Redesigning calibration algorithms for transmission electron microscopes

Citation for published version (APA):

Document status and date:
Published: 24/10/2018

Document Version:
Publisher’s PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:
• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher’s website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the “Taverne” license above, please follow below link for the End User Agreement:
www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:
openaccess@tue.nl
providing details and we will investigate your claim.
Redesigning Calibration Algorithms for Transmission Electron Microscopes

Nityanand Panpalia
Redesigning Calibration Algorithms for Transmission Electron Microscopes

Nityanand Panpalia

Eindhoven University of Technology
Stan Ackermans Institute / Software Technology

The design described in this report has been carried out in accordance with the TU/e Code of Scientific Conduct.
Abstract
The current calibration algorithms, developed in Delphi, are part of a legacy software stack. Here, legacy implies software with no or minimal tests. Such code is difficult to maintain or extend for adding new features, limiting improvement options. Testing for every release has to be done manually, which is a repetitive task. This is an inefficient use of a skilled software engineer’s time. This report describes a project to redesign the calibration algorithms in AutoStar, to make them testable and modifiable for further automation. AutoStar is a component for prototyping and developing functionality to automate microscope control. The project uses an incremental and iterative approach to redesign the algorithms. Each increment involves gathering requirements, designing, implementing the design, testing, and evaluating the results. The project is divided into two phases. The first phase involves developing a prototype to prove the feasibility of the project. The second phase involves redesigning the first calibration algorithm in C++. Module tests written for the redesigned code show the algorithms can be successfully redesigned in AutoStar. Usage of certain design principles and patterns makes the redesigned algorithms testable and modifiable.

Keywords

Preferred reference

Partnership
This project was supported by Eindhoven University of Technology and Thermo Fisher Scientific.

Disclaimer
Reference herein to any specific commercial products, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the Eindhoven University of Technology or Thermo Fisher Scientific. The views and opinions of authors expressed herein do not necessarily state or reflect those of the Eindhoven University of Technology or Thermo Fisher Scientific, and shall not be used for advertising or product endorsement purposes.

Disclaimer Liability
While every effort will be made to ensure that the information contained within this report is accurate and up to date, Eindhoven University of Technology makes no warranty, representation or undertaking whether expressed or implied, nor does it assume any legal liability, whether direct or indirect, or responsibility for the accuracy, completeness, or usefulness of any information.

Trademarks
Product and company names mentioned herein may be trademarks and/or service marks of their respective owners. We use these names without any particular endorsement or with the intent to infringe the copyright of the respective owners.

Copyright
Copyright © 2018. Eindhoven University of Technology. All rights reserved.
No part of the material protected by this copyright notice may be reproduced, modified, or redistributed in any form or by any means, electronic or mechanical, including photocopying, recording, or by any information storage or retrieval system, without the prior written permission of the Eindhoven University of Technology and Thermo Fisher Scientific.
Abstract

The current calibration algorithms, developed in Delphi, are part of a legacy software stack. Here, legacy implies software with no or minimal tests. Such code is difficult to maintain or extend for adding new features, limiting improvement options. Testing for every release has to be done manually, which is a repetitive task. This is an inefficient use of a skilled software engineer’s time. This report describes a project to redesign the calibration algorithms in AutoStar, to make them testable and modifiable for further automation. AutoStar is a component for prototyping and developing functionality to automate microscope control. The project uses an incremental and iterative approach to redesign the algorithms. Each increment involves gathering requirements, designing, implementing the design, testing, and evaluating the results. The project is divided into two phases. The first phase involves developing a prototype to prove the feasibility of the project. The second phase involves redesigning the first calibration algorithm in C++. Module tests written for the redesigned code show the algorithms can be successfully redesigned in AutoStar. Usage of certain design principles and patterns makes the redesigned algorithms testable and modifiable.
Foreword

Within Thermo Fisher Scientific Electron Microscopy, we create electron microscopes that need to be calibrated to get the best, accurate results. The calibration functionality is therefore an essential part of the tools to produce the final product. The current version of this tool is already available since a long time and needs to be replaced with a new version that is both better structured and maintainable and ready for future evolution. Since this is an essential tool we can only replace it with a new version if we can prove that the new one behaves the same (or better). We have defined this project to explore the possibilities and approach needed to move from the old code base (in Delphi) to a new version using a new framework (in C++ and Python) that is currently in development.

Fortunately, Nityanand has shown interest in this project and has picked it for his graduation thesis. After an initial exploration phase, we have chosen to create a proof of concept implementation of a mechanism that allows us to keep using the old user interface while we replace each calibration algorithm with a new implementation in the new framework, one at a time. This would allow us to deliver iteratively without disrupting the user experience. He designed and implemented this approach and demonstrated this successfully. For the following part, he continued the investigation in how to create a new generic approach for executing calibration tasks within the available framework. He had to understand a selected representable algorithm written in a functional manner in Delphi and find out how to match this to the task based approach of the new framework. In the end, he created an almost fully complete algorithm, demonstrating the soundness of his approach.

We run the project in an agile manner, using a TDD approach. During the project, he had to make several changes both due to new insights after getting more in-depth knowledge and the evolution of the used framework. The TDD approach proved to work helping to prove that already implemented functionality didn’t break after a refactoring. He also proved himself capable of progressing the project, come up with the right questions and involving people when needed.

As always, the context changed during the time of the project and shifted the priorities. In this case, it put the proof of concept approach, using the old UI, on hold. We now focus on a complete new implementation; both the algorithms and user interface will be developed new while maintaining the same functional results. We will start using his results immediately for the algorithms. His newly defined interactive task approach is, in line with the framework, generic and will even be used for another interactive tool. We will also try to continue using TDD as he proved this is possible and helps in creating quality software.

PROJECT MENTOR
ir. Paul Janson PDEng
October 2018
Preface

Systems that do not have any tests written for them are considered as legacy systems. Such systems are difficult to maintain or extend. In addition, it is extremely difficult to write tests for such systems because the code is not structured to write tests. There are two ways to solve this problem — either refactor the legacy system incrementally to start covering them with tests or redesign the legacy system. If there are other drawbacks of the legacy system then it can be more beneficial to use the second approach. An example of such a drawback is a legacy system developed in an old programming language. This project explores the second option by redesigning legacy calibration algorithms in a new programming language. This report describes how the redesign was carried out and the challenges and risks faced during this redesign.

This project has been carried out as a graduation assignment for the Software Technology (ST) Professional Doctorate in Engineering (PDEng) program of the Eindhoven University of Technology (TU/e). PDEng ST is a two-year doctorate level program that focuses on strengthening the technical and non-technical competencies of trainees. This ten-month graduation project has been carried out at Thermo Fisher Scientific and the report is one of the graduation deliverables.

Nityanand Panpalia

8 October 2018
Acknowledgements

As I sit down to write these final pages, I realize that my time at the TU/e is coming to an end. Successful completion of this course and project would not have been possible without the continuous support and guidance of several colleagues, friends, well-wishers and loved ones. I would like to take this opportunity to express my gratitude to all people who guided me, helped me, and supported me during this project and without whom this project would not have been a success.

First, I would like to bow before the Almighty God for showering his blessings and giving me the strength to carry out this project with utmost dedication and enthusiasm.

I am deeply grateful to the people from Thermo Fisher Scientific who made this project possible. I would like to thank my company supervisor, Mr. Paul Janson for his support, guidance, and enthusiasm towards this project. His valuable advice and feedback helped me improve the quality of work delivered.

I would also like to express my gratitude towards my university supervisor, Mr. Andrei C. Jalba who provided valuable suggestions and feedback to improve the report. His feedback helped me deliver a report that meets the university standards.

From the TU/e, I would like to acknowledge Ms. Yanja Dajsuren, Mr. Ad Aerts, and Ms. Desiree van Oorschot for their support throughout the duration of the program. In addition, I would like to thank the OOTI coaches who helped me improve professionally during the program. I would also like to thank my OOTI friends for their support and advice during the project.

I would like to thank Mr. Andrea Pasqualini for reviewing the design and implementation of the calibration algorithms. I would also like to express my gratitude to Mr. Peter Van Merkerk for providing AutoStar support. I would also like to express my gratitude to the calibrations team for their support and enthusiasm throughout the duration of the project.

I would like to thank my wife for her continuous support that allowed me to focus on the project. I would like to express my heartfelt gratitude to my mother for her sacrifice that enabled me to join the program. I would like to thank my sister for her unconditional love and support. Further, I would like to express my gratitude to my uncle and aunt for their unflinching support that motivated me to join the PDEng program and enabled me to continue with the program during difficult times.

Last, but not the least, for those whom I failed to mention on this page. I offer my apologies but know, that they have made this experience special. My warmest thanks and sincere regards to everyone once again.

9 October 2018
Executive Summary

Thermo Fisher Scientific is involved in research, development, production, service, sales, and marketing of high-end transmission electron microscopes. Transmission electron microscopes can generate high resolution, high quality images. To achieve such high quality, the microscopes need to be calibrated. At present, calibration is achieved through algorithms developed using the Delphi programming language.

The current calibration algorithms have the following drawbacks:

- The current calibration algorithms are part of a legacy software stack. Here, legacy implies software without any or minimal tests. Such code is difficult to maintain or extend for adding new features, limiting improvement options.
- Testing for each software release is done manually either on the microscope or on a Virtual Machine (VM). This is a waste of time of a skilled software engineer.
- Delphi is not a popular programming language. Hence, it is difficult for Thermo Fisher Scientific to find skilled software engineers who know the language or want to learn it.
- The current algorithms need manual interaction to calibrate the microscope. This is waste of time of a skilled system engineer that can be reduced.
- The current implementation heavily uses COM interfaces. COM interfaces once published cannot be changed. This introduces strong coupling and high maintenance overhead.

With a view to overcome these drawbacks, the goal of the project is

- To redesign the calibration algorithms in C++
- To make them testable and thus, maintainable
- To redesign such that further automation is possible
- To redesign such that the users are immune to the changes (redesigned algorithms should work with existing Delphi user interface)

The redesign must integrate the calibration algorithms in a C++ component called AutoStar. AutoStar is a component that provides a framework, called Task Core Library, to automate algorithms to be run on the microscope. In total, there are 17 calibration algorithms. However, it is not feasible to redesign all of them within the project duration. Instead, the objective is to create a design that achieves the above-mentioned goals and to prove its correctness by redesigning one or more calibration algorithms. The Delphi calibration algorithm user interface redesign is outside the project scope. 32-bit -64-bit incompatibility between Delphi user interface and AutoStar component led to the user interface integration being deemed out of scope for the project.

I used an incremental and iterative approach to execute the project. Following this approach, I divided the project into multiple increments and executed each increment in iterations. Each increment involved gathering requirements, designing, implementing the design, testing, and evaluating the results. I divided the project into two phases. The first phase involved developing a prototype to prove the feasibility of the project. The second phase involved redesigning the first calibration algorithm in AutoStar.

The prototype proved that it is possible to pass information, such as acquired microscope images and markers, from a C++ component to the Delphi user interface for display.
As part of the second phase, I extended the AutoStar component to support the calibration algorithms. Unfortunately, due to changes in the underlying AutoStar component, I had to redesign the extension a second time. After developing the extension, I developed the first algorithm using the incremental and iterative approach. I used a Test-Driven Development (TDD) approach to implement the calibration algorithms. The TDD approach was novel for me and brought an additional overhead of creating test infrastructure upfront that wasn’t factored well in the initial planning. Lower layer components did not provide the required functionality that further delayed the first algorithm development. In the end, half of the first algorithm could be developed. Since I could not develop and integrate one algorithm with the Delphi user interface, I wrote module tests, simulating AutoStar clients to prove the correctness of the redesign.

My calibration algorithm redesign shows that the calibration algorithms can be successfully integrated in the AutoStar C++ component. A TDD approach during implementation made the calibration algorithms testable. Coverage metrics of over 90% show that I employed TDD successfully. Certain design principles (For example Single Responsibility and Dependency Inversion) and template method design pattern used during implementation lead to a good separation of concerns and reduced strong coupling. This allowed the redesign to be modifiable for further automation.
# Table of Contents

Abstract .......................................................................................................................... iii

Foreword ......................................................................................................................... v

Preface ............................................................................................................................... vii

Acknowledgements .......................................................................................................... ix

Executive Summary .......................................................................................................... xi

Table of Contents .............................................................................................................. xiii

List of Figures .................................................................................................................... xvii

List of Tables .................................................................................................................... xix

1. Introduction ................................................................................................................. 1
   1.1 Context ..................................................................................................................... 1
   1.2 Outline ..................................................................................................................... 3

2. Stakeholder Analysis .................................................................................................. 5
   2.1 Introduction .............................................................................................................. 5
   2.2 Stakeholder Groups ............................................................................................... 5
      2.2.1 Project Steering Group (PSG) ......................................................................... 5
      2.2.2 Calibrations Group ......................................................................................... 6
      2.2.3 AutoStar group ............................................................................................... 6
      2.2.4 Systems Group ............................................................................................... 7
      2.2.5 University Group ............................................................................................ 7
   2.3 Stakeholder Matrix ................................................................................................. 7
   2.4 Communication Plan ............................................................................................. 8

3. Problem Analysis ....................................................................................................... 11
   3.1 Context ..................................................................................................................... 11
   3.2 Roadmaps ............................................................................................................... 12
      3.2.1 Business Roadmap ......................................................................................... 12
      3.2.2 Technology Roadmap .................................................................................... 12
      3.2.3 Product Roadmap ......................................................................................... 13
   3.3 Project Scope and Goals ....................................................................................... 14
   3.4 Design Opportunities ............................................................................................ 14
      3.4.1 In Scope .......................................................................................................... 14
      3.4.2 Out of Scope .................................................................................................. 15

4. Domain Analysis ....................................................................................................... 17
   4.1 AutoStar .................................................................................................................. 17
      4.1.1 Task Core Library ......................................................................................... 18
4.1.2. Task States................................................................. 19
4.1.3. Task Threading ...................................................... 19
4.2 TEM Object Model Proxy (OMP) ..................................... 20
4.3 Instrument Object Model (IOM) ...................................... 20
4.4 4+1 View Model .......................................................... 21
4.5 TICS Analyzer .............................................................. 22

5. Feasibility Analysis .......................................................... 23
5.1 Prototype Goals and Requirements .................................... 23
5.2 Prototype Design .......................................................... 24
5.2.1. Wrapper Layer ..................................................... 26
5.2.2. Core Layer ............................................................ 27
5.2.3. Communication Layer ............................................ 28
5.2.4. Delphi – C++ Interaction ......................................... 28
5.2.5. Acquiring an image ................................................ 31
5.2.6. Displaying Images and Markers via Callbacks ............... 33
5.3 Prototype Evaluation ..................................................... 33

6. System Requirements ...................................................... 37
6.1 Introduction ................................................................. 37
6.2 Customer Objective and Application Views ......................... 37
6.3 Use Cases ................................................................. 39
6.4 Functional View .......................................................... 42
6.4.1. Functional Requirements ........................................ 42
6.4.2. Non-Functional Requirements .................................. 43

7. System Architecture & Design ........................................... 45
7.1 Introduction ................................................................. 45
7.2 Logical View ............................................................... 46
7.2.1. Rationale for Extension ....................................... 46
7.2.2. Design Choices ................................................... 46
7.2.3. Task Core Library Extension .................................. 50
7.2.4. User Actions ....................................................... 52
7.2.5. State Transitions .................................................. 52
7.3 Development View ....................................................... 54

8. Implementation .............................................................. 57
8.1 Introduction ................................................................. 57
8.2 CountToElectron Algorithm ........................................ 57
8.3 Algorithm Implementation ........................................... 59
8.3.1. CountToElectron Flow ....................................... 60
8.3.2. Threshold Computation ....................................... 61
8.3.3. Conversion Factor Computation .............................. 62

9. Verification & Validation .................................................. 63
9.1 Introduction ................................................................. 63
List of Figures

Figure 1. TEM Control Software ................................................................. 2
Figure 2. Stakeholder Matrix ............................................................. 8
Figure 3. Calibration Component in TEM software stack ....................... 11
Figure 4. Business Roadmap: Manual to Automated Tests ...................... 12
Figure 5. Product Roadmap ................................................................. 13
Figure 6. AutoStar in TEM Software ................................................... 17
Figure 7. AutoStar Task Core Library .............................................. 18
Figure 8. Task States and Possible Transitions .................................. 19
Figure 9. Object Model Proxy in Software Stack ................................ 20
Figure 10. Server exposing IOM interfaces ......................................... 21
Figure 11. 4+1 View Model ................................................................. 21
Figure 12. C++ Calibration Prototype (Calibrator) in TEM software stack ... 24
Figure 13. Prototype Design ................................................................. 25
Figure 14. Wrapper Layer ................................................................. 26
Figure 15. Core Layer ...................................................................... 27
Figure 16. Communication Layer ....................................................... 28
Figure 17. Calibrator Object Creation .............................................. 31
Figure 18. Callback Setup ................................................................. 32
Figure 19. Image Acquisition .............................................................. 32
Figure 20. Display Acquired Images via Callback .................................. 33
Figure 21. Displaying message in Delphi UI ...................................... 34
Figure 22. Multiple Image Acquisition .............................................. 35
Figure 23. Image Markers ................................................................. 35
Figure 24. CAFCR model ................................................................. 37
Figure 25. Customer objectives and Application View: CAFCR model ....... 38
Figure 26. AutoStar architecture ......................................................... 45
Figure 27. Task Core Library Extension – Option 1 .............................. 47
Figure 28. Task Core Library Extension – Option 2 .............................. 48
Figure 29. Task Core Library Extension – Option 3 .............................. 50
Figure 30. Task Core Library Extension ............................................. 51
Figure 31. Accept User Action ............................................................ 52
Figure 32. Pause Request ................................................................. 53
Figure 33. Abort Sequence ................................................................. 53
Figure 34. Calibrations in AutoStar – Package Organization ................. 54
Figure 35. CountToElectron Algorithm .................................................. 58
Figure 36. CountToElectron Algorithm in AutoStar ............................. 59
Figure 37. CountToElectron Algorithm Sequence ................................ 61
Figure 38. Threshold Computation Sequence ..................................... 62
Figure 39. Conversion Factor Computation Sequence ........................... 62
Figure 40. TDD Cycle ..................................................................... 64
Figure 41. Incremental and Iterative Approach .................................... 71
Figure 42. Project Planning – High-Level Activities .............................. 72
Figure 43. Project Planning – Detailed Activities ................................ 73
Figure 44. Milestone Trend Analysis .................................................. 76
Figure 45. Resolving power of microscopes ........................................ 85
Figure 46. Transmission Electron Microscope compared with a slide projector 86
Figure 47. Transmission Electron Microscope (TEM) ............................ 87
Figure 48. Components of an Electromagnetic Lens System .................. 88
List of Tables

Table 1. Stakeholder Analysis: PSG Group.............................................................. 6
Table 2. Stakeholder Analysis: Calibrations Group............................................. 6
Table 3. Stakeholder Analysis: AutoStar group .................................................... 7
Table 4. Stakeholder Analysis: Systems Group..................................................... 7
Table 5. Stakeholder Analysis: University Group.................................................. 7
Table 6. Communication Plan .............................................................................. 9
Table 7. Task State Description ......................................................................... 19
Table 8. DLL vs OBJ ......................................................................................... 29
Table 9. Calling conventions ............................................................................. 29
Table 10. “Flattening” the object vs COM interface comparison ...................... 30
Table 11. Use Case: Execute a calibration algorithm ........................................ 39
Table 12. Use Case: Pause a calibration algorithm .......................................... 40
Table 13. Use Case: Resume a calibration algorithm ....................................... 40
Table 14. Use Case: Abort a calibration algorithm ........................................ 40
Table 15. Use Case: Repeat a calibration step ................................................. 41
Table 16. Use Case: Skip a calibration step..................................................... 41
Table 17. Use Case: Continue the next calibration step .................................... 41
Table 18. Use Case: Accept calibration results .................................................. 42
Table 19. Use Case: Display a microscope image ............................................. 42
Table 20. Functional Requirements .................................................................. 43
Table 21. Non-Functional Requirements ............................................................. 44
Table 22. Calibration code coverage metrics .................................................... 65
Table 23. Cyclomatic Complexity Metric .......................................................... 66
Table 24. Risk Exposure .................................................................................... 74
Table 25. Risk Analysis ..................................................................................... 75
Table 26. Comparison of electron sources ....................................................... 87
1. Introduction

This project is being carried out as a graduation assignment for the Software Technology Professional Doctorate in Engineering (PDEng) program (OOTI) of the Eindhoven University of Technology (TU/e). It is a two-year doctorate level program that focuses on strengthening the technical and non-technical competencies of trainees. This is achieved by focusing on the design and development of software in an industrial setting. The first 14 months involve advanced training and education, including three small industry-driven training projects. This is followed by a major 10-month design project in a company [1].

