PA1417 Basic System Verification - Lab Assignments

Team Information

- Team Member 1: Erik O.
- Team Member 2: Karl W.

Work Distribution

For Assignment 1:

- Team Member 1: Made first draft of full assignment.
- Team Member 2: Completed and complemented it.

For assignment 2:

- Team Member 1: Made full assignment.
- Team Member 2: Reviewed it and made minor changes.

For assignment 3:

- Team Member 1: Wrote answers to text questions.
- Team Member 2: Started to design test cases.
- Team Member 1: Mocked the validator and wrote initial test cases.
- Team Member 2: Completed test cases, refactored slightly and completed written assignment.

Assignment 1: The Test Design Technique

1. The 4-Step Test Design Technique

This structured approach helps create comprehensive test suites to verify system behavior. Here's how to implement it:

Step 1: Identify actions and expected outcomes

- **Objective**: Identify all possible interactions with the system under test
- **Process**:
  - List user actions (clicks, inputs, etc.)
  - List possible expected outcomes
  - Note system triggers (API calls, scheduled jobs)

- **Example** (for login feature):
  - Action: User submits credentials
  - Expected: Successful/unsuccessful authentication

Step 2: Identify Conditions

- **Objective**: Determine factors affecting outcomes
- **Process**:
  - Identify all conditions (parameters, system states) that will affect the outcome of the action.
- **Techniques**:
  The following techniques are usefule for deciding what input values are relevant for the test case:
  - **Boundary Value Analysis**: Test edge cases (value at boundaries, +/- 1)
  - **Equivalence Partitioning**: Define conditions based on expected outcomes
- **Example** (for age validation):
  - Boundaries: -1, 0, 17, 18, 120, 121
  - Partitions: Invalid (<0), Underage (0-17), Valid (18-120)

Step 3: Determine Combinations

- **Objective**: Create test scenarios
- **Process**:
  - Create all combinations or filter for relevant combinations based on domain knowledge.
- **Example** (for log in authentication):
  1. **Valid password + Invalid username**
  2. **Invalid password + Valid username**
  3. **Invalid password + Invalid username**
  4. **Valid password + Valid username**

Step 4: Define Expected Outcomes

- **Objective**: Clearly define the expected behavior of the system.
- **Process**:
  - For each combination, assign the expected outcome.
  - Combinations can be collapsed if various values of the condition do not impact the expected outcome.
- **Example** (for the combinations in Step 3 example):
  1. **Failure**: Ensures the system correctly denies access when the username is incorrect, even if the password is valid.

2. **Failure**: Checks that the system rejects login attempts when the password is incorrect, even if the username is correct.
3. **Failure**: Verifies that the system does not authenticate users when both credentials are incorrect.
4. **Successful login**: Verifies that the system does authenticate users when both credentials are correct.

Why This Method Works

1. **Covers Important Cases**
   Helps make sure all the key parts of the system are tested.
2. **Saves Time and Effort**
   Focuses on the most useful tests, so you don't waste time on unimportant ones.
3. **Easy to Understand and Track**
   Makes it clear what's being tested and why.
4. **Can Be Reused**
   Works well for testing new features and checking that old ones still work.

2. Boundary Value Analysis and Equivalence Partitioning

1. Explanation of Boundary Value Analysis (BVA) and Equivalence Partitioning (EP)

Boundary Value Analysis (BVA):

- A testing technique where test cases are designed around the **edges** of input ranges.
- Tests values **at, below, and above** boundaries to catch common errors (e.g., off-by-one).
- **Example**: For a system accepting inputs between 1 and 10, test values like 0, 1, 10, and 11.

Equivalence Partitioning (EP):

- Divides input data into **groups (partitions)** where all values in a partition behave similarly.
- Tests one representative value per partition to minimize redundancy.
- **Example**: For age validity (0–120), partitions could be:
  - Invalid (age < 0 or > 120),
  - Underage (0 ≤ age < 18),
  - Valid (18 ≤ age ≤ 120).
  - Test values is then selected in each of these categories, e.g. -5, 5, 20 and 130.

