Test plan overview

This is the test plan for the Privacy protection for information control application requested by SINTEF ICT. This test plan will be based on IEEE829-1998, the IEEE standard for software test documentation, with some adaptions to fit this project better. The purpose of testing is to find bugs and errors and correct them, and to make sure the program is working as expected. The purpose of this test plan is to make sure the tests will be executed as planned, and that they are well documented.

0.1 Test Methods

There are two main types of software testing: black-box testing and white-box testing.

Black-box testing

This is a method that test the functionality of an application. For this type of testing, knowledge about the application's code and structure is not required. The test cases are based on external descriptions of the software, e.g. specifications, functional requirements or designs. Black-box tests are usually functional, but they can also be non-functional. This type of testing can be applied to all levels of software testing.

White-box testing

This method is used for testing internal structures of an application. For white-box testing it is required to have both knowledge about the code and structure of the application, as well as knowledge about programming to design the test cases. This type of testing is normally done at unit level where it can test paths within a unit or paths between units, but it can also be used at integration and system levels of testing. This method can uncover many errors and problems, but it is not a good test method for finding out whether the program is fulfilling the requirements or not.

0.2 Testing approach

Our main focus will be on white-box testing. This is a program that is going to be used for research, which means that black-box testing will not be very useful as the client want to work on and test algorithms themselves. Our main task is to deliver a good framework with the necessary tools, and a working, learning algorithm so that further testing can be done with ease. Since one of the system requirements is high modularity, it will be a goal to have the tests be as little dependent on other modules as possible. There will be no training needs, as the testers are also involved in the programming.

What will be tested:

- **Unit testing:** This will be used for testing the functionality of the modules, so that we can ensure that they are working as intended.
- **Interface testing:** As our algorithm is based on case-based reasoning (CBR), it will be important to test that it is learning from new data.

What will not be tested:

- **Usability testing:** This program is intended for further research by the client, and not for use by customers. Since we are not delivering a program ready for users, there is no need to perform end-user tests to see how users interact with the program, and whether the product is accepted by users or not.
- **Learning of our algorithm:** For the same reasons we will not perform any tests on the quality of the GUIs. The GUIs that will be included is there to make testing easier for the client, not to provide the best possible interaction with end-users.
- **Run time:** We will not do designated tests for checking and optimizing the run time. This is because the main classification algorithm can be easily changed. Run time will only be looked into if the program is very slow even for small data sets.

0.3 Test case overview

Under is the test cases and their identifiers. The identifiers are named UNIT-XX, where XX is the number of the test case.

Unit tests:

- UNIT-01: Command line interface (CLI) functionality
- UNIT-02: P3P parser
- UNIT-03: Local database
- UNIT-04: Graphical user interface (GUI) functionality
- UNIT-05: Algorithm classification
- UNIT-06: Algorithm learning
- UNIT-07: Packet passing through network to community database

Under is the test case template for the unit tests

Item	Description
Name	The name of the test
Test identifier	The identifier of the test
Person responsible	The person responsible for making sure the test is executed correctly and on time.
Feature(s) to be tested	What kind of functionality that is being tested.
Pre-conditions	What code and environment that has to be in place before the test can be executed.
Execution steps	Stepwise explanation of how to perform the test.
Expected result	The expected output/result for the test to be successful.

0.4 Test cases

Item	Description
Name	Command line interface (CLI) functionality
Test identifier	UNIT-01
Person responsible	Henrik Knutsen
Feature(s) to be tested	All possible commands when running the program with the CLI. That input with incompatible commands does not run.
Pre-conditions	Code for input handling for all possible commands. Error handling for invalid inputs.
Execution steps	1. Test all the commands one by one. 2. Test the combinations of invalid inputs.
Expected result	1. The program runs using the input variables. 2. Error warning: Invalid arguments. Abort startup.