1.1 Context

The project titled “Redesigning Calibration Algorithms for Transmission Electron Microscopes” is the 10-month OOTI design project being carried out at Thermo Fisher Scientific. Thermo Fisher Scientific is a world leader in serving science with products in various segments ranging from Analytical Instruments, Life Science Solutions, Specialty Diagnostics as well as Laboratory Products and Services. Electron Microscopes were added to the Analytical Instruments product range through the acquisition of Field Electronics and Ion Company (FEI) in 2016.

Microscopes can be broadly divided into three categories: optical, charged particle (electron and ion), and scanning probe. A charged particle microscope uses charged particles and electromagnetic or electrostatic lenses to see objects as small as a tenth of a nanometer. A Transmission Electron Microscope (TEM) is an example of a charged particle microscope. More details on the three microscope types and the operating principles of TEM can be found in Appendix A.

There are four main components to a TEM: an electron optical column, a vacuum system, the necessary electronics, and a control software. The electronics includes lens supplies for focusing, magnifying, and deflecting the beam and the high voltage generator for the electron source. These components are spoken about in more detail in Appendix A.

The control software can be partitioned into a server software and an application software. The server software, called the TEM server, provides microscope control and safety. The application software implements advanced functionalities and user interfaces. Figure 1 shows a representation of the control software.

With an increasing complexity of electron-optical techniques and a focus on automation, more and more microscope operations are being added to the control software. This not only allows the microscope to be operated by users with less specialized training but also increases the complexity of the control software.

This project deals with the calibration algorithms used as part of the control software for calibrating TEMs. Calibration is the process of understanding how an instrument / device behaves by establishing a relationship between a known value (standard) and a measured value. In general terms, a calibration algorithm is a procedure that asks the microscope for a certain tilt or shift and then measures what the system delivers. This relation is then used by the microscope software to calibrate its responses for accurate results.
This project is about redesigning the calibration algorithms, which are currently written in Delphi, in C++. These calibration algorithms are exposed to the end-users through a user interface, which is also in Delphi. The current calibration algorithms have the following drawbacks:

- The current calibration algorithms are part of a legacy software stack. Here, legacy implies software without any or minimal tests [2]. Such code is difficult to maintain or extend for adding new features, limiting improvement options.
- Testing for each software release is done manually either on the microscope or on a Virtual Machine (VM). This is a waste of time of a skilled software engineer.
- Delphi is not a popular programming language. Hence, it is difficult for Thermo Fisher Scientific to find skilled software engineers who know the language or want to learn it.
- The current algorithms need manual interaction to calibrate the microscope. This is a waste of time of a skilled system engineer that can be reduced.
- The current implementation heavily uses COM interfaces. COM interfaces once published cannot be changed. This introduces strong coupling and high maintenance overhead.

The goal of the project is
- To redesign the calibration algorithms in C++
- To make them testable and thus, maintainable
- To redesign such that further automation is possible
- To redesign such that the users are immune to the changes (redesigned algorithms should work with existing Delphi UI)

The C++ component in which the calibration algorithms must be redesigned is called AutoStar. AutoStar is a component that provides a framework, called Task Core Library, to automate algorithms to be run on the microscope. In total, there are 17 calibration algorithms. However, it is not feasible to redesign all of them within the project duration. Instead, the objective is to create a design that achieves the above-mentioned goals and to prove its correctness by redesigning one or more calibration algorithms. The Delphi calibration algorithm user interface redesign is outside the project scope.
I used an incremental and iterative approach to execute the project. Following this approach, I divided the project into multiple increments and executed each increment in iterations. Each increment involved gathering requirements, designing, implementing the design, testing, and evaluating the results. I divided the project into two phases. The first phase involved developing a prototype to prove the feasibility of the project. The second phase involved redesigning the first calibration algorithm in AutoStar.

During the redesign in AutoStar, I identified that the AutoStar Task Core Library must be extended to support the calibration algorithms. Post extension, the Task Core Library underwent changes that lead to rework of the extension. Using a Test-Driven Development (TDD) approach to implementation and adding missing functionality in the lower-layer components lead to more development time than estimated. This resulted in development of only 50% of the first algorithm. The unit and module tests written as a part of TDD implementation were sufficient to prove the correctness of the design.

The results show that the AutoStar framework can be extended to realize the calibration algorithms in C++. A TDD approach and use of design principles like Dependency Inversion ensures that the redesigned calibration algorithms are testable. Use of design principles (such as Single Responsibility principle, Dependency Inversion principle) and use of template method design pattern ensure that the algorithms are modifiable for further automation.

1.2 Outline

The purpose of this report is to give the readers a clear idea of how TEM calibration algorithms are redesigned. The next chapter introduces the project stakeholders and their role in the project. Chapter 3 explains the problem in detail and shows how the problem is aligned with the business, technology, and product roadmap of Thermo Fisher Scientific. This chapter also describes the design opportunities of the project. Chapter 4 describes the underlying domain in which the problem must be solved.

Chapter 5 presents a prototype developed to prove the feasibility of the project. This chapter describes the prototype goals, design, and the results achieved.

Chapter 6 elicits the functional and non-functional requirements of the project using the CAFCR approach. Using the requirements from Chapter 6, Chapter 7 presents the design for the calibration algorithms. It presents the design choices made during the redesign along with their justification. Chapter 8 describes one of the calibration algorithms chosen for implementation. Further, this chapter shows how this algorithm was implemented. Chapter 9 describes how the redesigned algorithms were validated using the TDD approach. It also presents some metrics that were computed to fulfill certain requirements.

Chapter 10 shows the results of the project and traces them back to the functional and non-functional requirements. It also presents the conclusions that can be drawn from the project, lessons learnt and possible future work.

Chapter 11 describes how the project was planned and the risks managed. It also presents the approach followed to execute the project. Chapter 12 reflects on the project and revisits the design opportunities.
2. Stakeholder Analysis

In this chapter, the project stakeholders are identified and the following questions answered for each:

1. What is the stakeholder role?
2. How much is his / her interest in the project?
3. How much influence does he / she have over the project?
4. What is most important to the stakeholder (Goal of the project)?
5. How can the stakeholder contribute to the project?
6. What is the strategy for communication with the stakeholder?

The answer to the first five questions decided the communication strategy. Along with project planning, this was the first activity carried out during the project.

2.1 Introduction

A stakeholder is a person or entity who directly or indirectly influences or gets influenced by a project. Stakeholder analysis is important to:

- Identify the message to be conveyed to the stakeholders. This is important to persuade them to support and engage with the project.
- Manage stakeholder expectations by adapting the communication style to their needs.

There are two major organizations involved in this project: Thermo Fisher Scientific and the Eindhoven University of Technology (TU/e). From these organizations, there are different groups and individuals who are interested in the project. The next section provides an overview of these groups and the individuals from these groups who are stakeholders in the project.

2.2 Stakeholder Groups

2.2.1. Project Steering Group (PSG)

This group is composed of stakeholders from the company and the university. This group includes the trainee, a university supervisor, and a company supervisor. The main goal of this group is to ensure that both the university and company requirements are met for a successful project. This group meets periodically to track the progress of the project and provides constructive feedback to the trainee to improve the project quality and to ensure process adherence. The stakeholders in this group are described in Table 1.
Table 1. Stakeholder Analysis: PSG Group

<table>
<thead>
<tr>
<th>Stakeholder</th>
<th>Role</th>
<th>Goal</th>
<th>Contribution</th>
</tr>
</thead>
</table>
| Paul Janson          | Company Supervisor and Calibration Algorithm Owner | • Redesign calibration algorithms to make them maintainable           | As a project mentor and supervisor,  
• Helps with the domain, infrastructure, and environment  
• Reviews the redesigned calibration algorithms  
• Reviews report write-ups (for content and confidentiality)  
• Evaluates the trainee (Initial, Intermediate, and Final) |
| Andrei Jalba         | University Supervisor (TU/e)       | • Project process is followed  
• A good report is delivered that meets the university requirements   | • Reviews the final report  
• Reviews the process followed during the project  
• Provides inputs for algorithm improvements  
• Evaluates the trainee (Initial, Intermediate, and Final) |
| Nityanand Panpalia   | OOTTI Trainee                       | • Successful project completion by fulfilling company and university criteria for on-time graduation | • Owns the project |

2.2.2. Calibrations Group

This group from Thermo Fisher Scientific is responsible for the planning, development, testing, and maintenance of the calibration algorithms for TEM microscopes. Currently, most of the calibration algorithms are developed in Delphi. Once the calibration algorithms have been redesigned in C++, this group will merge with the AutoStar group. The stakeholders from this group are described in Table 2.

Table 2. Stakeholder Analysis: Calibrations Group

<table>
<thead>
<tr>
<th>Stakeholder</th>
<th>Role</th>
<th>Goal</th>
<th>Contribution</th>
</tr>
</thead>
</table>
| Ronnie Smets      | Scrum Master                  | • Track Progress  
• Effort Estimations – Indicators can provide guidance for other legacy code re-design | • Reviews project plan and suggest improvements |
| Frank Cornelissen | System Test and Integration Engineer | • Develop test framework for system level testing of redesigned algorithms | • Provides a microscope test slot for testing calibration algorithms  
• Provides results of system level tests |

2.2.3. AutoStar group

The AutoStar group from Thermo Fisher Scientific is responsible for developing and maintaining the AutoStar framework. The AutoStar framework is a framework for prototyping and developing functionality to automate microscope usage. Since calibration algorithms need to be redesigned for automation, they must conform to the AutoStar framework. As a result, there are stakeholders from the team who are explained in Table 3.
Table 3. Stakeholder Analysis: AutoStar group

<table>
<thead>
<tr>
<th>Stakeholder</th>
<th>Role</th>
<th>Goal</th>
<th>Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Andrea Pasqualini</td>
<td>Architect</td>
<td>● Redesigned algorithms should be consistent with the Auto-Star framework.</td>
<td>● Reviews project requirements</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>● Reviews redesigned calibration algorithms for their adherence to Auto-Star framework.</td>
</tr>
<tr>
<td>Peter Van Merkerk</td>
<td>Senior Software Designer</td>
<td>● Redesigned calibration algorithms should adhere to AutoStar framework, process, and coding guidelines.</td>
<td>● Provides pointers to AutoStar framework, coding guidelines, tool, and infrastructure (development and testing)</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>● Reviews the code</td>
</tr>
</tbody>
</table>

2.2.4. Systems Group

This group at Thermo Fisher Scientific is responsible for improving existing and developing new microscope functionalities. To do so, they extensively use the existing algorithms. They also know a lot about in-field usage of the calibration algorithms. This group primarily comprises of physicists and people with a non-software background. The stakeholders from this group are described in Table 4.

Table 4. Stakeholder Analysis: Systems Group

<table>
<thead>
<tr>
<th>Stakeholder</th>
<th>Role</th>
<th>Goal</th>
<th>Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Walter Van Dijk, Tjerk Spanjer, and Andreas Voigt</td>
<td>Scientist and Application Expert</td>
<td>● Improve algorithm feedback and automation</td>
<td>● Helps understand the existing calibration algorithms</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>● Provides input for better algorithm automation based on their in-field usage experience</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>● Provides input for algorithm improvements</td>
</tr>
</tbody>
</table>

2.2.5. University Group

This group is primarily composed of the program director and professional coaches who help the trainee with different aspects of the project. The stakeholders from this group are explained in Table 5.

Table 5. Stakeholder Analysis: University Group

<table>
<thead>
<tr>
<th>Stakeholder</th>
<th>Role</th>
<th>Goal</th>
<th>Contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Yanja Dajsuren</td>
<td>Program Director</td>
<td>● University requirements are met for a successful project</td>
<td>● Reviews the process followed during the project</td>
</tr>
<tr>
<td>Judith Strother</td>
<td>Professional Coach for Technical Report</td>
<td>● Report meets the technical report standards (university and industry)</td>
<td>● Reviews the drafts of the report for language and presentation improvements</td>
</tr>
</tbody>
</table>

2.3 Stakeholder Matrix

A stakeholder matrix indicates the level of influence and interest each stakeholder has in the project. It also demonstrates how much a stakeholder wants to be aware of the project status and progress. The goal of creating this matrix is to:

- Tailor the communication plan for each stakeholder
- Identify the stakeholders who are in High influence – Low interest group. This helps to decide on a strategy to increase their interest in the project.
As seen in Figure 2, there are four matrix groups.

1. High influence – High Interest – The stakeholders in this group are the key players in the project who are involved in decision making. It is important to engage and consult with them regularly to avoid surprises and deliver a successful project. The communication plan should reflect such an engagement. The PSG group along with Mr. Andrea Pasqualini from the AutoStar group fall under this category.

2. High Influence – Low Interest – The stakeholders in this group are important for the project but do not show sufficient interest. The goal is to increase their level of interest. If that is not possible, it is important to opportunistically consult and engage them in common interest areas.

Mr. Walter Van Dijk has a higher interest than the other stakeholders in the Systems group since he is the domain expert for the Calibrations group.

2.4 Communication Plan

As shown in Table 6, a communication plan was created based on the stakeholder matrix, stakeholder goals, and their potential contribution.
The PSG group is the one that is most closely involved in the project. Further, the stakeholders from this group fall in the High Influence – High Interest category. Hence, they need to be continuously engaged and consulted with. This is reflected in the communication plan above. However, the communication method and frequency varies from one stakeholder to another (even if they fall in the same influence-interest category). For example, with Mr. Paul Janson, a daily meeting was agreed upon since he works on the algorithm development and is interested in the day to day progress. On the other hand, a weekly sync was sufficient with Mr. Andrea Pasqualini to discuss the design. Design reviews and discussions did not need a daily meeting.
3. Problem Analysis

This chapter describes the problem in detail. It provides information on how the problem arose and why it needs to be addressed. Further, this chapter also relates the problem to the company’s business and technology roadmap.

3.1  Context

A TEM is composed of several different hardware elements. TEM is inherently complex due to the amount and configuration of internal hardware that it contains. TEMs can generate high quality, high resolution images. To generate such images, the microscope needs to be calibrated. Calibration algorithms are developed as part of a TEM control software stack. It has a user interface and an algorithm code, both of which are currently written in Delphi. Figure 3 shows the calibration algorithms in the TEM software stack.

![Figure 3. Calibration Component in TEM software stack](image)

As shown in Figure 3, the User Interface interacts with the **Calibration** component via the **ICalibration** interface. The **Calibration** component in turn talks to the TEM server via adaptors for performing microscope operations and acquisitions. The adaptors talk to the TEM server via the IOM interface. In some cases, the Calibration component bypasses the adaptors and directly accesses the TEM server using the IOM interface. IOM stands for Interface Object Model and is the interface to the TEM server. Adaptors are components, written in COM, that make interfaces available for different programming languages and technologies. IOM and Adaptors are described in more detail in **Chapter 4**.
3.2 Roadmaps

The sections below explain the business, technology and product roadmap for TEM software.

3.2.1. Business Roadmap

Thermo Fisher Scientific recently transitioned from releasing the TEM software once every six months to once per quarter. This change was undertaken to satisfy customers who demanded a faster turnaround time for their software issues. Currently, most of the calibration algorithm testing is done manually either on the microscope or on the Virtual Machines (VMs). There are minimal smoke tests that run on VMs simulating different microscope configurations. Currently, there are no unit tests run on the local machine. Figure 4 depicts this scenario as an inverted triangle on the left side. The microscope time is limited as they are in high demand for testing other components. As a result, it takes around four to six weeks for every version to be tested before release. This puts significant pressure on the developers to fix bugs that are found very close to the release dates. This testing time must be minimized to sustain the recent transition to release TEM software every quarter. The sustenance can be achieved by automating majority of the tests and reducing the dependency on the VMs and microscopes as shown in the triangle on the right side of Figure 4.

Figure 4. Business Roadmap: Manual to Automated Tests

As the non-inverted triangle shows, majority of the tests should be automated and run on the local machine as unit tests. A few scenarios that cannot be tested on a local machine should be tested on the VM. Rare scenarios that can neither be tested on a local machine nor a VM should be tested manually on the microscope.

The market focus for TEM software expands towards application development, robustness, ease of use, and automation. The broad spectrum of product configurations adds a maintenance burden, thus slowing down development. This increases the need for having a robust software with a strong test suite to reduce the maintenance time and costs.

3.2.2. Technology Roadmap

At Thermo Fisher Scientific, the scientists work on leading cutting-edge research in both hardware and software. The scientists usually develop prototypes to prove the feasibility of their research. Once the feasibility is proven, the software teams convert the prototype into production quality code by integrating the prototype into the existing software stack. In the case of calibration algorithms, this conversion of research to production code could mean either improving an existing algorithm or adding new ones. However, the legacy nature of the algorithms makes it very difficult to make such updates possible. Hence, there is a need to redesign the calibration algorithms so that the latest research in calibration algorithms can be productized.

