Master's Thesis

Master's Programme in Computer Science

Smoke Testing Display Viewer 5

Matias Isosaari

November 30, 2022

Faculty of Science

University of Helsinki

Contact information

P. O. Box 68 (Pietari Kalmin katu 5)
00014 University of Helsinki,Finland

Email address: info@cs.helsinki.fi URL: http://www.cs.helsinki.fi

HELSINGIN YLIOPISTO – HELSINGFORS UNIVERSITET – UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Koulutusohjelma — Utbildningsprogram — Study programme
Faculty of Science	Master's Programme in Computer Science

Tekijä — Författare — Author

Matias Isosaari

Työn nimi — Arbetets titel — Title

Smoke Testing Display Viewer 5

Ohjaajat — Handledare — Supervisors

Antti-Pekka Tuovinen

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
Master's thesis	November 30, 2022	38 pages

Tiivistelmä — Referat — Abstract

In this thesis, software industry testing standards were reviewed to help improve NAPCON Display Viewer 5 (DV5) test coverage. DV5 is the graphical user interface part of the NAPCON Operator Training Simulator (OTS) and has a large existing codebase with no existing testing procedures. To help improve build stability a shallow and wide smoke testing approach would be adopted to cover as much core functionality as possible, before more rigorous end-to-end testing could be implemented. The thesis study was conducted using the design science methodology.

As part of this thesis, a smoke testing tool was created to test DV5 display and faceplate opening functionality. DV5 simulator configurations can consist of up to several hundreds of displays and display construction requires a lot of manual work. Using the created tool, tests were conducted in several display repositories to determine the functionality of the display sets. With only shallow automated testing, minor errors were found in most of the tested repositories, demonstrating the usefulness of automated testing with large display sets. In addition to the display testing results the display testing execution helped demonstrate some of the DV5 performance issues. Based on these findings, the smoke testing scenarios could prove useful in the future in DV5 performance testing.

ACM Computing Classification System (CCS)

Software and its engineering → Software creation and management → Software verification and validation → Software defect analysis → Software testing and debugging

Avainsanat — Nyckelord — Keywords

Testing, Software development

Säilytyspaikka — Förvaringsställe — Where deposited

Helsinki University Library

Muita tietoja — övriga uppgifter — Additional information

Software study track

Contents

1 Introduction **1**

2 Methodology **2**

3 Software testing **3**

3.1 Levels of testing 3

3.2 Dynamic testing 4

3.3 Black-Box testing 5

3.3.1 Equivalence partitioning 6

3.3.2 Boundary value analysis 6

3.3.3 Decision tables 7

3.3.4 State transition testing 7

3.3.5 Evaluation 8

3.4 White-Box Testing 8

4 Testing process **10**

4.1 Goals of testing 10

4.2 Faults errors and crashes 10

4.3 Test execution 12

5 Methods of testing **13**

5.1 Functionality testing 13

5.2 Regression testing 13

5.3 Ad-hoc testing 13

5.4 Soak testing 14

5.5 Smoke testing **14**

5.5.1 Scope of smoke testing 15

5.5.2 Process of smoke testing 16

5.5.3 Benefits of smoke testing 17

5.5.4 Smoke testing GUIs 17

5.6 Sanity testing **18**

6 DV5 Smoke testing implementation **20**

6.1 Display Viewer 5 20

6.1.1 DV5 window types 20

6.2 Testing DV5 22

6.2.1 Goals of testing DV5 23

6.2.2 Challenges of testing DV5 23

6.3 TestControlWindow 24

6.3.1 Preliminary steps for testing 26

6.3.2 Testing displays 26

6.3.3 Testing faceplates 28
6.3.4 Technical implementation 29
6.4 Evaluation of results 30
6.4.1 Mass display testing 30
6.4.2 Performance and memory usage 32
6.5 Limitations and further development ideas 34
7 Discussion 36
References 37

1 Introduction

Software testing is a part of the software development process where predetermined tests are performed on the system under testing to discover faults and errors before the product is released. These tests are performed in accordance with a testing plan, and are run by designated software testers who document and report the findings of the tests. Running these tests can be time and resource intensive, therefore it is important that the tests are run on a build that is stable enough for rigorous testing scenarios.

To discover errors as early as possible in the development and to ensure that the software build is stable enough to be meticulously tested, daily builds and smoke tests are employed as a form of preliminary testing. Smoke testing is a shallow and wide approach, where the aim is to cover as much of the functionality of the software with as few test cases as possible. Smoke tests can help improve the build stability by detecting errors very early in the testing pipeline. This makes correcting the mistakes easier and faster and ensures no time is wasted on testing builds that won't function properly.

Applications that have little to no established testing can benefit in a short amount of time from simple smoke tests implemented to bolster the stability of the builds by ensuring core functionality of the software performs as intended.

The goal of this thesis was to find ways to increase the stability and robustness of Display Viewer 5 (DV5) by studying software testing literature and introducing new testing procedures to DV5. The focus was narrowed to creating a smoke testing tool for one core functionality of DV5, display opening. Following the design science methodology, the tool would be a solution to a practical engineering problem and its effectiveness could be analysed after being implemented.

The structure of the thesis is as follows. Chapter 2 describes methodology used in the thesis. Chapter 3 generally presents software testing and its levels, focusing on dynamic testing methods. Chapter 4 describes aspects of the software testing process. Chapter 5 lists methods of software testing, including smoke testing, used in the practical implementation of the thesis. Chapter 6 describes the implementation of the smoke testing tool made to test DV5. Chapter 7 contains discussion on the thesis results.

2 Methodology

Design science methodology is chosen to be the study method of the thesis as it is suitable for practical engineering problems. The thesis will aim to define the problem, the solution and evaluate the results. According to guidelines described by Hevner et al. the goal of the study is to produce a viable artifact for a relevant business problem. The effectiveness of said artifact will then be demonstrated and evaluated. (Hevner et al. 2014) Furthermore the results should be provided in a form understandable to both technology-oriented and management-oriented audiences.

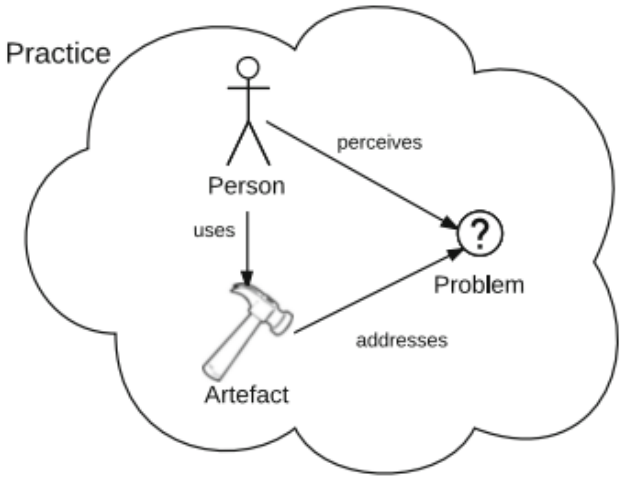

Figure 1: Components of design science (Johannesson & Perjons, 2014)

Figure 1 describes the relations of the key components in design science research. Artefacts are created and used by people to solve problems in practices. These artefacts are tailored specifically to solve the perceived problems, and design science is “the scientific study and creation of artefacts as they are developed and used by people with the goal of solving practical problems of general interest” (Johannesson & Perjons, 2014).

The starting point of this thesis was to find out how to improve the stability of the Display Viewer 5 (DV5) software using smoke testing to test one of the core functions of DV5, display opening. The solution should be a tool to help testers smoke test display openings in such a manner that the testing would be at least partially automated and easy to execute even without technological knowledge. The solution would be evaluated based on its ability to either A. detect faults in existing display opening logic or in the existing display repositories, or B. provide clear indication that the display opening logic functions as intended and does not crash the application.

3 Software testing

Software testing is an examination performed on software to ensure its quality and performance, using predefined methods to map and minimise errors. Testing is performed because it adds value to the product by increasing its quality and reducing the chance that the product would not function as predetermined. Testing takes resources away from product development, so it must be carried out with determination and a central goal in order to get the most out of testing. The goal of testing is to ensure that every functionality of the software is implemented according to its definitions (Spillner & Linz, 2021). Testing is performed at every stage of software development and should take up considerable time to do right. For example, up to 30% of a developer's time should be spent on testing. (Fairbanks, 2010).

Testing is usually done by testers who use tools such as testing frameworks to conduct their tests. Such tools often provide the elemental classes and context for making test cases, an environment to run them separately or share elements between them and ways to parse and present the results of the tests. The programmer on the other hand needs to know what goes in and what should come out. The expected results often require preset conditions that need to be coded to make assertions based on them. The programmer needs to provide the accepted results for each test case.

The following subchapters briefly explain the levels of testing, dynamic testing and two of its forms, Black-Box and White-Box testing.

3.1 Levels of testing

The levels of testing can be roughly divided into four different levels. These are unit testing, integration testing, validation testing, and system testing (Pressman, 2005). Of these, unit testing focuses on individual, isolated components. Tests at this level form a module that acts as a unit. These modules, on the other hand, consist of testing techniques that aim for maximum error detection and maximum coverage. The purpose of unit testing is to ensure that the code performs the tasks and functionalities designed for it as intended. This inspection is carried out at a really low structural level. Usually, the programmers themselves perform this level of testing (Pressman, 2005).

