Assignment 4 - G03P02

Group information

- Ana Inês Oliveira de Barros - `up201806593@fe.up.pt`;
- João de Jesus Costa - `up201806560@fe.up.pt`

Use cases

We selected the following use cases because they were both common use cases for the application and the most interesting to test/draw a model for:

- Add a new project;
- Play/pause a project;
- Edit a project's color;

Use case 1 -- Add a new project

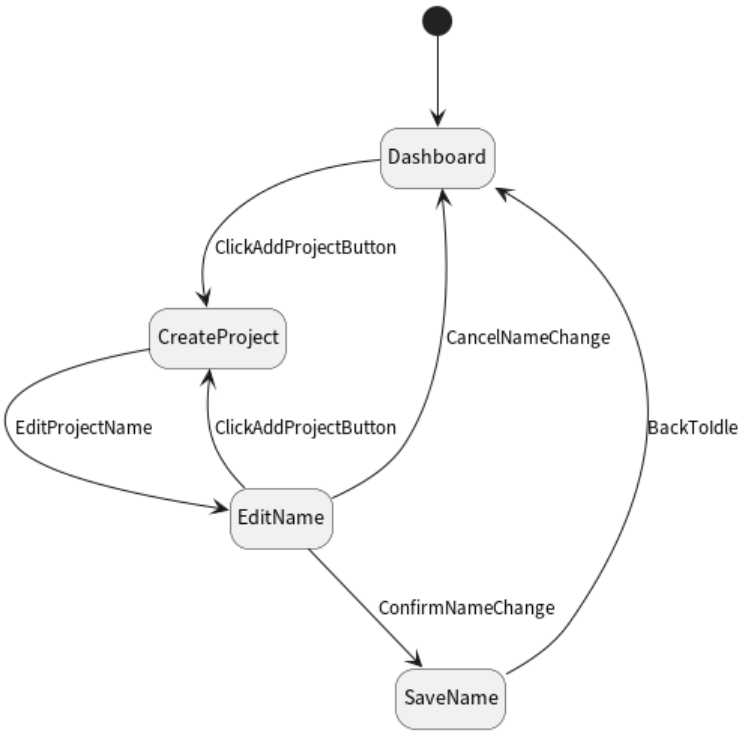
Adding a new project is fundamental for using the application.

State diagram

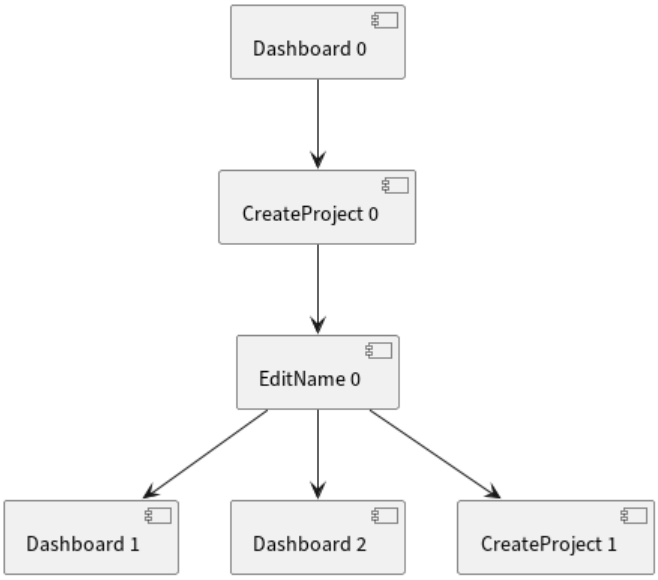
The most important aspects of this use case are:

- When a user **adds** a project while **adding another one** i.e., still editing its name, the name change is confirmed and the user is prompted for the name of the new project;
- Upon creating a new project, the user is immediately prompted to edit the project's name. However, there is the possibility to cancel this action (the project will keep the default name).

We decided that the application's end-state is the moment the application is closed. However, we chose to omit it since every state would have a transition to it.



Transition tree



With 3 leaf nodes in the tree, we need to create 3 tests to cover all states and transitions of the system.

Transition table

	ClickAddProjectButton	EditProjectName	ConfirmNameChange	BackToIdle	CancelNameChange
Dashboard	*CreateProject*				
CreateProject		*EditName*			
EditName	*CreateProject*		*SaveName*		*Dashboard*
SaveName				*Dashboard*	

According to the table, there are 14 sneak paths.

Tests

The tests are numbered in the same order as the leaf nodes in the tree
(left to right).

1. Start app ⇒ Create project ⇒ Edit name ⇒ Confirm name change
    - **Verification**: verify project exists and is called "test project".
2. Start app ⇒ Create project ⇒ Edit name ⇒ Cancel name change
    - **Verification**: verify project exists and is called "New project" (default name).
3. Start app ⇒ Create project ⇒ Create new project ⇒ Cancel name change
    - **Verification**: verify both projects exist.