The current calibration algorithms are developed in Delphi. Delphi is not a very popular programming language, making it difficult for Thermo Fisher Scientific to hire skilled engineers who either know the language or want to learn it. This is another technology reason to redesign the calibration algorithms.
3.2.3. Product Roadmap

While the calibration algorithms are written in Delphi, the TEM server is developed in C++. The server provides interfaces to its clients for controlling and performing operations on the microscope. A client is any component that requests the TEM server for a service that it provides. These interfaces are developed using Component Object Model (COM) to enable clients to be developed in a different programming language or platform. As shown in Figure 3, the current calibration algorithms use adaptors which are COM interfaces to access TEM server functionality. There are multiple adaptors, each providing access to a specific TEM server functionality. In some cases, the calibration algorithms directly make calls to the TEM server, bypassing the adaptors.

Figure 5 shows the future roadmap to use a layered architecture with applications calling TEM Object Model Proxy (OMP) in the server layer. The TEM OMP calls the IOM interface, to access and control TEM server functionality. AutoStar is the application layer component in which the calibration algorithms need to be redesigned. AutoStarServer component is a COM server that enables communication between a user interface, developed in Delphi and AutoStar, developed in C++. The AutoStar and AutoStarServer components are described in detail in Section 4.1.

The COM dependency with adaptors needs to be removed because changing COM interfaces once published breaks binary compatibility as well as backward and forward compatibility. If a new functionality has to be introduced, a new COM interface must be added. If a functionality has to be deleted, the entire COM interface providing access to the functionality must be deleted. This introduces strong coupling and high maintenance overhead. As a result, COM dependency should be reduced.
As TEM software undergoes changes to realize the market requirements, as far as possible, the users should be kept immune to these changes. In case of calibration algorithms, it would mean that the user interface stays the same while the algorithms are redesigned in AutoStar. In addition, the current lack of automation means that the calibration algorithms need an operator to interact with the microscope. This is a drawback since it consumes the time of a skilled system engineer, which can be put to better use. Hence, it is important to automate the calibration algorithms to eliminate operator interaction.

3.3 Project Scope and Goals

Based on the various roadmaps for TEM software, the goal of the project is

- To redesign the calibration algorithms in C++
- To make them testable and thus, maintainable
- To redesign such that further automation is possible
- To redesign such that the users are immune to the changes (redesigned algorithms should work with existing Delphi UI)

As shown in Figure 5, the C++ component in which the calibration algorithms must be redesigned is AutoStar. In total, there are 17 calibration algorithms. However, it is not feasible to redesign all of them within the project duration. Instead, the objective is to create a design that achieves the above-mentioned goals and to prove its correctness by redesigning one or more calibration algorithms. The Delphi UI redesign is outside the project scope.

The Delphi UI – AutoStar connection was deemed out of scope due to a risk identified midway through the project (see risk CS_R_011). The risk identified was that AutoStar is a 64-bit component while the Delphi UI is 32-bit. This means that the two cannot interact directly. Rather, the AutoStarServer component (see Figure 5), which is a COM server, must be modified so that the two components can interact. As a result, the last goal of keeping the users immune to the changes, was deemed beyond the scope of the project.

3.4 Design Opportunities

3.4.1 In Scope

Based on the project scope and goals, the following design opportunities arise.

3.4.1.1 Maintainability

One of the key quality requirements of the project is maintainability. Maintainability touches three different aspects [3]:

- The skills of the developer performing maintenance
- Technical properties of the system under maintenance
- Requirements engineering

For this project, the skills of the developer are an organizational aspect which is out of scope. Maintainability is defined only in context of the other two aspects – the technical properties of the system and requirements engineering.

From a technical property standpoint, as per ISO/IEC 25010 standard, maintainability has several sub-characteristics, namely, modularity, reusability, analyzability, modifiability, and testability [4]. Two of these sub-characteristics, namely testability and modifiability, are potential design opportunities for the project.

Testability

According to ISO/IEC 25010, testability specifies the degree to which a software facilitates testing in a given test context [4]. In other words, it measures the ability to detect, isolate, and fix defects. Hard-to-test software is difficult to modify. Having a
good test coverage ensures that changes can be made with confidence. A failing test can quickly point to the code that causes a functionality to break. This is because tests can be run against changes made to a baseline. Any failures will be in the changes added on top of the baseline. Further, good design principles can also aid in isolating defects. The calibration algorithms need to be redesigned such that it is easy to write tests as well as easy to detect, isolate, and fix defects.

**Modifiability**
According to ISO/IEC 25010, modifiability specifies the degree to which a product or system can be effectively and efficiently modified without introducing defects [4]. The future roadmap of calibration algorithms involves automating them as well as improving the algorithm logic. This will only be possible if the redesigned calibration algorithms are modifiable. Good design principles play an important role in making a software modifiable. These principles need to be analyzed and the right ones chosen to improve modifiability. Further, testability also improves modifiability because it makes it easier to run regression tests after program changes.

**3.4.1.2 Usability**
According to ISO/IEC 25010, usability specifies the degree to which a product or system can be used by users to achieve specified goals with effectiveness, efficiency, and satisfaction [4]. End-users have used the calibration algorithm UI for several years. The calibration algorithms should be redesigned in such a way that the users are not affected by the redesign. They should be able to use the UI which they understand and are comfortable using since many years. In other words, the usability of the calibration algorithms should not be affected by the redesign.

**3.4.1.3 Extensibility**
Extensibility specifies the degree to which a product or a system can be extended. An extension could either mean adding a new functionality or extending an existing functionality. Since redesigning all 17 calibration algorithms is out of scope for the current project, the new design should be extensible. The design should allow calibration algorithms to be easily added.

**3.4.2. Out of Scope**
The following design criteria are out of scope for the project and hence not considered.

**3.4.2.1 Portability**
According to ISO/IEC 25010, portability is the degree to which a system, product or component can be transferred from one hardware, software or other usage environment to another. Portability is composed of three sub-characteristics, namely, adaptability, installability, and replaceability [4]. Installability is a design criterion for the AutoStar component of which the calibration algorithms will be a part. Installability is the degree to which a specific product or system can be successfully installed and/or uninstalled in a specified environment [4]. It is a criterion for the component as a whole and out of scope for the project.

**3.4.2.2 Security**
According to ISO/IEC 25010, security is the degree to which a system or product protects information and data so that persons or other products or systems have the degree of data access appropriate to their types and levels of authorization [4]. Confidentiality is a security aspect which is a design criterion for AutoStar. AutoStar handles this aspect by using licenses. The calibration algorithms also need to address confidentiality since not all algorithms should be made available for all users. The use of licenses needs to be investigated to see if it can serve the purpose for calibrations. However, this aspect has been kept out of scope for the project.
4. Domain Analysis

To successfully develop a solution, it is important to understand the system domain. The term domain refers to the technologies, tools, and methodologies that are used to develop a solution. This chapter discusses the domain in which calibration algorithms need to be redesigned.

4.1 AutoStar

Figure 6 shows where AutoStar resides in the overall TEM software stack.

![Figure 6. AutoStar in TEM Software](image)

The user interface is Delphi UI that users can use to control the microscope. The UI interacts with the AutoStarServer. AutoStarServer is a COM server that enables communication between a user interface, developed in Delphi and AutoStar, developed in C++. This COM server is needed as an intermediary because the UI is a 32-bit application while AutoStar is a 64-bit component. Hence, it is not possible to directly make calls to AutoStar interfaces from the Delphi UI. AutoStar is a component for prototyping and developing functionality to automate microscope control. One of the project requirements (see function requirement CS_FR_001) is that the calibration algorithms must use the AutoStar framework for redesign, which is a framework that supports automation. This means that the calibration component shown in Figure 3 must become part of the AutoStar component shown in Figure 6. Due to the 32-bit–64-bit incompatibility between the UI and AutoStar component, the AutoStarServer component would also have to be modified to redesign the calibration algorithms in AutoStar. This incompatibility was identified mid-way through the project. TEM OMP is an abstraction on top of TEM server that is used by AutoStar to access and control the microscope.
4.1.1. Task Core Library

AutoStar provides a task core library that must be used by every algorithm to be automated. The structure of this library is shown in Figure 7. The task core library is composed of the common interface (ITask) and base class (Task template class).

![Diagram of Task Core Library](image)

**Figure 7. AutoStar Task Core Library**

A task is a unit of work, consisting of one or more actions on the microscope that is atomic. In other words, every automated algorithm implemented in AutoStar is a task. One AutoStar task running on the microscope cannot be interrupted by another task. All tasks share a common interface ITask, as shown in Figure 7 that is used for

- Starting the tasks
- Pausing and aborting tasks which are interruptible
- Subscribing to task state change, execution progress, and task messages
- Accessing subtasks

To minimize the implementation effort of tasks, the Task template base class has been provided, as shown in Figure 7, which implements the ITask functions. An algorithm implementation must derive from the Task template base class. A task can be in one of eight possible states, namely, Idle, Running, Pausing, Aborting, aborted, Failed, and Completed. Figure 7 shows that the state transitions are handled by the TaskStateMachine class while the states are held in the TaskState enumeration. These states are explained in Section 4.1.2.
4.1.2. Task States

As mentioned in the Section 4.1.1, a task can have eight possible states, namely, Idle, Running, Pausing, Paused, Aborting, Aborted, Failed, and Completed. These states are listed and described in Table 7.

<table>
<thead>
<tr>
<th>State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Idle</td>
<td>Task has not been started yet</td>
</tr>
<tr>
<td>Running</td>
<td>Task is currently executing</td>
</tr>
<tr>
<td>Pausing</td>
<td>Task has been requested to pause but is still executing</td>
</tr>
<tr>
<td>Aborting</td>
<td>Task has been requested to abort but is still executing</td>
</tr>
<tr>
<td>Paused</td>
<td>Task has been paused</td>
</tr>
<tr>
<td>Completed</td>
<td>Task has been completed successfully</td>
</tr>
<tr>
<td>Failed</td>
<td>Task execution encountered an error</td>
</tr>
<tr>
<td>Aborted</td>
<td>Task execution has been aborted</td>
</tr>
</tbody>
</table>

Figure 8 shows the lifecycle of a task.

![Figure 8. Task States and Possible Transitions](image)

A task always starts in an *Idle* state. When a task is executed, it goes into the *Running* state. During the task execution, it is possible to pause the task. The task then enters the *Pausing* state from where it can either go into either a *Paused* or *Aborting* state. A task cannot go directly from *Running* to *Paused* or *Aborted* state since it must finish what it is executing and reach a stage where it is safe to pause or abort. This is achieved by introducing *Pausing* and *Aborting* states. The *Running*, *Pausing*, and *Aborting* states are called as active states since the task is still executing in these states. If the task completes successfully, then it enters the *Completed* state. If it fails, then it enters the *Failed* state. A task can also be aborted by a user that makes it enter the *Aborted* state via the *Aborting* state. The *Completed*, *Failed*, and *Aborted* states are called end states. The only way to come out of these states is to restart the task execution that will make the task enter the running state again.

4.1.3. Task Threading

The task core library does not create threads. ITask’s *Execute* method is synchronous, that is, it returns only after the task has completed, failed or aborted. This implies that
the clients need to use another thread to pause or abort task execution. For instance, let us assume that a task is being executed by the UI on thread 1. For some reason, the user decides to pause the task. The UI must create a new thread, say thread 2, to call the pause method. Thread 2 will pause the execution of thread 1. Event callbacks are performed in the thread context of the caller of ITask’s Execute method.

4.2 TEM Object Model Proxy (OMP)

The TEM Object Model Proxy or OMP in short, is a server abstraction layer on top of the IOM interface of the TEM server. This is shown in Figure 9.

![Figure 9. Object Model Proxy in Software Stack](image)

The TEM OMP is used by the various applications, including the AutoStar component to access and control the TEM server. It
- Provides a C++ API for the TEM server functionality
- Hides COM as far as possible
- Hides IOM interface version differences
- Provides a layer to add workaround for bugs in TEM server

A major challenge for IOM clients is that IOM interfaces change frequently and these changes are often not backward compatible. Due to the way COM method calls are resolved (through vtable index), the client must be recompiled against the new IOM interface. In other words, a separate client binary must be created for each IOM version. To alleviate the burden on the applications, a TEM OMP component provides a constant C++ interface regardless of the server version on the system.

4.3 Instrument Object Model (IOM)

IOM is a collection of COM interfaces that clients use to communicate with the TEM server. IOM interfaces have a well-defined structure with respect to the microscope. The only way to access these interfaces is via a top-level Instrument as shown in Figure 10.
The instrument represents the overall microscope. For instance, if a client application wants to access the FluScreen, it must request an instrument object first. Using the instrument object, it must request the Detector object and then use this object to access the FluScreen object.

**4.4 4+1 View Model**

Figure 11 shows the 4+1 view model that describes the architecture of a system.

- **Logical View** – The logical view describes the functionality provided by the system to the end users. The logical view diagrams that have been used in this project to describe the system functionality are class diagrams, sequence diagrams, and state diagrams. This is usually one of the first views to consider along with the scenarios while designing a system.
- **Process View** – The process view describes the concurrency and synchronization aspects of a system. Aspects like timing, resource usage, and fault tolerance are handled by the process view. The project uses activity diagram from the process view to describe the Count2electron algorithm execution sequence.
• Development View – The development view describes the organization of the system in its development environment. Component diagrams and package diagrams from the development view have been used in this project.

• Physical View – The physical view describes how different components of the system are deployed on the hardware. No diagrams from this view have been used in this project.

• Scenarios or Use Cases – Scenarios describe the behavior of the system as seen by the users of the system. Use case descriptions which explain usage scenarios have been used in this project to gather the customer requirements.

The 4+1 view model has been extensively used across different chapters to describe the architecture, design, and implementation of the calibration algorithms (see chapters 7 and 8) and the prototype design (see chapter 5). The 4+1 view model has also been used to analyze the problem (see chapter 3) and the domain (see current chapter).

4.5 **TICS Analyzer**

TICS analyzer is a tool that generates coding standard violations for a software. The goal of the analyzer is to improve the software code quality. The quality is specified by a set of rules which can be modified. These rules have been categorized into ten different levels. Higher the level, the more important the rule. To improve code quality, TICS Analyzer has been used in this project. Further, a requirement has been added to fix all violations of rules above Level 8 (requirement CS_NFR_004). ■
5. Feasibility Analysis

As part of feasibility study, a C++ prototype of the calibration algorithms was developed. This chapter explains the features of this prototype along with the motives behind developing it. Although there were several motives, the most important one was to prove the feasibility of certain concepts, which determined if the project was doable.

5.1 Prototype Goals and Requirements

Due to the complexity of the domain and the lack of documentation on the calibration algorithms, first, a prototype was developed. The prototype was developed to achieve the following goals:

- To demonstrate progress to the stakeholders to increase their interest and subsequently, their involvement in the project
- To understand the current and future calibration algorithm domain
- To learn Delphi programming language to comprehend the calibration algorithms and their interaction with other components
- To feed the knowledge gathered by developing the prototype into the new design
- To prove feasibility of certain features or concepts. The most difficult feature, namely, image display via callbacks, was chosen as a requirement to confirm the feasibility of the project

The following requirements were identified for the prototype. These requirements were derived from the overall system requirements, described in Chapter 6, to prove that the project is feasible. The requirements are listed below.

- PROTO_R_001: The prototype should demonstrate that the Delphi calibration UI can display messages passed from the C++ calibration algorithms (derived from functional requirement CS_FR_013).
- PROTO_R_002: The prototype should display multiple images captured by the C++ calibration algorithm in Delphi UI (derived from functional requirement CS_FR_011).
- PROTO_R_003: The prototype should prove that image information can be passed to the Delphi UI via callbacks. As per [6], for Delphi and C++ programming languages, a callback is a function (in most cases, a pointer to the function) passed as a parameter. This allows lower layers of a software stack to call a function defined in a higher layer (also derived from functional requirement CS_FR_011).
- PROTO_R_004: The prototype should prove that markers / highlighters can be drawn on these images. In other words, to draw on top of the acquired images (derived from functional requirement CS_FR_012).

Figure 12 shows how the C++ calibration prototype, circled in blue and called Calibrator, fits in the TEM software stack. The calibration user interface (UI) calls the Delphi calibration component via the ICalibration COM interface. This Delphi calibration component, in turn, calls the C++ calibrator component which implements the prototype features. The calibrator calls the TEM OMP to acquire simulated images for display in the UI.
5.2 Prototype Design

In the subsequent sub-sections, the logical view from the “4+1” view model is used to describe the design of the prototype. The “4+1” view model is a collection of five views that can be used to describe the architecture of a system [5]. These five views are explained in Section 4.4. For representation, the prototype can be thought of as being composed of three layers – wrapper layer, core layer, and communication layer.

Figure 13 shows the complete prototype along with its partitioning into the three layers. The wrapper layer is marked in blue, the core layer is marked in green, and the communication layer is marked in red. These layers are described in detail in the following sub-sections.
Figure 13. Prototype Design
5.2.1. Wrapper Layer

The top most layer of the prototype is called the wrapper layer. It serves as an entry point for calls made from the Delphi calibration component. Further, it also holds information on the callback handlers, which need to be called for displaying messages and captured images in the UI.

**Figure 14. Wrapper Layer**

*Figure 14* shows the classes that comprise the wrapper layer. The responsibilities of these classes are described below.

**CalibrationWrapper** – This class holds all the functions which can be called from the Delphi calibration component. This includes functions to execute a calibration algorithm and to setup the callback handlers. *CalibrationWrapper* class uses the *ICallBackHandler* to pass on the handler information for storage.

**CallBackHandler** – This class implements the *ICallBackHandler* interface. This class stores callback handler information and executes them by calling functions of the *CallBackEvent* class. An instance of *CallBackHandler* can have multiple *CallBackEvent* instances, one instance for each function or procedure (handler) to be called back.

**UI** – This class implements the *IUI* interface. The *UI* class is responsible for forwarding the UI calls to the Delphi calibration component by calling the correct callback handler (via the *ICallBackHandler* interface). Further, this class uses the same function names as the functions in the UI class of the Delphi calibration component. This is to maintain consistent meaning for the UI calls across the two calibration components.
ImageHelper – This class implements the IHelper interface. This class is responsible for saving the image header and data into a file using a format which the Delphi calibration component understands.

CallBackData – This class stores the callback information, i.e., pointers to the Delphi ‘object’ and ‘procedure’ (function) that need to be called. The object pointer is a reference to the Delphi class which makes the call to set the callback handler. The procedure pointer is a function pointer to a Delphi procedure. Further, the procedure pointer takes the object pointer in its parameter list.

CallBackEvent – This class provides functions to execute the callback handlers. These functions access the object and procedure reference from the CallBackData class and execute them.

5.2.2. Core Layer

The core layer holds the logic of selecting the right calibration algorithm and executing it. For the prototype, the notion of executing a calibration algorithm involves reporting some messages back to the calibration UI and acquiring simulated images from TEM OMP for display.

![Figure 15. Core Layer](image)

Figure 15 shows the classes that comprise the core layer. The responsibilities of these classes are described below.

ICalibrator – This interface acts as the entry point for the core layer. The CalibrationWrapper class from the Wrapper Layer uses this interface to execute a calibration algorithm.

