VL Software Testing (188.280)

Exercise block 3 (30 Points)

Assignment 01 - Model-based Testing RingBuffer (8 Points)

Motivation and Goal

Designing and writing test cases for state-based systems is usually a manual process: The state model is transformed into a transition tree to systematically derive test cases (see exercise block 1, assignment 03). In *model-based testing*, these steps are *automated* and test cases can be generated directly from state models.

In this exercise, you should achieve following goals:

- *Develop a test model* for the RingBuffer in form of a Java class implementing FsmModel from ModelJUnit. Use your (updated) state diagram from exercise block 1, assignment 03 as basis.
- Run the model to *generate test cases* that cover all states and transitions
- *Add an adapter to the model* to call the methods of the class RingBuffer in testing
- Run the extended model to *cover the RingBuffer implementation*

Requirements and Setup

- ModelJUnit: https://sourceforge.net/projects/modeljunit/ (further details: *M. Utting and B. Legeard. Practical model-based testing: a tools approach. Elsevier, 2010*)
- Download and extract the source code package *exercise03.zip* from TUWeL. The folder related to this assignment is *01-RingBufferModelBasedTest*. Use this folder to store all files you produced in this assignment.

Instructions

- Starting point: The assignment folder contains an empty model class *RingBufferModel.java* and a JUnit test case to run the model via *mvn test*
- Update (if necessary) your state diagram from exercise block 1, assignment 03 according to the feedback comments you received on your submission. Put the figure into the directory \docs in the assignment folder
  - You only need to consider scenarios where the capacity of the RingBuffer is set to three or more elements, i.e. RingBuffer(>=3)
  - You can use a tool or draw the state diagram by hand using pen and paper and submit a photo. Make sure the drawing is well structured and clearly readable!
- Develop a test model matching your state diagram for the class *RingBuffer* using the framework *ModelJUnit* by implementing the missing parts in *RingBufferModel*
  - The model should provide actions for testing *enqueue(), dequeue()* and *peek()*
  - The model should reflect the states *empty*, *filled* and *full* of the *RingBuffer*

    Note: Use counter variables to keep track of how many elements are in the buffer

    and to determine the current state of the buffer in the function *getState()*. The model is an abstraction, so avoid using another data structure to represent the buffer and do not actually store data elements in the model.
  - Add *guard functions* to the model to make sure that the actions *dequeue* and *peek* are only called if the *RingBuffer* is in a state where this action is valid (e.g., *filled* or *full*)
  - Add additional actions (e.g., *dequeueFromEmptyBuffer*) and matching guard functions to represent the case of an empty buffer
- Run the model-based test and check the output produced by ModelJUnit in the file *\target\surefire-reports\at.tuwien.swtesting.RingBufferModelTest-output.txt*.
  - **Make sure that all states and transitions are covered**; if necessary, increase the length of the generated sequences set via *tester.generate(length)* in the test
- Create a new model *RingBufferModelWithAdapter* (e.g. by copying the existing model class) that contains calls to the methods of the *RingBuffer* implementation
  - Extend the action methods of the test model with calls to the corresponding methods of the class *RingBuffer*
  - Add checks to the action methods that compare the response (i.e., returned values, thrown exceptions) to the expected response using JUnit assert methods

    Note: Use the model's state variables (counters) to define the expected response

    In case of expected exceptions, make sure that (1) exceptions are actually thrown and (2) they are of the correct type and contain the correct message.
  - Add further actions for the methods *capacity()*, *size()*, *isEmpty()* and *isFull()*
- Write another JUnit test *testModelWithAdapter()* to *RingBufferModelTest* to exercise *RingBufferModelWithAdapter*
- Run the new test and measure the coverage of the *RingBuffer* implementation (see the generated coverage report at *\target\site\jacoco\index.html*)
  - Make sure that the class *RingBuffer* (not including *Iterator*) is fully covered

Assignment 02 - Model-based Testing RingBuffer 2 (6 Points)

Motivation and Goal

In exercise block 1, assignment 02 you have used equivalence partitioning on the parameter *capacity* of the constructor *RingBuffer(int capacity)*. From the perspective of modeling the RingBuffer's states and transitions, each valid partition of the parameter *capacity* leads to a slightly different model variant. The goal of this exercise is:

- *Understand how to work with a* ***partial test model*** representing only the scenarios where the capacity of the RingBuffer is set to one element, i.e. *RingBuffer(1)*
- Thus, instead of creating one big model that is able to handle all capacity values (i.e., all equivalence partitions) you should implement "partial" models, each covering the state transitions for a different specific equivalence partition.

