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

The 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. The 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 it can be ensured that every module is working as intended.
- **Interface testing:** As the algorithm is based on case-based reasoning (CBR), it will be important to test that it is learning from new data.

What will <u>not</u> be tested:

- **Usability testing:** This program is intended for further research by the client, and not for use by customers. Since this program is not supposed to be ready for end-users, there is no need to perform end-user tests to see how the users interact with the program, and whether the program is accepted by users or not.
- **Graphical user interface (GUI) testing:** For the same reasons there will not be any tests on the quality of the GUI. The GUI 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:** There will be no designated tests for checking and optimizing the run time of the algorithm. This is because the main classification algorithm can be easily changed with another algorithm. 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.

Under is the template to be used when executing a test.

Item	Description
Name	The name of the test
Test identifier	The identifier of the test
Person responsible	The person that executed the test
Date of execution	The date when the test was executed
Execution steps	Stepwise explanation of how to perform the test.
Steps executed	The steps executed by the tester.
Expected result	The expected output/result for the test to be successful.
Step results	The results of the performed steps. SUCCESS / NO SUCCESS for each step.
Test conclusion	SUCCESS / NO SUCCESS for the entire test. Only successful if every step is successful.
Comments	Comments, if necessary, on the test.

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	That all possible commands are working correctly when using the CLI. That input with incompatible commands does not run.
Pre-conditions	Code for input handling for all possible commands. Code for error handling for invalid inputs.
Execution steps	1. Start the program with all possible commands. Run once for each command. 2. Start the program with all possible combinations of incompatible commands. Run once for each combination of incompatible commands.
Expected result	1. The program starts successfully, performing the action connected to the command. 2. The program aborts startup. Gives error warning: Invalid arguments.

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 their 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	1. All required fields and values are obtained and written down. 2. The P3P XML is parsed successfully. Its content is printed to console. 3. The results are identical. The the two outputs on paper and in console have the same fields, with 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. 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. 5. Compare the written policy A and the read policy A. 6. Compare the written policy B and the read policy B.
Expected result	1. Policy A is successfully written to the database file. 2. Policy B is successfully written to the database file. 3. Policy A is successfully read from the database file. 4. Policy B is successfully read from the database file. 5. The written policy A and the read policy A are identical. They both have the same fields, with the same values. 6. The written policy B and the read policy B are identical. They both have the same fields, with the same values.

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

Item	Description
Name	Algorithm classification
Test identifier	UNIT-05
Person responsible	Henrik Knutsen, Dimitry Kongevold
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 distance algorithm on the single policy and the history. 3. Manually calculate and write down the distances between the single policy and each of the policies in the history. 4. Compare the distances that are returned by the algorithm with the manually calculated distances. 5. Run the reduction algorithm with input to find the k nearest policies. 6. Manually find the k policies with the lowest distance. 7. Compare the policies returned by the reduction algorithm and those found manually. 8. Run the conclusion algorithm.
Expected result	1. The policies are successfully stored in the database. 2. The distance algorithm runs successfully for the chosen input, and returns the distances between the single policy and each of the policies in the history. 3. The distance between the single policy and the policies in the database are found and written down. 4. The distances returned by the algorithm, and the respective distances calculated manually are identical. 5. The reduction algorithm runs successfully for the chosen input and returns the k nearest policies. 6. The k nearest policies are found and written down. 7. The k policies returned by the reduction algorithm and the k policies found manually are the same policies. 8. The conclusion algorithm runs successfully.

Item	Description
Name	Algorithm learning
Test identifier	UNIT-06
Person responsible	Henrik Knutsen, Neshahavan Karunakaran
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. Code for writing to the database. Algorithms for classification and learning must be implemented.
Execution steps	1. Obtain and write down 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. Accept the recommendation. 5. Choose to save the new policy to history. 6. Obtain and write down the contents of the weights file. 7. Compare the contents of the weights files obtained in steps 1. and 6.
Expected result	1. The contents of the weights file is obtained and written down. 2. All the chosen policies are successfully stored in the database. 3. The algorithm classifies the new policy, gives a recommendation on what action to take, and asks the user if the recommendation is accepted. 4. The user is asked whether to save the new policy to history or not. 5. The new policy is successfully written to the database. 6. The contents of the weights file is obtained and written down. 7. At least one of the fields in the weights files obtained from steps 1. and 6. 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.
Execution steps	1. Start the program locally. 2. Start the (virtual) server. 3. Send policy A from the local client to the server. Print the contents of policy A. 4. Receive packet A at the (virtual) server. Print the contents of policy A. 5. Compare the sent policy A and the received policy A. 6. Send policy B from the (virtual) server to the client. Print the contents of policy B. 7. Receive policy B at the local client. Print the contents of policy B. 8. Compare the sent policy B and the received policy B.
Expected result	1. The program starts successfully. 2. The (virtual) server starts successfully. 3. Policy A is sent to the server. Its contents are printed to console. 4. Policy A is received at the (virtual) server. Its contents are printed to console. 5. The contents of the sent policy A and the received policy A are identical. 6. Policy B is sent to the local client. Its contents are printed to console. 7. Policy B is received at the local client. Its contents are printed to console. 8. The contents of the sent policy B and the received policy B are identical.

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 schedule

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 it is not possible to test every possible input / output because there is too many different combinations of input. So there is a chance a test will pass all the combinations used in the test cases, but still fail at some later point for some other combination. This will be a problem for the tests UNIT-02 P3P parsing, 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 necessary amount policies that are as different as possible, so that as many of the extremes as possible will be covered.

The algorithm classification tests have the same issue. There is simply too many combinations of learning base and input to cover everything. So these tests will have to be performed in such way that as many of the extremes as possible is covered.