Item	Description
Name	P3P parser
Test identifier	UNIT-02
Person responsible	Henrik Knutsen
Feature(s) to be tested	That the P3P parser correctly parses all the required fields and their values.
Pre-conditions	The P3P parser must be implemented. Need to have P3P policies with a wide range of cases.
Execution steps	1. Manually go through a P3P XML and obtain all the required fields and the values. 2. Run the same P3P XML in the P3P parser and print the parsed elements and their values to console. 3. Compare the results from the two parsing methods.
Expected result	The two parsing methods give identical output. They must both have the same fields, each containing the same values

Item	Description
Name	Local database
Test identifier	UNIT-03
Person responsible	Henrik Knutsen
Feature(s) to be tested	Writing to and reading from the local database. That the serialization of the database is working.
Pre-conditions	Code for writing to and reading from the database file must be implemented. Need to have two different P3P policies.
Execution steps	1. Write policy A to the local database. 2. Write policy B to the local database. 3. Read policy A from the local database. 4. Read policy B from the local database.
Expected result	The written policy A and the read policy A must be identical. The written policy B and the read policy B must be identical.

Item	Description
Name	Graphical user interface (GUI) functionality
Test identifier	UNIT-04
Person responsible	Henrik Knutsen
Feature(s) to be tested	That all the interactable elements, buttons, lists etc., is working as intended.
Pre-conditions	GUI with all the necessary listeners must be implemented. Code for running the program with the GUI must be implemented.
Execution steps	1. Run the program using the GUI. 2. Test all the interactable elements.
Expected result	All the interactable elements is triggering the right methods when used.

Item	Description
Name	Algorithm classification
Test identifier	UNIT-05
Person responsible	Henrik Knutsen
Feature(s) to be tested	That the k-nearest neighbor algorithm bases its decision on the k most similar policies
Pre-conditions	Code for reading from the weights file must be implemented. A working k-nearest neighbor algorithm that uses the weights must be implemented. Need one policy to test on, and a set of policies to be used as history.
Execution steps	1. Load a set of policies into the history. 2. Run the k-nn algorithm on the policy to be classified and the history. 3. Manually go through the policies and verify the output of the algorithm.
Expected result	The algorithm finds the most similar policy.

Item	Description
Name	Algorithm learning
Test identifier	UNIT-06
Person responsible	Henrik Knutsen
Feature(s) to be tested	That the weights file is updated when a new policy is added to history.
Pre-conditions	Code for reading from and writing to the weights file must be implemented. Algorithms for classification and learning must be implemented.
Execution steps	1. Get the contents of the weights file. 2. Load a set of policies into the history. 3. Run the classification algorithm on the single policy and the history. 4. Choose to store the new policy, the context and the action. 5. Get the contents of the weights file. 6. Compare the contents of the weights files obtained in steps 1. and 5.
Expected result	The two weights files obtained in steps 1. and 5. are different

Item	Description
Name	Packet passing through network to community database
Test identifier	UNIT-07
Person responsible	Henrik Knutsen
Feature(s) to be tested	That packets can be sent between the client program and the community database.
Pre-conditions	A running local client. A (virtual) server. Code for sending and receiving packets must be implemented.
Execution steps	1. Start the program locally. 2. Start the (virtual) server. 3. Send packet A from the local client. 4. Receive packet A at the (virtual) server. 5. Send packet B from the (virtual) server. 6. Receive packet B at the local client.
Expected result	The received packet A is identical to the sent packet A. The received packet B is identical to the sent packet B.

0.5 Test pass / fail criteria

A test is passed if the given execution steps and input produce the expected results. If they do not, the test is failed.

0.6 Test scheudule

Under is the table with the scheduled execution dates of the unit tests.

Test identifier	Execution date
UNIT-01	Saturday 22.10
UNIT-02	Saturday 22.10
UNIT-03	To be determined
UNIT-04	To be determined
UNIT-05	To be determined
UNIT-06	To be determined
UNIT-07	To be determined

0.7 Risks and contingencies

For some of the tests we will not be able to test every possible input / output. So there is a chance a test will pass for all the combinations we will be testing for a specific test case, but still fail at some later point for some other combination. This can be a problem for the tests UNIT-02 P3P parser, and UNIT-05 Algorithm classification.

The P3P policies have a huge variety in which elements they contain, and certain elements have a N-to-1 relation, so it will be impossible to test if everything is parsed correctly for every possible P3P policy. The best way to prevent this is to handpick a set of policies that have as different content as possible, so that as many of the extremes as possible will be covered.

We got the same issue for testing the algorithm classification. There is simply too many combinations of learning base and input to cover everything. So again we have to do our best with regards to also covering as many extremes as possible.