The next level again focuses on integrating the modules together. The goal is to create a complete application package. Integration testing is an important part of the testing process, because although the operation of the modules as isolated units has been observed at the previous level, it cannot be

certain that the units will play together as desired. Functionality may be degraded, for example, by data loss between interfaces, messages may not be transmitted, and other possible means.

Thus, important test items at this level are the execution order of the program and the input and output between the different modules (Pressman, 2005). As with unit testing, the programmer performs this level of testing.

At the validation testing level, it is verified that the program implements the objectives and functionalities set for it in the software requirements. Thus, once integration testing is complete, the focus shifts to application functionality, behaviour, and performance. This level of testing is usually performed by independent testers.

System testing, on the other hand, focuses on testing the operation of software with its proper use within a nearby environment. This review includes, for example, the operation of software in the system hardware and databases. The purpose is therefore to ensure that all elements of the system perform the tasks assigned to it with the required performance in the environment in which it is set. This includes backing up and restoring them (Pressman, 2005).

3.2 Dynamic testing

Dynamic testing refers to the fact that the software under testing needs to be executable and be able to process data fed to it and produce an output that can be analysed. The goal of this type of testing is to see if the requirements of the software that are laid out in the specifications are implemented and that they are implemented in the correct manner. This should be achieved using broad test coverage to test as many requirements for failures as possible in the least amount of effort.

The following steps are a part of the dynamic testing technique implementation (Spillner & Linz, 2021):

- Test condition and precondition definition and objective specification
- Individual test case specification
- Test execution specification

Adherence to this process depends on a variety of factors, for example including planned system usage, skill level of team members, experience of team leadership, availability of time and other resources.

The aspects of the system that require dynamic testing should be taken into account at the outset. After that the necessary conditions for fulfilling the requirements are defined for test cases, along with an analysis of the risks should the system fail under a specific test (Spillner & Linz, 2021).

Individual requirements and the test cases designed to test them should be able to be linked in a manner which allows for the discovery of change in outcome as a result of a change in system requirements or test cases to be able to be traced back between them. This is called traceability and it enables, for example, the testers to be able to define the state of test coverage and through that to be able to define a sufficient threshold for test coverage for dynamic testing. Usually this threshold is set between a hundred and a thousand test cases (Spillner & Linz, 2021).

The test cases should also fulfil certain specifications. For starters they should be designed with preconditions in mind. They should also include the expected result for the test cases input values upon the test object. These should be well defined and documented in order for the analysis of the test cases generated reaction in the test object to be able to bear fruit. The danger of foregoing these steps can result in incorrect analysis of test results. The test cases are usually arranged in a sequence called a test suite or scenario, which is defined in the test execution schedule. The test execution schedule contains the list of test cases grouped by their intended objective (Spillner & Linz, 2021).

The following chapters explain Black-Box and White-Box testing, that are forms of dynamic testing.

3.3 Black-Box testing

Usually the basis for Black-Box testing rests in the specifications, documentation and interface descriptions. These represent the expected behaviour of the system under use. The test cases therefore are designed with this behaviour in mind. If the behaviour that is laid out in the documentation is described in an inaccurate form, or even contains errors, Black-Box testing will not reveal these faults (Homès, 2013). Black-Box testing can also be performed on systems without any specifications. In these cases, the test cases are based on what the tester expects as an outcome and the results are visible observations of the functionality of the system. Most integral part of Black-Box testing is observing the functionality of the tested object from the outside, without paying any attention to what is happening on the inside.

Black-Box testing is oftentimes used to test the functionality of user interfaces. In order to interact with the software's user interfaces data will have to be supplied. Planning this input data is the work of the tester. The interaction can also take the form of taking the role of a user and, through some sequence of actions described in the planned test cases, bring forth an outcome that may or may not be in line with the expected outcome described in the specifications of the system. The outcomes of these tests are gathered and analysed. The outcomes can be in the form of data output by the system or in the form of screen printouts or some other form of perceivable output (Homès, 2013).
Next few subchapters will go over the different varieties of Black-Box testing techniques and through their differences and their areas of strength

3.3.1 Equivalence partitioning

When using the technique known as equivalence partitioning the set of all possible values of input data is analysed and subdivided into partitions. These partitions are also known as classes. The criteria for placement into a particular partition is decided by the assumption that all values belonging to the same class will be processed in the same manner by the software under testing. This allows for the streamlining of the testing process since testing only one member of a class should be sufficient given our assumption of homogeneity inside a given class (Spillner & Linz, 2021).
The aim of equivalence partitioning is to reduce the amount of test cases through a systematic process. This process also has the added benefit of forcing the tester to conduct a detailed analysis of the characteristics of the set of possible input and output values. The benefit comes in the form of increased efficiency (Homès, 2013).

3.3.2 Boundary value analysis

Boundary value analysis is used on top of equivalence partitioning. The reason being that test cases defined using equivalence partitioning classes are most vulnerable for producing errors in the edge cases with respect to classes. So the value of boundary value analysis is that the technique helps in uncovering errors in the system that are not detected using equivalence partitioning techniques. It requires that the classes defined by equivalence partitioning be ordered in some way and that the boundaries are identifiable (Spillner & Linz, 2021).

It should also be noted that boundary value analysis depends on the assumption that the classes defined by equivalence partitions are identified correctly. Should this assumption not be true, then the boundary values identified by further analysis might prove to be incorrectly chosen, leading to possible undiscovered defects in the system under testing. The technique is not available for use when the variables classified by the equivalence partitions are not liner values (Homès, 2013).

3.3.3 Decision tables

Variables are considered as independent elements when designing test cases while using equivalence partitioning and boundary value analysis, while in reality there is interaction between variables (Homès, 2013).

Decision tables serve as a way to identify which combinations of values might lead to a failure when used as input in the system under testing. Decision tables also serve as a way to document different combinations of input values. These combinations reveal results that might have been not thought about otherwise, which leads into the fact that it is of use to test these combinations systematically. It is also a fact that conditions have an effect on other conditions. Through this interaction it can be found that some conditions are mutually exclusive, some produce redundancies, others influence each other in different ways (Spillner & Linz, 2021).

3.3.4 State transition testing

Program flow can be described using states and the transitions between them. These states transition from one to another based on a number of factors, for example function calls or user input. When a state can have two or more potential next states, guard conditions need to be used in order to clearly define what state is the intended following state when a certain event occurs.

3.3.5 Evaluation

Black-Box testing does not differentiate between poorly established preliminary system requirements or system specification. If these starting points are made with defects or outright errors, Black-Box testing will not be able to detect if the system is working as intended. It is merely able to detect if the system works as specified by these original set specifications. These erroneous requirements can be caught by an astute tester during test case planning. Reviews are also a step in the software production cycle where specifications should be looked at and fixed if inconsistencies are discovered (Spillner & Linz, 2021).

Black-Box testing techniques are unable to perceive if the system has functionality that is not contained in the initial system specifications. This is a problem, since these unforeseen functionalities can present issues in regards to the security of the system. The emergence of the unspecified functionalities is not guaranteed, since the test cases are not designed with these functionalities in mind. Hence if they are to appear in the testing process, they are the result of happenstance (Spillner & Linz, 2021).

3.4 White-Box Testing

In contrast to Black-Box testing, White-Box testing relies on the knowledge of the inner workings of structure of the program. White-Box testing is also known as structural testing or code-related testing, referring to the fact that access to the source code is of paramount importance. White-Box testing can be applied to all levels of tests (Spillner & Linz, 2021).

The idea that all parts of code of the program under testing are executed at some point is a key concept of White-Box testing. Testing is broken down between each process and test cases are designed and executed based on this configuration. The object's specifications have an effect on the test cases through for example identification of defective behaviour. The degree of test coverage is also determined beforehand (Spillner & Linz, 2021).

Statement testing is interested in the test object's individual statements. A number of these statements are executed based on the predetermined ratio of code coverage. Once the tests have been executed, it needs to be checked that all statements that were included in the test cases were run as a part of the test. The success of the test is determined by examining if the aimed code coverage has been achieved (Spillner & Linz, 2021).

Decision testing focuses on the decisions inside the source code. These decisions affect the flow of program execution, and these ramifications are assessed as a part of decision testing. While statement testing looks at the coverage of statement execution, decision testing examines branch coverage. Branching of the program is triggered by conditional statements within the source code, such as “If” and “Case” statements. These branches are often modelled by graphs, with edges portraying a possible branch between states, which are of course displayed as nodes of the graph (Spillner & Linz, 2021).

4 Testing process

Software testing is a planned procedure performed on software to map and minimise faults and errors. Testing should be planned and executed with a central goal in mind.
The following subchapters describe the goals of testing, what kinds of errors can be encountered and the execution of testing.