All tests pass successfully.

Use case 2 - Delete a project

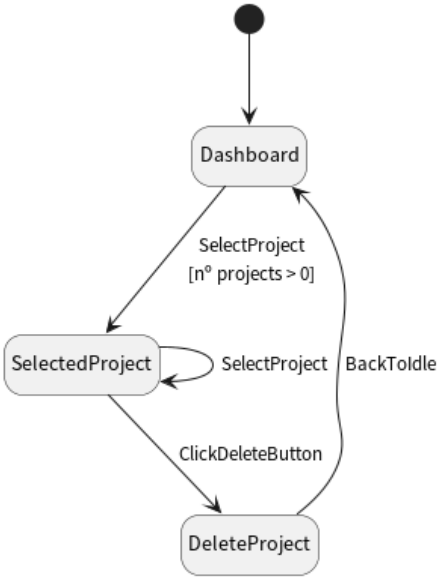
This is the main function of the application.

State diagram

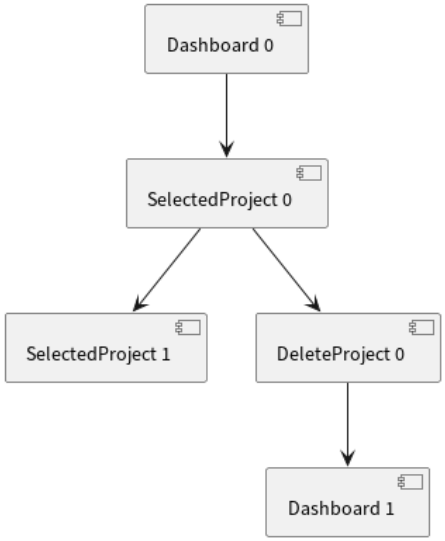
The most important aspects of this use case are:

- It is possible to delete any created project (even if it is running).
- The user can only shave one selected project at a time.
- While having one selected project, the only way to have no selected projects is to delete the current project.

We decided that the application's end-state is the moment the
application is closed. However, we chose to omit it since every
state would have a transition to it.



Transition tree



With 2 leaf nodes in the tree, we need to create 2 tests to cover all states and transitions of the system.

Transition table

	SelectProject [nº projects > 0]	**SelectProject**	**ClickDeleteButton**	**BackToIdle**
Dashboard	*SelectedProject*			
SelectedProject		*SelectedProject*	*DeleteProject*	
DeleteProject				*Dashboard*

According to the table, there are 8 sneak paths.

Tests

The tests are numbered in the same order as the leaf nodes in the tree (left to right). Each test implies some previous setup.

1. Start app ⇒ Select one project ⇒ Select another project
    - **Setup:** Create two projects.
    - **Verification:** Check if the correct project is selected.
2. Start app ⇒ Select project ⇒ Delete project
    - **Setup:** Create one project.
    - **Verification:** Check if the project no longer exists.

All tests pass successfully.

Use case 3 - Edit a project's color

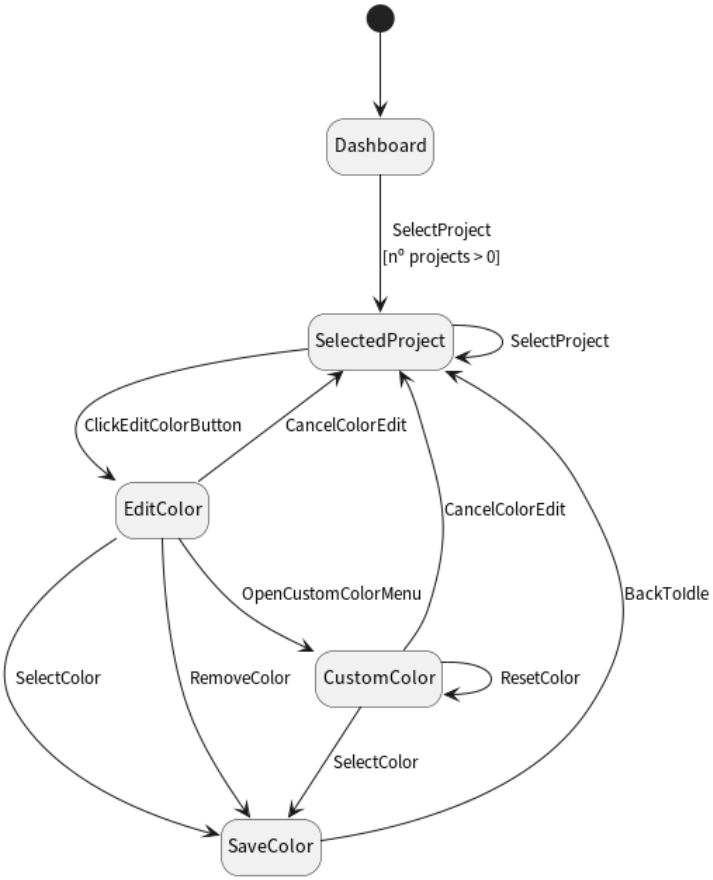
Color coding tasks is a common organization method for time scheduling/organization methods. A time scheduling application should have this feature.

