Abstract

Automated software testing is often very helpful when performing functionality testing. It makes it possible to have a lot of user actions performed within the application without the need for a human interaction. But would it be possible to extend the behavior of regular functionality testing scripts and use them for performance testing? This way we could have regular application usage testing performed automatically during longer runs as well as investigate how well the application performs over time.

This report presents the process of making test automation scripts run in a manner that makes it possible to analyze the tested application’s performance and limitations over time – not just its functionality. Additionally, a research is performed on how to choose proper test automation suites, appropriate tools and in which way to make it possible to efficiently gather performance related data during the test automation runs.

This work has been done at the company Tobii Technology and it is used to test a desktop application they develop, called Tobii Studio. As a result of our work, we have implemented a test suite that can run automated tests over a long period of time while monitoring runtime performance for both the application and the computer’s hardware. The resulting tests can be used repeatedly by Tobii to help them identify performance issues for common test cases, and newer versions of Tobii Studio can be tested in the future to verify that a certain level of performance is maintained. The designs of our tests are so general that it will be possible for Tobii to continue extending our suite with more functionality.
# Contents

Abstract .................................................................................................................. Error! Bookmark not defined.
Introduction ........................................................................................................ 5
Background ......................................................................................................... 6
Thesis goal definitions ....................................................................................... 6
General test automation background ............................................................... 7
Advantages and disadvantages of automated testing ..................................... 7
Costs of automated testing ............................................................................. 8
Project background .......................................................................................... 10
Tobii Studio ....................................................................................................... 10
How is Tobii Studio tested? ............................................................................... 13
Research ........................................................................................................... 14
Interviewing Tobii employees ........................................................................ 14
Interviewing developers ................................................................................ 14
Interviewing training and support ................................................................ 14
Interviewing testers and managers ................................................................. 14
Which parts of Tobii Studio should we create tests for? ............................... 15
Limiting areas of testing .................................................................................. 15
What kind of results should our testing obtain? .......................................... 16
Discovering what information is beneficial .................................................. 16
Gather data regularly or at events? ............................................................... 16
Generate raw data ............................................................................................. 17
Evaluating test automation tools .................................................................... 17
Test automation tool criteria provided by Tobii .......................................... 17
Methods and difficulties of comparing tools ............................................... 18
Evaluating runtime performance data gathering tools ............................... 19
Research workflow ......................................................................................... 19
Researching data gathering with TestComplete ........................................ 20
Researching profiling tools ......................................................................... 23
Researching monitoring tools ...................................................................... 24
Implementing tests ......................................................................................... 26
Test structure .................................................................................................. 26
Structure of existing test scripts ................................................................. 26
Planned structure for new test scripts .......................................................... 26
Introduction

*Tobii Technology* is a world leading manufacturer of *eye tracking* technology, based in Stockholm, Sweden. Eye tracking is used to determine where in 3D space a user is looking.

A lot of interesting things have been done with Tobii’s eye tracking technology. It has been used for research such as psychology and physiology studies about ocular movements, logical thinking, dyslexia and diagnosis of different diseases. Other applications of the technology are usability studies for websites, software and testing of advertisements. Eye tracking is also used as a means of controlling a computer; as an accessory for regular users or for handicapped users who cannot operate standard computers. Tobii have recently released the game *EyeAsteroids*, which is the first and only purely eye-controlled arcade game in the world.

Tobii produce their own eye tracking hardware and software. One of their software applications is called *Tobii Studio*, is used to is used for analyzing eye tracking data.

The main purpose of this thesis is to create a prototype for automated performance tests of the Tobii Studio application. However, an important part of the thesis has been to interview company employees in order to define the scope of the work and learn what needs to be done. This led us to perform research on suitable tools for test automation as well as tools for gathering runtime performance data during automated test runs.
Background

Thesis goal definitions

In this section we have basic definitions of goals with the thesis. These goals mostly concern what should have been achieved at the end of the thesis - not how our work process should look like on the way. This goes hand in hand with the general development and testing process at the company, because they encourage agile development. One way for them to make sure that we actually choose ways of working that generate good results in the end is to have us interview employees at the company and make those interviews help us define how we should proceed. We could choose how to approach different issues as long we were concerned about if it would benefit the company. And the interviews' main purpose would be to give us enough knowledge to be able to make such decisions on our own.

After discussions with Tobii we could reach conclusion regarding what the goals of the thesis in general should be. Below is a list of things that Tobii wanted us to achieve during the thesis.

- **Interview employees to define the different goals**
  Because of the fact that Tobii knows what they want to be done, but not exactly how we should do it, we need to spend time on figuring this out. A requirement in the process of researching this is to set up interviews with employees from all over the company. Approximately at least ten people should be interviewed. The goal of these interviews is to gather knowledge related to all the requirements below so that we can define our goals based on actual needs and knowledge at the company.

- **Create automated performance tests**
  We should create tests that are able to find out how much the Tobii Studio application can handle before crashing and freezing. These tests should be possible to run on different computers in order to make it possible to test application limitations on different hardware. They can be considered to be performance tests run in a limitations testing purpose.

- **Gather runtime performance data**
  When testing application limitations it is beneficial if the test provides information about what might have caused a potential malfunction. Therefore we need to find a way of collecting runtime performance data. We also need define what kinds of data are needed and helpful for the company. The results could be analyzed and used by Tobii to help extend system recommendations for their Tobii Studio customers depending on which kinds of tests the customer will be using. What kinds of data that would benefit the company is a research task - we need to find out what needs that exist and then design well fitting data gathering according to what we learn.

- **Present gathered runtime performance data in an easy way**
  It is not enough to just gather performance related data - we need to present it in a way that makes it easy to understand it and benefit from it. If the results from the tests are too complicated to work with, they will be worth nothing. Therefore we should not only focus on gathering the right types of data, we should also focus on the presentation of it and how it is stored.
• Design tests that test well fitting areas of the application
Many different kinds of activities can be performed in Tobii Studio, and there are many variables that potentially can affect the performance of these activities. It is not feasible to create automated tests for all possible use cases. What kinds of tests that we should create is a research task - we need to perform research and learn about problems, different possible solutions and then design well fitting testing functionality according to what we learn.

General test automation background
Successful test automation projects speeds up the execution of tests, as they can run unsupervised. However, automation can also complicate the development of the tests. Besides the initial investments of learning and creating the automation, continuous maintenance of the tests is also required [8]. It is possible that automating a test will result in a more cumbersome test process rather than a simplified one. As automation is a difficult process, it is recommended to evaluate the potential benefits and pitfalls before starting. [1]

Advantages and disadvantages of automated testing

Advantages
• Automation could be a motivation to run tests more often
• Reproducibility can increase test credibility and reliability, since you can run a scripted tests again in exactly the same way – unlike manual tests which aren’t recorded.
• Less time for supervising tests could leave more time for other activities; such as analyzing the test results
• Automatic logging is both time saving and prevents the uncertainty of manual logging
• Long tests can be performed without any manual work. In a fully automated test, no testers have to be present during the execution, which means that long tests can be run during odd hours when staff might not be available.
• The ability to execute tests without present staff could also means that it’s possible to schedule tests to run during nights and weekends when more hardware is available than during office hours.
• Could improve test coverage [1]
• Test credibility could be improved [1]

Disadvantages
• The initial time it takes to create the automated tests could mean that the the production of test results is delayed compared to manual testing. The idea that bugs and ther issues should be found as early as possible in a development process is considered an axiom within software testing. [11]
• Errors might not be detected until the entire test is run, as there is no one supervising the test
• Difficulties in finding the correct test tool. Certain lacking features or other problems with the tool's design might not be visible to the testers until they've reached a certain level of experience of using the tool. [2]
• Tests in general, and particularly automated tests, are easily made obsolete by changes in the tested application. [6]
• Unexpected errors and failures can occur during testing. The automation has to be designed in such a way that errors are handled and testing proceeds in a desired manor. Failures have to be handled in a way that makes it possible to understand why the test failed.

• "Highly repeatable testing can actually minimize the chance of discovering all the important problems, for the same reason that stepping in someone else’s foot-prints minimizes the chance of being blown up by a land mine." [12]

### Costs of automated testing

Costs are related to:

- The number of times the tests are executed. (The initial automation effort does not pay off until the tests have been run a few times.)
- Accuracy and trustworthiness of the results.
- The required maintenance of the automated tests to keep up with changes in the tested application and the environment.

The startup investment for static tests* is low compared to random tests* and model-based testing*. (*See definitions in section [Types of automation tests]) However contiguous investments are needed as static tests need to be manually updated as the tested software changes and as other functionality needs to be tested. Random and model based tests a higher startup investment, but can be updated automatically so the continuous costs are lower. [11]

### Cost analysis

Test automation is not always an obviously better alternative to use than manual testing. Even though it may seem nice to have tests run on their own, there are a lot of factors that can cause the use of automated tests to exceed the costs of regular manual tests.