Calibrator – This class implements the ICalibrator interface. It calls a factory to create the right calibration algorithm object and executes it. This makes the Calibrator independent of how the algorithm objects are created.

AlgorithmFactory – This class is a factory to create the correct calibration algorithm object.
ICalibrationAlgorithms – This interface holds the functions common across all the calibration algorithms.

Algorithm1 and Algorithm2 – These are concrete algorithm classes which are created by the AlgorithmFactory and holds the calibration algorithm logic. For the prototype, Algorithm1 reports back some UI messages and acquires microscope images, from the communication layer, for display in the calibration UI.

5.2.3. Communication Layer
The communication layer interacts with the TEM OMP interface to acquire simulated images.

![Diagram of communication layer classes]

**Figure 16. Communication Layer**

Figure 16 shows the classes that comprise the communication layer. The responsibilities of these classes are described below.

ImageAcquisition – This class implements the IImageAcquisition interface and is responsible for acquiring images from TEM OMP. Once an image is acquired, it calls the correct CallBackHandler function to display the acquired image in the calibration UI.

MicroscopeImage – This class holds the information of the acquired images, which is the image metadata and the raw image.

5.2.4. Delphi – C++ Interaction
A Delphi linker cannot link C++ objects in a Delphi application. As a result, a few conditions need to be met for calling a C++ object from a Delphi class. These conditions are discussed in detail in the following sub-sections.

5.2.4.1 Object Structure
Delphi and C++ have different object structures. On one hand, the memory layout of a Delphi class always has a pointer to the Virtual Method Table (VMT) as its first field. On the other hand, C++ classes may or may not have a VMT pointer in its memory layout. However, it is possible to force the C++ class to have a VMT by having no data members and making at least one-member function virtual [7]. In this way, the object files produced by Delphi and C++ compilers can be linked. This matching is necessary when using an interface style of sharing objects. More on this in Section 5.2.4.4.
5.2.4.2 Linking Process

There are two ways of linking source modules from Delphi and C++ – either dynamically by compiling them into Dynamic Link Libraries (DLLs) or statically by compiling into Intel relocatable object files (OBJs). The DLL and OBJ are compared in Table 8 [8].

**Table 8. DLL vs OBJ**

<table>
<thead>
<tr>
<th>Comparison Feature</th>
<th>DLL</th>
<th>OBJ</th>
</tr>
</thead>
<tbody>
<tr>
<td>Executable footprint</td>
<td>Smaller footprint since the DLL is compiled separately from the main executable</td>
<td>Larger footprint since the modules are linked statically</td>
</tr>
<tr>
<td>Maintenance</td>
<td>Easier to maintain</td>
<td>More difficult to maintain</td>
</tr>
<tr>
<td>Runtime execution</td>
<td>Slower since DLL must be loaded separately</td>
<td>Faster because there are no external calls to make</td>
</tr>
<tr>
<td>Programming Overhead</td>
<td>More overhead to match calling conventions, prevent name mangling, and know the DLL location.</td>
<td>Less overhead than DLLs</td>
</tr>
<tr>
<td>Sharing code</td>
<td>Easier to share since multiple applications can call the same DLL. A DLL bug fix will fix it for all applications</td>
<td>Sharing more complicated than DLLs and has limitations.</td>
</tr>
</tbody>
</table>

**Design Decision:** The prototype uses DLL as the linking process because

1. The programming overhead is not high due to good documentation on how to match the calling conventions and prevent name mangling. Further, a lack of clear documentation on how to use OBJs to achieve linking made it easier to choose DLL linking mechanism.
2. DLL mechanism is not much slower than OBJ since Windows uses a DLL architecture and the code for using DLLs is highly optimized.

5.2.4.3 Calling Convention

The calling convention determines

- the order in which the parameters, passed to a function, are placed on the stack
- who is responsible for maintaining the stack. i.e. if the calling function or the called function cleans up the stack

By default, Delphi’s calling convention is different from C++’s. Hence, appropriate directives need to be used to match the calling convention between Delphi and C++ code. There are four different calling conventions to choose from, namely, stdcall, cdecl, pascal, and fastcall. The four calling conventions are compared in Table 9 [8].

**Table 9. Calling conventions**

<table>
<thead>
<tr>
<th>Feature</th>
<th>Stdcall</th>
<th>Cdecl</th>
<th>Pascal</th>
<th>FastCall</th>
</tr>
</thead>
<tbody>
<tr>
<td>Parameter Ordering on the stack</td>
<td>Right to Left order</td>
<td>Right to Left order</td>
<td>Left to Right order</td>
<td>First three parameters on CPU register – EAX, EDX, and ECX. Rest on the stack from left to right</td>
</tr>
<tr>
<td>Stack Maintenance Responsibility</td>
<td>Called routine</td>
<td>Calling routine</td>
<td>Called routine</td>
<td>Called Routine</td>
</tr>
<tr>
<td>Programming language using as default</td>
<td>32-bit Windows APIs C++</td>
<td>Borland Delphi 1.0</td>
<td>16-bit Delphi 1.0</td>
<td>Delphi 2.0</td>
</tr>
<tr>
<td>C++ Directive</td>
<td>_stdcall</td>
<td>_cdecl</td>
<td>_pascal</td>
<td>_fastcall</td>
</tr>
<tr>
<td>Delphi Directive</td>
<td>Stdcall</td>
<td>CDecl</td>
<td>Pascal</td>
<td>Register (optional since it is the default calling convention)</td>
</tr>
</tbody>
</table>
**Design Decision:** Although cdecl is the default calling convention for C++, the prototype uses the stdcall calling convention since it creates smaller executables as compared to cdecl. This is because the stack cleanup is performed by the called routine or function. The stack cleanup is performed by the called routine forfastcall as well but is not used due to lack of standardization. The Cdecl convention is useful when a function takes variable number of arguments.

### 5.2.4.4 Exporting methods from C++ to Delphi

After ensuring that the object structures match, the next step is to find a way to share code between the two languages. There are two ways of doing so [7]:

1. “Flattening” the object – In C++, by default, the symbol names are mangled whereas in Delphi they are not mangled. “Flattening” an object means exporting a simple C function for each member method, the constructor, and the destructor. This is achieved by using an “extern C” directive.

2. Using COM interfaces - COM provides a specification for producing cross-language and cross application object oriented architectures. This approach uses interfaces which are defined as abstract classes. Abstract classes are pure virtual classes which ensure that the object structure between Delphi and C++ are matched (see Section 5.2.4.2). This allows sharing of objects between the two languages.

The pros and cons of the two approaches are mentioned in Table 10.

Table 10. “Flattening” the object vs COM interface comparison

<table>
<thead>
<tr>
<th>Pros</th>
<th>Using COM interfaces</th>
</tr>
</thead>
<tbody>
<tr>
<td>• One less level of indirection (no virtual function class) results in a faster execution.</td>
<td>• Much more convenient to use</td>
</tr>
<tr>
<td>• Easier and faster to code</td>
<td>• OO approach</td>
</tr>
<tr>
<td>• Portability across operating systems</td>
<td>• Windows specific</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Cons</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>• Not an OO approach since it uses C style functions</td>
<td>• Lot more coding on the C++ side to achieve this convenience. In other words, lot more source files.</td>
</tr>
<tr>
<td></td>
<td>• Execution speeds slower than “flattening” the object approach</td>
</tr>
</tbody>
</table>

**Design decision:** “Flattening” the object approach is used for the prototype since it is faster to code and has faster execution time. A non-OO approach is not a concern for achieving the prototype goals.

The code snippet given below shows how “flattening” the object approach has been used to share code between Delphi and C++. The code snippet is the C++ side of the code. It shows that the calibrator object must be created and destroyed on the C++ side. Further, the `CalibratorCreate` method creates and returns the `CalibrationWrapper` object to the Delphi code. This object is passed by Delphi to every function it calls on the C++ side. For instance, the snippet below shows the function signature of `CalibratorExecuteSelectedAlgorithm` method. This method takes the C++ object as one of its arguments and uses it to call the correct instance of the `ExecuteSelectedCalibration` method. In this way, multiple instances of `CalibrationWrapper` class are handled.

Exports.h:
```
#define DLLAPI extern "C" __declspec(dllexport)
DLLAPI void *CalibratorCreate();
```
31

DLLAPI void CalibratorDelete(void * calibrator);
DLLAPI void CalibratorExecuteSelectedAlgorithm(void * calibrator, int32_t algorithmType);

Export.cpp:
using namespace Connector;
#define C(p) ((CalibrationWrapper *) p)

DLLAPI void * CalibratorCreate()
{
    return new CalibrationWrapper();
}

DLLAPI void CalibratorDelete(void * calculator)
{
    delete calculator;
}

DLLAPI void CalibratorExecuteSelectedAlgorithm(void * calibrator, int32_t algorithmType)
{
    C(calibrator)->ExecuteSelectedCalibration(algorithmType);
}

5.2.5. Acquiring an image
One of the prototype goals was to acquire a simulated image from TEM OMP.
Acquiring an image is a three-step process described in the following sub-sections.

5.2.5.1 Object Creation and Call Back Setup
The first step is to create the CalibrationWrapper object and pass it back to the Delphi calibration component. Since the prototype uses stdcall calling convention, the C++ object creation and destruction must happen in the C++ calibrator. This is because in stdcall convention, the stack cleanup is the responsibility of the called function. Figure 17 shows how the objects are created.

![Figure 17. Calibrator Object Creation](image-url)

In Figure 17, CalibratorCreate function is called from the Delphi calibration component. The CalibratorCreate function call creates an object for some of the internal classes and returns an instance of CalibrationWrapper to the Delphi calibration component. This instance is later passed to the CalibrationWrapper destructor as a parameter for deletion.

5.2.5.2 Callback Setup
After the CalibrationWrapper object is obtained by the Delphi Calibration component, it sets up the callbacks by calling the functions defined in the CalibrationWrapper class. Figure 18 shows the sequence of function calls that happen.
There is one callback setup function defined for each callback event in the `CalibrationWrapper` class. Figure 18 shows two callback setup functions, namely, `UIWaitForNextEvent` and `DisplayAcquiredImageEvent`. Each of these two functions result in a function call to the `CallBackHandler` class for setting up the callbacks. New instances of `CallBackEvent` and `CallBackData` classes are created for each callback setup. The callback information is stored as member data in `CallBackData` class. This information is composed of pointers to the Delphi ‘object’ and ‘procedure’ (function) that needs to be called. The callback information is passed from the Delphi calibration component, as part of each callback setup call, to `CalibrationWrapper` class, i.e., as arguments to `UIWaitForNextEvent` and `DisplayAcquiredImageEvent` functions.

### 5.2.5.3 Image Acquisition

After object creation and callback setup, the next step is to acquire the simulated image from TEM OMP. Figure 19 shows the sequence of steps needed to acquire the image.

As shown in Figure 19, an image is acquired as part of a calibration algorithm execution. The call to execute a calibration algorithm comes from Delphi calibration component in the form of a function call, `ExecuteSelectedCalibration`. This results in a function call to the `Calibrator` class which asks the `AlgorithmFactory` to create the selected `Algorithm` object. As an example, an object of `Algorithm1` class is created, which in turn creates the object of `ImageAcquisition` class in its constructor. Once the `Calibrator` receives the `Algorithm1` object, it calls the `Calibrate` function on the object.
As part of the calibrate function, Algorithm1 requests the ImageAcquisition for an image from TEM OMP.

5.2.6. Displaying Images and Markers via Callbacks

Figure 20 shows how the captured images and their metadata are sent back to the Delphi calibration component.

![Figure 20. Display Acquired Images via Callback](image)

After acquiring every simulated image from the TEM OMP, the ImageAcquisition class calls the callback function DisplayAcquiredImage available in the CallBackHandler class. The raw image along with the image number are passed as arguments. Upon triggering of this event, the CallBackHandler handles the event by

- Creating an ImageHelper instance.
- Invoking the ImageHelper functions to save the image information in a file. The image information is comprised of the raw image data and image header. The image header includes the image metadata such as the height and the width as well as marker information to draw on top of the captured image.
- Calling the Execute function in the CallBackEvent class to execute the callback function.

The CallBackEvent class fetches the Delphi object and procedure from the CallBackData class. The procedure named DisplayImage is called to display the image and markers / highlighters, if any. The file path and the image number are passed as arguments to the procedure. This sequence is repeated for every image captured.

5.3 Prototype Evaluation

To conclude the feasibility analysis, a successful demonstration of the prototype was shown to the stakeholders. All the goals listed for the prototype were met.
Figure 21. Displaying message in Delphi UI

Figure 21 shows the message, circled in red, sent from C++ calibration algorithm to the Delphi UI. This shows that it is possible to display messages passed from the C++ calibration algorithms in the Delphi UI (requirement PROTO_R_001).

Figure 22 shows multiple images that were acquired from TEM OMP and passed to the Delphi UI for display. These images were passed to the Delphi UI using callback functions. This shows that it is possible to display multiple images captured by the C++ calibration algorithm in Delphi UI by passing image information via callbacks (requirement PROTO_R_002 and PROTO_R_003).
Figure 22. Multiple Image Acquisition

Figure 23 shows an enlarged view of the sixth image in the UI, titled ‘Cross-correlation’. This image shows red circles drawn on top of the cross-correlated image. These circles show that markers can be drawn on top of acquired images. In addition, the circles show that this information can be passed back to the Delphi UI for display, satisfying requirement PROTO_R_004.

Figure 23. Image Markers

Further, majority of the goals were also achieved. For example, majority of the code to acquire an image is heavily leveraged from the AutoStar implementation. This helps in understanding a part of the future calibration algorithm domain. In addition, there are some learnings for the algorithm design. The wrapper layer can become a translation component between the old Delphi calibration and new C++ calibration components. This layer can later be discarded when all calibration algorithms have been redesigned in AutoStar. The core layer logic should use the AutoStar framework since the core layer houses the calibration logic. The communication layer, which heavily leverages code from AutoStar can be discarded in the new design since AutoStar already includes the feature to acquire an image.
6. System Requirements

In this chapter, the requirements of the project are elicited. The first section provides an overview of the CAFCR method that is used to derive the requirements. These requirements are derived in the subsequent sections.

6.1 Introduction

The “CAFCR” model has been used for determining the requirements for the project. CAFCR is an acronym for five different views, namely, Customer objectives view, Application view, Functional view, Conceptual view, and Realization view. Figure 24 shows the CAFCR model.

![CAFCR Model]

The Customer objectives view deals with what the customer wants to achieve. The Application view deals with how the customer wants to realize his goals. These two views provide the justification for the Functional view which captures the requirements, both functional and non-functional. The Conceptual and Realization views describe how the product can realize the requirements which have been listed as part of the Functional view [9]. For the project, the first three views have been used to extract the requirements.

6.2 Customer Objective and Application Views

This section describes how the requirements were gathered using the first three views of the CAFCR model. The customer objectives can be captured in terms of customer key drivers. These drivers provide a direction to capturing requirements. These drivers are few and sharp and cover the most important functions of the system. The key drivers can be related to a much longer list of requirements. The Application view can be seen as a further refinement of the Customer objective view.

Figure 25 shows the key drivers for the project along with their extension to the application view.
Figure 25. Customer objectives and Application View: CAFCR model
6.3 Use Cases

The customer objectives and application views are used as a basis to identify the use cases for the project. All the use cases are derived from the existing functionalities of the Delphi calibration algorithms and AutoStar. The calibration algorithm functionalities need to be retained or improved while the AutoStar functionalities need to be adhered to, as mentioned in the application view.

The primary actor in these use cases is the end user. The end user can be a factory engineer or service engineer or a scientist. Further, in these use cases, Calibration System (CS) is used to specify the redesigned calibration component in C++.

As mentioned in Section 4.1, every AutoStar algorithm must have the capability to pause, abort, resume, and execute. Tables 11–14 show the use cases related to these AutoStar functionalities. These use cases were identified by talking to the stakeholders in the AutoStar group and going through the AutoStar documentation.

Table 11 shows the basic use case of executing a selected algorithm. Most algorithms are a series of steps where a user action is required based on the result of the intermediate step. This is captured in the extensions part of the use case. Based on the selected user action, other use cases get invoked.

<table>
<thead>
<tr>
<th>Use Case</th>
<th>CS1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Execute a calibration algorithm</td>
</tr>
<tr>
<td>Context</td>
<td>The user wants to run a calibration algorithm</td>
</tr>
<tr>
<td>Scope</td>
<td>Calibration System(CS)</td>
</tr>
<tr>
<td>Level</td>
<td>User Goal</td>
</tr>
<tr>
<td>Primary Actor</td>
<td>User</td>
</tr>
<tr>
<td>Success Guarantee</td>
<td>The calibration algorithm is run</td>
</tr>
</tbody>
</table>

**Main Success Scenario**

1. User: Selects a calibration algorithm in the user interface for execution
2. CS: Executes the selected algorithm
3. CS: Sends results to the user
4. User: Accepts the results (CS8)

**Extensions**

Condition: CS sends intermediate results to the user

1. a. CS: Sends result of a step to the user
2. b. User: Selects a user action based on the intermediate step (CS5, CS6, CS7)
3. c. CS: Performs the next calibration step based on selected user action

Table 12 shows the use case of pausing an algorithm. This use case is derived from the AutoStar use cases and is considered a basic functionality for any algorithm in AutoStar. The extension handles the case where algorithm execution is complete or failed. In both cases, the pause request is ignored.
Table 12. Use Case: Pause a calibration algorithm

<table>
<thead>
<tr>
<th>Use Case</th>
<th>CS2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Pause a calibration algorithm</td>
</tr>
<tr>
<td>Context</td>
<td>The user wants to pause a calibration algorithm</td>
</tr>
<tr>
<td>Scope</td>
<td>Calibration System(CS)</td>
</tr>
<tr>
<td>Level</td>
<td>User Goal</td>
</tr>
<tr>
<td>Primary Actor</td>
<td>User</td>
</tr>
<tr>
<td>Success Guarantee</td>
<td>The calibration algorithm is paused</td>
</tr>
</tbody>
</table>

Main Success Scenario

1. User: Selects the option to pause an algorithm
2. CS: Pauses the running algorithm
3. CS: Notifies the user that the algorithm is paused

Extensions

Condition: Algorithm execution has completed
(2) a. CS: Ignores the request
Condition: Algorithm has already been paused or aborted or failed
(2) b. CS: Ignores the request

Table 13 and Table 14 show the use cases of resuming and aborting an algorithm execution. These two use cases are considered basic functionality in AutoStar and handled similar to the pause use case.