4.1 Goals of testing

The goal of testing is to ensure that the program meets the required quality requirements before the program is published. This is done by telling the development team at different stages of development what is wrong with the software at the moment. The company's financing may be tied to the achievement of certain milestones, the achievement of which can be demonstrated by tests. This is an important process because if there are bugs in the software after its release, customers will face them, lowering their opinion of the product, leading to poor reviews and declining sales. This decline in reputation can be serious as it can lead to the loss of company assets. In addition, the reputation for bad reviews of one product will be remembered in the company's history by customers, also affecting future earnings. For example in game development, console and mobile developers want to be sure that the games they publish on their platforms exceed the quality standards they set. This is because they don't want software released on their platform to negatively impact the reputation of the entire platform (Schultz & Bryant, 2016).

4.2 Faults errors and crashes

The cornerstone for the planning of test cases lies in the test basis, which is formed from the documentation which describes the expected behaviour of the test object when given predetermined input data. Through this description of requirements for the software we can draw conclusions on what constitutes faulty behaviour for some functionality of the software under testing. An important term that is used when talking about testing terminology is "defect". The definition of a defect is failing to match the expected behaviour defined in the requirements when given a certain input. Expected behaviour should be documented beforehand.

Faults are the causes of system failures, which can be seen as incorrect output or crashes. Faults are only perceived when the system is running or under testing. It is important to be clear what is meant by a fault and the causes of said fault. Faults arise from coding errors, which cause system failures that again fall under the term. Faults can also be linked in the way of masking other faults. So fixing one fault can lead to another fault that was dormant before fixing the first fault causing a system failure after the fix. This phenomenon is called defect masking. Also it should be noted that not all faults in the system trigger system failures. Some lie undetected since their effects are minor or just not caught by the test cases (Spillner & Linz, 2021).

There are a multitude of reasons why software might contain errors. People might make mistakes due to pressure that arises via tight scheduling deadlines or because the technology being used is unfamiliar to them. It might also be the case that the source code or the project in question is overly complex, containing large amounts of components all interacting with each other. The participants of a project might have misunderstandings regarding the project requirements or specifications, leading to incorrect implementations that result in compatibility or other issues (Spillner & Linz, 2021).

Most dangerous form of error for the software development process is in its documentation. It boils down to the simple idea of the fact that if there are errors in the assumptions then you cannot derive correct outcomes. So checking the test basis for errors, ambiguous language and items that should be addressed being missing is very important. Once the test basis checks have been made and deemed satisfactory the test conditions should be ordered by priority, which enables testers to focus on the most important functions (Spillner & Linz, 2021).

It is also important to communicate when errors are found in the source code or in the documentation. The report should be addressed to the author as well as to the management, so that they become known to other interested parties of the development process, such as analysts, product owners and others. While it is human nature to not be receptive to criticism, it is important to acknowledge the positive implications of errors being discovered. Those being the facts that the defects of the product can be fixed and thus the quality of the product can be improved. Disclosing these defects in the test object requires soft skills, which when wielded correctly, can result in positive reception of these types of reports (Spillner & Linz, 2021).

4.3 Test execution

When an error is detected in the software, an error report is generated. The report is a detailed instruction on the recurrence of the fault, or the exact circumstances from the time the fault occurred. Analysing the report includes error verification. If the error can be verified, it will be taken into account in the development to correct the error. In some cases, the development team may decide that the bug is too small to use resources to fix it. Once the bug is fixed, the fix will be included in the next release version of the software.

Test execution can be done either manually or it can be automated using test automation framework or software. Automation can be either partial or complete and helps the tester test large tasks with minimal inputs or overseeing required. Test automation is favourable to most companies, since almost any type of task can be at least partly automated, making it significantly more efficient and less costly to execute in the long run. Automated tests can be run as often as required and can be used to gather data and statistics about the run test cases.

5 Methods of testing

There are several commonly used methods for testing software that are somewhat different in procedure. Generally in testing, the main focus is on testing the functionality of the software and increasing the quality and robustness of the product by detecting any unwanted behaviours.

The following subchapters briefly explain different types of testing techniques, with further emphasis on smoke testing.

5.1 Functionality testing

In functionality testing, all the functionalities related to using the software are considered and the tester looks for bugs in all the areas of the software. Functionality testing is the primary form of testing and is designed to ensure that the software works well, looks right, and does not crash as a result of normal operations (Novak, 2010). All unpredictable results can be interpreted as errors that should be reported in the functionality testing.

5.2 Regression testing

Regression testing tries to make sure that items that have been previously fixed or found to be working are not broken in new versions of the software. Sometimes bugs reappear when the source code changes with versions (Novak, 2010). To do this, it's a good idea to go through the problem areas again when making changes. Regression testing is facilitated by checklists of items to be tested that can be subjected to weekly testing. Timed implementation of automated tests also makes it easier to find bugs in previously running code.

5.3 Ad-hoc testing

The simplest form of testing, or ad-hoc testing, is the most basic way to look at how the program works. In ad-hoc testing, rules and checklists are forgotten and the software is used normally. Testing is then performed without a plan or structure. However, the findings are documented even if the process is not documented. The advantage of ad-hoc testing is that serious errors can be detected

quickly. If the tester is familiar with the program being tested, he or she can effectively detect the most striking bugs during a natural use session. However, ad-hoc testing should not be the primary method of testing, as unstructured testing leaves a lot of stones unturned and leads to poor testing coverage. On the other hand, the errors found during the ad-hoc testing may reveal possible shortcomings in the rest of the testing process itself, as the errors found were not found in the formal testing and were therefore not tested.

Some bugs can only be found by using the program naturally. Sometimes a bug can be found by chance even years after the software is released. Most developers encourage users to report bugs on the feedback form at both the alpha, beta, and release stages.

5.4 Soak testing

In soak testing, the program is left on for an extended period of time, for example, in pause mode or to perform some repetitive operations. The goal of soak testing is to test long-running sessions and detect errors such as memory leaks or program crashes. Soak testing is easy to implement but time consuming. The most common test items are the programs menu, mode, or idling for long periods of time, for example, around the clock.

5.5 Smoke testing

Smoke testing is a testing method that is used to ascertain whether or not the software under testing is stable enough for more detailed testing. A smoke test is defined by the ISTQB glossary as ”A test suite that covers the main functionality of a component or system to determine whether it works properly before planned testing begins” (ISTQB Glossary, 2022). It is important to note that smoke testing is a type of preliminary testing and should therefore aim to find simple flaws in the software crucial enough to warrant the rejection of a new build without bothering with the details too much. The focus is on the important functionality of the software. It should however cast a wide enough net that all areas of the software’s functionality are touched on somewhat. When smoke testing is done on a specific build, it is called a build verification test (BVT) (Chauhan, 2014).

Passing the smoke test indicates that the software did not contain any simply found fatal flaws, meaning that further testing is needed to find them if they exist. If the smoke test fails (meaning the

build is not stable), the build is rejected and it is up to the development team to make a new build with these issues taken care of.

Sanity test, while being related to smoke testing, is not an equivalent term. While both can be used as a basis for rejecting a build, the crucial difference comes from the reason for the rejection. For sanity testing, the reason for a rejection comes from the fact that the new build did not contain a working implementation of the newly added functionality, whereas smoke testing tests for crucial functionality as a whole.

5.5.1 Scope of smoke testing

Smoke testing falls under the category of preliminary testing. It is the first type of testing a build is subjected to after the build is finished. The testing itself is done by the QA team, but under certain conditions the development team may perform smoke testing. It is a form of high level testing, meaning it does not delve too deep into the details. Details of individual modules are covered in later stages using functional and regression testing. It should also be noted that all parts of the software should be inspected, creating a somewhat wide net to catch possible flaws from all parts of the application. Smoke testing scenarios can also be for example run in parallel with load tests to do performance analysis on software (Khan & Amjad, 2016).