Requirements and Setup

- ModelJUnit: https://sourceforge.net/projects/modeljunit/ (further details: *M. Utting and B. Legeard. Practical model-based testing: a tools approach. Elsevier, 2010*)
- Download and extract the source code package *exercise03.zip* from TUWeL. The folder related to this assignment is *02-RingbufferModelBasedTest2*. Use this folder to store all files you produced in this assignment.

Instructions

- Starting point: The assignment folder contains an empty model class *RingBufferModel.java* and a JUnit test case to run the model via *mvn test*
- *Adapt your state diagram* from the previous assignment 01 (above) so that it corresponds to the scenarios where the capacity of the RingBuffer is set to one element only, i.e. RingBuffer(1)
  - Put the figure into the directory *\docs* in the assignment folder
  - You can use a tool or draw the state diagram by hand using pen and paper and submit a photo. Make sure the drawing is well structured and clearly readable!
- *Develop a test model* (i.e., an additional partial model) matching your state diagram for the class *RingBuffer* with *capacity=1* using the framework *ModelJUnit* by implementing the missing parts in *RingBufferModel*
  - The model should implement actions and guards for *enqueue*, *dequeue* and *peek* as well as the representation for the possible states of the *RingBuffer*
- *Run the model-based test* and make sure that all states and transitions are covered; if necessary, increase the length in *tester.generate(length)*
- *Extend the model with an adapter* that contains the calls to the corresponding methods of the *RingBuffer* implementation and add checks that compare the responses of these calls (i.e., returned values, thrown exceptions)
  - Add further actions for the methods *capacity()*, *size()*, *isEmpty()* and *isFull()*
  - Write another JUnit test *testModelWithAdapter()* to *RingBufferModelTest* to extend the new *RingBufferModelWithAdapter*
- Run the extended test and measure the coverage of the *RingBuffer* implementation (see the generated coverage report at *\target\site\jacoco\index.html*)

Assignment 03 - Maintaining Model-based Tests (6 Points)

Motivation and Goal

In model-based testing, updating or extending the tests is simple. You only have to update/extend the test model and re-run test generation.

In this exercise, you should achieve following goal:

- *Adjust and extend the existing test model* to match the changes in the system

Requirements and Setup

- ModelJUnit: https://sourceforge.net/projects/modeljunit/ (further details: *M. Utting and B. Legeard. Practical model-based testing: a tools approach. Elsevier, 2010*)
- Download and extract the source code package *exercise03.zip* from TUWeL. The folder related to this assignment is *02-RingbufferModelBasedTest2*. Use this folder to store all files you produced in this assignment.

Instructions

- Starting point: The assignment folder contains the class *RingBuffer* with the additional method *setCapacity(int newCapacity)*

  ```
  /**
   * Updates the size of the buffer while preserving FIFO ordering of elements.
   * @param newCapacity of the updated buffer.
   * @throws RuntimeException if the new capacity is smaller than the current size.
   */
  public void setCapacity(int newCapacity)
  ```

  Copy your test and model from assignment 01 (above) to the folder *\src\test* and run it on the extended RingBuffer via *mvn test.*

  - Note: Check that the test from assignment 01 above still runs on the extended RingBuffer without any problems before you continue.
- *Update your state diagram* from the previous assignment 01 (above) so that it now corresponds to the extended RingBuffer implementation with the ability to dynamically change the capacity via the new method *setCapacity()*
  - Put the figure into the directory *\docs* in the assignment folder
  - You can use a tool or draw the state diagram by hand using pen and paper and submit a photo. Make sure the drawing is well structured and clearly readable!
- *Adjust your test model* (copied from assignment 01) to match the updated state diagram by including an action and guard method for *setCapacity()*
- *Run the model-based test* and make sure that all (old and new) states and transitions are covered; if necessary, increase the length in *tester.generate(length)*
- *Adjust the model with the adapter* to also call the new method *setCapacity()*
- Re-run the extended test, make sure it passes successfully, and measure the coverage of the *RingBuffer* implemenation (see the generated coverage report at *\target\site\jacoco\index.html*)
  - Make sure that the extended *RingBuffer* (not including *Iterator*) is fully covered
  - Include the output files from *\target\surefire-reports\...* in your submission

Assignment 04 - Model-based Testing for Bugzilla (8 Points)

Motivation and Goal

In the previous exercise, you have learned how to create a model for model-based testing with ModelJUnit. You will now explore its application for model-based testing of a larger system via the graphical user interface.