Table 13. Use Case: Resume a calibration algorithm

<table>
<thead>
<tr>
<th>Use Case</th>
<th>CS3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Resume a calibration algorithm</td>
</tr>
<tr>
<td>Context</td>
<td>The user wants to resume a calibration algorithm</td>
</tr>
<tr>
<td>Scope</td>
<td>Calibration System(CS)</td>
</tr>
<tr>
<td>Level</td>
<td>User Goal</td>
</tr>
<tr>
<td>Primary Actor</td>
<td>User</td>
</tr>
<tr>
<td>Success Guarantee</td>
<td>The calibration algorithm is resumed</td>
</tr>
</tbody>
</table>

Main Success Scenario

1. User: Selects the option to resume the algorithm
2. CS: Resumes the paused algorithm
3. CS: Notifies the user that the algorithm is resumed

Extensions

Condition: Algorithm execution has completed
(2) a. CS: Ignores the request
Condition: Algorithm has already been resumed or aborted or failed
(2) b. CS: Ignores the request

Table 14. Use Case: Abort a calibration algorithm

<table>
<thead>
<tr>
<th>Use Case</th>
<th>CS4</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Abort a calibration algorithm</td>
</tr>
<tr>
<td>Context</td>
<td>The user wants to abort a calibration algorithm</td>
</tr>
<tr>
<td>Scope</td>
<td>Calibration System(CS)</td>
</tr>
<tr>
<td>Level</td>
<td>User Goal</td>
</tr>
<tr>
<td>Primary Actor</td>
<td>User</td>
</tr>
<tr>
<td>Success Guarantee</td>
<td>The calibration algorithm is aborted</td>
</tr>
</tbody>
</table>

Main Success Scenario

1. User: Selects the option to abort the algorithm
2. CS: Aborts the running algorithm
3. CS: Notifies the user that the algorithm execution is aborted

Extensions

Condition: Algorithm execution has completed
(2) a. CS: Ignores the request
Condition: Algorithm has already been aborted
(2) b. CS: Ignores the request
Condition: Algorithm execution has failed
(2) b. CS: Ignores the request
The Delphi calibration software is run on the Virtual Machines (VMs) to understand the existing features. The features to retain are identified in discussions with Mr. Paul Janson, owner of the calibration algorithms. Tables 15–19 show the use cases for these features. Usability is a key driver in identifying these features as the new algorithms must work with the existing Delphi UI.

Table 15 shows the use case to repeat an algorithm step. This is a feature in the existing calibration system identified to be retained. The extension section of the use case guards against the possibility where a request to repeat is received by the CS when the algorithm isn’t expecting such a request.

### Table 15. Use Case: Repeat a calibration step

<table>
<thead>
<tr>
<th>Use Case</th>
<th>CS5</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Repeat a calibration step</td>
</tr>
<tr>
<td>Context</td>
<td>The user wants to repeat a calibration algorithm step</td>
</tr>
<tr>
<td>Scope</td>
<td>Calibration System(CS)</td>
</tr>
<tr>
<td>Level</td>
<td>User Goal</td>
</tr>
<tr>
<td>Primary Actor</td>
<td>User</td>
</tr>
<tr>
<td>Success Guarantee</td>
<td>The calibration algorithm step is repeated</td>
</tr>
</tbody>
</table>
| Main Success Scenario | 1. User: Selects the option to repeat a calibration step  
2. CS: Repeats the calibration step |
| Extensions | Condition: Algorithm not expecting a user intervention  
(2) a. CS: Reports a message and ignores the request |

Table 16 and Table 17 show use cases for the skip and continue user actions. The skip action should skip the next calibration step while continue should continue with the next step. Both use cases are similar to the pause use case described in Table 15.

### Table 16. Use Case: Skip a calibration step

<table>
<thead>
<tr>
<th>Use Case</th>
<th>CS6</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Skip a calibration step</td>
</tr>
<tr>
<td>Context</td>
<td>The user wants to skip a calibration algorithm step</td>
</tr>
<tr>
<td>Scope</td>
<td>Calibration System(CS)</td>
</tr>
<tr>
<td>Level</td>
<td>User Goal</td>
</tr>
<tr>
<td>Primary Actor</td>
<td>User</td>
</tr>
<tr>
<td>Success Guarantee</td>
<td>The calibration algorithm step is skipped</td>
</tr>
</tbody>
</table>
| Main Success Scenario | 1. User: Selects the option to skip a calibration step  
2. CS: Skips the calibration step |
| Extensions | Condition: Algorithm not expecting a user intervention  
(2) a. CS: Reports a message and ignores the request |

### Table 17. Use Case: Continue the next calibration step

<table>
<thead>
<tr>
<th>Use Case</th>
<th>CS7</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Continue the next calibration step</td>
</tr>
<tr>
<td>Context</td>
<td>The user wants to continue with the next calibration algorithm step</td>
</tr>
<tr>
<td>Scope</td>
<td>Calibration System(CS)</td>
</tr>
<tr>
<td>Level</td>
<td>User Goal</td>
</tr>
<tr>
<td>Primary Actor</td>
<td>User</td>
</tr>
<tr>
<td>Success Guarantee</td>
<td>The next calibration algorithm step is executed</td>
</tr>
</tbody>
</table>
| Main Success Scenario | 1. User: Selects the option to continue with the next calibration step  
2. CS: Continues with the next calibration step |
| Extensions | Condition: Algorithm not expecting a user intervention  
(2) a. CS: Reports a message and ignores the request |
Table 18 shows the use case of accepting calibration results. This is needed at the end of calibration execution when the user reviews the results and decides if it is should be accepted.

Table 18. Use Case: Accept calibration results

<table>
<thead>
<tr>
<th>Use Case</th>
<th>CS8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Accept calibration results</td>
</tr>
<tr>
<td>Context</td>
<td>The user wants to accept the results of the calibration algorithm</td>
</tr>
<tr>
<td>Scope</td>
<td>Calibration System(CS)</td>
</tr>
<tr>
<td>Level</td>
<td>User Goal</td>
</tr>
<tr>
<td>Primary Actor</td>
<td>User</td>
</tr>
<tr>
<td>Success Guarantee</td>
<td>The results of the calibration algorithm are accepted</td>
</tr>
</tbody>
</table>
| Main Success Scenario | 1. User: Selects the option to accept the calibration results  
2. CS: Accepts the calibration results |
| Extensions | Condition: Algorithm not expecting a user intervention  
(2) a. CS: Reports a message and ignores the request |

Table 19 shows the use case of capturing an image from the microscope and displaying it to the user. This is a very important use case since it is heavily used by majority of the 17 calibration algorithms. Further, this use case would test interactions with the lower layers to acquire an image from the microscope. The extension section of the use case deals with a condition where the algorithm must post-process the acquired image to arrive at some conclusions. This can be shown to the user in the form of markings (for example circling image spots) on the displayed image.

Table 19. Use Case: Display a microscope image

<table>
<thead>
<tr>
<th>Use Case</th>
<th>CS9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Display a microscope image</td>
</tr>
<tr>
<td>Context</td>
<td>The user wants to display an image acquired from the microscope</td>
</tr>
<tr>
<td>Scope</td>
<td>Calibration System(CS)</td>
</tr>
<tr>
<td>Level</td>
<td>Sub-Function Goal</td>
</tr>
<tr>
<td>Primary Actor</td>
<td>User</td>
</tr>
<tr>
<td>Secondary Actor</td>
<td>TEM Server</td>
</tr>
<tr>
<td>Success Guarantee</td>
<td>The results of the calibration algorithm are accepted</td>
</tr>
</tbody>
</table>
| Main Success Scenario | 1. User: Selects the option to execute a calibration algorithm  
2. CS: As part of algorithm execution, requests the TEM Server for an image  
3. TEM Server: Provides an image to CS  
4. CS: Reports the image to the user |
| Extensions | Condition: Algorithms requires post-processing on the image  
(4) a. CS: Performs post processing like identifying and marking image spots and reports the updated image to the user |

6.4 Functional View

In the functional view of the CAFCR model, the functional and the non-functional requirements of the project are determined. These requirements are derived from the key drivers, application view, and the use cases defined in the previous sections.

6.4.1 Functional Requirements

Table 20 shows the functional requirements for the project. A ‘key driver’ column in the table traces every requirement back to one of the key drivers identified in Section 6.2.
Table 20. Functional Requirements

<table>
<thead>
<tr>
<th>Requirement ID</th>
<th>Description</th>
<th>Priority</th>
<th>Key Driver</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS_FR_001</td>
<td>The CS must be integrated in AutoStar</td>
<td>High</td>
<td>Redesign in C++</td>
</tr>
<tr>
<td>CS_FR_002</td>
<td>The CS must be able to execute the selected (chosen) calibration algorithm.</td>
<td>High</td>
<td>Redesign in C++</td>
</tr>
<tr>
<td>CS_FR_003</td>
<td>The CS must be able to pause an algorithm execution</td>
<td>High</td>
<td>Redesign in C++</td>
</tr>
<tr>
<td>CS_FR_004</td>
<td>The CS must be able to resume an algorithm execution</td>
<td>High</td>
<td>Redesign in C++</td>
</tr>
<tr>
<td>CS.FR_005</td>
<td>The CS must be able to abort an algorithm execution</td>
<td>High</td>
<td>Redesign in C++</td>
</tr>
<tr>
<td>CS_FR_006</td>
<td>The CS must be able to repeat an algorithm execution</td>
<td>High</td>
<td>Redesign in C++</td>
</tr>
<tr>
<td>CS.FR_007</td>
<td>The CS must be able to skip an algorithm step</td>
<td>High</td>
<td>Redesign in C++</td>
</tr>
<tr>
<td>CS.FR_008</td>
<td>The CS must be able to continue with the next algorithm step</td>
<td>High</td>
<td>Redesign in C++</td>
</tr>
<tr>
<td>CS.FR_009</td>
<td>The CS must be able to accept the results of an algorithm</td>
<td>High</td>
<td>Redesign in C++</td>
</tr>
<tr>
<td>CS_FR_010</td>
<td>The CS must be able to acquire multiple microscope images for display</td>
<td>High</td>
<td>Redesign in C++</td>
</tr>
<tr>
<td>CS.FR_011</td>
<td>The CS must be able to display images to the user using the Delphi UI</td>
<td>High</td>
<td>Redesign in C++</td>
</tr>
<tr>
<td>CS.FR_012</td>
<td>The CS must be able to perform post-processing on the acquired images</td>
<td>High</td>
<td>Redesign in C++</td>
</tr>
<tr>
<td>CS.FR_013</td>
<td>The CS must be able to report messages to the user using the Delphi UI</td>
<td>High</td>
<td>Redesign in C++</td>
</tr>
<tr>
<td>CS.FR_014</td>
<td>The CS must perform error handling by throwing exceptions</td>
<td>High</td>
<td>Redesign in C++</td>
</tr>
<tr>
<td>CS.FR_015</td>
<td>The CS must be able to report progress to the user using the Delphi UI</td>
<td>Medium</td>
<td>Redesign in C++</td>
</tr>
<tr>
<td>CS.FR_016</td>
<td>The CS must be able to report state change to the user using the Delphi UI</td>
<td>Low</td>
<td>Redesign in C++</td>
</tr>
</tbody>
</table>

All the functional requirements derive from the either the ‘Redesign in C++’ or ‘Usability’ key driver. Further, all use cases translate to one or more functional requirements. Functional requirements CS.FR.002 to CS.FR.005 are derived from CS.FR.001. In other words, the requirements CS.FR.002 to CS.FR.005 arise because the calibration algorithms must be integrated in AutoStar.

6.4.2. Non-Functional Requirements

Table 21 shows the non-functional requirements of the project. These requirements have been derived from the key drivers and the application views.
Table 21. Non-Functional Requirements

<table>
<thead>
<tr>
<th>Requirement ID</th>
<th>Description</th>
<th>Priority</th>
<th>Key Driver</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS_NFR_001</td>
<td>The CS must have statement, branch, and decision coverage greater than 90%</td>
<td>High</td>
<td>Testability</td>
</tr>
<tr>
<td></td>
<td>with unit tests</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CS_NFR_002</td>
<td>The CS must be designed such that it is easy to add new calibration</td>
<td>High</td>
<td>Extensibility</td>
</tr>
<tr>
<td></td>
<td>algorithms</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CS_NFR_003</td>
<td>The CS must adhere to AutoStar coding standards</td>
<td>High</td>
<td>Modifiability</td>
</tr>
<tr>
<td>CS_NFR_004</td>
<td>All TICS analyzer violations above Level 8 must be resolved</td>
<td>High</td>
<td>Modifiability</td>
</tr>
<tr>
<td>CS_NFR_005</td>
<td>Each file in the CS must have a cyclomatic complexity less than or equal to</td>
<td>High</td>
<td>Modifiability</td>
</tr>
<tr>
<td></td>
<td>four</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CS_NFR_006</td>
<td>The CS must be easy to modify for future changes to automate the user</td>
<td>Medium</td>
<td>Modifiability</td>
</tr>
<tr>
<td></td>
<td>interactions</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CS_NFR_007</td>
<td>Code when compiled must have zero errors and warnings</td>
<td>High</td>
<td>Production Ready</td>
</tr>
<tr>
<td>CS_NFR_008</td>
<td>The CS must be designed such that it is easy to add tests for the new</td>
<td>Medium</td>
<td>Testability</td>
</tr>
<tr>
<td></td>
<td>calibration algorithms</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Testability**

Code coverage metrics indicate how many lines of code or branches or decisions have been covered by test runs. The lower the coverage, the lower the quality of the performed tests. Code coverage is an indicator of “Testability”. This leads to requirement CS_NFR_001.

**Modifiability**

A coding standard is a set of rules that engineers should follow. Such rules target language pitfalls, code constructions to avoid, naming conventions and program layout. Having a coding standard makes it easier to understand the intentions of the program, making it easier to modify. This is the basis for requirement CS_NFR_003.

AutoStar uses TICS Analyzer to measure software code quality and maintain a coding standard. The quality is specified by a set of rules which can be modified. These rules have been categorized into ten different levels. Higher the level, the more important the rule. To improve code quality, all violations of rules above Level 8 need to be fixed, which is requirement CS_NFR_004.

Cyclomatic complexity is one of the oldest software metrics that counts the number of independent paths through a program. The higher the cyclomatic complexity, the more difficult it is to understand a program. Hence, it is an important metric to measure modifiability of the code. Although TICS Analyzer checks for cyclomatic complexity, due to its importance, a separate non-functional requirement to keep the cyclomatic complexity less than or equal to four has been added. In general, cyclomatic complexity of five is considered ok but anything above ten warrants a refactoring. This is the basis for requirement CS_NFR_005.

**Production Ready**

To develop a working software, it must be compiled. If the compiler throws errors, they must be fixed for the program to run. Compiler warnings do not need to be solved. However, they must because some warnings can indicate major program flaws which can affect the software on deployment. This is the basis for requirement CS_NFR_007.
7. System Architecture & Design

This chapter describes the design of the calibration algorithms in AutoStar.

7.1 Introduction

One of the requirements for the project is that the redesigned calibration algorithms should be part of the AutoStar component (see requirement CS.FR_001). This means that the calibration algorithms should either use or extend the AutoStar design, as necessary. Hence, the calibration algorithms must follow the AutoStar design. Figure 26 shows the layered multi-tier AutoStar architecture.

![AutoStar Architecture Diagram]

**Figure 26. AutoStar architecture**

Each tier in the architecture can only talk to a tier below it. A tier can have several layers. Layers in a tier can interact with each other. For instance, in Figure 26, User Interface layer, AutoStar, TEM OMP, and TEM Server are tiers. C++ Tasks, C++ algorithms are examples of layers inside a tier.

The top most layer is the user interface layer through which the AutoStar algorithms are made available to the end-users. AutoStar tier interacts with the TEM OMP tier that in turn talks to the TEM server via the IOM interface. AutoStar is composed of several layers. There are three layers to implement automated algorithms, namely, C++ tasks, C++ Algorithms, and API C++ Algorithm. There are two support layers that are used by most of the other layers inside AutoStar. The first is the supporting library layer which houses libraries like boost. The other layer is the Infrastructure layer that houses functionalities for logging and unit testing.

In the following sections, a combination of views from the 4+1 view model, described in Chapter 4, are used to describe the calibration algorithm redesign. Further, the sections also describe how the existing AutoStar design is extended, the rationale behind doing so, and the design decisions made. The current AutoStar design is explained in Section 4.1 that is a pre-requisite for this chapter. Section 4.1 explains the responsibility of each class in the Task Core library that is important to understand the design choices made in Section 7.2.2.
7.2 Logical View

One of the first views to consider while designing is the logical view that describes the functionality provided by the system to the end user. As described in Section 4.1, each AutoStar algorithm must derive from the Task Core Library. In other words, each AutoStar algorithm is a task.

The calibration algorithms can be viewed as a series of steps. After each step execution, the results are provided to the user who then takes an appropriate action based on the feedback. The action a user takes can be one of the following.

1. Skip – This action requires the algorithm to skip the next step.
2. Repeat – This action requires the algorithm to repeat the previous step.
3. Next – This action requires the algorithm to execute the next step.
4. Accept – This action requires the algorithm to accept the results.

The algorithm continues based on the user action.

7.2.1 Rationale for Extension

Each calibration step must be implemented as an AutoStar task (see requirement CS_FR_001 and AutoStar’s Task Core Library). However, the Task Core Library cannot support the calibration algorithms, in its current form, due to the following reasons.

1. The current AutoStar design does not have any provisions for user interaction. It is designed for fully automated algorithms, which can only be either paused or aborted.
2. The current AutoStar design has provisions for running a sequence of tasks. However, the calibration algorithms require some processing in between two tasks based on a user action. This processing can involve
   a. determining the next task to be executed
   b. generating suitable input for the next step of the calibration algorithm

7.2.2 Design Choices

Based on the rationale mentioned in Section 7.2.1, the Task Core Library needs to be extended by introducing an InteractiveTask interface and InteractiveTask template class that implements the interface. There are three different design choices considered for the extension.
7.2.2.1 Option 1: Interactive Task derives from Task template class

In this design choice, shown in Figure 27, the InteractiveTask interface derives from ITask interface while the InteractiveTask template class derives from Task template class. The Task template class implements the ITask functionality while InteractiveTask template class implements the InteractiveTask interface. The InteractiveTaskA acts as an entry point for the calibration algorithms. This task delegates algorithm execution to one of its sub-tasks.
Pros:
1. In theory, such an inheritance allows the `InteractiveTask` template class to use the `ITask` functionality defined in the `Task` template class.

Cons:
1. Each task has its own finite state machine, depicted by `TaskStateMachine` class. As a result, deriving from the `Task` template class means overriding most of its member functions to handle two finite state machines—one for the `InteractiveTask` template class and the other for the sub-task being executed, which derives from the `Task` template class.

   For instance, the `pause` function in `Task` template class pauses the current task execution. The `pause` function in `InteractiveTask` template class must pause itself along with any sub-task which is executing. This involves keeping track of the current sub-task being executed and overriding the pause function defined in `Task` template class.

   Such an extension negates the advantages of inheritance and makes the code difficult to read. This, in turn, makes the code difficult to extend, which is a requirement for the project (see non-functional requirement CS_NFR_002).

7.2.2.2 Option 2: Interactive Task derives from Composite Task

In this design choice, the `InteractiveTask` template class derives from `CompositeTaskBase` class as shown in Figure 28.

![Figure 28. Task Core Library Extension – Option 2](image-url)
In this option, the InteractiveTask template class derives from CompositeTaskBase and implements InteractiveTask functionality.