2. Comparison of Usability

Criteria	BVA	EP
Focus	Edge cases and boundaries	Grouping similar inputs
Strengths	Detects boundary-related errors (e.g., off-by-one)	Reduces redundant test cases
Weaknesses	Requires precise boundary definitions	May miss edge-case defects
Best Used	For numerical ranges and discrete values	For categorizing inputs into valid/invalid groups
Combined Use	EP defines partitions; BVA tests their edges	Ensures efficiency and thoroughness

---

3. Application to the Age Validity Scenario

Equivalence Partitions (EP):

1. **Impossible**:
   - Age < 0 (e.g., -5)
   - Age > 120 (e.g., 130)
2. **Underage**: 0 ≤ age ≤ 17 (e.g., 10)
3. **Valid**: 18 ≤ age ≤ 120 (e.g., 25)

Boundary Values (BVA):

- **Boundary values**: 0, 120, 18
- **Boundary values analysis**: -1,0,1, 119, 120, 121, 17, 18, 19

Test Cases:

Value	Expected Result	
-1	Impossible	
0	Underage	
1	Underage	
17	Underage	
18	Valid	
19	Valid	

Value	Expected Result	
119	Valid	
120	Valid	
121	Impossible	

Conclusion: Combining BVA and EP ensures coverage of edge cases and minimizes redundant testing.

3. Designing Test Cases

This answer will follow the design test technique to identify all relevant test cases for the scenario:

- Step 1: Identify actions and expected outcomes
- Step 2: Identify Conditions
- Step 3: Determine Combinations
- Step 4: Define Expected Outcomes

A *test case* is a precise description of a single test. At minimum it contains:

- ID
- Action: an activity of the system under test that we evalaute
- Inputs: the list of conditions that represent the situation
- Expected outcome: the behavior the system is expected to exhibit given the inputs

The scenario

To open the door at the entrance of a company building from the outside, one must either:

- Hold a valid company card to a sensor for at least two seconds, or
- Have the door automatically unlocked by the porter
- The door can always be opened from the inside

Step 1: Identification of Conditions and Actions

1. **Action(s)** and **expected outcome**:
    - door *can be* [opened, not opened]

Actions:

- 1. Opening the door with a valid card.
- 2. Opening the door with an invalid card.
- 3. Opening the door from the inside.

- 4. Porter opens the door.

Outcomes:

- Door opens; door remains closed

Step 2: Identify Conditions:

2. **Condition(s)**:
   - card *can be* [valid; invalid; not present]
   - duration at sensor *can be* [less than two seconds, two seconds or more]
   - door opener location *can be* [inside; outside]
   - porter action *can be* [opened, not openend]

Step 3-4: Determine Combinations and Expected Outcomes

Test cases are collapsed below because certain parameters doesn't matter for the door, given some circumstances (e.g. if card is invalid, the time at sensor is not relevant).

#	Card	Duration at sensor	Door opener location	Porter action	Door
1	valid	two seconds or more	outside	not opened	opened
2	valid	less than two seconds	outside	not opened	not opened
3	invalid	-	outside	not opened	not opened
4	not present	-	outside	not opened	not opened
5	-	-	outside	opened	opened
6	-	-	inside	-	opened

Assignment 2: Unit Testing

1. Mocking

Explanation of Mocking

Mocking is a technique used in software testing for simulating dependencies like method calls, external services, API-calls. In a unit test we want to test an isolated part of the code.

When a test relies on dependencies that are unavailable or unpredictabe like an API-call for today's weather, mocking saves the day by replacing these dependencies with the variables of our choice. This allows tests to focus exclusively on the logic we are trying to test.

Purpose of Mocking in Unit Testing

The main purpose of Mocking is to **isolate the code under test** by removing external dependencies. There are also other benefits that come with it. Here are a few:

Speed & Efficiency & Cost

- **Skip slow tasks** (like waiting for network calls) to make tests run faster.
  Example: Avoid real API calls or database operations that take a long time to complete.

Consistency

- **Prevent random failures** caused by things you can't control (like a broken internet connection or changing data).
  Example: Use fake data so your test works the same way every time.

Verification

- **Check if your code talks to other parts correctly**.
  Example: Confirm if a method was called with the right inputs, like checking if `sendEmail("user@test.com")` happened.

Control & Predictability

- Lets you **create fake scenarios** (like errors or special cases) to see how your code reacts.
  Example: Make a fake API return an error code (like "500") to test your error-handling code.

Dependency not yet implemented

- Lets you **test dependencies not yet created** to test specific unit without the dependency implemented.
  Example: Make an API call to an API not yet implemented.

Impure / non-deterministic dependencies

- Lets you **test dependencies that are non-deterministic** in a consistent way.
  Example: A dice roll function is random and we might want to mock it so that we get deterministic values.