A common case is that automated testing will cost more in the beginning, during its implementation, and then have reduced execution costs over time compared to manual testing. Therefore the major challenge in this regard is to make sure that the use of automated tests is effective enough to be beneficial in the long run. [4]

Automated tests can affect costs in many different ways. For example, it can be possible to reduce staff involvement during testing by having automated tests, which would cause savings compared to manually running the tests. However, the automated tests’ outcome may require a larger staff involvement to process and analyze, thereby increasing the costs. [5]

Hoffman divides the cost types of automated testing into two segments:

- **Fixed costs**
  Consists of elements that are not affected by the number of times they are used. For example hardware, testing software licenses, automation environment implementation and maintenance, and training. [5]

- **Variable costs**
  Has the ability to increase or decrease depending on how many tests that are implemented, how many times the tests are run, how much maintenance tests need and how complex the test results are to analyze. [5]
Return on investment (ROI)

A common way of determining if the use of test automation is beneficial is to calculate the return of interest (ROI) value. This topic was broadly discussed in the publication X by Michael Kelly [4]. A classic way of doing this calculation is by dividing the net benefits of the automation effort by the cost of the automation effort. This can be expressed as:

\[
ROI = \frac{Benefit}{Cost}
\]

Kelly defines the cost as a set of different parameters (described below), and the benefit to be the result of calculating how much savings are given by automation over a given period of time.

**Cost of automation** = price of hardware + price of software + time to develop scripts + 
(time to maintain scripts * number of times scripts are executed) + 
(time to execute scripts * number of times scripts are executed)

The cost of manual test execution is defined in a similar manner:

\[
Cost\ of\ manual = time\ to\ develop\ test\ cases + 
(time\ to\ maintain\ test\ cases * number\ of\ times\ tests\ are\ executed) + 
(time\ to\ execute\ manual\ testing * number\ of\ times\ tests\ are\ executed)
\]

The definition of the benefit then becomes the following:

\[
Benefit = cost\ of\ manual - cost\ of\ automation
\]

**Break-even point**

With these formulas it is possible to calculate a break-even point; when does test automation start to pay off? Figure 1 illustrates a simple example of this. The initial cost of manual tests is lower, but as the cost per executed test is higher the cost grows for each number of executed tests, until it finally reaches a point where manual tests will cost more than automated tests.

![Testing cost graph example](image)
Is it really possible to compare manual testing to automated testing?

However, these formulas contain a couple of problems and incorrect assumptions that Kelly clarifies:

1. Automated testing cannot be compared to manual testing. They neither are the same nor perform the same testing tasks – therefore they do not provide the same information about the tested application. It lies in the nature of automated tests to be simpler than manual tests. Manual tests are more effective when it comes to complex test scenarios. This is because automated tests have to be repeatable and be compatible with the scripting language and environment being used, and therefore are categorized into certain test classes.

2. The purpose of manual tests often differs from the automated tests'. All automated tests cannot perform the same tasks as manual tests. And there are automated tests that would never be performed in a manual manner. Therefore it is not actually possible to compare the costs of an amount of test executions between the two types – since they do not perform the same tasks.

Analyzing our project costs

We have considered calculating the costs for the work done in this thesis project and then analyzing the ROI of the work done compared to what it could have cost if it was done manually. However, a few factors made us decide not to spend time on this.

First of all, we do not spend a constant regular time working with the test automation. Therefore it is very difficult to estimate how much time has been spent. Most importantly, however, is the fact that the kinds of tests that we create are only suitable for use in an automated testing environment. To do these kinds of tests manually is not an option, and we therefore find it even more difficult to try to evaluate how much time the automation is worth in manual hours.

Project background

Tobii Studio

This section is about Tobii Studio, the desktop application that is the target of our testing. Some parts of the report rely on the fact that the reader is fairly familiar with basic concepts of the application. Therefore we provide a brief terminology, and various descriptions about the application as a method of preparation.

Terminology

- **Project**
  Contains an arbitrary amount of tests and participants. Tobii Studio can have one project open at a time.

- **Test**
  The basic testing unit. A test contains an arbitrary amount of stimuli that are used during a recording. It also contains an arbitrary amount of participants and recordings.
• **Recording**
  A recording resides in a test and collects data about the test during a test run. A recording has a participant connected to it.

• **Participant**
  Participates in a test’s recording. Holds brief personal information about the test person. A participant can be used in multiple recordings for different tests in a project.

• **Stimulus**
  A media element type that can be put in a test. This is what a participant will be looking at during a test. These are the currently available stimulus types:
  - Instruction
  - Image
  - Video
  - Web
  - Screen recording
  - External video
  - Scene camera
  - Questionnaire
  - PDF element

**Description**
The description below is a quotation from the Tobii Studio manual and describes what Tobii Studio is and what it can do:

"Tobii Studio offers a comprehensive platform for the recording and analysis of eye gaze data, facilitating the interpretation of human behavior, consumer responses, and psychology. Combining an easy preparation for testing procedures and advanced tools for visualization and analysis, eye tracking data is easily processed for useful comparison, interpretation, and presentation. A broad range of studies is supported, from usability testing and market research to psychology and ocular-motor physiological experiments. The Tobii Studio’s™ intuitive workflow, along with its advanced analysis tools, allow for both large and small studies to be carried out in a timely and cost-efficient way, without the need for extensive training."

A typical usage of Tobii Studio is to either create a new project or open an existing one. Inside the project a new test is created or an existing test is selected. One or more stimulus items can be added to the test. It is possible to mix different types of stimulus in the same test. Before starting the recording it is needed to either create a new participant or use an already existing one. After the participant has been selected, the eye tracker needs to be calibrated to make sure that it can locate the user’s eyes. When the test is setup a recording is typically started. During the recording the stimuli items are shown to each participant, one at a time, and data is collected from the eye tracker. When the test is finished, it is possible to review and analyze the gathered results from existing recordings. The application provides several ways of analyzing the results, for example heat maps, areas of interest (AOI), statistics and charts.
Below are multiple lists of features and usage areas in Tobii Studio. They have been copied from the manual.

**Usability studies**
Tobii Studio is very well suited for evaluating user experience in regard to:
- Websites
- Software
- Computer games
- Interactive TV
- Handheld devices
- Other physical products

**Advertising testing**
Tobii Studio is ideal for testing advertising design on a variety of media:
- Packaging and retail shelf design and placement
- Web advertising
- TV commercials
- Print advertising, digitally scanned and presented on a monitor
- Print advertising, using the actual physical print

**Psychology & physiology research**
Tobii Studio is suitable for a wide range of psychological and physiological experiments, such as:
- Infant research
- General psychological response studies, including the use of scan-paths, gaze distribution, gaze
- response times, and manual response times
- Studies of autism, ADHD, and schizophrenia
- Reading studies
- Studies of ocular-motor behavior and vision deficiencies.

Figure 3: Example of the Design and Record tab in Tobii Studio. 7 movie stimuli are visible.

How is Tobii Studio tested?

Agile methods are used for development and testing. Development is performed in sprints. When the developers have finished a sprint, the QA division will begin testing the former sprint. There are several different types of tests. One important type is the one that contains all the scripts testing Tobii Studio functionality briefly and makes sure that everything works as expected.

During automated testing of the application no actual persons are used as participants – instead they are replaced with virtual participants and the gaze data to the eye tracker is generated through simulation. Tobii is currently testing Tobii Studio using the application TestComplete 8, developed by SmartBear. Using TestComplete, Tobii execute several of their automated test suites on a regular basis to test Tobii Studio. The tests are written as scripts in the programming language jScript.

Tests are selected within TestComplete and are then set up as scheduled jobs on a server and executed automatically when there is available time. The test results are automatically verified to indicate if tests passed or failed. Logging is also performed so that it is possible to review which actions were taken in a test. If a test fails a tester has the possibility to review the logs to find out where and when the problem occurred.
Research

Interviewing Tobii employees

An early part of our work was to conduct interviews with employees at Tobii. This was one of the project requirements - we needed to gain knowledge about different aspects of the situation. We received a list of suggested people to speak to. This list contained of about ten people and we planned to interview each one of them for about 30 minutes. During the interviewing process we received recommendations about other people to speak to. In the end we had interviewed more people than we knew about from the beginning, which proved to be a nice result.

The persons we interviewed were from different parts of the company. We focused more on some departments than others, but we did not exclude anyone that we still had questions for. Before we interviewed anyone we created an interview template of questions that we had as a base for each interview. We used several different templates depending on which department the interviewed person was coming from. In the later parts of our interviewing process we felt that we did not really need any template, since we were familiar with what questions that were appropriate and we could easily adapt our questions to fit the current interview situation.

Interviewing developers

During these interviews we received a lot of valuable information about the process of how Tobii Studio was designed which helped us get a general idea of which way we should approach our testing. We could ask them detailed questions about which features they would like to see us implement in our project. The developers thought that using a profiling tool during our tests would be very attractive. If problems are found during testing, having data from a profiler could provide more detailed information useful for developers. The developers also provided a lot of ideas for specific types of tests that we could create for certain areas of the software.

Interviewing training and support

The training and support divisions informed us on some of the more common requests from the customers using Tobii Studio, which provided us with some good goals to start focusing on. The support division thought that having tests that run for long periods of time would be a good idea; by stressing and finding the limitations of the software the results generated could help the support division to provide better information to users of the software. It could also help with keeping the system requirements updated.

Interviewing testers and managers

During interviews with the testers we got a better understanding of which approaches to testing would work best for Tobii and complement the existing tests the most.