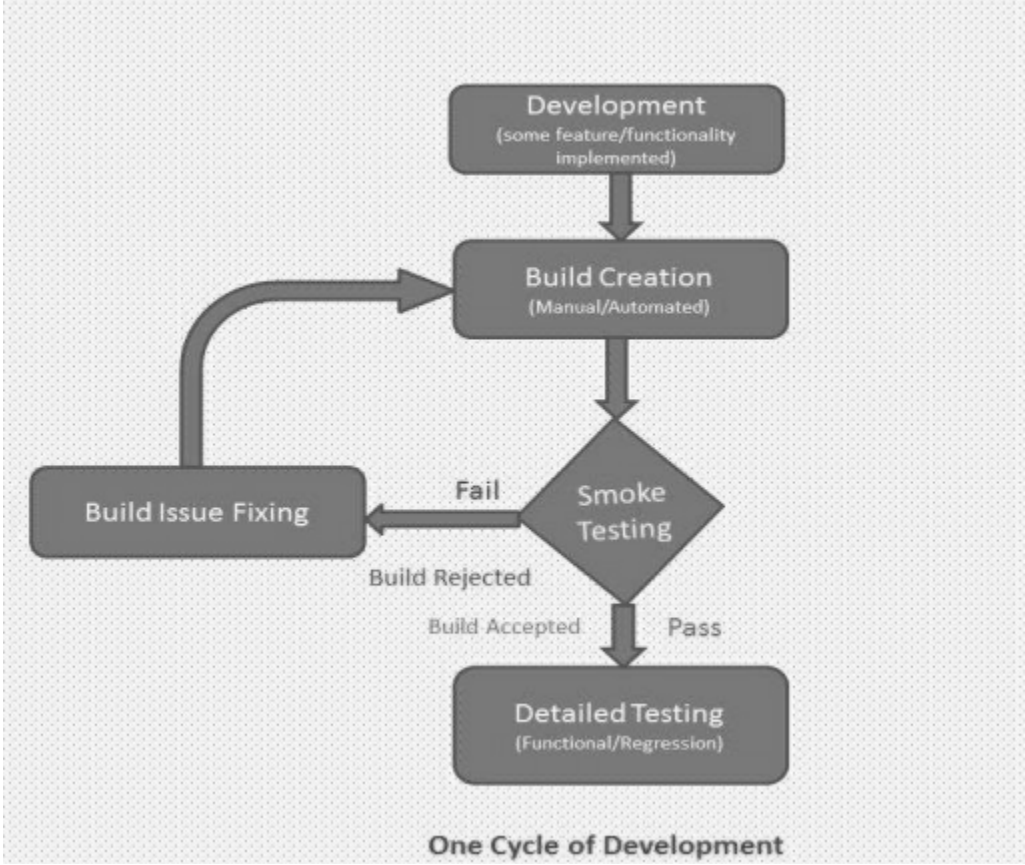

Figure 2: Smoke testing in development cycle (Chauhan, 2014)

5.5.2 Process of smoke testing

Smoke testing should be done by either the QA team or the developers. If the QA team conducts the smoke testing then it is done before any other testing takes place. Alternatively the development team can conduct the smoke testing before the build is submitted to the QA team. Figure 2 describes how smoke testing verifies the build and leads to the build either being accepted for further testing or rejected, needing to be fixed in the next build.

First part of the smoke testing process is to identify the smoke test cases. We must find the least amount of test cases that cover as wide an array of functionality of the application as possible. This is done in order to make the process as fast as possible, maintaining the efficiency of the overall process, that being the principal reason to smoke testing in the first place. After the smoke testing test cases have been identified, comes the creation of said tests and if possible, automating them. After the cases have been implemented then all that is left is to run these tests on new builds and take their

results into consideration on whether or not the build should be accepted or rejected. After this comes the maintenance of the test cases over time in order to keep them relevant.

According to McConnel, most development teams create a penalty for breaking the daily build (McConnell, 1996). While the motivation behind this (emphasis on the health of the project and trying to minimise the occurrences of build being broken) is solid, the method can be considered archaic for today's working environment.

5.5.3 Benefits of smoke testing

Smoke testing offers good cost-to-benefit-ratio since it is fairly simple to implement and maintain while still improving the robustness and quality of the software under testing. Compared to a more comprehensive end-to-end integration testing method, smoke testing can save resources in many places. Some of the benefits of smoke testing include: saving time, cost and effort, reducing the risks of errors in integration of parts, improving the quality of basic functions, and improving the assessment of progress for the core functionality of the software (Chauhan, 2014).

The savings in time, effort and cost are due to the nature of smoke testing. Smoke testing usually covers only the basic functionality of the software under testing, therefore maintaining tests is relatively easy and cheap compared to a more comprehensive approach. Smoke testing will ensure the build is stable enough for rigorous testing so no time or resources are wasted on testing unstable builds.

Due to the low time cost smoke tests can often be executed e.g. daily. Daily or nightly build and smoke tests help reveal bugs early in the development process. When the application is tested for build verification each day, it becomes much easier to determine where the build breaking error occurred. If the product passed smoke tests yesterday, but no longer passes them, something must have happened in between that broke the build (McConnell, 1996).

5.5.4 Smoke testing GUIs

One of the challenges of smoke testing is testing software with graphical user interface. While creating smoke tests for conventional software can be simple, smoke testing software with GUI is usually more complicated and requires specialised testing tools (Memon et al. 2003). For this reason, smoke testing GUIs are often neglected which can lead to more expensive testing required later. One

way to deal with applications with GUIs is to go around the GUI and test the underlying logic instead. This can help in testing the functionality of the software, but ultimately neglects the perspective of the end user. Using GUI testing extensions or specialised frameworks can help create smoke tests for GUI applications. One of such frameworks is DART (Daily Automated Regression Tester) (Memon et al. 2003) that can be used to automate the entire smoke testing process for GUIs, including GUI analysis, smoke test generation and execution (Memon et al. 2005). While such frameworks already exist and can help detect large number faults in code, usage of these GUI smoke tests frameworks should go hand in hand with code-based smoke tests, since there are some classes of faults that automated GUI testing cannot detect (Memon & Xie, 2004).

5.6 Sanity testing

Like smoke testing, sanity testing is a shallow and wide approach to preliminary testing before wider system testing is conducted. Also like smoke testing, sanity testing can be used to accept or reject a build. Unlike smoke testing, which is usually conducted on an unstable build, sanity testing is conducted when a build is considered stable to verify the correctness of the application under testing.

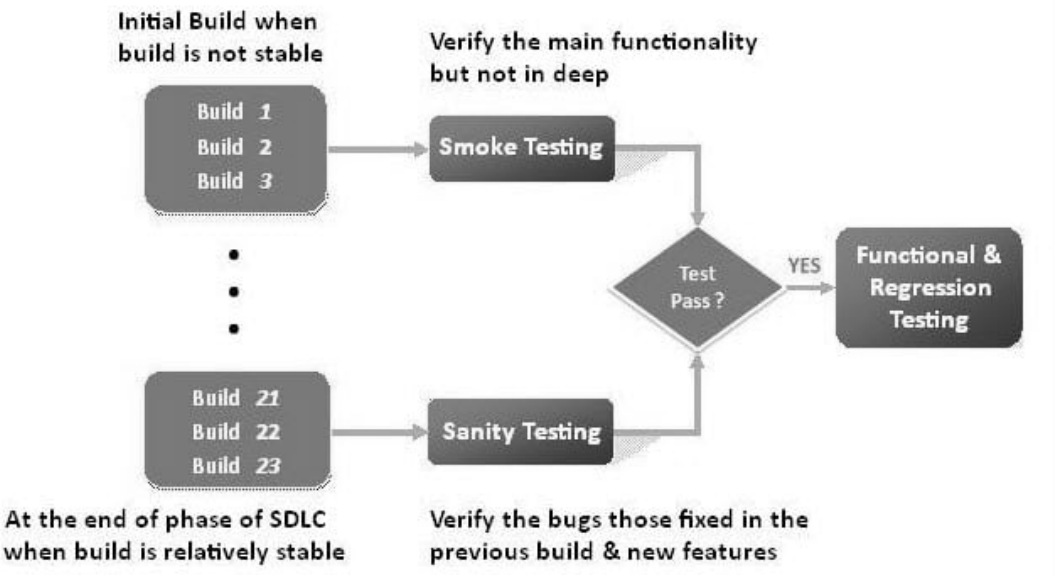

Figure 3: Use of smoke and sanity testing in product life cycle (Hamilton, 2022)

Figure 3 depicts the use of smoke testing and sanity testing throughout the products life cycle. Where smoke testing is performed on the initial unstable builds to verify the core functionality of the system

under testing, sanity testing is reserved for later, when builds are stable and the focus shifts from whether the application works to how well it works. The main goals of sanity testing are to determine that the build contains the specified changes and if they're working as intended.

6 DV5 Smoke testing implementation

The goal of this thesis was to find ways to increase the stability and robustness of DV5 by introducing new testing procedures. In order to address this problem, a smoke testing tool would be implemented to test DV5 core functionality. The functionality of the tool would cover opening and testing the displays and faceplates of DV5. The following chapters describe DV5 the software, the goals and challenges of testing it, and describe the TestControlWindow that was created and used to smoke test DV5 displays.

6.1 Display Viewer 5

NAPCON Display Viewer 5 (DV5) is a software used to emulate the user interface of an industrial process plant's (e.g. refinery) distributed control system (DCS). It is used together with NAPCON ProsDS (process simulator) and NAPCON Informer (database) to run the NAPCON Operator Training Simulator (OTS). NAPCON OTS is used for example by Neste in Finland to train refinery operators. Using DV5 operators can be trained to control the plant's processes in simulated scenarios.
DV5 is a desktop application run locally and it is developed with C# and Windows Presentation Foundation (WPF) technologies. DV5's main function is to serve as the UI for the operator to view and control the emulated process.
DV5 is split into multiple customised versions, called "flavours", with distinct styles. Each of these flavours are designed to imitate the style of a different DCS. They all share mostly the same use cases, with each having some unique functions and needs. The work done in this study has been mainly implemented to support the testing of the DV5 DNA flavour (used for example in the Neste Porvoo refinery in Kilpilahti) with a plan to support more flavours in the future.

6.1.1 DV5 window types

DV5 is used to open and control views that display process areas and equipment. These views exist in a process hierarchy that contains the areas and process displays of the emulated DCS.