Pros:
1. Such an inheritance provides a clear separation of responsibility – The CompositeTaskBase handles the synchronization of the state machine transitions between the parent task and its sub-tasks while the InteractiveTask handles the user actions. Most of the CompositeTaskBase functions do not need overriding which overcomes the disadvantages of Design Option 1. For instance, the pause function in the CompositeTaskBase class pauses itself along with any sub-task, which is being executed. This is precisely the behavior needed for InteractiveTask template class.

Cons:
1. The biggest disadvantage of inheriting from the CompositeTaskBase class is that it does not hold any intelligence. In other words, this class merely executes the list of sub-tasks it holds. As explained in Section 7.2.1, the calibration algorithms need some processing in between two tasks based on a user action. Hence, CompositeTaskBase is deemed unsuitable for extension for calibration algorithms.
2. The intent of CompositeTaskBase is to act as a task container. Hence, any changes made to the class would inadvertently also affect InteractiveTask template class, which might break the inheritance.

7.2.2.3 Option 3: Interactive Task derives from ITask
In this design option, as shown in Figure 29, the IInteractiveTask interface derives from ITask interface while the InteractiveTask template class derives from IInteractiveTask interface. As a result, the InteractiveTask template class implements the functionality of both the ITask and IInteractiveTask interfaces.
Figure 29. Task Core Library Extension – Option 3

Pros:
1. This option overcomes the disadvantages of Option 1 and 2. A complete implementation of ITask interface in InteractiveTask template class means that the finite state machine transitions can be handled as done in CompositeTaskBase class, while allowing for processing in between two sub-task executions.

Cons:
1. The ITask functionality must be implemented, most of which is already available in the CompositeTaskBase class.

The advantages of Option 3 outweigh the disadvantages and hence, is chosen for the final design.

7.2.3. Task Core Library Extension

Based on the chosen design option 3, the Task Core Library is extended to support calibration algorithms. This extension is shown in Figure 30. The CompositeTaskBase class is removed from the diagram since it plays no role in the extension.
The responsibilities of the classes added as part of the extension are described below.

**InteractiveTask** – This is the interface that defines the user actions taken during a calibration algorithm execution. A client uses this interface to specify the user action taken. The possible four user actions are next, repeat, skip, and accept.

**InteractiveTask** – This is a template class that implements the ITask and IInteractiveTask interfaces. This template class takes a class as a parameter. Typically, this parameter is an algorithm interface. Such a template class allows an algorithm to use the implementation of ITask and IInteractiveTask interfaces without having to redefine them again for every algorithm.

**InteractiveTaskA** – This is a dummy calibration algorithm task which implements the IInteractiveTask interface. The template binding relationship with the InteractiveTask template class provides the actual parameter for the template class. The parameter is IInteractiveTaskA. Since IInteractiveTaskA is a parameter to the InteractiveTask template class, InteractiveTaskA class has access to the implementation of ITask and IInteractiveTask interfaces.

**IWaitForUserInput** – This interface exposes the function that needs to be called to resume execution after receiving a user action.
**WaitForUserInput** – This is a task that waits for user action. Once the user action has been received, it returns control back to the caller. This task is called when a calibration algorithm must wait for a user action.

**WaitForInput** class allows the algorithm to wait for a suitable user action. The user action is based on the result of an intermediate algorithm step. Based on the intermediate result, the user can choose to tweak certain microscope configurations (for example, center the electron beam) before deciding to continue with algorithm execution. This class is an inflection point to add automated logic to replace the microscope configuration tweaks and the user action, satisfying non-functional requirement CS_NFR_006.

### 7.2.4. User Actions

This section shows how user actions are handled for the calibration algorithms. Figure 31 shows how *Accept* user action is handled.

![Figure 31. Accept User Action](image)

Upon receiving an *Accept* function call, the *InteractiveTask* template class gets the current sub-task. If the current sub-task is a *WaitForUserInput* task, then the user action is set to *accept* and the wait for user input is ended to return the execution control to the caller of the *WaitForUserInput* task. This caller is typically a calibration algorithm, like the dummy *InteractiveTask* shown in Figure 30. If the current sub-task is any other sub-task, including no sub-task, a runtime exception is thrown to inform the user that the algorithm is not waiting for any user action.

The same flow holds true for the remaining three user actions, namely, *skip*, *repeat*, and *next*. The corresponding user action is set if the sub-task is a *WaitForUserInput* task. For example, if *Skip* function is called, the user action is set to *Skip*.

### 7.2.5. State Transitions

This section shows how state transitions are handled while waiting for a user input. In other cases, the state transitions are handled as done by the original Task Core Library.

#### 7.2.5.1 Pause Request

Figure 32 shows how a pause request is handled when a calibration algorithm is waiting for a user input.
Upon receiving a Pause function call, typically from a higher layer, the InteractiveTask template class sends a HandlePauseRequest call to its state machine. This call changes the InteractiveTask state from Running to Pausing. This state change is then notified to all subscribers. Next, the InteractiveTask template class gets the current sub-task being executed. If this sub-task is a WaitForUserInput task, then the pause request is ignored since the task is already waiting for a user action.

A resume request is like a pause request. The WaitForUserInput task also ignores a resume request since its waiting on a user input and there is no code to resume executing.

### 7.2.5.2 Abort Request

Figure 33 shows how an abort request is handled while a calibration algorithm is waiting for a user input.

Upon receiving an Abort function call from a UI, the InteractiveTask template class sends a HandleAbortRequest call to its state machine. This call changes the InteractiveTask state from Running to Aborting. This state change is then notified to all subscribers. Next, the InteractiveTask template class gets the current sub-task being executed. If this sub-task is a WaitForUserInput task, then the task state is changed from Running to Aborting by sending an Abort request to the parent class of WaitForUserInput, which is Task template class. Further, EndWaitForUserInput function is called to stop waiting for a user input so that the task execution can be aborted.
7.3 Development View

The development view focuses on the module organization of the software. A package diagram, as shown in Figure 34, shows the organization of classes into packages. The diagram shows a visualization of the namespaces. Apart from the new packages added, the diagram shows only those AutoStar packages that are of interest for designing and implementing the calibration algorithms.

![Package Organization Diagram](image)

**Figure 34. Calibrations in AutoStar – Package Organization**

The root package of the AutoStar component is named *AutoStar*. The *AutoStar* package contains three sub-packages, namely, *Infra*, *Algorithms*, and *Tasks*.

The *Algorithms* package holds structures that implement the mathematical computations for an algorithm. The computations related to calibration algorithms are packaged under *Calibrations* package which is nested under *Algorithm* package.

The *Infra* package holds elements which can be used by most other packages. Two such nested packages are *Logging* and *Imaging*. The *Logging* package provides logging capabilities whereas the *Imaging* package provides image analysis capabilities.

The *Tasks* package holds all the classes and interfaces of the Task Core Library. The *Calibrations* package is fully contained inside the *Tasks* package and holds all classes pertaining to calibration algorithm implementation. The other package fully contained inside the *Tasks* package is the *Basic* package which provides capabilities to acquire an image by making calls to the TEM OMP. Further, the *Calibrations* package imports the elements of *Basic* package that allows the use of unqualified names from the *Basic* package. The *Calibrations* package also requires *Infra* and *Algorithms:Calibrations* packages for its full operation as shown by the “use” dependency.

The current chapter explains the classes under the *Tasks* package in Section 8.2.3. The classes under other packages are introduced in the next chapter (Chapter 8) since these are classes and, in some cases, packages that have been added while implementing the first calibration algorithm. The distribution of classes and hence responsibility across these packages is one way of introducing separation of concern across the system (Single Responsibility principle). The Single Responsibility Principle states that each class...
must have only one responsibility. Such a separation of concern improves the testability and modifiability of the system. Such separation of concern enables the design to evolve to include \textit{WaitForInput} class and hence, to satisfy the non-functional requirement \textit{CS\_NFR\_006} (see Section 7.2.3).
8. Implementation

This chapter describes one of the calibration algorithms that is chosen for implementation. Further, this chapter explains how the chosen algorithm is implemented by extending the Logical View defined in the previous chapter.

8.1 Introduction

The first algorithm chosen to be redesigned using the extended AutoStar Task Core library is the CountToElectron algorithm. This algorithm is one of the least complex but needs image acquisition from the microscope, which is a key basic requirement for most of the other calibration algorithms. This makes the CountToElectron algorithm a good starting point to redesign and implement, to prove the correctness of the extended AutoStar Task Core library.

8.2 CountToElectron Algorithm

The CountToElectron algorithm provides a measure of the dose on a specimen (see below). In general terms, a dose is a measure of the number of electrons a pixel has been exposed to. To determine the dose, the algorithm computes a conversion factor based on the following formula.

\[
Conversion\text{factor} = \frac{1}{6,022 \times 10^{-19} \times \text{sum}\text{counts}} \times \frac{\text{exposure time}}{\text{screen current}}
\]

Where,
- \(6,022 \times 10^{-19}\) C is the charge of one electron
- \(\text{exposure time}\) is the window of observation for which a specimen is exposed to electrons
- \(\text{screen current}\) is the current measured on the fluorescent screen
- \(\text{sum}\text{counts}\) is the sum of the value of each pixel that is proportional to the number of photoelectrons received

This conversion factor is stored as the result of the algorithm along with the screen current, voltage, and binning for which it is measured. This conversion factor is used by applications to determine the maximum intensity of the electron beam allowed for a specimen. This is extremely useful for life science applications where it is important to know how much damage can be caused to a specimen by viewing it. The activity diagram, shown in Figure 35, depicts how the current Delphi algorithm computes the conversion factor.
The algorithm first checks to see if the pre-conditions required for algorithm execution are met. For example, one pre-condition is that the electron beam should fit within the camera read out area. If all conditions are met, the current microscope state is stored for later restoration.

Next, the algorithm checks to see whether the intensity of the electron beam is within the required threshold for a valid conversion factor determination. This check is performed as part of ‘Compute Threshold’ activity. To perform this check, the algorithm first determines the lower and upper thresholds. The lower threshold is determined by capturing a noisy image and calculating the mean of all pixels in a rectangular read out area around the center of the image. The upper threshold is achieved based on the camera saturation point. Once the thresholds are determined, a bright image with binning value of one is captured. Binning is a process of combining information from
Adjacent pixels in a Charge Coupled Device (CCD) during readout. A binning value of one means that each pixel information is read individually with no combining of pixel information. Again, the mean of all pixels in the rectangular read out area is computed. This mean value should be within the lower and upper thresholds. If not, the user is asked to adjust the beam parameter (intensity, position etc.) and try again. Once this condition is satisfied, the beam should not be touched.

Next, the 'Compute Conversion Factor' activity is performed. First, using the same bright image, the conversion factor for binning value of one is computed based on the conversion factor equation. Next, the conversion factor is computed with the remaining binning values supported by the selected camera. The steps to do so are the same wherein a bright image is captured with the correct binning. Next, the image mean is computed that is then used to compute the conversion factor.

After the computation is done for all binning values, the user is asked to review and accept the results. Upon acceptance, the results are stored; else the calibration ends without storing the results.

### 8.3 Algorithm Implementation

Figure 36 shows how the CountToElectron algorithm is implemented using the extended AutoStar Task Core Library.

![Figure 36. CountToElectron Algorithm in AutoStar](image-url)
Although the entire Task Core Library, as shown in Figure 30, is used to execute the CountToElectron algorithm, only the classes which are directly inherited from, are shown in Figure 36 to focus on the new classes added.

Two tasks have been defined for the algorithm execution – ComputeThresholdTask and ComputeConversionFactorTask. Each of these two classes derive from the Task template class and are used by CountToElectronInteractiveTask class to execute the algorithm.

The main responsibilities of the classes are described below.

**CountToElectronInteractiveTask** – This class derives from the InteractiveTask template class and acts as the entry point to the calibration algorithm. It populates the task sequence and executes the algorithm steps as defined in Figure 35. This class also provides the necessary inputs for the ComputeThresholdTask and ComputeConversionFactorTask classes. Further, CountToElectronInteractiveTask class processes the output of the two classes and stores the calibration results.

**ComputeThresholdTask** – This class is responsible for determining the lower and upper threshold values as well as determining the correct beam intensity and size for a bright image with binning value of one. This class implements all the actions mentioned as part of Compute Threshold activity in Figure 35.

**ComputeConversionFactorTask** – This class is responsible for computing the conversion factor for all the remaining binning values (other than one), which the camera supports. This class implements all the actions mentioned as part of Compute Conversion activity in Figure 35.

**ElectronConversionEfficiency** – This class is part of the Algorithms package and implements the mathematical equation to compute the conversion factor.

**ImageAnalysis** – This class is part of the Imaging package and implements the functions to compute the mean value for a read-out area of an image. This class is used by the ComputeThresholdTask class to determine the threshold values.

### 8.3.1. CountToElectron Flow

Figure 37 shows how the algorithm is implemented by the CountToElectronInteractiveTask. The purpose of this diagram is to show how the CountToElectronInteractiveTask interacts with the ComputeThresholdTask and ComputeConversionFactorTask to implement the algorithm. The details of the ComputeThresholdTask and ComputeThresholdTask are covered in Sections 8.3.2 and 8.3.3 respectively. The algorithm follows a sequence as described by Figure 35.

As shown in Figure 37, upon receiving a request to start calibration, the CountToElectronInteractiveTask initializes the sequence of tasks to be called. Three tasks are initialized – ComputeThresholdTask, WaitForUserInput, and ComputeConversionFactorTask. Next, the pre-conditions are checked to see if the algorithm can be run. If not, an error message is reported back to the client and the task is failed. If the pre-conditions are met, the current microscope state is saved. Next, the first task to compute the threshold is executed. After the task is complete, the algorithm waits for a user input by calling the WaitForUserInput. The user can choose to repeat the task or to continue algorithm execution by choosing ‘Next’ as the user action. If the ‘Next’ user action is chosen, the ComputeConversionFactorTask is executed. After this task is completed, the results are reported back to the user and the algorithm again waits for user feedback. The user can choose to repeat the task, to cancel the task or to accept the results. If the results are accepted, they are stored and the microscope state restored.
8.3.2. Threshold Computation

Figure 38 shows how the ComputeThresholdTask determines whether the beam intensity is within the threshold limits or not.

The ComputeThresholdTask is called by the CountToElectronInteractiveTask. To compute the threshold, the ComputeThresholdTask gets the camera acquisition settings. Based on these settings, it acquires a noisy image. A noisy image is an image in which the electron beam does not reach the CCD. Next, the mean of a readout area of such an image is calculated. This mean is used to compute the lower threshold value. The upper threshold is computed based on the camera saturation point. Next, an image mean is captured in the same way for a bright image. If this image mean is within the threshold range, then the conversion factor for this bright image is captured. If not, the user is asked to adjust the beam settings so that the bright image mean can fall within the threshold and repeat the task (see Section 8.3.1 to know how ComputeThresholdTask can be repeated).

Ideally, the conversion factor computation should be done by the ComputeConversionFactorTask. However, to avoid capturing the bright image with binning of one again, the factor is computed as a part of this task. This is to improve the algorithm performance.
The ComputeThresholdTask interacts with lower layer components to achieve some of its tasks, most notably, to acquire the noisy and bright images. However, to focus on how the algorithm steps have been mapped to the tasks, the lower layer calls have been omitted.

Figure 38. Threshold Computation Sequence

8.3.3. Conversion Factor Computation

Figure 39 shows how the conversion factor can be computed. This task has not yet been implemented.

Figure 39. Conversion Factor Computation Sequence

The ComputeConversionFactorTask is called by the CountToElectronInteractiveTask. The binning values are passed as an input to the task. For every binning, an image is acquired based on the camera settings. Next, the image mean is computed for the complete image. Based on the mean, the conversion factor is computed by calling ComputeEfficiency function of ElectronConversionEfficiency class. After conversion factors for all the binning values have been computed, the results are stored.
9. Verification & Validation

This chapter presents the results of the verification performed on the implementation of calibration algorithms in AutoStar.

9.1 Introduction

According to [10], Verification and Validation (V&V) is a collection of analysis methods that can be used to find deficiencies and potential risks in proposed solutions. Verification methods are used to determine if a product is being developed properly. In other words, verification checks if the software meets its specifications. Verification methods include static testing where no code is executed. For example, walkthrough, inspection, and reviews. Validation methods are used to determine if the correct product is being developed. In other words, validation checks if the software meets the customer needs and requirements. Validation methods include dynamic testing where code is executed. For example, black box and white box testing.

9.2 Test-Driven Development (TDD)

One of the key drivers for the project is testability. This driver leads to a requirement of having statement, branch, and decision coverage higher than 90% (CS_NFR_001). To satisfy this key driver and requirement, a Test-Driven Development (TDD) approach is used for algorithm implementation. TDD is a software development process which focuses on writing unit tests before writing production code. Unit testing is a type of white-box testing where individual units of a software are tested. A unit can be defined as the smallest testable part of a software. TDD can be described in terms of three rules [11].

1. Write just enough unit test to fail. Even compilation errors are considered as failures.
2. Write production code only to pass a failing unit test.
3. Write no more production code than necessary to make the failing unit test pass.

Although TDD uses unit tests for development, it is not the same as basic unit testing. There is a subtle difference between the two. While TDD uses the three rules to allow the tests to drive the development, the unit tests only verify the behavior of a unit. Further, TDD serves as a way of incrementally designing a system. Unit testing is mostly about writing tests after the code has been written. Such tests are useful to find bugs but do not influence the system design. The tests in TDD serve the following purposes:

1. They seed the design.
2. They record the intentions of the designers.
3. They act as an invariant on the code base.

The above mentioned three rules were used to develop the calibration algorithms by repeating the cycle shown in Figure 40.
The first step is to add a new test. This test targets the smallest behavior which can be tested and has not been coded yet. By writing the test before the code, the focus is on the requirements. Next, all existing tests are run to see that the new test fails, which increases the confidence. If the test passes, it is time for review to see whether the behavior was already inadvertently coded or the previous test was not atomic enough. If the test fails, just enough code is written to make the test pass. Next, all tests are run to ensure that they all succeed and that the added code has not broken existing behavior. Next, a check is made to see if any refactoring is needed as a result of adding the new code. This is done by, for example, checking if good design principles are being used or if there is any code duplication or if the correct coding guidelines have been followed. If not, the code is modified and the tests run again to check that the code behavior has not changed. During this code modification, no new behavior is introduced. If any of the tests fail, then those failing tests are fixed. This cycle is repeated to implement the calibration algorithms.

For the calibration algorithms, TDD tests have been written at inflection points. An inflection point is a narrow interface to a set of classes. If any changes are made to the classes behind an inflection point, the change is either detectable at the inflection point, or inconsequential for testing [2]. This has been achieved by writing tests for methods exposed by `IInteractiveTask`, `IWaitForUserInput`, and `IComputeThresholdTask` interfaces.