Interviewing several departments and employees was very good for getting a general understanding of the software, situation at the company, what kinds of requests and expectations there were for testing. However we quickly realized that there were so many different perspectives and requests on what an automated test project could achieve that we had to narrow the scope of our thesis; if we decided to try every idea that was presented we would’ve never been able to finish in our limited time frame.
Together with the managers we agreed that we would only focus on certain areas of testing.

**Which parts of Tobii Studio should we create tests for?**

Because of the fact that Tobii Studio is a large application with a lot of different features there are huge amounts of things to create tests for. Here we focus on how we should limit our testing scope to testing cases that are beneficial for Tobii and of a well-fitting size. Our time spent on interviewing employees proved to be very useful, since it really helped us in understanding the issues of Tobii Studio and what would be fitting to test.

**Limiting areas of testing**

**Approach discussion**

After learning about how to possibly proceed we needed to decide more precisely which areas of the software that we should focus our testing process on. As we saw it, we had four possible approaches for limiting the testing area. These can roughly be summarized as:

1. Create a list of all the features and just pick enough list items to keep us busy during the entire project.
2. Find out which the highest prioritized features of the application to test are and focus our testing on them.
3. Find out which the most stable features of the application are and focus our testing on them.
4. Find out which features of the application that are often used by customers.

Neither Tobii nor we thought that it would be such a good idea to just pick items from a list and test them without any actual context or correspondence between them. To focus on testing prioritized features seemed more sensible. However, their current testing processes seemed to be good enough to manage this kind of testing already. Therefore we found an interesting contrast in the idea of testing areas that are considered to be working correctly - these areas would surely be able to withstand longer test runs. However, we did not feel that it would fulfill any real needs to test features that are considered fully working and without any real known issues.

The final approach, based on features that are both widely used by customers and that also are well fitting for long test runs, was considered a well fitting choice by us as well as people at Tobii. Results from work on that approach would surely be beneficial for Tobii, and we could easily find an appropriate amount of features to test.

**Approach selection and definition**

In compliance with Tobii we concluded that our work should be focused on approach number 4 from the list above; to learn which features that are both popular and have known performance related issues and focus our testing on those parts of Tobii Studio.

In addition to deciding which kinds of features we should test, we also needed to actually define what this specifically means. Based on what we learned through our interviews and what came up from discussions with Tobii, our final decision landed on testing of the recording functionality of Tobii Studio. It is a widely used feature that is well fitting for test automation and long runs. Our input to the recordings should be as simple as possible - tests with single stimulus types of a certain amount.
Tobii thought it would be a good idea for us to work incrementally in an agile manner, meaning that we should try to complete smaller increments at a time to make sure that we do not end up with a lot of loose ends that make the result unusable. We would start out with recordings of tests containing various amounts of images, and then progress to other media types and also possibly extend the testing to include additional things apart from only recordings. Due to our agile way of working, it was well fitting to define quite open goals with the primary goal of getting the basic functionality finished and possibly adapt it to something better along the way.

**What kind of results should our testing obtain?**

**Discovering what information is beneficial**

To be able to know what kind of results we should gather from our testing, we needed to find out what employees at Tobii actually are interested in knowing. We knew beforehand that we were likely to get different answers from different parts of the company, depending on what each employee is working with.

Based on our interview questions to the different employees we drew the following conclusions regarding what different departments on the company generally are interested in when it comes to performance testing on Tobii Studio:

- **Development**
  Detailed runtime performance data on how well Tobii Studio is running when it is used.

- **Testing**
  Information about how well Tobii Studio is working functionally as well as the response time when using different GUI components.

- **Support**
  Limitation numbers on long Tobii Studio can run and how many times certain features can be used before the program crashes or stops responding.

This is the knowledge we had quite early in the work process. At this point we were not ready to make any final decision regarding specific data to collect. We thought that it was enough to just pinpoint what kind of information that could be of use, and then use this knowledge for further and deeper investigations.

**Gather data regularly or at events?**

Something we discussed in the beginning of our work was with what sample method we should retrieve runtime performance data - should we sample at constant and regular intervals or should we do it more irregularly, for example at certain application events? We brought this up for discussion with different employees and they did not all share the same opinions of course.

We preferred the approach of sampling data at a constant and regular rate. We based this opinion on the fact that we planned on making it possible to change the sample interval, so it would always be possible to get a suitable granularity. Also, this approach guarantees that a data certain amount of data will be collected. In contrast, by using event based sample points data will be gathered slower if Tobii Studio starts running slower - leading to less data being collected. This could possibly lead to loss of important information. A regular sample rate may be able to display what happens between two events and possibly help finding the reason to why the application get decreased performance.
Additionally, it is easier to get reproducible results with regular intervals, since the data is always gathered at the same rate. It cannot be guaranteed that the results are reproducible when using the event based approach.

In the end, we ended up choosing to go with the regular intervals approach. We based this on discussions with the employees as well as our own opinions. We did not find any overwhelming argument for choosing an event based approach; it was easier to find advantages for the regular intervals approach at the same time as it did not have any disturbing disadvantages like the event based had.

Additionally we came up with the idea of logging events on our own. By using a regular interval data collecting method and also generating an event log with time stamps we would be able to see and know what happens at certain times during the measurements data. The response from Tobii employees to this idea was overall positive and we therefore decided to extend our selected functionality with this feature.

**Generate raw data**

We agreed with Tobii quite early that we should focus on gathering data that can be available in an independent raw text format without being dependent on a specific application to use the data. It would not be very agile or future development friendly to lock our implementation to a specific application that is needed to use the gathered data. To have some way of visualizing the data would probably be beneficial, but not to force the use of a specific application.

**Evaluating test automation tools**

As part of our research we were assigned the task of evaluating test automation tools on the market. There were two purposes of this; to find a suitable test automation tool for our work and also to see if we could find a tool that was worth to replace Tobii’s current tool with. Tobii currently used TestComplete 8, which is developed by SmartBear.

**Test automation tool criteria provided by Tobii**

During our process of searching for test automation tools we were given some constraints by Tobii regarding different aspects. Below is a list of the criteria we took into consideration.

- **Price**
  We were told to look for rather cheap solutions first and foremost. Buying a more expensive tool was not off the table. However an expensive tool would have to be very well motivated, due to the fact that buying expensive software licenses for the company is a costly procedure. We were given a certain pricing range that we should try to stay within.

- **Ease of use**
  It would be a huge advantage if the tool we chose could be used by the testers at Tobii, since our work might need to be updated in the future. And to have a complex tool would possibly make our work unusable in the future.

- **Desktop support**
  Tobii Studio is a desktop application and the test tool should therefore be built for testing such an application.

- **Framework support**
Tobii Studio is mainly built using C# in the .NET framework and therefore the testing tool needs to support testing such an application.

- **Operating system support**

  The tool must be able to run on different versions of Windows: XP, Vista and 7.

### Methods and difficulties of comparing tools

There are many possible approach methods when researching tools. You could read product descriptions, compare lists of features, read user reviews, look at the size of the market share, etc.

Hendrickson* recommends downloading trial versions of different tools and having an existing test case. With each tool you should then try to automate that case and evaluate how you experience the tools' ease of use. Test tool marketing and reviews are often focused on presenting the tool's general ease of use and GUI-features, but not what it can do in terms of programming. This is a reason that investigation of test tools based on reviews and feature lists is extremely difficult. If the programming lacks features that prevent you from completing your tests it might be something you don't find out until months later when you have started maintaining the scripts. [2]

### Progress of tool research

Here we present the work flow during our automation tool research.

Some advice on the methods of select a test tool are provided in the publication “Making the Right Choice - The features you need in a GUI test automation tool” by Elisabeth Hendrickson. Hendrickson recommends downloading some trial versions of tools and begin experimenting with them to get a general feel for the tool. We decided to go with this approach.

To begin with we had to decide which tools to try out. We started looking at lists of common test tools. To be able to narrow down the field we reviewed the lists of features, to be able to exclude certain tools that definitely did not have the features that were necessary. 

Lots of tools called "performance test tools" seemed to be aimed only at web and HTTP load testing and could easily be excluded from further investigation.

We could also discard a lot of tools that simply were too expensive. However finding out accurate prices of test tools proved to be a real challenge, as many of them simply do not have price quotes available on their websites. We had to look at test tool comparison articles to find out the costs, and sometimes we could not conclude the exact price of a tool.

Having narrowed down our field of possible tools we had to decide which tools to look more closely at first. We tried to get a picture of which tools had the biggest market share, but this also proved to be difficult.

We decided that market share should not be too much of a factor, since it is definitely not a guarantee for either quality or popularity, but can be caused by things like marketing and availability.

After we had decided on a few candidates that we wanted to try for ourselves, we started downloading some trial versions. However, as we were new to testing and automation in general and we found that trying to learn several new tools at once was an overly time consuming task. We felt that we only had time to test the more basic look and feel of the tool, and could not follow Hendrickson's advice about looking at the programming possibilities.
We decided that we should start looking more closely at the tool that the company Test Complete, which the company already had purchased licenses for. After spending a few days investigating these tools we thought they seemed quite easy to use and capable of providing us with what we needed to perform our tasks.

After interviewing key testing personnel at the company we found out that they didn't have a huge amount of knowledge or preferences about other automation or test tools than Test Complete.