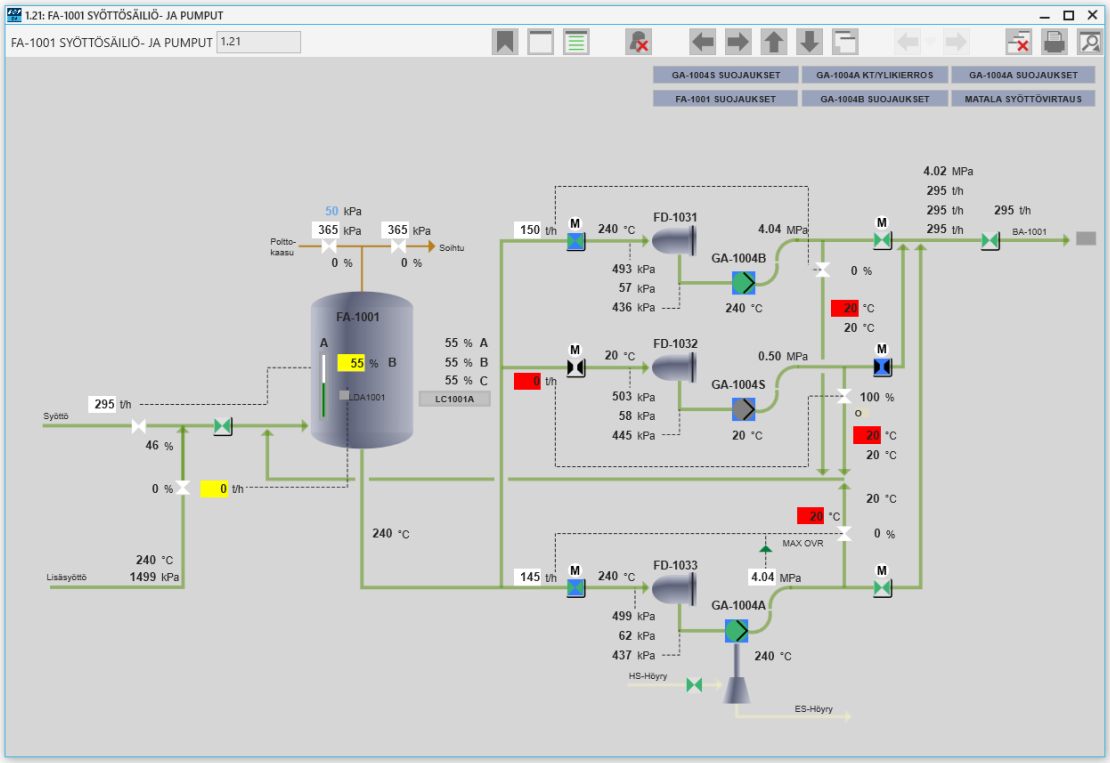

Figure 4: DV5 DNA Picture window

A Picture window (Figure 4) is a window type in DV5 DNA that is used to display a view of a process area. The user can open multiple picture windows to view several different areas at once. From the picture window the operator trainee can monitor, access and manage process equipment. Managing the equipment is done via faceplates.

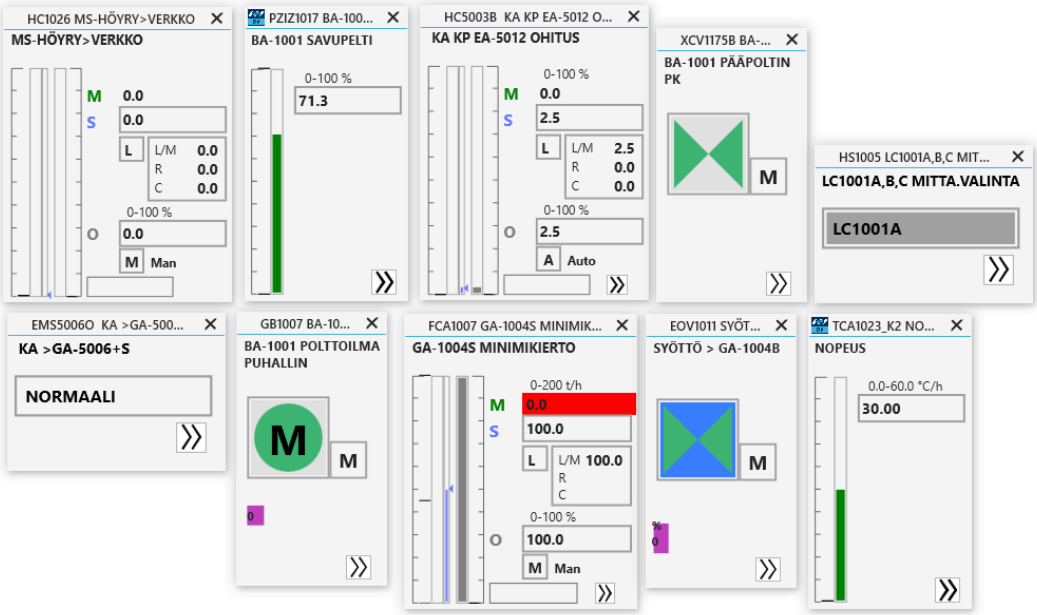

Figure 5: DV5 DNA Faceplates

Faceplates (Figure 5) are small windows that open from selected equipment in the picture windows. Using faceplates the operator trainee can view more detailed information about the equipment and modify some of the values to affect the process.

DV5 simulators can contain hundreds of views that contain information from thousands of database values.

6.2 Testing DV5

Currently DV5 has no defined or documented testing procedures. The implemented features are manually tested by developers and verified by the customer. A plan is set to adopt a testing and release schedule that will allow better defined and documented testing processes. To accommodate proper DV5 testing, smoke testing should be implemented as part of this testing and release schedule and smoke tests should be executed on all projects that are currently being actively developed. Smoke testing should also be considered for all projects when changes are made in shared libraries which are utilised by all DV5 simulators.

6.2.1 Goals of testing DV5

The first goal of testing DV5 is to determine if the current version is stable and working as intended. If the build verification testing fails, the build should be fixed immediately before it is passed further. If a bug is discovered in quality assurance, a ticket should be raised to fix the bug in a future version. When a build is broken, besides fixing the issue, it should be determined why the build was broken. Another goal of testing DV5 should be to figure out what causes build breaking errors in the project and help correct the process of developing DV5 so that such errors wouldn't occur. For example, if build breaking issues often occur due to changes in commonly used libraries, then the dependencies between simulators should be made more clear and extra scrutiny to changes in commonly used assets should be used. Same applies to theme resources shared between multiple DV5 flavours in the same project. Whatever the cause of the issue is, it should be considered in the future if it is something someone else can be confused with as well. When common pitfalls are documented they can be more easily avoided.

Testing should also include performance testing as a way of helping increase DV5 overall performance. According to Hooda & Chhillar, the lack of performance testing is the root cause of most of the failures that occur in the software industry. Testing performance manually is very difficult and for that reason performance testing tools are used (Hooda & Chhillar, 2015). Including the aspect of performance in testing DV5 can help with demonstrating that it performs well enough to fulfil customer set criteria. It can also help comparing between different projects to find bottlenecks and help increase performance (Khan & Amjad, 2016). In the case of DV5, performance smoke testing could be considered for display navigation, opening displays (partially covered in this implementation) and triggering alarms.

While Fairbanks' 30% time spent on testing may seem like a high investment, it would be an accurate estimate to reach appropriate testing levels for DV5.

6.2.2 Challenges of testing DV5

The first challenge to testing DV5 is that currently there is no proper testing implemented. Setting up new testing operations and routines takes time and resources and is a large commitment when deliveries to clients must continue. However, while the commitment may seem daunting, according to McConnell, testing should be considered even more seriously when under constraints of resources

and time, since under pressure developers may take shortcuts leading to faults that wouldn't happen under normal circumstances (McConnel, 1996).

DV5 has a large legacy codebase and technical debt which makes architectural changes that would benefit testing difficult. It is split into several projects (.NET "Solutions"), some of which are further split into different theme "flavours". These different projects each represent a different simulator, cover a lot of similar core functionalities and share some common resources amongst each other. These shared dependencies pose a challenge in development, since modifications to shared resources affect multiple solutions. In testing DV5, these shared libraries should take some priority to cover test cases that can affect each simulator. Despite the shared libraries, and common use cases, each simulator needs to be considered on its own when designing tests, since they have many simulator specific quirks and exceptions.

Creating automated end-to-end GUI test cases would be extremely costly, since the simulators so far haven't been developed with that in mind. Maintaining such tests would as well take up considerable resources. Some flavour specific functionalities only appear in certain flavours and certain display sets so there are no proper models for testing them all from developers perspective. For this reason, creating shallow and wide smoke testing for each core functionality, while creating a long term testing plan would be the ideal way to get the most benefit in the short and long term.

DV5 simulators can have hundreds of displays, which makes it hard to validate each manually, therefore automated tests are necessary.

6.3 TestControlWindow

The TestControlWindow is the result of the display smoke testing research for DV5. It is this study's artefact to the problem of increasing DV5 test coverage with simple tools using a shallow smoke testing approach. Of the core functionalities of DV5, it can be used to test the display and faceplate opening in DV5. The TestControlWindow is designed to be used by either a developer or a simulator engineer to test the DV5s faceplate and display opening functionality on large sets of displays. It can be used to validate the process hierarchy and opens each display in the hierarchy. This should be done periodically during normal development and e.g. daily during display construction for new simulators to verify the integrity of the built displays and related process.

