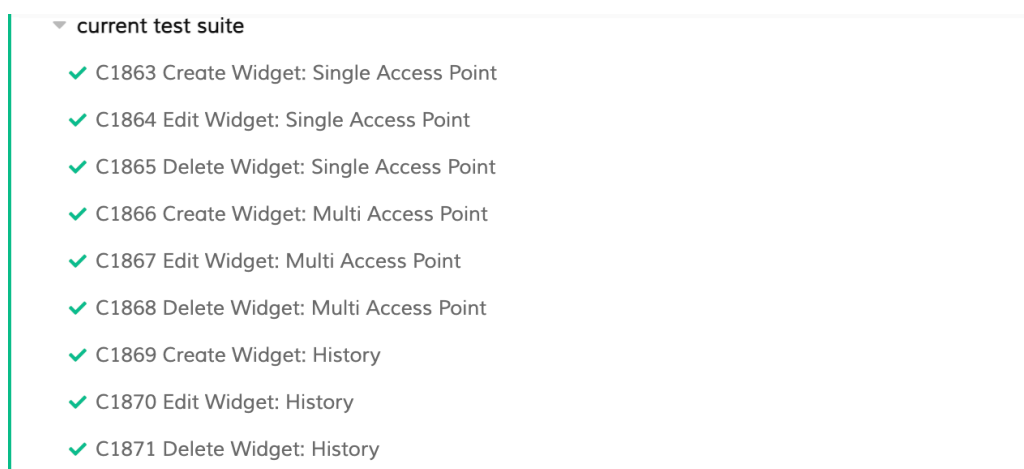
A screenshot of a code editor showing a Cypress test function. The code is as follows:

```
340 it(TEST_CASE_ID_C1880, {config: function () {
341
342   // spy on endpoints for page
343   cy.intercept('POST', DELETE_WIDGET).as('deleteWidget');
344
345   cy.get(`[id=${DASHBOARD_WIDGET_DELETE_BUTTON_ID}${newWidgets.response.body.id}]`).click();
346
347   cy.get(`[id=${DASHBOARD_WIDGET_DELETE_DIALOG_DELETE_BUTTON_ID}]`).click();
348
349   //confirm dashboard widget deletion
350   confirmWidgets( alias: 'deleteWidget', DASHBOARD_WIDGET_ID, newWidgets, {createNot: false, editNot: false, deleteNot: true}).then((newWidget) => {
351     newWidgets = newWidget;
352   })
353 })
354 })
355 })
356 })
357 }
```

screenshot for delete widget

This is a test code for widget deletion. According to the code, we can see this test is getting the `delete_button_id` and clicking on it. And then confirm if deleted the widgets we just created, edited.

E.g.

A screenshot of a test suite list in a web application. The list is titled "current test suite" and contains ten items, each with a green checkmark icon and a text description:

- ✓ C1863 Create Widget: Single Access Point
- ✓ C1864 Edit Widget: Single Access Point
- ✓ C1865 Delete Widget: Single Access Point
- ✓ C1866 Create Widget: Multi Access Point
- ✓ C1867 Edit Widget: Multi Access Point
- ✓ C1868 Delete Widget: Multi Access Point
- ✓ C1869 Create Widget: History
- ✓ C1870 Edit Widget: History
- ✓ C1871 Delete Widget: History

screenshot for showing the repeating create, edit, delete widgets

And in all the different widgets we created in this file, they all had these same functionalities. So we can transfer this code block as a command as well as the editing widgets.

E.g.

A screenshot of a code editor showing a Cypress command definition. The code is as follows:

```
Cypress.Commands.add( name: "confirmDeleting", fn: (newWidgets) => {  
  // spy on endpoints for page  
  cy.intercept('GET', GET_AUTH_USER).as( alias: 'authUser');  
  cy.intercept('POST', DELETE_WIDGET).as( alias: 'deleteWidget');  
  
  cy.get(`[id=${DASHBOARD_WIDGET_DELETE_BUTTON_ID}${newWidgets.response.body.id}]`).click();  
  
  cy.get(`[id=${DASHBOARD_WIDGET_DELETE_DIALOG_DELETE_BUTTON_ID}]`).click();  
  
  //confirm dashboard widget deletion  
  confirmWidgets( alias: '@deleteWidget', DASHBOARD_WIDGET_ID, newWidgets, {confirmCreating: false, confirmEditing: false, confirmDeleting: true}).then((newWidget) => {  
    newWidgets = newWidget;  
  });  
});
```

screenshot for delete widget as a command

We can call it in the test file by wrapping this code block as a command in commands.js.

E.g.