State diagram

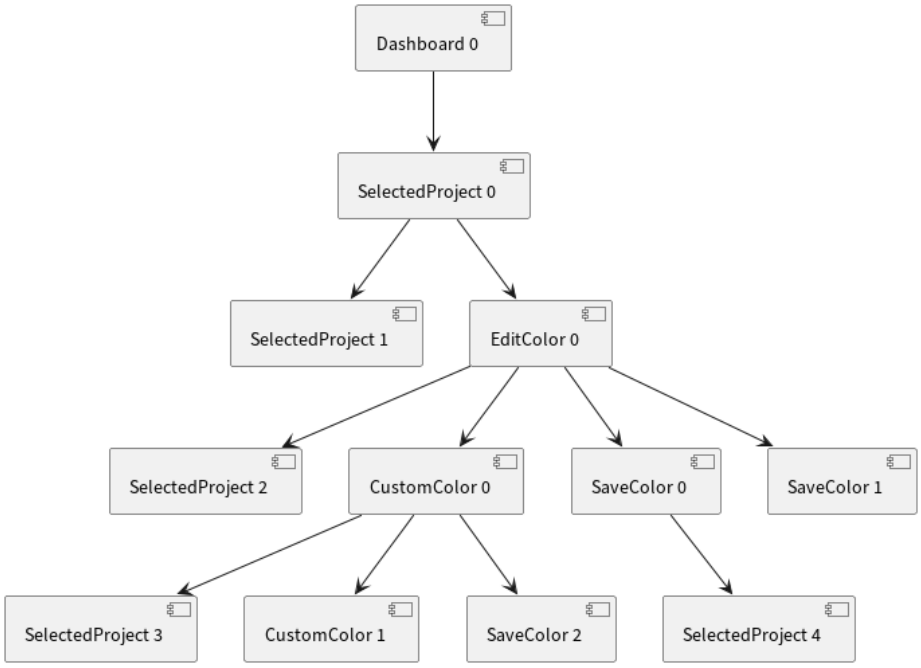
The most important aspects of this use case are:

- It is possible to choose a custom color instead of the predefined one.
- We omitted the states around the custom color selection menu: there are 5 tabs reachable from all other tabs. This leads to an *explosion of states* that doesn't add anything to the analysis.

We decided that the application's end-state is the moment the application is closed. However, we chose to omit it since every state would have a transition to it.



Transition tree



With 6 leaf nodes in the tree, we need to create 6 tests to cover all states and transitions of the system.

Note: The first test case is equal to the first test case of the second use case, so the test implementation will be the same.

Transition table

	SelectProject [n° projects > 0]	SelectProject	ClickEditColorButton	CancelColorEdit	SelectColor	RemoveColor	OpenCustomColorMenu	ResetColor	BackToIdle
Dashboard	*SelectedProject*								
SelectedProject		*SelectedProject*	*EditColor*						
EditColor				*SelectedProject*	*SaveColor*	*SaveColor*	*CustomColor*		
CustomColor				*SelectedProject*				*CustomColor*	
SaveColor									*SelectedProject*

According to the table, there are 35 sneak paths.

Tests

The tests are numbered in the same order as the leaf nodes in the tree (left to right). The setup of all tests involves adding a project to the list.

1. Start app ⇒ Select one project ⇒ Select another project
    - **Setup:** Create two projects.
    - **Verification:** Check if the correct project is selected.
2. Start app ⇒ Select one project ⇒ Edit color ⇒ Cancel color edit.
    - **Setup:** Create one projects.
    - **Verification:** Check if the project remains colorless.
3. Start app ⇒ Select one project ⇒ Edit color ⇒ Open custom color menu ⇒ Cancel color edit
    - **Setup:** Create one project.
    - **Verification:** Check if the project remains colorless.
4. Start app ⇒ Select one project ⇒ Edit color ⇒ Open custom color menu ⇒ Reset color selection
    - **Setup:** Create one project.
    - **Verification:** Check if the project is white (we set the color in order to test it).