We finally decided that:

- The existing tool, Test Complete, had such a big advantage in being well known among the testers at the company.
- To be able to motivate investing time and money in purchasing a new tool would require it to be a lot better than the existing tools.
- With our limited time frame we didn't have enough time to test other tools enough to conclude with certainty that we preferred them greatly over the existing tools.

Therefore our conclusion was to stick with the existing tool, Test Complete.

**Evaluating runtime performance data gathering tools**

Since a part of the goals of our testing has been set to gathering runtime performance data during test runs, we have created the need of finding a suitable tool for this purpose. The procedure of defining at which level we should gather data has been revised several times during the thesis.

When we were interviewing developers we learned that they would probably be the most interested in detailed runtime information. This would allow us to analyze memory leaks, poor resource management and other performance related issues. If this analysis was done by a profiling tool we would also be able to see exactly where in the program's code the issues exist. An advantage of using a profiling tool is that the usefulness of the tests and the number of times they could be run would increase greatly if they could produce results helpful for solving the issues, not just detecting when they occur.

In contrast to using external tools, we knew that it could also be possible to gather some sorts of data from within TestComplete by writing scripts that access certain system and process information.

**Research workflow**

Below is a diagram illustrating our workflow of the entire runtime performance data gathering tools research process. It simply shows graphically which decisions we made and what paths we have taken.

**Description:**

- We have enumerated all the steps taken by us chronologically. The first thing we did is of course at the start position and has the number 1 in its transition label. By following the transitions by each label's number it is possible to see in what order we worked.

- Since we have made several decisions regarding which way to go, some crossings exist in the diagram. In combination with the notation mentioned in the previous bullet, the path that we took is highlighted in green. The paths that were not taken are in black.
- The rhombs/diamonds represent decisions. Each decision has a corresponding question. At a rhomb, the question is put on the incoming transition label, and the possible answers to the questions are outgoing from the rhomb. We wanted to put the questions inside the rhombs, but this was not supported by the tool.

![Diagram of research workflow](image)

**Researching data gathering with TestComplete**

To have the test scripts not only test Tobii Studio, but also to manage the runtime performance data gathering was an attractive option to check out. Therefore we started to look into which possibilities we had for creating such functionality within our scripts. TestComplete 8 makes it possible to create process objects that can access a variety of information about a process. [http://smartbear.com/support/viewarticle/12761/](http://smartbear.com/support/viewarticle/12761/). We wanted to try to use this functionality to gather data about the Tobii Studio process' processor and memory usage. These two properties are describes as:

- Returns the approximate percentage of CPU time spent on executing the process. [http://smartbear.com/support/viewarticle/17478/](http://smartbear.com/support/viewarticle/17478/)
- Returns the total amount of memory (in kilobytes) allocated by the process. [http://smartbear.com/support/viewarticle/16353/](http://smartbear.com/support/viewarticle/16353/)

After we had concluded that it was possible to access this information about the Tobii Studio process, we just needed to actually make sure that the data was collected regularly. Since we had decided to have constant sample points and not base our gathering on events in the application, we thought that a classic timer functionality could be a good way of achieving this.

**Timer functionality**

In TestComplete it is possible to create timers that fire on a regular basis, based on the interval that they are set to [http://smartbear.com/support/viewarticle/13306/](http://smartbear.com/support/viewarticle/13306/). We thought that we should be able to define a timer that fires at a given interval and each time lets the callback function gather data about the Tobii Studio process.
We started out on a basic level just to make sure that our timer worked as we wanted it to. At this point the scripts were very simplified; we just started Tobii Studio without actually doing anything in the application with our scripts. We only had it running in order to be able to gather process data about it. Then we had our timer firing regularly and the callback function was gathering processor and memory information about the Tobii Studio process.

Our simple tests worked well enough for us to think it was worth proceeding. We did this by incorporating the data gathering in some already existing test scripts that used Tobii Studio to a much larger extent than we had during our initial timer test runs. This proved to be devastating to our timer’s firing accuracy. It seemed like the operating system was too busy working with Tobii Studio and TestComplete, because our timer started to fire extremely irregularly - no way near the expected timings.

To verify that the timers actually were irregular, we created a few experiment scripts. We repeatedly executed a few tests designed to fire timers regularly each 10th second over a period of a couple of minutes. A standard result was that timers hardly never fired with exactly 10 seconds in between. Usually a few timers fired with roughly 8-12 seconds in between. Occasionally up to 4 timers fired at once during the same 10 second interval, and other times up to 40 seconds would pass without a single timer firing.

Obviously we could not proceed with these kinds of results - they felt like they were totally unpredictable. And the major reason for choosing to work with timers in the first place was to get regular and predictable sample intervals.

In an attempt to resolve the situation we managed to receive help from the TestComplete support community. The help we got from the community was not official from SmartBear, but it helped us pinpoint the issues. A conclusion we reached was that Windows 7, due to the fact that it is not a real-time operating system, does not guarantee that a timer always gets to fire its event when it is supposed to. However, it is likely that the timer will be able to fire its events once the operating system finds time to let it do its work.

Additionally, a part of the reason for getting such bad timer performance during heavy system load could depend on the fact that TestComplete’s timers are simply wrappers to the operating system's timers and that it takes a lot of time to manage this functionality from TestComplete’s level. Since we have previous knowledge about how the operating system’s scheduler generally works, this seemed like a valid conclusion and we felt that it was too troublesome to proceed down this road. Especially since we could not be sure that we could make it work out in the end. We therefore decided to drop this and try something else.

WMI objects

Our conclusion regarding the use of timers and process objects was that it was not possible to get accurate sample intervals and that it therefore was not possible to gather data regularly. During our contact with the support community we received a tip about the use of WMI objects as an alternative to timers. WMI objects are provided by the operating system which makes it possible for us to use various processes from within the script. WMI is described as the following by MSDN:

"Windows Management Instrumentation (WMI) is the Microsoft implementation of Web-based Enterprise Management (WBEM), which is an industry initiative to develop a standard technology for accessing management information in an enterprise environment. WMI uses the Common Information Model (CIM) industry standard to represent systems, applications, networks, devices, and other managed components." [9]
What we learned was that it is possible to use Windows' Performance Monitor via a WMI object in a script, and that way gather data via Performance Monitor. This seemed like a nice alternative because we could use a dedicated application for gathering data, while still maintaining the control inside our scripts.

However, we discovered a major drawback with this approach - we would need to explicitly tell the WMI object of the Performance Monitor when to refresh and sample data. This lead us back to our previous issue about timers; that we actually could not create such a behavior that regularly does a certain action - in this instance it would be to call the refresh function of the WMI object. Therefore we reached the conclusion that the WMI object approach was likely to fail as well.

Asynchronous function calls and stopwatches

By knowing that TestComplete's timers were not fit for us we thought of new ways to gather data regularly. We knew that TestComplete's stopwatch functionality was likely to work because they are widely used in already existing scripts at Tobii. Stopwatch objects are used in such a way that you tell it how long it should wait when it is started. Once it has been started, it will hold further script execution until it has waited for the given amount of time. In other words, it acts as a script execution pause function.

After discussing how stopwatches could possibly replace a timer's functionality we came up with the idea to make an asynchronous function call. When we looked in the TestComplete manual, it also proved to be possible to perform such calls. That article describes the functionality as the following:

"An alternative approach is to call the method or property that pauses the script execution asynchronously. This means that after a call to the method (property) the script engine continues executing further script statements and does not wait until this method returns the execution control (or until the get or set methods of the property return)."

We thought that we might be able to trap the asynchronous function call inside the called function in order to create a sort of pseudo threaded call. By not letting that call return, it would resemble the creation of a new thread since the calling function would continue executing its routines without being concerned about the called function's return state.

The point of trapping the asynchronous call inside its function was to make it use a stopwatch and on a regular basis gather data by either accessing process information or by using a WMI object to use Performance Monitor. We could stop the function from returning by locking the control flow inside a regular loop with a condition that would always be true.

Our algorithm design was simple and we thought that it would be easy to implement. But we go trouble once again. We discovered that it did not seem to be possible to make asynchronous calls to other script unit functions - only to application object methods. Because we wanted our call to be to a script function, not an application object, we reached the conclusion that we had to drop this approach.
Researching profiling tools

Based on what we learned about the developers’ needs we started looking into the market of profiling tools. One of the developers had performed a smaller investigation earlier, where the person looked into three different profilers. However, the developer had not decided which tool was the most fitting for their cause. Instead we used this knowledge to have these three tools as candidates to our own work. These tools were:

- ANTS Performance Profiler, from Red Gate.
- ANTS Memory Profiler, from Red Gate.
- .NET Memory Profiler, from SciTech.

Additionally, we added our own candidate to the list; AQTime. It is developed by SmartBear, the same company that makes TestComplete. It seemed like an interesting alternative, since it possibly could lead to good integration with the existing test suite.

At this stage we had four tools to look into and learn about. Due to the fact that Tobii earlier had found interest in three of these tools we were satisfied with our setup.

Our first procedure was to look at the list of features in each application and try to exclude some of them early on. We thought that it seemed like a good way to start out, instead of downloading all of them immediately and spend a lot of time on testing them manually. After looking into the programs’ features we narrowed down our search to AQTime and ANTS Performance Profiler. This was based on the following:

- AQTime comes from SmartBear. To have two products from the same developer would probably lead to a good integration and also make it easier for company employees to use it, since the interface should be familiar. Also, its features seemed well fitting for our testing.