In this exercise, you should achieve following goals:

- *Develop a test model* using ModelJUnit *for Bugzilla's status workflow*, for which you already have implemented a test in exercise block 2 (*ChangeStatusTest*)
- *Reuse the page objects* you created in exercise block 2 as adapter to access Bugzilla
- Run the model to *generate test cases that cover all states and transitions of Bugzilla's bug status workflow*

Requirements and Setup

- ModelJUnit: https://sourceforge.net/projects/modeljunit/ (see above)
- Selenium WebDriver and Page Object Design Pattern (see *Exercise block 02*)
- Virtual Bugzilla Appliance 5.0 (see *Exercise block 02*)
- The assignment folder *04-BugzillaModelBasedTest* contains sample test code to be extended and a *pom.xml* for building the project and running the tests

Instructions

- Starting point: Run *mvn test.* Make sure the build is successful before you continue. Output see: *\target\surefire-reports\...*
- First, develop a **test model for Bugzilla's status workflow** by extending the provided model class *StatusModelWithAdapter*
  - Represent Bugzilla's bug status as model state (using an enumeration)
  - Add action methods for each status transition, e.g., *changeToNew(),* and corresponding guard methods

    Note: The action method should call the adapter to perform the actual change of the status on the Bugzilla Web page. In the first step, leave the *BugzillaAdapter* implementation empty, so you can run and validate the model without connecting to Bugzilla during model development.
  - Run the model and make sure all states and transitions of the Bugzilla status workflow are covered; adjust the length of the generated sequences via setting *tester.generate(length)* in the test if necessary
- Second, adjust the provided **JUnit test *ChangeStatusModelTest***
  - Edit the test to point *BASE_URL* to your Bugzilla server
  - Adjust the setup/teardown methods (i.e., see the existing code) to open a new Web browser instance and perform a login and logout to Bugzilla
  - Create an instance of *BugzillaAdapter* and configure it for testing

  - Run the test and make sure that login/logout work correctly before you proceed
- Finally, implement the class ***BugzillaAdapter*** **interacting with Bugilla**
  - The responsibility of the adapter is to provide all methods necessary for interacting with the Web pages of the Bugzilla application. The model should not need to know any technical details about how to access Bugzilla.
  - The adapter uses *page objects* to access Bugzilla
  - Make sure that the adapter is instantiated and configured in the JUnit test
  - The adapter should provide a method to create a new bug entry for each new test sequence (i.e., whenever *reset(true)* is called in the model)
  - The adapter methods for changing the bug status should return true if the status change was successful
- Run the model-based test on Bugzilla and make sure it passes successfully. Check that all states and transitions of the Bugzilla status workflow are covered.
  - Include the output files from *\target\surefire-reports\...* in your submission

Assignment 05 - Model-based Testing in Practice (2 points)

Motivation and Goal

Reflect on the application of model-based testing in practice by reading the book chapter *Model-Based Test-Case Generation In ESA Projects*. It describes the model-based testing approach presented in the guest lecture by Stefan Mohacsi, ATOS.

Reference: Stefan Mohacsi, Armin Beer: Model-Based Test-Case Generation In ESA Projects, In: D. Graham and M. Fewster (Eds.) *Experiences of Test Automation: Case Studies of Software Test Automation*, Pearson Education, 2012

Instructions

- Read the book chapter carefully and answer following questions
  1. How does the described model-based testing approach overcome the typical limitations of (static) automated regression testing? Answer this question by briefly referring to the described case.
  2. What is the difference between online and offline model-based testing? Which of these two approaches has been used in the case study?
  3. What test strategies are compared in calculating the ROI? When has the break-even point been reached? What are potential problems in this calculation?
- Document each of your answers in a short paragraph in the text file *answers.txt* in the folder *05-Reading&Discussion*

Submission

- Please add your name, student-id, and exercise number in a header block at the beginning of every (source code) file.
- Pack the entire repository as zip archive and upload the archive via TUWeL until the submission date (before 23:55).

Keep in mind that incomplete submissions, submissions lacking information required for appraising, and submissions handed in after the deadline or per email will not be accepted.