PizzaDronz Test-Planning Document

Matthew Davidson

January 27, 2023

Consider the development of the payment subsystem from the PizzaDronz pizza delivery system, in which there are several requirements, however arguably the most important one is that all the card details inputted by the user are correct. This was identified as the requirement with ID F2 in the requirements document (`requirements.pdf`), with the individual unit requirements having ID F7, F8 and F9. These individual unit requirements identify the need to correctly verify the expiry date, the card number and the CVV number respectively.

1 Priority and Pre-requisites

1.1 F2

F2 is a system level requirement, identifying the need to correctly implement the credit card detail checking functionality of the system. This is a crucial part of the system, as allowing incorrect payment details when ordering pizza could lead to a loss of revenue, and restaurants using our system to move to competitors. Furthermore, payment systems are heavily regulated, so it is of upmost importance that these regulations are complied with to prevent fines or sanctions. This suggests that we give this subsystem more of our time budget when testing, to ensure that we meet this requirement. To reduce the amount of time spent overall on testing this system, we will use the partition principle to split the requirement into further unit requirements (F7, F8 and F9).

To test this requirement we need to know the inputs and outputs for the subsystem:

1.1.1 Inputs

- `cardNumber` - String of integers containing a customer's card number
- `expiryDate` - String containing the expiry date of the customer's card in the `MM/YY` format
- `cvv` - String of integers containing the customer's card's cvv number

1.1.2 Outputs

- `cardValid` - True if the card details are valid

1.1.3 Specification

- `cardValid` is true only when `cardNumber`,`expiryDate` and `cvv` are all valid, otherwise false.

1.2 F7

F7 is a unit level requirement that the card validation subsystem shall correctly validate customer's credit card expiry dates. This is necessary to ensure that a card is not being used outwith it's lifespan and to ensure that payment can be collected from the customer. This falls under the same high priority of F2 as it is part of the payment system. Making the assumption that as long as the expiry date on the card is a month that happens after the current month, then it is valid ie. if the current month is 01/23 then a card with an expiry of 02/23 would be valid but one with an expiry of 01/23 would be invalid.

To validate this requirement we need to look at the inputs and outputs:

1.2.1 Inputs

- `expiryDate` - String containing the expiry date of the customer's card in the `MM/YY` format

1.2.2 Outputs

- `validExpiry` - True if the expiry date is valid

1.2.3 Specification

- `validExpiry` will be true if the expiry month is after the current one, otherwise, if it is the current month, before the current month or an invalid date it will be false.

1.3 F8

F8 is a unit level requirement that the card validation subsystem shall correctly validate card numbers. This is necessary to ensure the user has entered a card that payment can be taken from and that they haven't entered something that isn't even a card number. Using the restriction principle I have made this easier by only validating the length of the card number and ensured that the entire string consists of digits. Ideally the luhn algorithm would be implemented to ensure that the card number is valid, however due to strict time budget this wasn't possible.

To validate this requirement we need to look at the inputs and outputs:

1.3.1 Inputs

- `cardNumber` - String containing the customer's card number as a string of digits

1.3.2 Outputs

- `validCardNumber` - True if the card number is valid

1.3.3 Specification

- `validCardNumber` will be true if the card number is exactly 16 digits long and all of the characters are a number (0-9), otherwise it will be false

1.4 F9

F9 is a unit level requirement stating that the card validation subsystem shall correctly validate customer's CVV numbers. This completes the payment card processing subsystem. To validate the CVV number we check that it is a 3 or 4 character string consisting exclusively of digits 0-9.

1.4.1 Inputs

- `CVV` - String containing the customer's CVV as a string of digits

1.4.2 Outputs

- `validCVV` - True if the CVV is valid

1.4.3 Specification

- `validCVV` will be true if the CVV is exactly 3 or 4 digits long and all of the characters are a number (0-9), otherwise it will be false

2 Scaffolding and Instrumentation

For the requirements outlined above, F2 will require the most extensive scaffolding if we plan to do extensive combinatorial testing. Otherwise F7-F9 should be relatively straight forward to test, requiring a little bit of scaffolding for randomly generated outputs, as well as testing the boundary and incorrect input conditions.

- To test F2, we require F7, F8 and F9 to be implemented, as well as the logic combining these. Then we need to test different combinations of correct or incorrect CVV numbers, card numbers and expiry dates. The tests should only pass when all three conditions are valid.
- To test F7-F9 we require some scaffolding that will generate random valid and invalid inputs, as well as some cases that should definitely not work.

3 Process and Risk

Ideally the above activities would be carried out early in the process so that any errors in the implementation can be identified and squashed as quickly as possible. There are no real prerequisites for these tasks, only that a vague structure for the system has been identfied to allow testing. However, due to the tight schedule for the project and a lack of people power, the implementation activities will be carried out first, and the testing later.

This introduces a schedule risk, as if a large quantity of errors are found during the testing stage, then it may mean that a lot more time is spent fixing these errors than was spent performing the implementation activities in the first place. There is also the risk that due to the tight schedule, little time is spent thoroughly testing the software, and so the final product is found to have a number of severe bugs after delivery.

There is also a risk that the randomly generated input used to test the above requirements isn't representitive of what a real user would enter. This could mean that users encounter bugs that were missed due to inadequate test data during development. I think this is unlikely due to the nature of the functionality of what is being tested, however it is still a possibility.