Several studies have shown that TDD provides the following benefits to developers – reduced maintenance, reduced defects, improved code quality, and improved developer confidence. Although the productivity goes down in most cases, the reduced maintenance compensates for the increase in development time. The subsequent code quality improvement further compensates for the development time. Hence, TDD is a derived requirement of the more basic requirement to make the software more maintainable [12].
9.3 Google Test and Google Mock

One of the requirements for the project is that the calibration algorithms must be integrated in AutoStar (requirement CS.FR.001). It is easier to use the tools used by AutoStar for testing the redesigned calibration algorithms. Currently, AutoStar uses Google Test and Google Mock for creating unit tests. Hence, the same tools were used for developing calibration algorithms using TDD approach.

As the name suggests, Google Test is a C++ testing framework developed by Google [13]. It is based on the xUnit architecture. Google tests use assertions to verify code behavior. An assertion is a statement that checks whether a condition is true. If a test crashes or an assertion fails, then the test fails; else it succeeds. In production code, often an object must collaborate with other objects to get their work done. This can pose certain challenges for testing:

- A dependency on slow collaborator objects can result in slow tests.
- A dependency on a volatile collaborator can result in intermittently failing tests.
- The collaborator may not exist or might be under development.

This problem can be solved by using a test double, which emulates the collaborator object. A mock is a test double that self-verifies based on pre-defined expectations [14]. Google Mock is a library for creating mock classes and using them. It is a part of the Google test framework.

9.4 Test Coverage

The following code coverage metrics have been used for the calibration algorithms to demonstrate that there is no unintended functionality and to evaluate the effectiveness of TDD [15].

1. Statement coverage – Statement coverage is a testing technique which aims to ensure that every statement of code is executed at least once during software testing.
2. Branch (Decision) coverage – Branch coverage is a testing technique which aims to ensure that each one of the branches from a decision point is executed at least once during software testing.
3. Condition / decision coverage – Condition / decision coverage is a testing technique which aims to ensure the following
   a. Each one of the branches from a decision point is executed at least once.
   b. Each condition in a decision must take all possible outcomes at least once.

A TDD approach to implementation means that these coverage metrics should be close to 100% since no production code is written before adding a test. These metrics are used as a measure of how successfully TDD has been implemented (which is one of the requirements) and not as a measure of correctness or completeness of code. Table 22 shows the coverage metrics for the files that implement the calibration algorithms.

Table 22. Calibration code coverage metrics

<table>
<thead>
<tr>
<th>File Name</th>
<th>Statement coverage</th>
<th>Condition / decision coverage</th>
</tr>
</thead>
<tbody>
<tr>
<td>ImageAnalysis.cpp</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>ElectronConversionEfficiencyp.cpp</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>InteractiveTaskCore.cpp</td>
<td>100%</td>
<td>86%</td>
</tr>
<tr>
<td>WaitForUserInputTask.cpp</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>ComputeThresholdTask.cpp</td>
<td>100%</td>
<td>100%</td>
</tr>
<tr>
<td>Factory.cpp</td>
<td>100%</td>
<td>N.A.</td>
</tr>
<tr>
<td>EmptyTask.cpp</td>
<td>100%</td>
<td>N.A.</td>
</tr>
</tbody>
</table>
The table shows only statement and condition / decision coverage. This is because condition / decision coverage includes branch coverage. In other words, 100% condition / decision coverage means 100% branch coverage. The numbers indicate that for most of the files, TDD has been well implemented. The InteractiveTaskCore file has 86% condition / decision coverage because it uses a flag from the existing AutoStar framework which is not public / protected to toggle its value for testing.

The above numbers show that the non-functional requirement of having statement, branch, and decision coverage above 90%, has been met (CS_NFR_001).

9.5 Cyclomatic Complexity

Cyclomatic complexity is one of the oldest software metrics that counts the number of independent paths through a program. The higher the cyclomatic complexity, the more difficult it is to understand a program. Hence, it is an important metric to measure modifiability of the code. Table 23 shows the cyclomatic complexity for the files that implement the calibration algorithms.

<table>
<thead>
<tr>
<th>File Name</th>
<th>Cyclomatic Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>ImageAnalysis.cpp</td>
<td>4.000</td>
</tr>
<tr>
<td>ElectronConversionEfficiency.cpp</td>
<td>2.000</td>
</tr>
<tr>
<td>InteractiveTaskCore.cpp</td>
<td>1.517</td>
</tr>
<tr>
<td>WaitForUserInputTask.cpp</td>
<td>1.166</td>
</tr>
<tr>
<td>ComputeThresholdTask.cpp</td>
<td>1.272</td>
</tr>
<tr>
<td>Factory.cpp</td>
<td>1.000</td>
</tr>
<tr>
<td>EmptyTask.cpp</td>
<td>1.000</td>
</tr>
</tbody>
</table>

The above numbers show that the non-functional requirement of having cyclomatic complexity less than or equal to four, has been met. (CS_NFR_005). □
10. Conclusions

This chapter presents the results of the project along with the conclusions. Further, this chapter describes the lessons learned and future work. This chapter also traces the results back to the requirements to show the requirements that were met.

10.1 Results

The three major goals of the project were to redesign the calibration algorithms in C++, to make them maintainable, and to redesign the algorithms with future scope for automation. These goals were translated into key drivers and requirements, as described in ‘System Requirements’ chapter. Next, a prototype was developed to prove the feasibility of the project. The prototype involved creating a C++ calibration component to acquire multiple images from TEM OMP, to process these images, to add overlays, and to display these images in the Delphi UI. In doing so, the prototype showed that certain requirements can be met (CS_FR_010, CS_FR_011, CS_FR_012, and CS_FR_013).

To meet the requirements, the AutoStar Task Core Library was analyzed. The analysis showed that the library had to be extended to support the calibration algorithms. This extension was realized by creating the InteractiveTask interface and InteractiveTask template class. The Delphi UI – AutoStar connection was deemed out of scope due to a risk found (see risk CS_R_011) while analyzing the AutoStar library. Hence, to prove that the AutoStar extension can support the calibration algorithms, module tests were created. These tests call the AutoStar Task Core Library extension as clients of AutoStar would, to show that the extension can support the calibration algorithm requirements. Further, these tests proved that the algorithms can be executed, paused, resumed, and aborted (satisfying requirements CS_FR_002, CS_FR_003, CS_FR_004, and CS_FR_005). Further, these tests also showed that the extension allows to repeat, skip, continue an algorithm step as well as accept calibration results (satisfying requirements CS_FR_006, CS_FR_007, CS_FR_008, and CS_FR_009). Further, 50% of the Count2Electron algorithm, the first calibration algorithm chosen for redesign, was implemented in AutoStar. The entire algorithm could not be implemented due to delays described in detail in Section 11.4 of the Project Management chapter.

The AutoStar Task Core Library extension and the calibration algorithms were implemented while conforming to the non-functional requirements. A TDD approach to implementation was used keeping testability in mind. Test coverage tools were used to measure coverage metrics of above 90%. Dependency Inversion principle was used to improve the testability of the code. Dependency Inversion isolates abstraction and details making the code easier to modify. This isolation is achieved by adding interfaces such that the high-level modules do not depend on low level implementation details. Adding interfaces allows modules to be decoupled and allows classes to be mocked and stubbed, which improves the ability to create unit tests. Further, Dependency Injection was used in the constructor of classes to inject instances of classes on which a class depends. This simplifies passing mocks and stubs to classes, thereby, improving the ability to write unit tests, partly satisfying non-functional requirement CS_NFR_008. However, it was not possible to design the test infrastructure. The test coverage metrics shown in Section 9.4 confirms that the redesigned calibration algorithms are testable, satisfying non-functional requirement CS_NFR_001.

The dependency inversion principle also reduces coupling in a system. This principle is used in the AutoStar extension to separate the abstractions like repeating, skipping or continuing an algorithm step from the algorithm implementation details. This makes it easier to add new calibration algorithms by adding a new algorithm interface and deriving from the InteractiveTask template class to realize a calibration algorithm, as shown in Figure 36. Hence, the design satisfies non-functional requirement CS_NFR_002.
The Single Responsibility principle was used to divide the system functionality into different classes and packages. This principle states that every class must have only one responsibility. This was achieved by separating the object creation, algorithm sequence, algorithm steps, and mathematical computations into different classes and in some cases, even packages. The algorithm steps were further separated into common steps (like repeat, skip, continue, and accept) and non-common steps (specific to each algorithm) by using the template class design pattern for the InteractiveTask class. This helped to improve the testability and modifiability, which are two key drivers for the project. Such separation of concern enabled the design to evolve to include WaitForInput class. WaitForInput class allows the algorithm to wait for a suitable user action. The action is based on the result of an intermediate algorithm step. Based on the intermediate result, the user can choose to tweak certain microscope configurations (for example, center the electron beam) before deciding to continue with algorithm execution. This class is an inflection point to add automated logic to replace the microscope configuration tweaks and the user action, satisfying non-functional requirement CS_NFR_006.

The redesigned algorithms were code reviewed weekly by Mr. Andrea Pasqualini, system architect of the AutoStar component. This ensured that the AutoStar coding standards were followed, satisfying requirement CS_NFR_003. Further, TICS Analyzer reports were generated to ensure that all violations above Level 8 were resolved, satisfying requirement CS_NFR_004. All files added have a cyclomatic complexity less than or equal to four, as shown in Table 23, thus, satisfying non-functional requirement CS_NFR_005.

To conclude, the results show that the AutoStar framework can be extended to realize the calibration algorithms in C++. Due to certain risk realizations, only two out of four Count2Electron algorithm increments could be developed. These increments are described in the detailed planning section of the Project Management chapter. However, these two increments along with the unit and module tests written for them were sufficient to prove the correctness of the design. The redesigned calibration algorithms are maintainable, testable, and modifiable for further automation. The Delphi UI cannot directly call the AutoStar component due to 32 bit – 64-bit incompatibility between the two. The AutoStarServer component must be modified to communicate between the two components so that users are not affected by the redesign.

10.2 Lessons Learned

The following lessons were learnt during the project.

- TDD approach worked well to improve testability of the calibration algorithms. The TDD approach provided early feedback on problems caused by class dependencies, which helped to improve the testability from the beginning of the project. When the underlying Task Core Library underwent modification, TDD allowed a faster redesign of the AutoStar Task Core Library extension. The existing unit tests provided confidence in the rework.
- TDD reduces productivity initially. This is because the development time goes up in creating the necessary mocks and stubs to write a failing test. This increased development time must be factored in planning if the required stubs and mocks aren’t available upfront. Further, TDD implementation needs a different mindset as compared to traditional development processes. If a developer has not used the TDD approach before, it takes time to adapt to the TDD approach and must be factored in planning.
- It is important to analyze the TEM OMP functionality needed to implement the calibration algorithms. Such an analysis will help to better estimate the time required for each calibration algorithm redesign.

10.3 Future Work

- The remaining two increments of the Count2Electron calibration algorithm must be implemented using the TDD approach.
The AutoStarServer component, which is a COM server, must be modified to allow Delphi UI to call the redesigned calibration algorithms in AutoStar via the COM server. The AutoStarServer is needed because Delphi UI is a 32-bit application while AutoStar is a 64-bit component.

The remaining calibration algorithms need to be implemented using the AutoStar framework extension that has been developed as part of this project. This effort will move all calibration algorithms from Delphi legacy code to testable and maintainable C++ code in AutoStar.

The testing framework can be improved for reuse when tests are added for new calibration algorithms. The test framework to mock lower layers and supply test data should be reusable across all calibration algorithms. This will make it easier and faster to create new tests.

The calibration algorithms can be automated by removing the user interactions that exist in the current calibration algorithms. This can be achieved by replacing the user interaction with an algorithm that performs the same step that the user performs manually during an algorithm interruption. The redesigned calibration algorithms make this substitution easier – replace the WaitForUserInput call with the automated logic.

Understanding the Count2Electron calibration algorithm took a significant amount of time due to lack of sufficient documentation. The algorithm documentation must be improved while redesigning them in AutoStar.
11. Project Management

This chapter describes how the project was planned and progress tracked. Further, a risk analysis was also carried out to identify risks and make mitigation plans.

11.1 Introduction
An incremental and iterative approach was used to execute the project as shown in Figure 41.

![Figure 41. Incremental and Iterative Approach](image)

Incremental development divides the project into smaller pieces or increments to be developed over time and integrated as they are completed. Iterative development sets aside some time to improve what has been developed [16]. Combining the two approaches, the entire project was divided into multiple increments, each of which was executed in iterations. Each increment was composed of gathering requirements, designing, implementing the design, testing, and evaluating the results. The evaluation or review phase lead to rework for some increments. At the end of each increment, the results were either integrated into the existing framework or released to the users depending on the increment type. For instance, at the end of a prototype increment, the prototype was released for use. On the other hand, at the end of an algorithm increment, the increment was integrated into the existing system and not released.

11.2 Project Planning and Scheduling
The first activity carried out for the project was to make a project plan. The project was executed in ten months, from January 2018 to October 2018. The incremental and iterative approach mentioned in Section 11.1 was used as a basis for defining the project activities. Figure 42 shows the planning overview containing the chronological sequence and duration of high-level activities carried out during the project.

### 11.2.1 Summary

<table>
<thead>
<tr>
<th>Activities/week</th>
<th>January</th>
<th>February</th>
<th>March</th>
<th>April</th>
<th>May</th>
<th>June</th>
<th>July</th>
<th>August</th>
<th>September</th>
<th>October</th>
</tr>
</thead>
<tbody>
<tr>
<td>Stakeholder Analysis</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
</tr>
<tr>
<td>Risk Analysis</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
</tr>
<tr>
<td>Domain Analysis</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
</tr>
<tr>
<td>Prototype (Feasibility Analysis)</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
</tr>
<tr>
<td>Count2Electron algorithm Development</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
</tr>
<tr>
<td>Technical Report Writing</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
</tr>
<tr>
<td>Project Defense Presentation</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
</tr>
<tr>
<td>Project Wrap-up</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
<td>1 2 3 4</td>
</tr>
</tbody>
</table>
The first activity for the project was to make a project plan. There was no earlier precedence of such refactoring available within the calibrations group. As a result, it was difficult to predict time estimates for the redesign at the beginning of the project. To deal with this, the project plan was revisited frequently. For example, at the end of the prototype activity, the plan was updated to estimate what could be accomplished in the subsequent months based on prototype learnings.

After the project plan, a stakeholder analysis and risk analysis were carried out. Both documents were revisited mid-way through the project to identify new stakeholders and risks. Next, a domain analysis was carried out to understand the AutoStar framework, existing calibration algorithms, and to identify and familiarize with the tools or software that would be used for the redesign. Next, a feasibility analysis was carried out by developing a prototype. The prototype development lasted for around three months. Finally, feedback from the prototype development was used to estimate four months of calibration algorithm development time.

The high-level activities were planned using an excel sheet. The calibration group, of which I was a part, used scrum for project execution. A Scrum is an incremental and iterative framework for product development. Scrum divides activities into smaller well-defined tasks. These tasks are called user stories. In accordance with the Scrum process, the activities of the project plan were broken down into user stories and tracked in sprints. A tool called Jira was used to track the sprint stories.

**11.2.2. Details**

Figure 43 shows the detailed project plan with the increments defined for the prototype and calibration algorithm development activities.

Following the incremental and iterative approach, the prototype was divided into five increments. The requirements were gathered upfront. Each prototype increment involved design, implementation, testing, and rework based on reviews. In other words, a single round of iteration (rework) was a part of each increment.

The algorithm development was divided into four increments. These increments were identified based on the domain analysis performed on the Count2Electron algorithm (see Section 8.2). These four increments are:

- Increment 1: Extend AutoStar Task Core Library to support the calibration algorithm requirements
- Increment 2: Develop the ComputeThreshold task (see section 8.3)
- Increment 3: Develop the ComputeConversionFactor task (see section 8.3)
- Increment 4: Implement the Count2Electron Interactive task (see section 8.3)

Each algorithm increment involved requirement gathering, design, implementation, testing and rework. However, only two of the four algorithm increments could be realized (Increment 1 and 2). The reasons are explained as part of the Milestone Trend Analysis.

The Delphi UI connection via the wrapper layer was deemed out of scope due to a risk identified midway through the project (see risk CS_R_011). The risk identified was that AutoStar is a 64-bit component while the Delphi UI is 32-bit. This means that the two cannot interact directly. Rather, the AutoStarServer component, which is a COM server, must be modified so that the two components can interact. This was deemed out of scope for the project after discussions with Mr. Paul Janson, owner of the calibration algorithms, due to the additional time required to modify the AutoStarServer.
11.3 Risk Analysis

Table 24 shows the risks that were identified for the project along with their impact and probability of occurrence during the project.
Table 24. Risk Exposure

<table>
<thead>
<tr>
<th>Risk ID</th>
<th>Risk</th>
<th>Risk Impact</th>
<th>Probability of Occurrence</th>
<th>Risk Exposure</th>
</tr>
</thead>
<tbody>
<tr>
<td>CS_R_001</td>
<td>Insufficient microscope time to test new calibration algorithms for all configurations.</td>
<td>Medium</td>
<td>High</td>
<td>High</td>
</tr>
<tr>
<td>CS_R_002</td>
<td>Incorrect estimates in project plan</td>
<td>Medium</td>
<td>High</td>
<td>High</td>
</tr>
<tr>
<td>CS_R_003</td>
<td>Impacted individuals are not kept informed</td>
<td>High</td>
<td>Medium</td>
<td>High</td>
</tr>
<tr>
<td>CS_R_004</td>
<td>Integration/Smoke testing environment is not available/usable</td>
<td>High</td>
<td>High</td>
<td>High</td>
</tr>
<tr>
<td>CS_R_005</td>
<td>Lower layer components do not provide required functionality for redesigning</td>
<td>Medium</td>
<td>High</td>
<td>High</td>
</tr>
<tr>
<td>CS_R_006</td>
<td>Poor quality design</td>
<td>High</td>
<td>Medium</td>
<td>High</td>
</tr>
<tr>
<td>CS_R_007</td>
<td>Stakeholders not available during critical times of the project</td>
<td>Medium</td>
<td>Medium</td>
<td>Medium</td>
</tr>
<tr>
<td>CS_R_008</td>
<td>Redesigned calibration algorithms cannot be integrated with Auto-STAR</td>
<td>High</td>
<td>Low</td>
<td>Medium</td>
</tr>
<tr>
<td>CS_R_009</td>
<td>Conflicting stakeholder requirements</td>
<td>Medium</td>
<td>Medium</td>
<td>Medium</td>
</tr>
<tr>
<td>CS_R_010</td>
<td>University and/or company requirements are not met</td>
<td>High</td>
<td>Low</td>
<td>Medium</td>
</tr>
<tr>
<td>CS_R_011</td>
<td>Incompatible Delphi UI and AutoStar</td>
<td>Medium</td>
<td>High</td>
<td>High</td>
</tr>
</tbody>
</table>

The risk exposure is a product of the risk impact and the probability of occurrence. A qualitative measure with a three-point scale (High, Medium, and Low) was used to grade the risks. This is because there were not too many risks and it was easy to keep track of them.