- ANTS Performance Profiler provides nice features for keeping track of what happens on the Garbage Collector heaps, which we knew was a very attractive features to some of the developers at Tobii.

- ANTS Memory Profiler did not seem to provide as beneficial features as ANTS Performance Profiler. We therefore excluded this candidate and put our focus on the performance profiler instead.

- .NET Memory Profiler did not have any features that were extraordinary or good enough to conquer out any of the other competitors. Therefore we excluded this one, since the others seemed to offer more.

By now we had concluded that both AQTime and ANTS Performance Profiler were two good profiling candidates. However, they had quite different work approaches and it was not clear to us which one was actually the better; both in terms of our use of them as well as future usability by Tobii’s employees. The research about how they work and what they do also made us realize that by gathering runtime performance data this way we would be working at very a low level in terms of limitations testing; would it really be needed and necessary for us to gather so much information about how the code runs and performs?

The developers' general opinion was that profilers seemed like a good way to go. On the other hand, testers and other employees thought that it might be too complex and time consuming to study the program in such detail when performing this kind of tests. We reached the conclusion that it would probably we wiser to narrow our work path down by not using profilers in our work, and instead focus on some more high level monitoring. However, our acquired knowledge
about profilers would still be of use to Tobii. We had also become more experienced in the area of runtime performance data gathering tools and had a greater understanding of what kind of functionality and level of detail we really needed.

Even though we did not feel that profilers would be of great use to our work, we left a door open for having a profiler run in the background of our tests. The purpose of this was not to actually have us analyze the data, but to provide the developers with detailed information about what happens in the program during our automated test runs. It was still not clear how much this would affect the overall performance on the testing system though. In case we later found that we have time to check this out, it would be something we could do in order to extend our measurements data.

Researching monitoring tools

After we reached the conclusion that it would be wise to narrow down our work to exclude code level analysis we started looking for other tools; in the monitoring area instead of profiling. Monitoring tools gather data about running processes and possibly some computer hardware, instead of digging into the tested application and analyzing it on code level, as profiling tools do. Several people at the company recommended us to look into Windows 7's own program called Performance Monitor. We had used this program in our past and were fairly familiar with it. Since it would be free of charge to use Performance Monitor, due to the fact that Windows 7 is the primary operating system used at Tobii, we decided to only look for free programs. This decision made us focus on the open source market.

We found a few interesting programs. Unfortunately most of them were Java-based and therefore did not provide us with any good support for monitoring .NET applications such as Tobii Studio. Additionally, lack of documentation, support and small user communities made it difficult for us to try out the open source monitors thoroughly - once we got stuck we could not really get through and proceed any further. Because of this our research in this area did not last very long, and instead we decided to go with Windows' Performance Monitor since it seemed like a strong and good candidate and we already knew that it could manage what we wanted to do.

Below is a list of final arguments to why we chose to use Performance Monitor.

- Its level of analysis is well fitting with what we want to achieve with our results.
- There is no need to buy any licenses for it, since it is included in Windows 7, which all the Tobii computers that will work with this already have. Therefore it will not cost Tobii anything to use this in the future.
- Performance Monitor is already installed on every Windows 7-equipped computer. So there is not even a need to install it. This also makes it easy to run our tests on a customer's computer if it has Windows 7, since it makes it possible to gather results there as well.
- Profilers are likely to demand more CPU and memory usage from the test system (based on the opinions of several Tobii employees and the fact that profilers have to work on a deeper level in order to analyze an application on code level, while Performance Monitor only keeps track of a set of performance counters provided by the operating system).
Figure 5: An example of Performance Monitor
Implementing tests

Test structure

Structure of existing test scripts

Tobii have an extensive collection of test scripts for Tobii Studio in their TestComplete projects. Generally, the test scripts are written for specific test cases and are not focused on being able to be used in more cases. The tests are run on Tobii’s test servers by using TestExecute, which is a command line based tool from SmartBear. Its purpose is to make it possible to execute TestComplete projects without the need of having TestComplete installed. At the beginning of a test run the scripts clean up in certain directories on the computer to make sure that Tobii Studio is in the same state every time. The logging system is event based and all the events that occur are written to an HTML-formatted file.

The entire Tobii Studio GUI is name mapped so that buttons, forms and similar components that need to be used during tests can be used when accessing the components from the test scripts. However, during our tests some parts of the application’s name mapping had not been updated yet, due to all the updates that were being added in time for the next big software release.

Tobii have a large set of test scripts, which are divided into the following categories:

- **Tests**
  These are the types scripts that perform actual testing functionality by performing a set of actions in Tobii Studio. Tests often use utility and sub utility scripts for help with common functionality.

- **Utilities**
  These are script types that perform specific actions in Tobii Studio that are common and can be used in different tests without being dependent on the context. Often uses sub utilities for help with smaller, specific actions.

- **Sub utilities**
  Test scripts of this type are focused on performing small actions in Tobii Studio. They are most often used by utilities to achieve a bigger action all together, but can sometimes be used directly by tests.

- **Data**
  These scripts can be used by any other kind. Their purpose is to supply the other scripts with files, paths, numbers, strings and similar kinds of data.

Planned structure for new test scripts

Our plan is to use TestComplete and try to adapt our work as much as possible to fit together with the current situation and structure that Tobii has. We believe that by following their standards our tests are more likely to work correctly as well as being beneficial in the long run. Because of the fact that they already have so much scripting functionality, we believe that we will be able to use a lot of their existing functionality for parts of our creations. Therefore it is important to have the same structure as they do.
We want to make our tests dependent on their helping scripts, but not make their tests dependent on ours. In other words; we should try to fit in with the situation and make our functionality work without changing theirs. Our scripts should at the same time be as general as possible, making it possible to reuse them for different tests without the need of writing new ones that are almost identical. An important factor in achieving this is to have parameters defining the behavior instead of hard-coded values.

We will focus on creating large test cases that run for hours. This will test the limitations of the application and will be able to detect how the performance is changed over time. Additional benefits of high volume tests are that they are believed to be able to dramatically increase the reliability of the tested software compared to smaller tests. Certain errors such as resource leaks, resource exhaustion errors, timing related error and buffer overruns might not be detected until a high volume test is performed. [14]

Since test runs might be very long, anything that could cause a test run to fail should be prevented as early as possible. So we need to check that the system that is running the tests is set up properly. This includes for example checking that necessary folders are not write-protected, if network resources are connected, and that Tobii Studio is in an acceptable state.

We will start development by concentrating on some simple well defined test cases first, to get comfortable with the test application and then work our way towards more difficult tests later on in the development. In the publication "Success with test automation", Bret Pettichord recommends not trying to start by doing too much, but rather try to get some first results quickly. Digging into the development in this manner will enable you us to quickly identify any testability issues. As the sooner they are identified, the better. [15] Some parts of the software might be more difficult to automate, and thus could be left alone. Due to our tight time plan we might need to focus on the parts that are relatively easy to automate.

Just as Tobii’s tests do, our tests should make sure that Tobii Studio is in a standard state when each test is starting. This is important in order to get reproducible results and to make sure that the test results are as unaffected by external events as possible. For example, the application should be restarted between every test run.

We also plan to extend the current logging functionality by keeping track of important events and control flow during the test run - not only events occurring in the application. This information can be important when analyzing performance over time and it is necessary to know what has happened at different times.

**Monitoring functionality**

**What is Performance Monitor?**

In this section we will give a description about how Performance Monitor works and also explain a set of important key words that are very common to use when talking about the application.

**Terminology**

Here is a list of words that are associated with the use of Performance Monitor. The descriptions are based on the knowledge and experience we have received by using it extensively.
• **Data collector set**  
  Contains an arbitrary amount of data collectors. The data collector set is used when starting and stopping monitoring.

• **Data collector**  
  Contains an arbitrary amount of counters of a certain kind.

• **Counter**  
  There are different kinds of counters. They can provide information about such things as the operating system, applications, services and the kernel.

• **Performance counter**  
  This is the kind of counters we are using. They makes it possible to get data about the computer's hardware, different parts of memory and resource allocation in processes.

• **Sample interval**  
  At what rate data should be gathered in a performance counter data collector.

• **Template**  
  A data collector set can be saved as a template. This outputs an XML file defining all the settings and components of the set.

• **Data manager**  
  Manages the creation of summary log files for a data collector set. Can output XML and HTML-based summaries for the entire monitoring run.

**How it works**

In order to be able to collect any sorts of data, a data collector set is needed. When a data collector set has been created, it needs to be filled with at least one data collector in order to make it possible to gather any data. The data collector set works as a container for all the data collectors that are needed. When a data collector is created inside a data collector set, you need to define which counters should be put inside it. In our case we work with performance counters, and we therefore can choose from all those counters that the operating system is providing us.

Each data collector has a sample interval and an output format. The sample interval defines at what interval all the containing counters should refresh and gather new data. The output format defines how the data should be presented in the output file. There are four types to choose from: binary, tab separated text, comma separated text, and SQL. A data collector can only have one output format - it cannot write to multiple files of different formats. However, it is possible to convert log files to other formats later on. Binary log files are read by and displayed in Performance Monitor. This makes it render a graph to show each counter's values graphically. It is also possible to use what is called Data Manager in a data collector set. It can create summary log files based on XML and HTML formats.