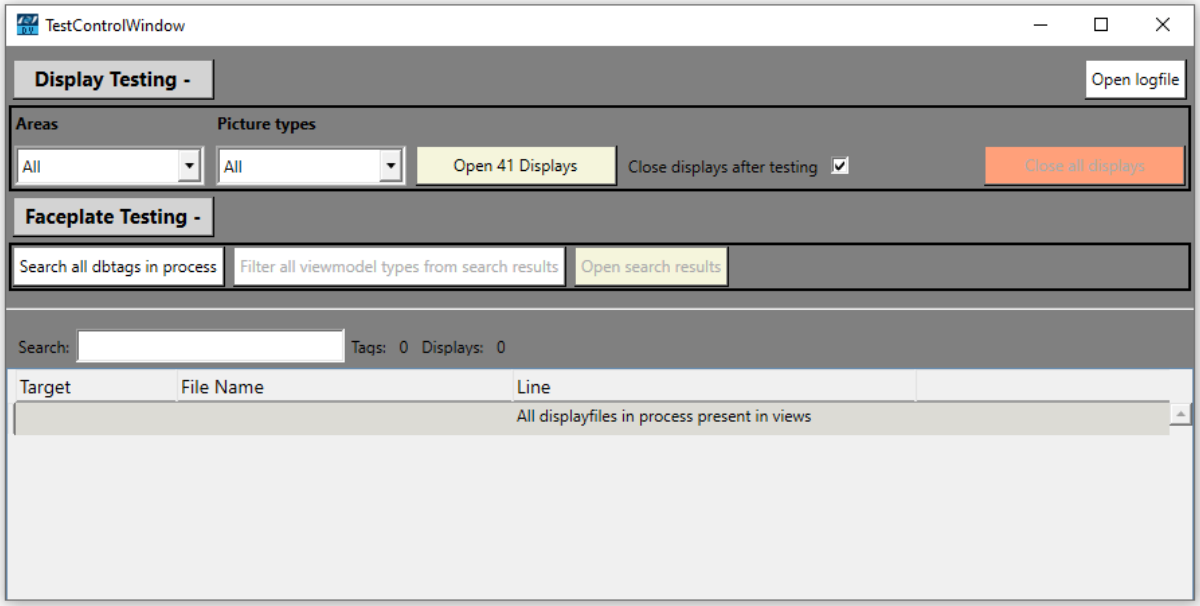

Figure 6: DV5 TestControlWindow

The TestControlWindow (Figure 6) can be opened from the DV5 system tray icon. When testing mode is enabled in the configuration, the option to open the TestControlWindow appears in the icon right click context menu. The TestControlWindow can be used by the developer or a designated QA person to test displays in DV5.

The TestControlWindow is split in two areas: the operating controls on the top of the window and the results area below the controls. The operating controls cover three main functionalities and are split as such: display testing controls, faceplate testing controls and search bar. The display testing controls include dropdown selections to filter based on areas and types along with methods to open and close selected displays. The faceplate testing controls allow searching the process for all existing DbTags, filtering them by types and opening all search results.

The results area is a datagrid that displays the result items. The result items can be either dbtags, displays, error messages or info messages. The search bar enables searching the views directory for keywords, such as dbtags or element names to find matches in the display files. If the keyword is found in a display, the corresponding display can be opened from the search result. If the keyword matches a dbtag, the corresponding faceplate for that dbtag can be opened from the search result. If multiple results are discovered, the results area can be scrolled down to view more results.

6.3.1 Preliminary steps for testing

Process file and views

When DV5 is launched it will alert if the file describing the process hierarchy (process.xml) is missing. If the file exists, no further validation is done. In order to ensure that the tested distribution contains the necessary files, further validation was necessary.

When TestControlWindow is initialized it reads the process file and generates a list of all the view files in the process. It then compares that list to the current views directory. If a display present in the process is missing from the directory, a search result item for that display is generated and displayed with a description informing it is missing from the views directory. This way the user knows if the distribution is whole and contains all the views required to be tested.

Unhandled Exceptions

Before display testing could be started, the applications exception handling needed to be improved, since some of the display exceptions caused DV5 to crash without logging the error. In order to catch and document exceptions in displays, the application needed more handling for uncaught exceptions. By adding a custom handler to Dispatcher.UnhandledException most of the fatal errors that would have previously crashed DV5 can be caught and logged. In case of an unhandled exception the user is prompted about the exception and logs are provided in notepad.

6.3.2 Testing displays

Display testing controls can be accessed in the TestControlWindow (Figure 7). The controls are accessible after expanding the Display Testing area.

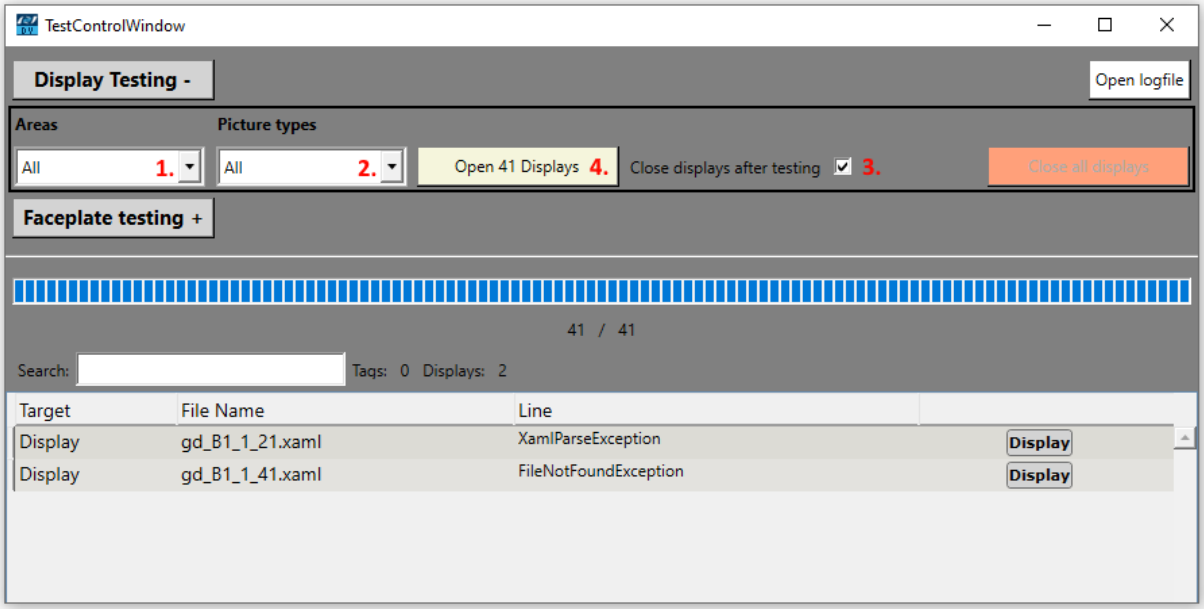

Figure 7: Testing displays with TestControlWindow

1. Select area.

   Each simulator is split into areas that contain pictures. From the dropdown the user can select either all areas or a specific area to test. When selected, the display count is updated on the open displays button.

2. Select picture types

   Each area lists the contained pictures under six picture types.

   These types are:

   - GdPictures
   - MwPictures
   - TrendPictures
   - IwPictures
   - OverviewPictures
   - ControlGroupPictures

   From the dropdown the user can select either all picture types or a specific picture type to be tested. When selected, the display count is updated on the open displays button.

3. Close displays selection

Before opening the selected displays, the user can select whether the displays should remain open or be closed periodically during testing. If this option is selected, windows are opened in sets of 10 and then closed to conserve memory, as it was discovered that opening too many displays will eventually crash the program.

4. Open displays

   Selected displays are opened one by one. A progress bar will update as displays are opened to notify the user of the test progress.

5. Testing results

   After all the displays are opened, any errors found in displays are displayed in the results grid. Each window opening command is logged and the stack trace for errors can be viewed in the logfile.

6.3.3 Testing faceplates

Faceplate testing controls can be accessed in the TestControlWindow (Figure 8). The controls are accessible after expanding the Faceplate Testing area. Faceplate testing is intended for manually testing faceplate functionality and to support development work, since previously operating all different faceplate types required lots of manual inputs and knowledge of the simulator to be able to find all different types of faceplates.

