

## Software Quality Assurance: SOFT3302

### Tutorial – Week 4

### Objectives

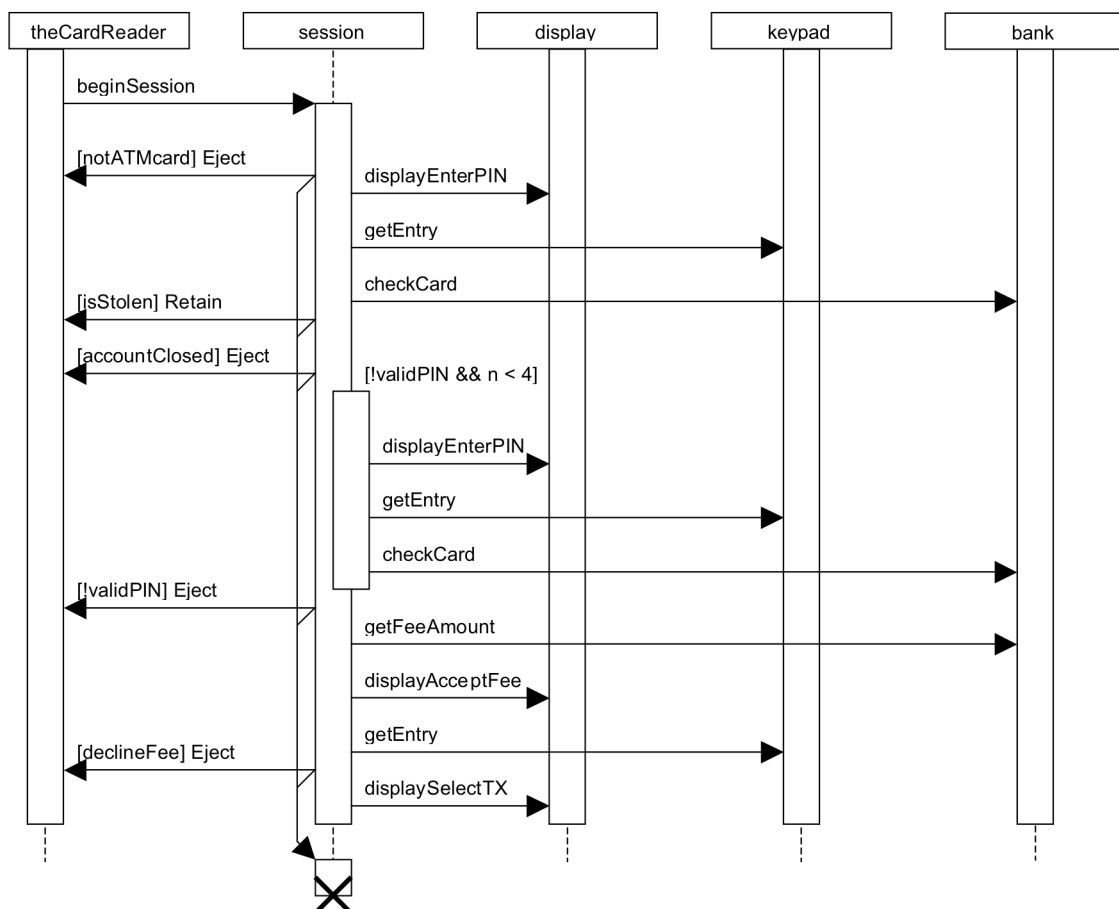
This tutorial gives experience using test design based on use-case scenarios. By the end of this tutorial you should have experience in developing test cases derived from use-case scenarios.

### Prework

Read the following derivation of a test-case from a particular use-case scenario path. (Taken from *Binder*, pp. 579ff.)

A *flow graph* is a highly-testable model of interactions. It may be derived from a sequence diagram and describe an actual sequence of messages passed or it may exist at a higher level and describe actor-system interaction in a use-case. A *scenario* is one path through the flow graph. A path set may be derived which provides branch and loop coverage of the graph.

The sequence diagram below shows what happens when a customer inserts a card into an hypothetical ATM. The range of options presented to the user allows for many difference scenarios each of which should be tested, but what are they? Paths traced on a flow graph derived from the sequence diagram correspond to test cases.