Monitoring is managed by selecting a data collector set and then pressing either the start or stop button, depending on the current status of the data collector set. During monitoring it will write to its output files, so there is no need for an explicit data export step.
How should Performance Monitor usage be automated?

Once it was decided that Performance Monitor should be used, we started to automate the use of it. We wrote scripts that started the application, selected a data collector set, and finally clicked the button for monitoring to start. When our Tobii Studio test run was over, we then let the script click the stop button. We had managed to automate the usage of our runtime performance data gathering tool.

Later we realized that it could be troublesome for Tobii in the future if our data gathering is based on the usage of a GUI. First of all, it is quite complex to create scripts that can create new data collector sets as well as data collectors. We based our scripts on a pre-existing data collector set that we had created manually beforehand. Even though we had researched which performance counters that were of interest to Tobii, we thought that it seemed better and more scalable in the long run to actually make it possible to let the script manage the creation of the data collector set.

We could think of two ways of proceeding; either to create complex scripts for using the Performance Monitor GUI for creation of data collector sets and setting all the needed settings, or to find a more script-friendly approach - preferably by using shell commands or something similar that is more text-based and script-friendly.

GUI automation with template

Instead of creating scripts that navigate the Performance Monitor GUI and perform a large amount of clicks, textbox typing and other needed GUI-based operations we came up with a shortcut that would still be GUI-based, but have some parts based in text file writing. To not have to write GUI-based scripts for all parts would be big time saver in the context.

The solution was found in the template feature of the application - it is possible to create data collector sets from file. And an already existing data collector set can of course be saved as a template. The template is an XML file. And we came up with the idea that we could use a basic XML template for creating a data collector set instead of doing it via the GUI. This would allow us to save time by only creating GUI scripts for the procedure of importing and XML template, and possibly also for exporting. What we would need to do instead was to write XML-formatted text to a file.

Before proceeding to far down the XML template road, we wanted to make sure that that was the only solution. Because it felt quite risky to try to create our own templates dynamically without any past experience with that part of Performance Monitor.

Command line from shell

We wanted to search for an alternative to automating the use of Performance Monitor with the help of templates - preferably without any GUI interaction at all. Due to the fact that we during our research phase had learned that it was possible to create a WMI object to access the Performance Monitor process and tell it to refresh a data collector set, we believed that we would be able to find more command based ways to go.

Quickly we found out that there actually exists a dedicated program for running Performance Monitor from Windows shell's with a command line. It is called Logman and it is simply meant for managing Performance Monitor with the command line [http://technet.microsoft.com/en-us/library/cc753820(WS.10).aspx]. Once we learned this, we excluded the GUI and template usage immediately. Because this was what we had been hoping to find. To access the shell from script is a simple operation and it is as simple to execute commands.
Automating Performance Monitor usage with Logman

Now that we had discovered a new way of automating the use of Performance Monitor it was just a matter of learning how to use the Logman commands and create scripts for constructing a working command line. All the needed basic operations are there; it is possible to create, delete, update, start and stop data collector sets as well as creating new ones from imported XML templates and exporting existing ones to templates. And during creation Logman provides all the parameters we needed when defining our data collector set.

We managed to create a parameter based set of scripts that can create a custom data collector set based on the values of the parameters. One of the most important features of this implementation is that it makes it possible to decide exactly which performance counters that should be used. Other features include the functionality to set output file paths and names, as well as sample interval. We did not implement any functionality for creating multiple data collectors inside our data collector set. We had never used multiple data collectors earlier because we did not feel any need for it. And therefore we did not feel any need for creating functionality for it in our scripts.

The scripts worked fine and we were satisfied with our scalable implementation - it was not restricted to any GUI usage that can be disturbed by response time, object naming and such. By using the command line we could guarantee a result as long as the command line is not incorrect.

Automating generation of raw data

As we concluded in the research section, we should make it possible to have a text based raw data file as an alternative to any runtime performance data that needs a specific application to be used. Our runtime performance data output file from Performance Monitor is in a binary format and needs to be opened with Performance Monitor in order for the data graphs to be displayed. Obviously this does not concur with our previous decision. The solution is simple and straightforward; we use command line program Relog. As input it takes any of the available output files from Performance Monitor and converts to another one of those. In other words; we can feed it with our binary log file and receive a text based (either tab- or comma separated) raw data file with the exact same data.

Presentation of test results

In a performance testing project, the results are naturally an important part to focus on. That also includes presentation of acquired results. To have results that are complex to read and understand is not beneficial. Therefore we have put a lot of effort in coming up with a way of presenting informative event logs and runtime performance data results. We want the presentation to be done in a way that enables the result reviewer to evaluate how the logged events can have affected the performance of the tested application. To have a graphical visualization is likely to lead to easier understanding on the data. During our work we have tried out different approaches to achieve what we want.
Implementation structure

Figure 9 shows the control flow of the applications we use as well as what files they output. Later in this section we will describe how the implementation works.

![Diagram showing control flow and output files.](image)

**Figure 6: Overview of the prototype**

How to start a test

1. Select which test cases you want to run. A collection of common test items with predefined values is provided, but you can also create a custom test case by simply setting some values.
2. Select which features, such as performance monitoring, that you want to run the tests with.
3. Start the test suite.

What happens during the test execution

The following list describes what happens during one automated test run. This process can be run an arbitrary amount of times in a row without having one run affect another. Each test run will create a set of output files in a separate folder.

1. When a new test is started, an initialization will verify that all necessary paths are set. If the project fails during this phase, it will first try to fix the problem, but if proves to be impossible to solve the issue, the affected test will be stopped.
2. If the performance monitoring feature is selected, TestComplete will start a shell that runs few Logman command lines. This causes Performance Monitor to run in the background and works according to the parameters that have been set (such as sample interval and which performance counters to use).
3. Tobii Studio environment is cleaned before the application is started. Once it is running, the selected project is imported and its containing test is run. When the test is finished, performance monitoring is.
4. When the execution is finished the final results are placed in the selected output folder.
Combining event logs and runtime performance data

Our first attempted solution was to combine the event logs and the runtime performance data into a single log and rendering this data as a combined graph. By placing the events on a timeline on the X-axis and the runtime performance data as elements in the plot area we could achieve a document that would be easy to review and could make correlations between events and performance very obvious.

We started working on a script that converted generated performance log data and event logs to Excel. Via Excel macros we would then format the performance data as a graph and plot the event messages along a timeline. With the timeline placed below the graph, the reader could easily see the relationship between the performance at a certain time and events that occurred during that time.

We soon found out that there could be some problems with formatting the data in this way, the event timeline along the X-axis could become quite messy if too many events occurred close to each other. Using a plug-in we managed to place the events on different heights, to increase the readability and avoid events being placed on top of each other.

We discussed our solution with Tobii and were told that it seemed like good idea to use Excel to handle raw data and that the timeline feature was very beneficial. This was working quite well until we found out that the actual test servers did not have Excel installed. Since results preferably should be viewable on the test computers, we had to think of a different solution. Tobii did not think it would be worthwhile to move the data to some computer with Excel installed - it would be too much effort and the work would probably not be used. And because we did not have any more time to spend on illustrating our event log we had to proceed and find a different solution.

Figure 7: Combined performance graph with timeline event in Excel

We discussed our solution with Tobii and were told that it seemed like good idea to use Excel to handle raw data and that the timeline feature was very beneficial. This was working quite well until we found out that the actual test servers did not have Excel installed. Since results preferably should be viewable on the test computers, we had to think of a different solution. Tobii did not think it would be worthwhile to move the data to some computer with Excel installed - it would be too much effort and the work would probably not be used. And because we did not have any more time to spend on illustrating our event log we had to proceed and find a different solution.
The final result presentation

We decided to output four different result files, generated once for each test run:

- A text document containing a brief test summary over which settings were used and also a listing of important events and at which time each one occurred.
- An HTML-based document from Performance Monitor. It provides an overview of all the measured performance counters.
- A performance graph that features the full runtime performance data of all the used performance counters used in Performance Monitor. This file is in binary format and needs to be opened with Performance Monitor.
- A text-based raw data version of the binary log file that Performance creates. This file is based on the exact same data as the binary file, but in a more accessible format.

Additionally an HTML-based detailed TestComplete log is created as well. This is the standard log file that Tobii use. We use it as well, because of the fact that it is integrated with TestComplete. However, we do not base our results on any of its contents.

To verify that the test has been performed correctly, a peek at the summarized event log is all that is needed. If an error has occurred, a look at the detailed log might be needed. A simple comparison with previous performance tests can be made by checking out the performance summary. If necessary you can also view the performance graph to get the full picture. The performance graph viewed side by side with an event log will show if a certain event caused a sudden change in performance.

Could we automate the result comparisons?

When working with automated tests, it is appreciated if not only the test execution, but also the gathering and management of results, requires as little manual work as possible. A test is often called semi-automated as opposed to fully automated if the results require some manual handling. [6]

For this project the way result management could be automated was not very obvious. We contemplated using some kind of automated script that would match the results of a test run
with the expected results. However since there was little information available about which levels of performance should be expected from the software, and so many different variables to look at, it would require a lot of research and communication with Tobii to find out how such a result comparison should be made.