2. Unit Testing for User Controller

Test Cases for get_user_by_email Function

Email	Format	Exists	Outcome
Provided	Correct	Yes, once	User returned
Provided	Correct	Yes, more	User returned
Provided	Correct	No	None returned
Provided	Incorrect	-	ValueError raised
Not provided	-	-	ValueError raised

Implementation

Link to test file in repository: https://github.com/eckepecke/bsv-edutask/blob/master/backend/test/test_usercontroller.py

Test Execution Output

```
================================== FAILURES ===================================
_____________________ test_get_user_by_email_not_found _____________________

mock_user_controller = <src.controllers.usercontroller.UserController object at 0x1034cd1d0>

    def test_get_user_by_email_not_found(mock_user_controller):
        # Assert that None is returned when incorrect email format is passed
        mock_user_controller.dao.find.return_value = []
>       result = mock_user_controller.get_user_by_email('error@example.com')

test/test_usercontroller.py:42:
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

self = <src.controllers.usercontroller.UserController object at 0x1034cd1d0>
email = 'error@example.com'

    def get_user_by_email(self, email: str):
        """Given a valid email address of an existing account, return the user object contained in the database associated
        to that user. For now, do not assume that the email attribute is unique. Additionally print a warning message containing the email
        address if the search returns multiple users.

        parameters:
            email -- an email address string

        returns:
            user -- the user object associated to that email address (if multiple users are associated to that email: return the first one)
            None -- if no user is associated to that email address

        raises:
            ValueError -- in case the email parameter is not valid (i.e., conforming <local-part>@<domain>.<host>)
            Exception -- in case any database operation fails
        """

        if not re.fullmatch(emailValidator, email):
            raise ValueError('Error: invalid email address')

        try:
            users = self.dao.find({'email': email})
            if len(users) == 1:
                return users[0]
            else:
                print(f'Error: more than one user found with mail {email}')
>               return users[0]
E               IndexError: list index out of range

src/controllers/usercontroller.py:37: IndexError
---------------------------- Captured stdout call -----------------------------
Error: more than one user found with mail error@example.com

---------- coverage: platform darwin, python 3.11.4-final-0 ----------
Name                                Stmts   Miss  Cover   Missing
-----------------------------------------------------------------
src/controllers/__init__.py             0      0   100%
src/controllers/controller.py          31     22    29%   24-27, 44-47, 59-62, 80-84, 99-103
src/controllers/taskcontroller.py      68     68     0%   1-139
src/controllers/todocontroller.py      21     21     0%   1-40
src/controllers/usercontroller.py      24      5    79%   42-46
src/util/dao.py                        67     51    24%   25-40, 54-65, 79-83, 101-118, 134-141, 156-162, 170-173, 184
src/util/validators.py                  7      4    43%   13-16
-----------------------------------------------------------------
TOTAL                                 218    171    22%
```

The implementation of the get_user_by_email is flawed. Docstring claims it will return None when user does not exist. Therefore the failure occurs during the tests.

Test Coverage Interpretation

The test coverage output shows that 79% of the lines in the UserController class, have been executed at least once. The lines 42-46 have been marked as missing, there is no test validating this part of the code. These lines refer to the update method.

Assignment 3: Integration Testing

1. Test Levels

Difference in Scope Between Unit and Integration Tests

Unit testing verifies that a single component (e.g., a class, method, or function) behaves as expected in isolation, often by mocking external dependencies. Integration testing checks if multiple components or systems work together correctly, validating interactions, data flow, and dependencies (e.g., APIs, databases).

Different Purposes of Mocking in Unit vs. Integration Tests

Mocking in integration testing, is used to isolate external systems or control the environment (like simulating a third-party API or a database), not the code units themselves. The goal in integration testing is to test how multiple components work together, but still may avoid hitting real external systems to improve test reliability and speed.

2. Integration Testing for DAO and MongoDB

List of Test Cases

Using the template for Test Design Technique

The template to arrive at test cases was used, see Appendix. The docstring for the function was used to go through the template.

Step 1: Identify Actions and Expected Outcomes

1. **Action(s)** and **expected outcome**:
    - create object *can* [return with correct data, return with incorrect data, raise WriteError, raise TypeError]

Actions:

- **Objective**: Clearly define the actions and their expected results.
- **Actions**:

- 1. Creating object with (1) the data* for the new object contains all required properties, (2) every property complies to the bson data type constraint, (3) and the values of a property flagged with 'uniqueItems' are unique among all documents of the collection.
- 2. ... (1) the data* for the new object **doesn't** contain all required properties, (2) every property complies to the bson data type constraint, (3) and the values of a property flagged with 'uniqueItems' are unique among all documents of the collection.
- 3. ... (1) the data* for the new object contains all required properties, (2) **not** very property complies to the bson data type constraint, (3) and the values of a property flagged with 'uniqueItems' are unique among all documents of the collection.
- 4. ... (1) the data* for the new object contains all required properties, (2) every property complies to the bson data type constraint, (3) and the values of a property flagged with 'uniqueItems' are **not** unique among all documents of the collection.
- 5. ... (1) the data* for the new object contains all required properties, (2) every property complies to the bson data type constraint, (3) and **there are no** values of a property flagged with 'uniqueItems'.
- 6. ... (1) **non-compliant data type**, (2) every property complies to the bson data type constraint, (3) and the values of a property flagged with 'uniqueItems' are unique among all documents of the collection.

What about an empty data param?

**data -- a dict containing key-value pairs compliant to the validator*

- **Expected Outcomes**:
  - Object returned with the newly created MongoDB document (parsed to a JSON object) containing the input data and an _id attribute.
  - Raises WriteError
  - Raises TypeError (?) (on non-compliant data type?)

Step 2: Identify Conditions

- **Objective**: List all conditions that influence the outcomes.
- **Conditions**:
  - Complete data set : can be complete, or incomplete
  - Complies with type constraints : compliant or not
  - Unique constraints:
    - uniqueItems flagged: unique or not unique
    - unqueItems not flagged: independent
  - data type is wrong
- **Techniques**:
  - **Boundary Value Analysis**: [Describe relevant boundaries]
  - **Equivalence Partitioning**: [Describe partitions]

Step 3: Determine Combinations

- **Objective**: Combine conditions to create test scenarios.
- **Combinations**:

#	Data Set	Type Constraint	Unique Constraint	Arg Type	Expected Outcome
1	Complete	Compliant	Unique	`dict`	Document created
2	Incomplete	Compliant	Unique	`dict`	`WriteError` raised
3	Complete	Non-compliant	Unique	`dict`	`WriteError` raised
4	Complete	Compliant	Non-unique	`dict`	`WriteError` raised
5	Complete	Compliant	Unique	Non-`dict`	`TypeError` raised
6	Incomplete	Non-compliant	Unique	`dict`	`WriteError` raised
7	Complete	Non-compliant	Non-unique	`dict`	`WriteError` raised
8	Incomplete	Non-compliant	Non-unique	`dict`	`WriteError` raised
9	Complete	Non-compliant	Unique	Non-`dict`	`TypeError` raised
10	Complete	Non-compliant	Non-unique	Non-`dict`	`TypeError` raised
11	Incomplete	Non-compliant	Unique	Non-`dict`	`TypeError` raised
12	Incomplete	Non-compliant	Non-unique	Non-`dict`	`TypeError` raised

Step 4: Define Expected Outcomes

- **Objective**: Assign expected results to each combination.
- **Expected Outcomes**:
  - Combination [1]: Document created
  - Combination [2]: WriteError
  - Combination [3]: WriteError
  - Combination [4]: WriteError
  - Combination [5]: TypeError

- Combination [6]: WriteError
- Combination [7]: WriteError
- Combination [8]: WriteError
- Combination [9]: TypeError
- Combination [10]: TypeError
- Combination [11]: TypeError
- Combination [12]: TypeError

Test cases

Combination 6-12 is excluded as its individual pieces are tested before, and expected to fail. For example: 6 is tested with 2, and is expected to fail at one point, and we will not learn more about the failure by this test.

#	Data Set	Type Constraint	Unique Constraint	Arg Type	Expected Outcome	Function Name
1	Complete	Compliant	Unique	`dict`	Document created	`test_create_suc`
2	Incomplete	Compliant	Unique	`dict`	`WriteError` raised	`test_create_inc`
3	Complete	Non-compliant	Unique	`dict`	`WriteError` raised	`test_create_typ`
4	Complete	Compliant	Non-unique	`dict`	`WriteError` raised	`test_create_uni`
5	Complete	Compliant	Unique	Non-`dict`	`TypeError` raised	`test_create_inv`

Pytest Fixture for Database Interaction

```
@pytest.fixture(autouse=True)
def clean_database():
    mongo_url = os.getenv("MONGO_URL",
"mongodb://root:root@mongodb:27017")
    db_name = os.getenv("MONGO_INITDB_DATABASE", "rootDb")

    client = pymongo.MongoClient(mongo_url)
    db = client[db_name]

    yield
    db.drop_collection("test_users")
```