*Note that this diagram does not use UML 2.0 notation.*

## ***Fault Model***

Each scenario is a path through the sequence diagram/flow graph. Each object or subsystem interface that participates in the scenario must be physically correct and must provide a correct implementation of its responsibilities. In addition, the overall design to produce the response must be correct. The following bugs can occur in a scenario implementation.

- Incorrect or missing output
- Acceptance of incorrect selection or object
- Action missing on external interface
- Missing function/feature in a participant
- Correct message passed to wrong object
- Incorrect message passed to right object
- Message sent to destroyed object
- Correct exception raised but caught by the wrong object
- Incorrect exception raised to the right object
- Incorrect usage of target environment services
- Incorrect or ineffective memory allocation/deallocation resulting in an abnormal end or degraded performance
- Unbalanced or incorrect serialisation of server resources, *e.g.* a single client can lock a shared remote object for an arbitrarily long interval
- Unnecessary polling
- Task deadlock
- Priority inversion leading to process starvation
- Inadequate performance

If a bug exists it must lie on at least one round-trip path through the sequence diagram. To reach the bug the path must be taken. The test model covers all round-trip paths to ensure these bugs are reached. Once reached, gross interface bugs are more likely to be triggered than performance, exception or memory bugs.

## ***Test Strategy***

Sequence diagrams are not ideal for testing. They strike a good balance between too much and too little detail but do not provide a good representation of repetitive, recursive or conditional tasks. Tracing paths on a sequence diagram can produce a visual mess. A sequence diagram can be turned into a flow graph, which is more easily testable.

## ***Test Procedure***

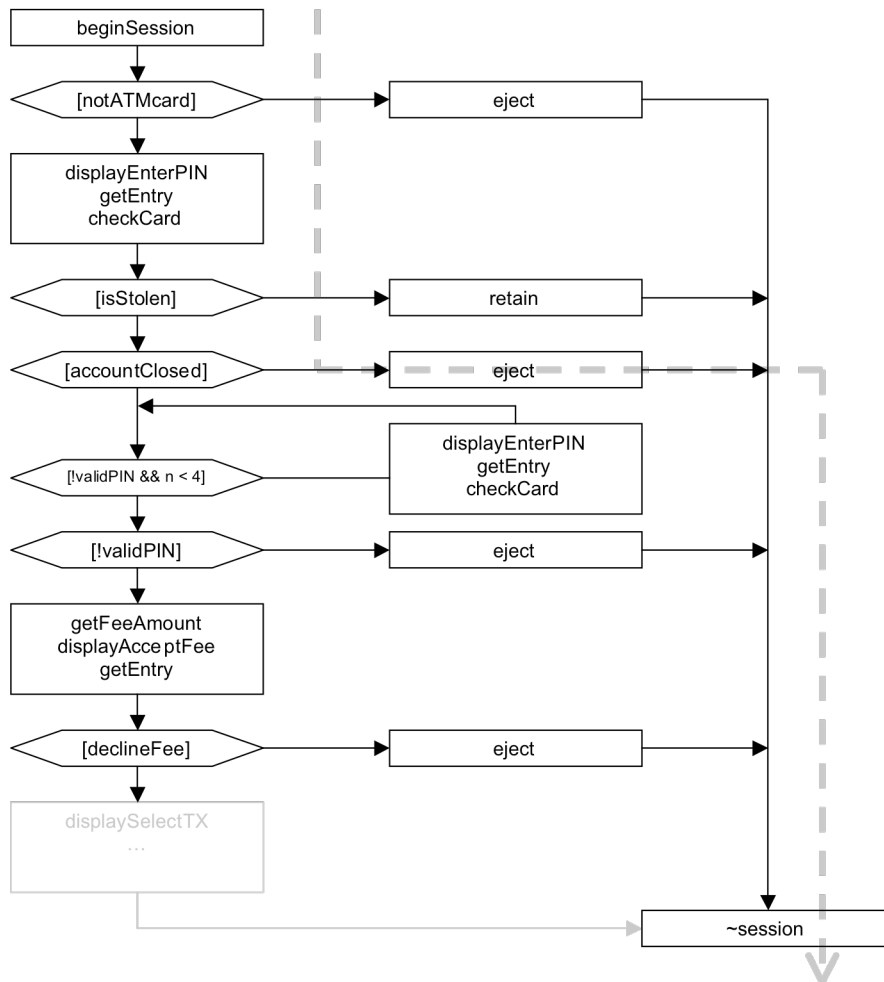
Translate the sequence diagram into a flow graph.