We spent a lot of time on finding out which tools we should use to gather the expected results, we also did not know which format the results could be presented in, until later on in the project. Because of our limited time frame we did not have time to build a results comparison script.

**Problems we faced during the implementation**

We had to rewrite the structure of our tests several times during development.

- Our initial test scripts were built in a way that was not compatible with the current way test scripts were run in Tobii. We used a feature of the test tool that enabled tests items to use input parameters. After implementing all our tests this way we found out that this solution was not compatible with the way the test were currently run in the company. They used the tool TestExecute to launch the tests. We had assumed that TestExecute, since it was built by the same company as TestComplete, could handle all necessary features. However, we never found any solution concerning how to set our input parameters via TestExecute, so we had to rebuild the entire structure of our scripts.

- *Name Mapping* is a feature in TestComplete that enables testers to separate the tested application’s object names from tests in order to improve the maintainability of the tests. By name mapping the GUI of an application you can call buttons, forms and other GUI-elements by different names in the test scripts, thereby not having to update the scripts if the elements change names in a new version of the application.

We started using the existing name mapping from the current test scripts. However we found that some parts of Tobii Studio did not have up to date mapping due to a new major release of the software. We spent some time trying to remap after deciding that it was too time consuming. Another issue would have been to merge our name mapping with the existing name mapping. After talking to the other testers at the company we decided that it was best to leave the parts of the software with insufficient name mapping alone.
Conclusions and results

Reflections on what was achieved

Did we complete all of our goals?
Below is the list of bullets used in the background section for describing and defining the goals of the thesis. Here we will verify that if we actually managed to do what was intended.

- **Interview employees to define the different goals**
  This was an open and agile goal definition. We ended up having interviewed about 15 people and that was enough for us to get the knowledge that we needed for proceeding with the other goals and defining them. It was very rewarding to get contact with so many different people at the company, because it really helped us a lot with understanding the situation.

- **Create automated performance tests**
  We have created automated performance tests that are designed in a general way in order to make it possible to reuse them for a lot of different tests. We focused entirely on the recording feature of Tobii Studio and managed to make it possible to test the majority of the available stimulus types. We are satisfied with our effort and think we have made our work beneficial for Tobii in the future.

- **Gather runtime performance data**
  We spent a lot of time researching this area. In the end we ended up having a solution that we were happy with, since it provided the functionality we had been looking for. The implementation can provide as coarse or fine grained data as the test requires. The fact that the result generation is fully automated and customizable means a lot.

- **Present gathered runtime performance data in an easy way**
  We always aimed at making the results visualized with graphs or similar. This is also what we ended up with. And according to other decisions we also made readable text based raw data available if someone in the future may want to use the data in some other way. He also have provided simple summaries that are short and concrete enough to be read without any visualization.

- **Design tests that test well fitting areas of the application**
  The recording functionality is perhaps the most used functionality in the entire application, since it is one of the corner stones of Tobii Studio. Now that we have covered that area almost entirely we think that we have achieved this goal. Additionally, it is easy for Tobii to extend our automation functionality to test other parts of the application.

Our experiences during the project development
Prior to this project we had an extremely limited experience with testing. We had done some exercises with simple unit testing, but had never tested for a real world project. Our experience with automation was nonexistent. It took us some time to get familiar with the test tools and the overall test mindset. The first couple of scripts took a long time to produce, as we were working in a tool that was new to us, in a language we were not used to. As the work progressed we started to become more comfortable with scripting and testing in general.
The resulting prototype

As a result of our work, we have implemented a test suite that can run automated tests over a long period of time while monitoring runtime performance for both the application and the computer's hardware. The resulting tests can be used repeatedly by Tobii to help them identify performance issues for common test cases, and newer versions of Tobii Studio can be tested in the future to verify that a certain level of performance is maintained. The design of our tests is so general that it will be possible for Tobii to continue extending our suite with more functionality.

The amount of functionality we ended up having created automated tests for was not as big as we had hoped for. This was caused by the limited time frame and the fact that our work started during the month when Tobii were about to release a new major version of the Tobii Studio application. The effect of this was that the testers and developers changed a lot of things in both the application and in the testing scripts. This made it difficult to keep up from time to time, and sometimes we had to redo some of our work.

Since our goal was to create an implementation that would be used in the future by we focused on creating a general implementation as possible. We documented our tests well and provided an interface that lets the other testers at the company easily extend our tests. Most of the existing tests at the company could, with only slight modifications, be used in combination with our scripts – thus providing extended logging and performance monitoring to areas which previously were not monitored.

Changes during the course of the project

We started the project with an agile approach. Together with the company we set a general goal for our project but let the details be developed in an agile manner, as we progressed and found out more about the restrictions and possibilities of the available tools and the possible solutions.

Several changes regarding choice of tools were made during the course of the project, and are described in detail in other parts of the report. In this section we try to summarize some of these changes.

- The method of handling parameters in our test suite had to be revised, since it was not compatible with TestExecute, the tool which executes scripts on Tobii's test server.
- After developing a solution for combining performance graphs and event timelines in Excel we found out that it could not be used on the test server.
- Through our research on how to gather runtime performance data we tried out a lot of different approaches until we found a one that suited our purpose well.
Future work

Possible improvements of our implementation
These are some recommendations for Tobii on how to further extend our test projects. Ideas that we have either started working on but had to scrap due to the limited time frame, or ideas that evolved during development but never were considered for implementation.

Result gathering

Log warnings when certain performance thresholds are exceeded
It is theoretically possible to implement this kind of functionality by using Performance Monitor and performance alerts. We only use performance counters in our implementation. The main reason to why we excluded performance alerts is that we did not have any real material to base the threshold values on. Additionally we found the documentation about how to combine standard performance logging with performance alerts to be lacking and did not figure out how to achieve it. If implemented, some time would also be required to decide which alert levels should be assigned to which tests.

Adaptable sample interval setter
In our implementation the tester has to decide the sample interval of the monitoring. A possible alternative to this is to make the script calculate an appropriate sample interval based on how long the automated run is expected to run (which is already calculated). To have this functionality removes some of the responsibility of the tester, who does not need to concern himself about setting this each time.

Result presentation

We did not focus all too much on the result presentation. However, we were still able to find some possibilities for improved presentation.

- Combining the performance log from Performance Monitor with the event log would increase readability. This is what we tried to do, but had to drop. It is still a nice feature though.
- A filter functionality for the event logs that would enable you to select only certain parts of the log. The filter could for example filter out certain kinds of events or focus on parts with a certain delay.
- An automated results comparison tool or script could compare two test results to see if the performance has changed.

Our conclusions regarding testing and automation in general
Concepts that could be beneficial for other test projects in the future:

- When using automated tests it is especially important that the results and logs produced can be trusted. In the early stages of implementation we often found ourselves having to manually observe automated tests in order to find out exactly what happened, since our logging was insufficient. After expanding the logging we could save a lot of time since we could actually get sufficient information even though something unexpected happened in the test.
• When working with large test projects it is important to refactor reoccurring code into separate functions. These functions should also be as general as possible. We found that lots of the older existing test scripts at the company included multiple versions of the same pieces of code, with slight differences, and thus it was quite difficult to keep track of what was being tested.

• It is worth spending time finding out the constraints of the test tool early, and develop your tests accordingly from the start, since it might be time consuming to refactor the code later.

• Do not assume functionality of your test tool. Try to find out early if certain operations such as syncing projects and moving stuff around actually work. You might not find out that certain important features are lacking until later on in the project, and had you known that earlier you would’ve developed in a way that would’ve been less dependent on those features.

• Do not develop for too long in a "simulated environment". Start running the tests on the intended platform early. This will help you find out if you have designed the tests in an incorrect way.

• While using a mapping layer between the script and the tested application generally simplifies the process of keeping the test up to date with the software, it is very important to check out the way in which the mapping is handled in the test tool. It is good to find out early if you think you can rely on features such as moving and copying mappings from different projects, and merging mapping with each other. If certain features are cumbersome to use, you might have to design around them.
References

   http://www.uppsatser.se/uppsats/1902327764/

   http://www.cnblogs.com/nckiki/articles/250803.html


   &ObjectType=ART


   http://softwarequalitymethods.com/Papers/autoarch.pdf

   www.smartbear.com


[9] “About WMI”, MSDN.com

[10] "Timer object“ Smartbear.com wiki,
    http://smartbear.com/support/viewarticle/13306/


    http://dl.acm.org/citation.cfm?id=1022494.1022536&coll=DL&dl=ACM&CFID=77462952&CFTOKEN=60911683

    http://www.prismnet.com/~wazmo/succpap.htm
Related work

Sara Forslin – “Automatiska GUI-test : En teoretisk och praktisk studie för hållbarare test” adresses some of the issues of creating maintainable automated tests. An evaluation of TestComplete is also featured. The report features advice on how to design the scripts to make sure that they will be kept up to date with as little work as possible, for instance discusses the importance of using a mapping layer when GUI-testing. Some recommendations for test designers is to focus on automating parts of the software which will provide the most benefit.

Automating Performance Testing of Interactive Java Applications
http://dl.acm.org/citation.cfm?id=1808268&bnc=1
The paper presents an approach on automatic testing of GUI applications’ perceptible performance with different capture & replay tools. It makes it possible to perform comparative studies of the perceptible performance, for example by testing an application’s latency distribution on different operating systems. It also makes it possible to compare results of regression performance testing between different interactive application versions.