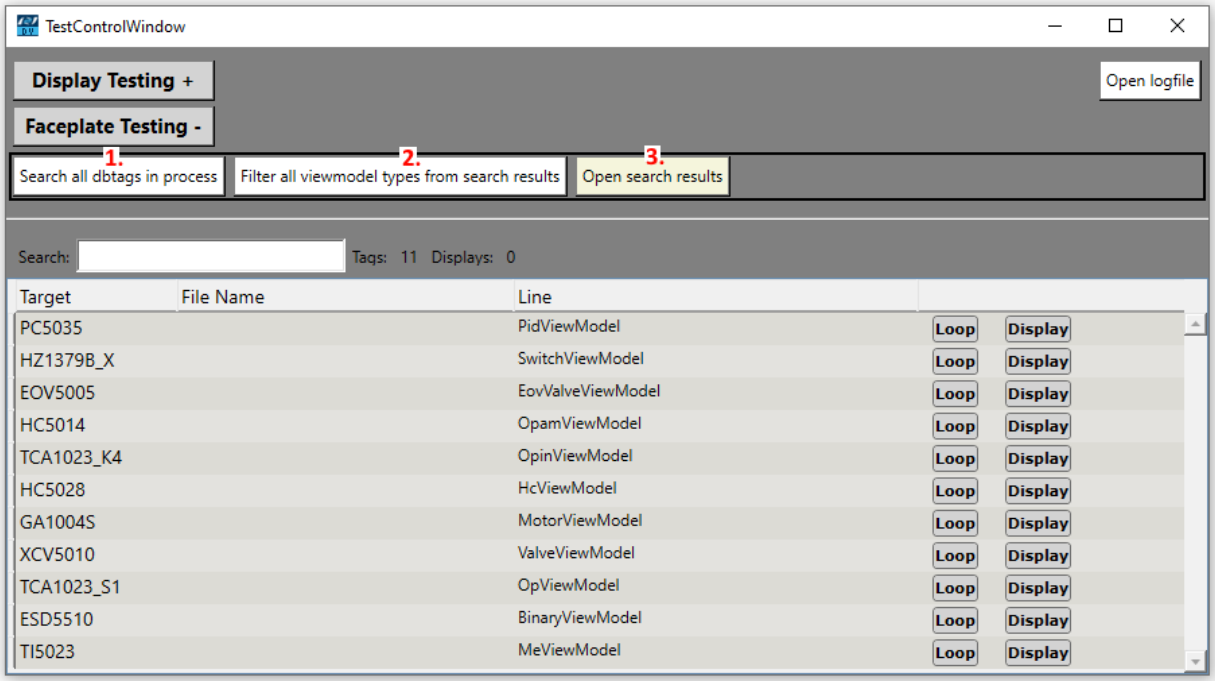

Figure 8: Testing faceplates with TestControlWindow

1. Search all DbTags

   The search function is used on all displays to fetch all dbtags in process. Each DbTag is only listed once.

2. Filter viewmodel types

   A DbTag representing each viewmodel type is selected at random from the results. The filtered results are displayed.

3. Open search results

   Each result is opened into a different type of faceplate window.

6.3.4 Technical implementation

TestControlWindow was implemented in the DV5 BaseLib, a common library shared by all DV5 projects. TestControlViewModel was extended from AppSearchViewModel to utilise the existing search functionalities in discovering and enumerating windows used in the open simulator. TestControlWindow uses the search functionality to list all displays and faceplates and uses the WindowFactory and FaceplateFactory to open the desired results.

In order to facilitate mass opening of displays, MemoizedXamlReader in BaseLib had to be modified so that defective displays wouldn't cause user interruptions. This functionality was put behind a configurability option "EnableTestingMode". When testing this configuration key needs to be set to either true or false (defaults to false) to access the testing features. When set to true, the TestControlWindow can be opened from the system tray icon via right click context menu.
When TestControlWindow is opened, it initializes a new TestControlWindowViewModel as its DataContext. TestControlWindowViewModel contains the business logic to test displays and faceplates. Display testing can be done without an active database as it only requires the file containing the process hierarchy (process.xml) and the display files. When doing faceplate testing, an active database connection is required as the viewmodel types are fetched via UaConnection.GetTypeDefinitionIdentifiers().

6.4 Evaluation of results

Smoke testing was done using the developed tool to mass open display sets from different simulator display repositories and results of findings were collected.

6.4.1 Mass display testing

Using the developed tools, mass display testing was performed manually on all available display repositories with DV5 DNA flavour and the results were documented. For most repositories testing took between five to fifteen minutes, depending on the intensity of detailed picture windows. For VY2, the simulator with the most displays, the process took significantly longer, varying around forty to sixty minutes, due to the high number of displays and the effects of improper memory management and memory not being released fast enough.
Among the results, some displays were found missing from the repositories, along with some errors in process validation.

- VY2
    - 945 displays
    - 2 errors (FileNotFoundException)
        - gd_B2_50_31_6.xaml

        - gd_B3_70_31_1.xaml
- REF3
    - 191 Displays
    - No errors
- RT3
    - 243 Displays
    - No errors
- Edupoli
    - 11 Displays
    - 1 Error (FileNotFoundException)
        - gd_T1_0_1.xaml
- Gen
    - 41 Displays
    - No Errors
- Rdam
    - 323 Displays
    - No errors
- Sing old displays
    - 381 Displays
    - 24 errors (XamlParseException)

      This was an early conversion batch on local test environment and the errors were later fixed in the following revisions of display conversion and construction
- Sing 2021 display conversion
    - 237 Displays
    - 1 Error (FileNotFoundException)
        - gd_B1_20_41_04_01.xaml
- Sing new displays (before july 2022)
    - 397 Displays
    - Anomaly

      When TestControlWindow is opened and views directory contents are compared to views in process, one display (SimuSws_Local_Panels.xaml) is determined to be missing. However, when all displays are opened, this display is opened along with the

rest. This turned out to be the result of display validation being case sensitive. The filename for was capitalised as SimuSws_Local_Panels.xaml in the process file but as SimuSws_Local_panels.xaml in the views directory. Case sensitivity should be considered during display construction and when handling displays in the code.

- JTER
    - 27 Displays
    - No errors in displays, but process validation failed due to missing parameter in process.xml (IsTeacherArea)
- CLG
    - 163 Displays
    - 2 Errors (FileNotFoundException)
        - mw_A1_SEQ6014_CST.xaml
        - Placeholder.xaml

Of the eleven tested repositories only four had no errors found in the displays or process file. The other seven had some errors such as missing display files or errors within the process file. This was discovered using the shallow smoke testing of mass opening the displays, which calls that further attention should be drawn to testing the repositories in more detail. The process of constructing the displays should also be refined to include smoke testing displays to ensure these errors are caught. The findings should determine that the tool can be evaluated as being functional, as it has helped discover unknown anomalies in most of the display repositories.

6.4.2 Performance and memory usage

Upon testing mass opening displays in DV5 it has become apparent that resources are not released properly after windows are closed. While garbage collection will eventually alleviate the issue, proper disposal should be implemented, since working with multiple displays will end up taking unnecessary amounts of memory.

In order to perform some rudimentary performance smoke testing, some examples were collected from different display sets to try to document the issues.

Object Type	Count Diff.	Size Diff. (Bytes)	Inclusive Size Diff. (Bytes)	Count	Size (Bytes)	Inclusive Size (Bytes)
PropertyChangedEventHandler	+11 127	+746 000	+249 239 584	11 197	751 328	257 758 000
EventHandler	+23 383	+1 549 448	+245 439 744	24 481	1 621 808	245 998 328
DV5.ViewModels.PictureWindowViewModel	+41	+6 560	+122 005 312	42	6 720	122 230 312
DV5.Views.Controls.PictureContent	+41	+29 288	+121 964 544	42	30 000	122 188 128
DV5.Views.Windows.PictureWindow	+36	+80 400	+106 617 416	37	82 824	106 831 232
DispatcherOperation	+2 928	+445 184	+94 220 720	2 937	446 536	94 226 112
List<AutomationPeer>	+2 585	+281 432	+69 207 712	2 664	289 792	69 374 536
UserControlAutomationPeer	+1 597	+229 968	+63 909 920	1 613	232 272	63 912 832
ScrollViewer	+76	+82 424	+53 085 376	84	91 432	53 207 616
ScrollViewerAutomationPeer	+72	+10 368	+52 077 080	75	10 800	52 078 688
PriorityItem<DispatcherOperation>	+2 928	+187 392	+47 193 760	2 937	187 968	47 196 912
Action	+2 813	+180 056	+45 944 352	2 825	181 240	45 946 200
List<DV5BaseLib.Factories.ViewModelFactory+ViewModelRequest>	+32	+8 456	+35 993 056	33	8 552	35 993 152
DV5BaseLib.Factories.ViewModelFactory+<>c__DisplayClass23_0	+32	+768	+35 992 832	32	768	35 992 832
DV5BaseLib.Factories.ViewModelFactory+ViewModelRequest	+677	+39 400	+35 984 600	677	39 400	35 984 600
Grid	+5 598	+2 770 176	+31 039 776	5 705	2 829 080	37 036 240
DynamicBlockLib.Views.Blocks.BinaryText	+351	+633 440	+31 011 632	351	633 440	31 011 632
List<DV5BaseLib.Lib.ValueReaders.ValueReader>	+1 217	+317 632	+28 619 136	1 218	317 728	28 620 480
UIElementCollection	+6 237	+249 480	+26 297 824	6 395	255 800	32 329 976
VisualCollection	+6 604	+539 104	+26 200 656	6 787	554 792	32 227 904

Figure 9: Managed memory, DV5.exe (OTS Gen) 20 items sorted by inclusive size diff

In the example in Figure 9, displays from the OTS Gen repository were opened and breakpoints were set before and after to document the memory overhead after closing all the displays. The objects are sorted by inclusive size dif which can be used to determine what objects are taking up the most memory resources. The inclusive size dif indicates that the propertychanged handlers are taking up large amounts of memory, likely caused by improper disposal of handlers.