A screenshot of a code editor showing a Cypress test case. The code is as follows:

```
it( TEST_CASE_ID_C1880, config: function () {  
  //confirm deleting  
  cy.confirmDeleting(newWidgets);  
})
```

screenshot for using the confirmDeleting in test case

Since there are seven different types of widgets, those test cases' editing and deleting can replace with those command.

2.1 Document the function

The documentation is almost the same as previous version because we only wrap the code as a command. There are only few parameters that need to expose.

2.2 Advantage and disadvantage:

One of the advantages of this way is to make the test code shorter and more clear. The original way need to repeat editing and deleting in each type of widget. It took six lines for each time. And currently, it only need 1 line to do this work. Overall, it saved 84 lines.

The disadvantage of this method is it's not as flexible as the function. Setting as a command can also take the parameters. But since “it” and “describe” are the interface that is implemented in Mocha. It will be tough for people to understand when passing several parameters. So wrapping the code block as a command is good to use when the parameters are few.

2.3 Why “createWidgets” is not commandable

This was a question coming up when we tried to wrap the createWidgets as a command. The reason we do not operate is that it may be over-engineering the function.

For createWidgets for different widgets, they have some similar parts. Take “SAP” and “History” widgets as an example.

E.g.

```
// Create widget
cy.useSpeedDial(DASHBOARD_SPEED_DIAL_ID, DASHBOARD_SPEED_DIAL_ACTION_CREATE_WIDGET_ID);

cy.get(`[id=${DASHBOARD_WIDGET_DIALOG_NAME_INPUT_ID}]`).type({text: 'SAP Widget'});
cy.useSearchableSelect(
  DASHBOARD_WIDGET_DIALOG_TYPE_SELECT_ID,
  DASHBOARD_WIDGET_DIALOG_TYPE_SELECT_SEARCH_ID,
  WIDGET_TYPE_SINGLE_ACCESS_POINT,
  DASHBOARD_WIDGET_DIALOG_TYPE_SELECT_CONTAINER_ID);

cy.useSearchableSelect(
  DASHBOARD_WIDGET_DIALOG_ACCESS_POINT_SELECT_ID,
  DASHBOARD_WIDGET_DIALOG_ACCESS_POINT_SELECT_SEARCH_ID,
  createAccessPointBody.accessPointInfo.name,
  DASHBOARD_WIDGET_DIALOG_ACCESS_POINT_SELECT_CONTAINER_ID);

cy.get(`[id=${DASHBOARD_WIDGET_DIALOG_SAVE_ID}]`).click();
```

screenshot for the “SAP” createWidgets

```
// Create widget
cy.useSpeedDial(DASHBOARD_SPEED_DIAL_ID, DASHBOARD_SPEED_DIAL_ACTION_CREATE_WIDGET_ID);

cy.get(`[id=${DASHBOARD_WIDGET_DIALOG_NAME_INPUT_ID}]`).type({text: 'History'});
cy.useSearchableSelect(
  DASHBOARD_WIDGET_DIALOG_TYPE_SELECT_ID,
  DASHBOARD_WIDGET_DIALOG_TYPE_SELECT_SEARCH_ID,
  WIDGET_TYPE_HISTORY,
  DASHBOARD_WIDGET_DIALOG_TYPE_SELECT_CONTAINER_ID);

cy.get(`[id=${DASHBOARD_WIDGET_DIALOG_SAVE_ID}]`).click();
```

screenshot for the “History” createWidgets

One of the differences between these two cases is typing text. This can be solved by passing as a parameter to the command.js.

Another difference is the parameters in useSearchableSelect command. This can only be solved by wrapping the parameters as JSON. Following this way, it can only make the parameter clearer, but it doesn't save the line.

The last difference is the number of useSearchableSelect. In "SAP," it needs to select two options before creating the widgets. In "History," it only needs to choose one. The way to solve this is using the if or switch condition.

Overall, we used wrapping the code block as a command in createWidgets, and it still can't make the code reusable. So we can keep this part.

Comparison tables

1 Comparison between functions and original code

	Lines	Time
Code without function	639	More
Code with function	571	Less
Improvement	11%	

2 Comparison between commands and original code

	Lines	Time
Code without command and with function	571	More
Code with command and with function	430	Less
Improvement	25%	

3 Finally comparison between code use and original code

	Lines	Time
Original code	639	More
Code with command and with function	430	Less
Improvement	32%	

Conclusion

By calling the function and wrapping it as a command in `create_widget_spec.js`, it saves the time to write the code and quickly reduces around 200 lines in a single test file.

So it's efficient work to apply code reuse in automation test.

Citation

[1] https://en.wikipedia.org/wiki/Code_reuse