Identify paths in the flow graph (bypassing loops if possible in this step).

- a) Choose the longest or most complex entry/exit path and colour/trace this path.
- b) Choose a new colour.
- c) Trace a new entry/exit path that has not been coloured yet.
- d) Repeat b) and c) until all decision branches have been coloured.
- e) Make a table containing a row for each path. Copy the conditions that must hold for each path in the table. For example, suppose that the condition is that  $[a==b]$ . The *true* path condition is  $[a==b]$  and the *false* path condition is  $[a!=b]$ .

Two more tests are needed to meet iteration coverage: one iteration; and the maximum or high iterations.

- f) Pick any path that leads to a loop entry.
- g) Copy the path conditions up to the loop entry to the table. Revise the condition for the loop predicate to allow one iteration. If possible, append different exit paths after the loop exit.
- h) Pick another path that lead to the loop entry.
- i) Copy the path conditions up to the loop entry to the table. Revise the condition for the loop predicate to allow the maximum number (or a high number of) iterations. If possible, append different exit paths after the loop exit.



## Tasks

1. Derive a test case for the path through the use-case indicated by the dotted grey line.

**Suggest numbering the boxes with A1, A2, ..., An for action boxes and C1, C2, ..., Cn for conditional boxes. The above path then becomes: [A1, C1, A3, C2, C3, A5, A11].**

2. Write down the path(s) that attempt(s) to effect a transaction with a stolen card.

**This path should not be possible but an attempt to traverse it should form part of a negative test. If it succeeds then it is an error.**

3. Consider the following constraints on the system: ‘the card is an ATM card’ and ‘the information on the card is readable’. Develop test cases that attempt to break these constraints.

Clearly neither of these is covered by the above diagram. Once test case might be to use another type of card or even to use something that isn't a card such as a piece of cardboard.

It's a useful reminder that what's being developed here is a methodology for testing not necessarily just a methodology for testing *software* although that's how it is applied here. The 'cardboard test' would form a necessary part of testing an actual ATM machine.

## Labwork

1. Develop a use-case/sequence diagram/flow graph for the transaction **Withdraw Notes**, which might be a typical ATM use case. Feel free to use the sequence diagram above as a fragment of the new sequence diagram.

In UML2 sequence diagram fragments are boxes containing the name of the sequence diagram referred to and the label 'sd' in the top-left corner. The number of lifelines/message entering and leaving the box must correspond to the number of lifelines/messages at the boundaries of the subsequence. Fragments may be parameterised.

Develop the use-case/sequence diagram with the class in the style of the above sequence diagram. Include error cases. Let students develop the flow graphs themselves.

2. Derive test cases to exercise this use-case.

All paths through the use-case should have a corresponding test case. This includes any loops, *e.g.* where an action may occur repeatedly. Such loops should appear in the flow diagram for the use-case and thus a test case should 'fall out' naturally when developing test cases based on use-case paths.

Some typical paths might be: 'account selected and money amount entered without error', 'account selected but money amount entered and then corrected', 'account selected and money amount entered but no notes of suitable denomination available.' All of these cases must be covered by the diagrams developed in (1).

3. Discuss how one might run these tests and record the results.

*Any tests on a live ATM system are undertaken at a student's own risk and should be limited to passive queries such as balance transactions.*

Seriously! Students may like to actually look at the interface of an ATM and develop real test cases and find a moment to try some of them out. (Warn students that should do this responsibly and carefully and not inconvenience other users and no responsibility will be taken if the ATM retains their card. Student do such testing on their own recognisance and at their own risk. Remind them that they will be working on a live system and so a test to withdraw \$1000 may actually withdraw that amount! Suggest they limit testing to passive operations such as balance enquiries. Such unwarranted testing on systems such as web sites may be construed as fraud or a denial of service attack.)

The important things to stress here are consistency (all cases and test results should be documented uniformly) and impartiality (there is no discretion on the part of the tester used in running the tests). All tests should be run and the tester cannot assume that because one test passes another test will and may therefore be skipped.