5. Start app ⇒ Select one project ⇒ Edit color ⇒ Open custom color menu ⇒ Select custom color (pink) ⇒ Set custom color
    - **Setup:** Create one project.
    - **Verification:** Check if the project's color is pink.
6. Start app ⇒ Select one project ⇒ Edit color ⇒ Remove color
    - **Setup:** Create one project with the color green.
    - **Verification:** Check if the project became colorless.
7. Start app ⇒ Select one project ⇒ Edit color ⇒ Select color (green)
    - **Setup:** Create one project.
    - **Verification:** Check if the project became green.

All tests pass successfully.

Sneak paths

We decided to test one sneak path from each use case.

Use case 1's sneak path

Cancel a name change while on the dashboard. This corresponds to being in the *Dashboard* state and receiving a **CancelNameChange** event.

The test fails, because **QF-Test** isn't able to deliver the event.

Use case 2's sneak path

Delete a project while on the dashboard (without selecting a project). This corresponds to being in the *Dashboard* state and receiving a **ClickDeleteButton** event (using the keyboard's delete key instead of the UI button to avoid selecting the project).

Nothing happens in this test (the project isn't deleted).

Note: In order to un-select the project, we had to add and delete a second project. This is the only way we found to be on the dashboard without any project being selected.

Use case 3's sneak path

Remove a project's color while on the dashboard (the color edit menu is not open). This corresponds to being in the *Dashboard* state and receiving a **RemoveColor** event.

The test fails, because **QF-Test** isn't able to deliver the event (the modal/button isn't visible)4.

Notes about the tests

There are some problems about using QF-Test on the jTimeSched application.

Firstly, the location of the application's jar file is hard-coded as an absolute path (at least when using the quick-start wizard). This makes it hard to run the tests on different machines/systems. Although there might be a work-around for this, we haven't found it (even when looking at the docs).

Secondly, jTimeSched has a tray icon. This is problematic for 2 reasons: it is constantly producing exceptions on systems without a tray area, and it might start minimized to the tray. The first problem, makes it so most tests on QF-Test fail, because it detects the exception about the missing tray icon when the app tries to update (e.g., when deleting a project). The second problem makes it so QF-Test isn't able to run the tests without some manual setup for the first attempt. For example, on windows 10 (tested on 3 computers), the first time jTimeSched is opened, it is automatically minimized to tray. Tests aren't able to be run with the app in this state. To fix this, the user needs to maximize the app and then close it while it is maximized. Windows should *remember* the maximized state, and start it in this state the next times.

Lastly, the application's state is saved every minute or so. In order to always start with a *clean slate*, the user should use `/dev/null` (or similar) as the working directory otherwise, tests expecting an empty dashboard might fail.

QF-Test tool feedback

Systems used with the tool:

- System 1 - Desktop
  - OS: Arch Linux x86_64
  - Kernel: 6.0.6-zen1-1-zen
  - Default shell: fish 3.5.1
  - Window manager: dwm
  - CPU: intel i5-6600k (boost mode enabled)
  - GPU: XFX RX 57000 XT Thicc III
  - RAM: 16GB DDR4 2666 MHZ
- System 2 - Laptop - ThinkPad T460
  - OS: Arch Linux x86_64
  - Kernel: 6.0.6-zen1-1-zen
  - Default shell: fish 3.5.1
  - Window manager: sway
  - CPU: intel i5-6300U
  - GPU: intel integrated graphics
  - RAM: 8GB DDR4
- System 3 - Desktop
  - OS: Windows 10 Pro
  - OS Version: 21H2
  - CPU: AMD Ryzen 5 3600X
  - GPU: Nvidia GeForce RTX 2060
  - RAM: 16GB DDR4 2133Mhz

Overall, using the QF-Test tool was a positive experience. It was easy to learn how to start using the tool by making simple tests. The application has an intuitive interface, but in order to take the most advantage of the tool and to find out all its features, we had to read the documentation and watch some tutorials (specially considering the performance differences of using dependencies for setup/teardown, instead of opening/closing the app for each test).

The tool was really useful for Model-based testing, but there were some problems along the way that were mainly due to bugs. The application we were testing had some features that involved the icon tray menu. Recorded actions that involved interaction with the tray icon were not reliable since they would only work sometimes. This also extends to tests. We don't know why, but some

tests sporadically fail for no apparent reason.

Another difficulty that arose was the verification of visual clues. We wanted to test whether some item was selected or with a different color, but it took a bit of time to understand that the tool was not working as intended (clicking the image was checking for text instead of the actual image).

The main *pain-point* of the using QF-Test is the difficulty in sharing the test with a teammate. The jar file location is hard-coded as an absolute path and we haven't found a way to make this dynamic between systems/computers. As such, when coding in a different machine, we had to reconfigure these paths.

Overall, the experience was positive with the exception of the application's jar file path problem.