Object Type	Count Diff.	Size Diff. (Bytes)	Inclusive Size Diff. (Bytes)	Count	Size (Bytes)	Inclusive Size (Bytes)
DispatcherOperation	+1 288	+160 456	+618 867 968	1 295	161 544	618 872 632
EventHandler	+181 312	+11 827 496	+469 898 832	182 071	11 877 136	470 478 440
PropertyChangedEventHandler	+34 593	+2 215 416	+466 530 168	34 631	2 218 512	467 124 448
PriorityItem<DispatcherOperation>	+1 288	+82 432	+309 469 096	1 295	82 880	309 472 120
DV5.Views.Windows.PictureWindow	+83	+169 944	+265 533 016	84	172 368	265 776 008
DV5.Views.Controls.PictureContent	+243	+149 752	+237 674 960	244	150 464	237 927 720
DV5.ViewModels.PictureWindowViewModel	+243	+38 880	+232 917 040	244	39 040	233 171 216
UserControlAutomationPeer	+27 499	+3 959 944	+219 187 816	27 506	3 960 952	219 189 336
List<AutomationPeer>	+24 675	+1 641 424	+214 301 968	24 746	1 648 784	214 464 584
Action	+454	+28 816	+144 076 248	463	29 632	144 077 432
DV5BaseLib.Factories.ViewModelFactory+<>c__DisplayClass23_0	+440	+10 560	+144 050 848	440	10 560	144 050 848
List<DV5BaseLib.Factories.ViewModelFactory+ViewModelRequest>	+440	+107 488	+144 040 856	441	107 584	144 040 952
DV5BaseLib.Factories.ViewModelFactory+ViewModelRequest	+10 043	+655 312	+143 934 408	10 043	655 312	143 934 408
UIElementCollection	+68 608	+2 744 320	+138 735 480	68 694	2 747 760	138 918 144
VisualCollection	+70 666	+3 820 016	+138 104 808	70 773	3 829 232	138 285 224
DV5.Views.Windows.MonitoringWindow	+160	+288 264	+133 738 984	160	288 264	133 738 984
Action<DV5BaseLib.ViewModels.UaViewModel>	+10 043	+642 752	+133 010 216	10 043	642 752	133 010 216
DV5BaseLib.Factories.ViewModelFactory+<>c__DisplayClass9_0<DV5BaseLib...	+10 043	+241 032	+132 805 224	10 043	241 032	132 805 224
Action<DV5BaseLib.ViewModels.UaControlViewModel>	+10 043	+642 752	+132 728 352	10 043	642 752	132 728 352
DV5BaseLib.Views.UaUserControlBase+<>c__DisplayClass26_0	+10 043	+241 032	+132 523 360	10 043	241 032	132 523 360

Figure 10: Another example of managed memory in DV5.exe (RT3 Displays) 20 items sorted by inclusive size diff

Likewise in the example in Figure 10 the DispatcherOpertations and EventHandlers are taking up large amounts of memory even after displays have been closed. This issue causes DV5 to slow down unnecessarily over long sessions or when operations are performed in quick succession.
Opening large amounts of displays can be used as means of load testing the system and can help determine if the system performs well enough under stress, and can help point out what is causing memory issues in the system.

6.5 Limitations and further development ideas

Over the course of creating the smoke testing implementation for display opening for DV5, some issues were discovered. Looking into these limitations and considering how to further develop smoke testing for DV5 can help create more stability in the builds.

- **GUI smoke testing**
  The current smoke testing implementation was made to use the existing logic and factories as if they were called by the GUI. The testing however is not proper end to end, since the GUI is effectively bypassed and any issues would go unnoticed. In the future, proper GUI testing may be implemented to take account of the real end user perspective.

- **Search function and process validation**
  Sometimes a view file can be in the views folder but not present in the process file, making it visible to the search function, but invisible in the simulator. Also when the view directory is scanned, the contents are checked against what exists in the process file, not the other way around. Therefore some views can exist in the views directory but not be present in the process file. This was considered when developing, but was left since the integrity of the process file should primarily be ensured in the display creation or conversion, not in DV5. The search function could however be modified to ignore view files not included in the process file. Additionally when the program is started, process.xml undergoes a validation process that may be outdated and require refining. For example Duplicate identifier in process.xml leads to no errors, instead the second instance will simply not be loaded into the process.

- **Separate test solution**

  This implementation was made to be included in the DV5 solution, partly due to ease of access, so that a potential tester wouldn't need additional tools to perform rudimentary smoke testing. Testing mode can simply be enabled from the configs. A separate testing solution would maybe be helpful however, when more test scenarios are included. One would be required to further automate the testing as well.

- **Cover more flavours**

  The current iteration of the TestControlWindow can only be operated in the DV5 DNA solution, on any of the DNA themes. Further development is required to make it functional with other DV5 solutions.

Overall the work provides a tool that is helpful for smoke testing a specific functionality of DV5. It can be used by both developers and simulator engineers since it can be used with limited technical knowledge of the software and doesn't require a tailored testing environment, but rather runs in any working simulator. It however only functions in one DV5 flavour currently and should be further developed to fit other DV5 flavours as well. It also lacks full automation and needs to be run manually from the application. The results for DV5 DNA testing show that it can be useful for testing displays especially during display construction, but more test cases need to be implemented before the stability of the entire application can be judged between builds.

7 Discussion

In the context of this thesis, a smoke testing tool was made for mass opening displays and faceplates in DV5. This tool, “TestControlWindow” bypasses GUI to access existing logic and factories to perform smoke testing on displays and faceplates. The tool was used to perform testing on eleven display repositories, with displays from different DNA simulators. From these repositories, only four proved to have no faults, while the other seven had faults ranging from missing displays, to minor errors in process validation. In addition to the discovered faults, the testing helped demonstrate performance issues tied to DV5 displays, and may be useful when conducting further performance analysis for DV5. These results can also help demonstrate how shallow testing can be used to discover faults in core functionalities of the program, such as display opening in this case.

These smoke testing scenarios could be implemented for other core functionalities in DV5 as well. Smoke testing can help test large amounts of code with simple test cases, but should be executed alongside proper end to end testing to gain the most value. Running smoke tests daily on new projects or weekly for projects in maintenance can help identify issues early in development. Smoke testing displays during display construction for new simulators would benefit the construction work since faults would be discovered earlier.

In order to further improve DV5 stability, testing should be planned and executed diligently and appropriately and by both developers during development and designated QA before releases. The testing sessions should be planned ahead and the test build should be determined stable before testing is conducted.

References

Chauhan, V. K. (2014). Smoke testing. International Journal of Scientific and Research Publications, 4(1), 2250-3153.

Fairbanks, G. (2010). Just enough software architecture: a risk-driven approach. Marshall & Brainerd.

Hamilton, T. (2022). Sanity testing vs. Smoke Testing – difference between them. Guru99. Retrieved May 19, 2022, from https://www.guru99.com/smoke-sanity-testing.html

Hevner, A. R., March, S. T., Park, J., & Ram, S. (2004). Design science in information systems research. MIS quarterly, 75-105.

Homès, B. (2013). Fundamentals of software testing. John Wiley & Sons.

Hooda, I., & Chhillar, R. S. (2015). Software test process, testing types and techniques. International Journal of Computer Applications, 111(13).

ISTQB Glossary. (2022). Retrieved November 10, 2022, from https://glossary.istqb.org/

Johannesson, P., & Perjons, E. (2014). An introduction to design science (Vol. 10, pp. 978-3). Cham: Springer.

Khan, R., & Amjad, M. (2016, April). Smoke testing of web application based on ALM tool. In 2016 International Conference on Computing, Communication and Automation (ICCCA) (pp. 862-866). IEEE.

McConnell, S. (1996). Daily build and smoke test. IEEE software, 13(4), 144.

Memon, A., Banerjee, I., Hashmi, N., & Nagarajan, A. (2003, September). DART: a framework for regression testing" nightly/daily builds" of GUI applications. In International Conference on Software Maintenance, 2003. ICSM 2003. Proceedings. (pp. 410-419). IEEE.

Memon, A. M., & Xie, Q. (2004, September). Empirical evaluation of the fault-detection effectiveness of smoke regression test cases for gui-based software. In 20th IEEE International Conference on Software Maintenance, 2004. Proceedings. (pp. 8-17). IEEE.

Memon, A., Nagarajan, A., & Xie, Q. (2005). Automating regression testing for evolving GUI software. Journal of Software Maintenance and Evolution: Research and Practice, 17(1), 27-64.

Novak, J. (2010). Game development essentials: game QA and testing. Delmar Cengage Learning.

Pressman, R. S. (2005). Software engineering: a practitioner's approach. Palgrave macmillan.

Schultz, C. P., & Bryant, R. D. (2016). Game testing: All in one. Stylus Publishing, LLC.

Spillner, A., & Linz, T. (2021). Software Testing Foundations: A Study Guide for the Certified Tester Exam-Foundation Level-ISTQB® Compliant. Dpunkt. Verlag.