Implementation of Test Cases

Link to test file(s) in repository

Test Execution Results

```
# Console output from pytest
(base) karl@supergo:~/rsync/dbwebb/sysver/lab-team-work$ docker exec -it
edutask-backend pytest test/util -v
========================================= test session starts
=========================================
platform linux -- Python 3.10.12, pytest-7.2.2, pluggy-1.5.0 --
/usr/local/bin/python
cachedir: .pytest_cache
rootdir: /app, configfile: pytest.ini
plugins: cov-4.0.0
collected 5 items

test/util/test_dao.py::test_create_success PASSED
[ 20%]
test/util/test_dao.py::test_create_incomplete PASSED
[ 40%]
test/util/test_dao.py::test_create_type_constraint_violation PASSED
[ 60%]
test/util/test_dao.py::test_create_unique_constraint_violation FAILED
[ 80%]
test/util/test_dao.py::test_create_invalid_arg_type PASSED
[100%]

============================================== FAILURES
==============================================
___________________________________ test_create_unique_constraint_violation
___________________________________

dao_fixture = <src.util.dao.DAO object at 0x7f02619d5840>

    def test_create_unique_constraint_violation(dao_fixture):
        # Data with unique constraint violation: Duplicate 'email'
        """Assert that the validator raises WriteError when unique
constraint is violated."""

        # First entry
        first_data = {
            "name": "Alice",
            "email": "alice@example.com"
```

```
        }

        # Create the first entry
        dao_fixture.create(first_data)
        # Duplicate data
        duplicate_data = {
            "name": "Bob",
            "email": "alice@example.com"
        }

        # Assert that creating this data raises a WriteError
>       with pytest.raises(WriteError):
E       Failed: DID NOT RAISE <class 'pymongo.errors.WriteError'>

test/util/test_dao.py:129: Failed
----------------------------------------- Captured stdout setup ----------
------------------------------
Connecting to collection test_users on MongoDB at url
mongodb://root:root@mongodb:27017

---------- coverage: platform linux, python 3.10.12-final-0 ----------
Name                                Stmts   Miss  Cover   Missing
-------------------------------------------------------------------
src/controllers/__init__.py             0      0   100%
src/controllers/controller.py          31     31     0%   1-103
src/controllers/taskcontroller.py      68     68     0%   1-139
src/controllers/todocontroller.py      21     21     0%   1-40
src/controllers/usercontroller.py      24     24     0%   1-46
src/util/dao.py                        67     36    46%   37-38, 79-83,
101-118, 134-141, 156-162, 170-173
src/util/validators.py                  7      4    43%   13-16
-------------------------------------------------------------------
TOTAL                                 218    184    16%

========================================= short test summary info
=========================================
FAILED test/util/test_dao.py::test_create_unique_constraint_violation -
Failed: DID NOT RAISE <class 'pymongo.errors.WriteError'>
====================================== 1 failed, 4 passed in 0.44s
======================================
```

Evaluation Statement

4 out of 5 test cases pass. The focus of the testing has been the create method in the DAO object. There are no rows which are missing that relates to the create method, it means that the method is fully tested.
The failed test case is related to the unique constraint.

References

- Lecture 1
- Lecture 2
- Lecture 3

Appendix

Template for Test Design Technique

Based upon our description of the test design technique, Copilot was used to generate a template for the test design technique. The template follows here. This template was used as a support for Assignment 3.2.

Step 1: Identify Actions and Expected Outcomes

- **Objective**: Clearly define the actions and their expected results.
- **Actions**:
  - [Action 1]
  - [Action 2]
  - ...
- **Expected Outcomes**:
  - [Outcome 1]
  - [Outcome 2]
  - ...

Step 2: Identify Conditions

- **Objective**: List all conditions that influence the outcomes.
- **Conditions**:
  - [Condition 1]
  - [Condition 2]
  - ...
- **Techniques**:
  - **Boundary Value Analysis**: [Describe relevant boundaries]
  - **Equivalence Partitioning**: [Describe partitions]

Step 3: Determine Combinations

- **Objective**: Combine conditions to create test scenarios.
- **Combinations**:
#	Condition 1	Condition 2	...	Expected Outcome

Step 4: Define Expected Outcomes

- **Objective**: Assign expected results to each combination.
- **Expected Outcomes**:
  - Combination [#]: [Expected Outcome]
  - Combination [#]: [Expected Outcome]
  - ...

Example Test Case Template

Test Case ID	Action	Conditions	Expected Outcome
TC-01	[Action]	[Condition values]	[Outcome]
TC-02	[Action]	[Condition values]	[Outcome]

[List test cases derived using test design technique]