For each risk identified in Table 24, Table 25 shows the response strategy, should the risk be realized.

Risk CS_R_002 and CS_R_005 were realized while working on the Count2Electron algorithm increments. These risks, which delayed the first and second increments of the calibration algorithm, are discussed in detail as part of Milestone Trend Analysis.

Risk CS_R_011 was identified after prototype development while analyzing the Auto-Star Task Core Library. The risk was that AutoStar is a 64-bit component while the Delphi UI is 32-bit. This means that the two cannot interact directly. The only strategy was to accept the risk and reduce the scope. As a result, Delphi UI – AutoStar connection was deemed out of scope for the project.
<table>
<thead>
<tr>
<th>ID</th>
<th>Project Impact</th>
<th>Response Strategy</th>
<th>Strategy Steps</th>
</tr>
</thead>
</table>
| CS_R_001 | • Poor feature quality  
• Customer dissatisfaction          | Mitigation        | • Strong unit test suite  
• Use Virtual Machines (VMs) to do as much testing as possible with integration testing  
• Reserve microscope slots early to get testing time on the machines  
• Make a list of all configurations under which a calibration algorithm can be used and do some smart testing (maximum configuration coverage with least testing).  
• Review calibration results with experts |
| CS_R_002 | • Affects project scope  
• Reduces number of calibration algorithms redesigned | Mitigation        | • Monthly review of the project plan with stakeholders to re-adjust, if required  
• Use an AGILE approach for project plan and execution |
| CS_R_003 | • Negative feedback  
• Potential reduced cooperation | Avoidance         | • Have the stakeholder analysis reviewed with experts to avoid missing a stakeholder in your communication plan |
| CS_R_004 | • Affects project scope | Continuity        | • Spend effort on investigation and extending the test environment |
| CS_R_005 | • Reduces number of redesigned calibration algorithms  
• More effort on investigation, reporting and possible implementation in lower layers.  
• Long throughput to solve these problems | Acceptance         | • Reduce scope and review project plan (Acceptance, no mitigation possible) |
| CS_R_006 | • Poor feature quality  
• Requirements not met          | Mitigation        | • Engage experts early for feedback and continuously iterate |
| CS_R_007 | • Project Delays                  | Mitigation        | • Plan meetings early, ask for alternate times if there are conflicts  
• Ask for vacation plans of the stakeholders and plan accordingly  
• Ask for back-up person to reply to questions |
| CS_R_008 | • Calibration Algorithm not aligned with AutoSTAR architecture.  
• Possible project failure | Mitigation        | • Read up on Auto-STAR architecture, processes, coding guidelines.  
• Integrate early (possibly the prototype) and seek feedback from Auto-STAR team  
• Involve them in code reviews |
| CS_R_009 | • Unsatisfied stakeholder          | Contingency       | • Schedule joint meetings to resolve differences and agree on a common set |
| CS_R_010 | • Project Failure  
• Delayed or no graduation          | Mitigation        | • Monthly PSG meetings to share progress and receive feedback. Take corrective actions, if necessary. |
| CS_R_011 | • Affects project scope          | Acceptance        | • Reduce scope and review project plan. |
11.4 Milestone Trend Analysis

Figure 44 shows a graph of milestone trend analysis for the prototype and count2electron algorithm development activities. The purpose of this graph is to highlight the delay in developing the first and second algorithm increments. Because of this delay, the third and fourth algorithm increments could not be developed.

Figure 44. Milestone Trend Analysis

Figure 44 shows that all the prototype increments were developed on time. However, the count2electron algorithm increments were delayed. The first increment was delayed by five weeks because the risk CS_R_002 (incorrect activity estimation) was realized. This activity underestimation happened for two reasons:

- After the AutoStar task library was extended as part of the first increment, the library was redesigned. Hence, the extension had to undergo redesign as well. This rework took an additional three weeks, which delayed the project (see risk CS_R_005).

- TDD approach to implementation brings an additional overhead of creating test mocks and stubs upfront. Further, it is a different approach to development which takes time to get used to. This wasn’t factored in the initial planning and delayed the increment by two weeks.

The second increment of the calibration algorithm was delayed by three weeks because:

- During the increment development, it was realized that some functionalities were not provided by TEM OMP, the lower layer component. These had to be implemented which increased the development time by two weeks.

- The TDD overhead that affected the first increment caused a delay of one week for the second. Additional mocks and stubs were needed for the second increment, which caused the delay.

11.5 Conclusions

The incremental and iterative approach worked well for the project. It allowed both the prototype and calibration algorithm development to be divided into complete, useful slices of functionality called as increments. Each increment was visible to the user either via a UI or unit tests. Further, reviewing each increment allowed changes to be made to the increment before being integrated into the system. When certain risks were realized, increments could be adjusted accordingly. Due to certain risk realizations,
only two out of four Count2electron algorithm increments could be developed. However, these two increments along with the unit and module tests written for them were sufficient to prove the correctness of the design.
12. Project Retrospective

This chapter presents a reflection on the past ten months that were spent on carrying out the project. In addition, the design opportunities that were identified at the beginning of the project are revisited and evaluated.

12.1 Reflection

One of the most important reasons I joined the PDEng program at the TU/e was because I was keen on learning how to architect and design a system. In addition, I wanted to apply theoretical concepts to industrial projects. The PDEng program provided me an opportunity to do so. The first year included three, six-week mini-projects. These projects were not long enough to understand the benefits of applying the design principles and development practices. This ten-month project at Thermo Fisher Scientific provided me an ideal platform to apply and understand the design aspects on an in-production legacy software.

However, this ten-month project was more than just a design project. It involved executing most of the stages of the software V-model. Further, I was the project owner. This meant that I had to either find solutions independently or ask the right questions to the right stakeholders for guidance. I also did not know much about the Transmission Electron Microscope domain. Hence, applying newly learnt concepts to a new domain was a concern and a challenge at the beginning of the project.

My concerns were allayed as I started working with my team. Everyone shared their knowledge openly, when approached, that helped me understand the domain and motivated me to give my best. Further, building a prototype at first also helped allay my concerns. Building the prototype gave me time to understand the domain and apply the design principles.

While implementing the redesigned calibration algorithms, I was skeptical about using the TDD approach, partly because I had never used it before. I found TDD tedious and noticed that it slowed down my speed of implementation. Thinking of creating tests before writing code was difficult. However, I persisted with the approach and the benefits shone through when the code had to be refactored or modified due to design changes. Having all the existing code covered by tests gave me the confidence to make changes. Further, I realized that the changes could be made faster since the tests isolated the failures.

Overall, working on this project has been a learning, motivating, and rewarding experience. I have grown both professionally and personally during my time working on this project. The project gave me an opportunity to exercise my design skills and work in a team with my stakeholders. It was a pleasure to work on this project because of the people associated with it.

12.2 Design opportunities revisited

In Section 3.4, three design opportunities were identified for the project, namely, maintainability, usability, and extensibility. Maintainability was defined in terms of two of its sub-characteristics, namely, testability, and maintainability. In this section, these design opportunities are revisited to see how they were realized in the project.

12.2.1. Testability and Modifiability

A TDD approach to development was used to improve their testability. Non-functional requirement CS_NFR_001 was added for test coverage as a metric to indicate how effectively TDD approach was used. Test coverage tools were used to monitor coverage regularly.
The characteristics of a testable software are adequate complexity, low coupling, and good separation of concerns. Dependency Inversion principle was used to reduce the coupling in the system. Dependency Inversion isolates abstraction and details making the code easier to modify. This isolation was achieved by adding interfaces such that the high-level modules do not depend on low level implementation details. Adding interfaces allowed modules to be decoupled and classes to be mocked and stubbed, improving the ability to create unit tests. A non-functional requirement on cyclomatic complexity, CS_NFR_006, was added to ensure that the system did not become overly complex. The Single Responsibility principle was used to achieve separation of concerns. This was achieved by separating the object creation, algorithm sequence, algorithm steps, and mathematical computations. The algorithm steps were further separated into common steps (like repeat, skip, continue, and accept) and non-common steps (specific to each algorithm).

Testability also improved the modifiability of the system because it made running tests easier after code changes. Low coupling and good separation of concerns also helped in improving the modifiability of the system.

Improving the testability and modifiability of the system made it more maintainable, which was one of the project goals.

12.2.2. Usability

To ensure that the end users are not affected by the algorithm redesign, it was decided to link the AutoStar calibration implementation with the Delphi UI. Although this connection could not be established within the project duration, a prototype was developed to demonstrate that it is feasible and can be taken up as future work.

12.2.3. Extensibility

The dependency inversion principle reduces coupling in a system. This principle was used in the AutoStar extension to separate the abstractions like repeating, skipping or continuing an algorithm step from the algorithm implementation details. This made it easier to add new calibration algorithms by adding a new algorithm interface and deriving from the InteractiveTask template class to realize a calibration algorithm. In this way, the system was made extensible.
# Glossary

<table>
<thead>
<tr>
<th><strong>API</strong></th>
<th>Application Program Interface</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>AutoStar</strong></td>
<td>AutoStar is a component in the TEM software stack that is used for prototyping and developing functionality to automate microscope control.</td>
</tr>
<tr>
<td><strong>CAFCR</strong></td>
<td>CAFCR is a multi-view method for architecting systems by balancing genericity and specificity. CAFCR is an acronym for five different views, namely, Customer objectives view, Application view, Functional view, Conceptual view, and Realization view.</td>
</tr>
<tr>
<td><strong>CCD</strong></td>
<td>Charge coupled device</td>
</tr>
<tr>
<td><strong>COM</strong></td>
<td>Component Object Model, a Microsoft software interface technology</td>
</tr>
<tr>
<td><strong>Cyclomatic Complexity</strong></td>
<td>Cyclomatic complexity is one of the oldest software metrics that counts the number of independent paths through a program.</td>
</tr>
<tr>
<td><strong>DLL</strong></td>
<td>Dynamic Link Library</td>
</tr>
<tr>
<td><strong>FEG</strong></td>
<td>Field Emission Gun</td>
</tr>
<tr>
<td><strong>FEI</strong></td>
<td>Field Electronics and Ion Company</td>
</tr>
<tr>
<td><strong>IOM</strong></td>
<td>Instrument Object Model is an interface to the TEM server.</td>
</tr>
<tr>
<td><strong>OOTI</strong></td>
<td>Ontwerpersopleiding Technische Informatica (see ST)</td>
</tr>
<tr>
<td><strong>PDEng</strong></td>
<td>Professional Doctorate in Engineering</td>
</tr>
<tr>
<td><strong>PSG</strong></td>
<td>Progress Steering Group that comprises of supervisors from the university and the company and evaluates the candidate for his performance during internship.</td>
</tr>
<tr>
<td><strong>ST</strong></td>
<td>PDEng degree in Software Technology. See [1]</td>
</tr>
<tr>
<td><strong>TDD</strong></td>
<td>Test Driven Development is a software development process which focuses on writing unit tests before writing production code.</td>
</tr>
<tr>
<td><strong>TEM</strong></td>
<td>Transmission Electron Microscope</td>
</tr>
<tr>
<td><strong>TEM OMP</strong></td>
<td>Transmission Electron Microscope Object Model Proxy. TEM OMP is an abstraction on top of TEM server that is used by AutoStar to access and control the microscope.</td>
</tr>
<tr>
<td><strong>TTM</strong></td>
<td>Time To Market</td>
</tr>
<tr>
<td><strong>TU/e</strong></td>
<td>Eindhoven University of Technology</td>
</tr>
<tr>
<td><strong>UI</strong></td>
<td>User Interface</td>
</tr>
<tr>
<td><strong>VM</strong></td>
<td>Virtual Machine</td>
</tr>
</tbody>
</table>
References


Appendix A: Transmission Electron Microscope

A microscope is an instrument that produces enlarged images of small objects, allowing the observer an exceedingly close view of the minute structures at a scale convenient for examination and analysis [17]. The word microscope is derived from the Greek words ‘mikros’ (small) and ‘skopeo’ (look at). Most microscopes can be classified as one of three basic types:

- **Optical Microscopes** – They use visible light and transparent lenses to see objects as small as about one micrometer, such as red blood cell (7 µm) or a human hair (100 µm). These are most commonly used in facilities ranging from high school science lab to doctor’s office.
- **Charged Particle (Electron and Ion) Microscopes** – These microscopes use a beam of charged particles instead of light and use electromagnetic or electrostatic lenses to focus the particles. They can be used to see features as small as tenth of a nanometer, such as individual atoms.
- **Scanning Probe Microscopes** – They use a physical probe (a very small, very sharp needle) which can scan over the sample in contact or near-contact with the surface. They too provide atomic scale resolution.

Figure 45 shows the image resolution power of the microscopes.

<table>
<thead>
<tr>
<th>Resolving power of microscopes</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 m</td>
</tr>
<tr>
<td>--------------------------------</td>
</tr>
<tr>
<td>1 m</td>
</tr>
</tbody>
</table>

**Figure 45. Resolving power of microscopes [18]**
A Transmission Electron Microscope (TEM) is a charged particle microscope. The first TEM was built by Ernst Ruska at the University of Berlin in 1931. Initially, this microscope used two magnetic lenses which was later increased to three to demonstrate a resolution of 100nm, twice as good as the light microscopes from those times.

**Operating Principles**

The TEM can be compared with a slide projector as shown in Figure 46.

![Figure 46. Transmission Electron Microscope compared with a slide projector](image)

In TEM, electron beams emitted from an electron source are made into parallel beams by the Electromagnetic condenser lens. These parallel beams are passed through the specimen and are then focused on to a fluorescent screen or an electronic imaging device such as CCD (Charge Coupled Device) camera. A fluorescent screen emits light when struck by electrons. The electron path from the source to the screen is under vacuum because electrons scatter or get absorbed on interaction with other gases. In addition, the specimen (object) must be very thin to allow the electrons to pass through [19].

**TEM Components**

*Figure 47* depicts a Transmission Electron Microscope.
There are four main components to a TEM: an electron optical column, a vacuum system, electronics, and control software.

**Electron Gun**

There are three main types of electron sources: tungsten, lanthanum hexaboride, and field emission guns. Table 26 highlights the important points for each of the three sources.

**Table 26. Comparison of electron sources**

<table>
<thead>
<tr>
<th>Tungsten</th>
<th>Lanthanum hexaboride</th>
<th>Field emission gun (FEG)</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Comprises of a filament, a Wehnelt cylinder, and an anode. These three together form a triode gun, which is a very stable source of electrons.</td>
<td>• These guns depend on thermionic emission of electrons from a heated source, lanthanum hexaboride crystal.</td>
<td>• In these guns, the electrons are extracted from a very sharply pointed Tungsten tip by an extremely high electric field.</td>
</tr>
<tr>
<td>• Least expensive</td>
<td>• More expensive than a Tungsten gun</td>
<td>• Most expensive of the three</td>
</tr>
<tr>
<td>• Offers lowest brightness</td>
<td>• Offers 10x more brightness than Tungsten guns</td>
<td>• Offers highest brightness.</td>
</tr>
<tr>
<td>• Suitable for magnifications up to 40-50kX</td>
<td>• Suitable for magnifications between 50-100kX</td>
<td>• Suitable for magnifications higher than 100Kx</td>
</tr>
</tbody>
</table>

Further, there are two types of field emission sources, namely cold field emission and Schottky (thermally assisted) field emission. Cold field emission offers very high brightness but varying beam currents. It requires frequent flashing to clean containments from the tip. On the other hand, Schottky field emission offers high brightness.
and high, stable current with no flashing requirements. In fact, the latest generation Schottky field emitters attain brightness levels close to cold field emission levels.

**Electromagnetic Lens System**

There are three kinds of lens system in a TEM – condenser lens system, objective lens system, and projection lens system. Figure 48 is an illustration of the components that comprise a lens system [20].

![Components of an Electromagnetic Lens System](image)

**Figure 48. Components of an Electromagnetic Lens System**

Each of the three lens systems has four major components [20]:

- **Deflectors** – Deflectors are responsible for correcting the electron beams that are not parallel to the optical axis of the lens. They bend the electron beam to send it as close to the optical beam as possible.

- **Lens** – Depending on which lens system they belong to; the lenses have different functions. The condenser lens system focusses the electron beam onto the specimen as much as necessary to suit the purpose. The objective lens produces an image of the specimen. The projection lenses follow the objective lenses and are used to focus, magnify, and project the image or diffraction pattern onto the viewing device.

- **Stigmators** – The lenses suffer from Lens Stigmatism. Lens Stigmatism refers to the condition where the magnetic field produced by one part of the lens is stronger than another part. A Stigmator solves this problem by producing a magnetic field to compensate for the asymmetric magnetic field produced by the main lenses. The Stigmators are mounted at the bottom of the lens system.

- **Aperture** – The apertures block the scattered electron beams which are not close to the optical axis of the lens. This is done because such beams do not focus well through the later lenses to form the image.

**Vacuum**

The entire microscope column from the source to the fluorescent screen is evacuated to create a vacuum. This is necessary because in vacuum, electron behave like light. The maximum vacuum level is around the specimen and the source while the minimum vacuum level is in the projection chamber and camera chamber. To avoid having to evacuate the whole column every time a new specimen is placed, several airlocks and separation valves are built in [19].

**Electronics**

The TEM houses sophisticated electron circuits to ensure an extremely stable voltage and current. This is necessary to obtain a very high resolution. The voltage or current is so stable that their values do not deviate by more than one part in ten million of the value selected [19].
Control Software

Modern electron microscopes employ a fast, powerful computer to control, monitor, and record the operating conditions of the microscope. This allows special techniques and algorithms to be embedded in the microscope in the control software so that the operator can carry them out using the same controls. As the microscope optical design gets more and more complicated, there is a greater need to simplify the microscope operation. The added benefit is that this will allow more users with less specialized training to operate the microscope. [19].
About the Author

Nityanand Panpalia received his Bachelor’s degree in Computer Science from Visvesvaraya Technological University, India in 2006. In 2008, he received his Master’s degree in Software Engineering from Birla Institute of Technology, Mesra, India. Nityanand did his Master’s thesis at Intel Technology India Private Ltd. The thesis was titled “Time Based Transaction Stitching” and involved developing time based algorithms to validate Intel’s First Generation Quick Path Interconnect (QPI) Protocol.

After graduation, Nityanand joined Intel as a full-time employee. He went on to work for eight years, primarily as a Software Engineer, developing in-house JTAG-assisted Debug and Trace Solutions for Intel Architecture.

In 2015, he changed roles to become a Functional Safety Certified Automotive Engineer and received ISO 26262 certification. In this role, he was involved in software verification of test libraries that were deployed in autonomous vehicles.

In 2016, he joined Eindhoven University of Technology (TU/e) as a Software Technology PDEng trainee. During the first year of this program, he worked on three mini-projects for Bosch Securities, TNO, and Philips Lighting. His final year internship was at Thermo Fisher Scientific where he worked on redesigning the calibration algorithms for Transmission Electron Microscopes.