Success with test automation
Bret Pettichord
http://www.prismnet.com/~wazmo/succpap.htm
This publication talks about how to really succeed with test automation in the real world. How to approach the tests in order to discover problems with testability in an early stage is one recommendation. Certain compromises might need to be made, and you might have the best rate of success if you don’t try to do too much, but rather focusing on the areas where the big gains are.

Cost Benefits Analysis of Test Automation
Douglas Hoffman
The paper presents research on how test automation is not always necessary, appropriate or cost effective. By analyzing expected return on investment (ROI) it is possible to determine if test automation can be profitable. These results are computed by comparing the costs and gains that are achieved through automated testing compared to manual testing. To compute ROI is not always helpful though, as intangible factor may interfere with the calculations, making them not display the real value of automation. The paper also discusses how to value test automation and shows that management issues can lead to mistakes and misplaced expectations.

The ROI of test automation
Michael Kelly
The article investigates different test automation offers guideline types and shows how their return on investment (ROI) is calculated. The purpose is to provide better understanding for test automation costs and the ability to what types of automation will yield the greatest ROI. Some of the conclusions that are reached are that the topic of test automation as well as the topic of calculating costs and benefits for test automation is very large and complex; and that it therefore is not possible to provide equations that will work for all kinds of projects. General guidelines, ideas for maximizing ROI and calculation examples with given formulas are provided.
The paper describes a broad class of testing techniques that are collectively called High Volume Test Automation (HVTA). The main purpose of the techniques is automated execution and evaluation of large numbers of tests that are run for the purpose of exposing functional errors that are otherwise hard to find. The authors propose to find additional existing industry HVTA projects in order to write case studies of them and later on be able to create their own to for the technique's implementation. HVTA techniques are more effective to use on already stable code, because it is more suitable to test for reliability on such code that passes basic functional tests.
Appendix

General test definitions

[1]

**Regression test**
A regression test is a test which is intended to run repeatedly during the development of a software. For instance running after a new feature has been implemented and comparing the results with previous results. A common issue in software development is able to ensure that a performance level is maintained between iterations.

**Precondition**
the state in which a software has to be in prior to executing a code. In testing this would be the input data which is needed to complete a test case.

**Post condition**
conditions which exist after a code has been executed. In a testing context this would be the expected results

**Test coverage**
the amount of code in a project which is being executed when a test suite is running (test coverage usually expressed as a percentage).

**Test Oracle**
Predicting a test result in advance and comparing the results with the prediction is an activity performed by so called test oracles. An oracle can either be a tester or a software.

**Test item**
A single test unit, which usually focuses on testing a small defined functionality / area of the software.

**Test group**
A collection of several test items

**Test project**
A collection of several test groups or items

**Test Suite**
A collection of several test projects
Definitions of Automated tests

**Manual testing** can be described as a situation where a person initiates each test, interacts with it, and interprets, analyzes, and reports the results.

**Automated testing** is basically a concept where a test is able to run without manual interference. There are a few different opinions on what is required for a test to be considered as automated. Many testers differentiate between *semi-automated* and *fully automated* tests. However, there is no hard line between the two.

A common example of elements required for a fully automated test:

- Ability to run two or more specified test cases
- Ability to run a subset of all the automated test cases
- No intervention needed after launching the tests
- Automatically sets-up and/or records the relevant test environment parameters
- Runs the test cases
- Captures the relevant results
- Compares actual with expected results and flags differences
- Analyzes and reports pass/fail for each test case and for the test run

If some of these elements are not present, you might say that the test is not strictly fully automated, and you could call it semi-automated instead.
Definitions of Performance Testing

*Performance testing* is a general term with sub-categories such as *load testing* and *stress testing*.

**Load testing Error! Reference source not found.**
- Measures performance, usually under *average* load, or at least a load when the system is still expected to function properly. The load level is often predefined.
- *Does not* attempt to break the system as stress testing does.
- The expose problems that might not occur during single tests (for example memory leakage).
- Huge datasets are preferred, since certain bugs only appear under those conditions.
- *Load* often refers to the number of concurrent user. So load testing is most relevant to multi-user systems. ‘*Virtual User*’ is a common term in the world of load testing.
- Criteria for passing a load test differs from company to company, there is no default criteria.
- Preferably simulates actual rather than modeling theoretical usage.
- Can measure both subjective user experiences and objective statistics.
- *Does not* focus on GUI-testing. (for example if the software to be tested has a button connected to a method, a load test might just test the method itself, while a regression test tool might simulate actually clicking the button and firing the method that way).

**Endurance testing Error! Reference source not found.**
A type of load test that runs a a large number of times or for long durations. Purpose is to find out if there are possible memory leaks or other problems that occur over time.

**Stress testing Error! Reference source not found.**
- Uses an unusually *high load*.
- Attempts to *break the system*.
- Tests robustness, availability and error handling. Test how gracefully the system can recover from being broken (for instance: shut downs and restarts of system parts).
- Might focus on trying to find the system’s weakest link.
- Loads can be anything from user amount, bandwidth, data. Although testing data amount is usually known as *volume testing*.

**Types of automation tests**
In the book “Software testing” Ron Patton, the following terms are brought up as examples of different approaches to automation:

**Static testing** – this is the simplest type of automation. Test scripts are written manually. According to Patton, tests are very well suited for regression testing as they test the same functionality every time.
This is the type of testing that we will implement in our thesis.

Other types of test include *random testing*, which tests operations of the software randomly, and *model based testing* which also generates random test cases but does this within a model
which is basically boundaries for the types of operations the software can perform. Test within these boundaries will be generated. Model based testing requires a lot more initial setup compared to static testing, which makes it an unrealistic choice for smaller test project, according to Patton.

Performance counters used in Performance Monitor

This section contains a listing of all the performance counters that we ended up using in Performance Monitor. They have been categorized in the same way as they are in the
application. Each performance counter's description is copied from Performance Monitor's own description.

.NET CLR memory

% Time in GC
is the percentage of elapsed time that was spent in performing a garbage collection (GC) since the last GC cycle. This counter is usually an indicator of the work done by the Garbage Collector on behalf of the application to collect and compact memory. This counter is updated only at the end of every GC and the counter value reflects the last observed value; it's not an average.

Gen 0 heap size
This counter displays the maximum bytes that can be allocated in generation 0 (Gen 0); its does not indicate the current number of bytes allocated in Gen 0. A Gen 0 GC is triggered when the allocations since the last GC exceed this size. The Gen 0 size is tuned by the Garbage Collector and can change during the execution of the application. At the end of a Gen 0 collection the size of the Gen 0 heap is in fact 0 bytes; this counter displays the size (in bytes) of allocations that would trigger the next Gen 0 GC. This counter is updated at the end of a GC; it’s not updated on every allocation.

Gen 1 heap size
This counter displays the current number of bytes in generation 1 (Gen 1); this counter does not display the maximum size of Gen 1. Objects are not directly allocated in this generation; they are promoted from previous Gen 0 GCs. This counter is updated at the end of a GC; it’s not updated on every allocation.

Gen 2 heap size
This counter displays the current number of bytes in generation 2 (Gen 2). Objects are not directly allocated in this generation; they are promoted from Gen 1 during previous Gen 1 GCs. This counter is updated at the end of a GC; it’s not updated on every allocation.

Large object heap size
This counter displays the current size of the Large Object Heap in bytes. Objects greater than 20 KBytes are treated as large objects by the Garbage Collector and are directly allocated in a special heap; they are not promoted through the generations. This counter is updated at the end of a GC; it’s not updated on every allocation.

Memory
Available MBytes
is the amount of physical memory, in Megabytes, immediately available for allocation to a process or for system use. It is equal to the sum of memory assigned to the standby (cached), free and zero page lists. For a full explanation of the memory manager, refer to MSDN and/or the System Performance and Troubleshooting Guide chapter in the Windows Server 2003 Resource Kit.

Process
% Processor Time
is the percentage of elapsed time that all of process threads used the processor to execution instructions. An instruction is the basic unit of execution in a computer, a thread is the object
that executes instructions, and a process is the object created when a program is run. Code executed to handle some hardware interrupts and trap conditions are included in this count.

**% User Time**
is the percentage of elapsed time that the process threads spent executing code in user mode. Applications, environment subsystems, and integral subsystems execute in user mode. Code executing in user mode cannot damage the integrity of the Windows executive, kernel, and device drivers. Unlike some early operating systems, Windows uses process boundaries for subsystem protection in addition to the traditional protection of user and privileged modes. Some work done by Windows on behalf of the application might appear in other subsystem processes in addition to the privileged time in the process.

**Private Bytes**
is the current size, in bytes, of memory that this process has allocated that cannot be shared with other processes.

**Thread Count**
The number of threads currently active in this process. An instruction is the basic unit of execution in a processor, and a thread is the object that executes instructions. Every running process has at least one thread.

**Working Set**
is the current size, in bytes, of the Working Set of this process. The Working Set is the set of memory pages touched recently by the threads in the process. If free memory in the computer is above a threshold, pages are left in the Working Set of a process even if they are not in use. When free memory falls below a threshold, pages are trimmed from Working Sets. If they are needed they will then be soft-faulted back into the Working Set before leaving main memory.