

Entwicklung und Implementierung eines Testkonzepts
für die Datenauswertung des RadarImagers unter
Verwendung von C++

T2000_02

für die Prüfung zum
Bachelor of Engineering

des Studiengangs Elektrotechnik - Elektronik
an der Dualen Hochschule Baden-Württemberg Stuttgart

von
Nico Durst-Claus

02. September 2024

| | |
|---|--------------------------------|
| Bearbeitungszeitraum | 27.05.2024 - 02.09.2024 |
| Matrikelnummer | 7199590 |
| Kurs | TEL22GR2 |
| Ausbildungsunternehmen | Balluff GmbH, Neuhausen a.d.F. |
| Betreuer des Ausbildungsunternehmens | Patrick Benz (B. Eng.) |

Eidesstattliche Erklärung

Ich versichere hiermit, dass ich meine Projektarbeit mit dem Thema:

„Entwicklung und Implementierung eines Testkonzepts für die
Datenauswertung des RadarImagers unter Verwendung von C++“

selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Ich versichere zudem, dass die eingereichte elektronische Fassung mit der gedruckten Fassung übereinstimmt.

Ort, Datum

Unterschrift

Gender-Hinweis

Aus Gründen der Lesbarkeit wird in dieser Projektarbeit darauf verzichtet, durchgehend geschlechterspezifische Formulierungen zu verwenden. Soweit personenbezogene Bezeichnungen nur in männlicher Form angeführt sind, beziehen sie sich auf Männer, Frauen und Diverse in gleicher Weise.

Dies soll jedoch keinesfalls eine Geschlechterdiskriminierung oder eine Verletzung des Gleichheitsgrundsatzes zum Ausdruck bringen. Entsprechende Begriffe gelten im Sinne der Gleichbehandlung grundsätzlich für alle Geschlechter. Die verkürzte Sprachform beinhaltet keine Wertung.

Sperrvermerk

Die gesamte Arbeit ist geistiges Eigentum des Verfassers und der Balluff GmbH. Die verwendeten Zitate und Abbildungen anderer Autoren sind als deren geistiges Eigentum durch das Copyright ebenfalls geschützt. Eine Verwendung dieser Arbeit, auch nur in Auszügen ist, ohne ausdrückliche Genehmigung des Verfassers und der Balluff GmbH, rechtswidrig.

©

Nico Durst-Claus
Balluff GmbH

Zusammenfassung

Hier steht die deutsche Zusammenfassung

Abstract

Hier steht die englische Zusammenfassung

Inhaltsverzeichnis

| | |
|---|------------|
| Abbildungsverzeichnis | I |
| Abkürzungsverzeichnis | II |
| 1 Einführung | 1 |
| 1.1 Problemstellung | 1 |
| 1.2 Zielsetzung | 1 |
| 1.3 Vorgehensweise | 1 |
| 2 Theoretische Grundlagen | 3 |
| 2.1 Grundlagen des RadarImagers | 3 |
| 2.2 Grundlagen des GenICam-Standards | 5 |
| 2.3 Grundlagen von C++ | 6 |
| 2.4 Grundlagen des Testings | 7 |
| 2.5 Grundlagen der Testautomatisierung | 10 |
| 2.6 Aufbau der Testumgebung | 11 |
| 3 Stand der Technik | 13 |
| 3.1 Impact Acquire SDK | 13 |
| 3.2 Methoden der Testkonfiguration | 14 |
| 3.3 Methoden des Loggings | 15 |
| 4 Konzeptentwicklung | 17 |
| 4.1 Definition der Anforderungen | 17 |
| 4.2 Konzeptionelle Gestaltung der Konfiguration | 18 |
| 4.3 Konzeptionelle Gestaltung des Loggings | 20 |
| 5 Implementierung des Testkonzepts in C++ | 24 |
| 5.1 Aufbau und Organisation des Projektordners | 24 |
| 5.2 Verarbeitung der Konfigurationsdatei | 25 |
| 5.3 Umsetzung der Konfigurationsmöglichkeiten | 27 |
| 5.4 Umsetzung des Loggings | 29 |
| 5.5 Entwicklung und Durchführung der Modultests | 30 |
| 5.6 Ablauf des Testprogramms | 33 |
| 6 Bewertung und Fazit | 36 |
| Literaturverzeichnis | III |
| Anhang | V |

Abbildungsverzeichnis

| | | |
|-----|---|----|
| 2.1 | RadarImager | 3 |
| 2.2 | Einzelbild und Bildstapel | 4 |
| 2.3 | Testpyramide | 9 |
| 2.4 | Testumgebung | 11 |
| 4.1 | Beispiel Log-Datei | 21 |
| 5.1 | Verzeichnisstruktur | 24 |
| 5.2 | Ergebnis Modultests | 32 |
| 5.3 | Flussdiagramm Hauptprogramm | 33 |
| 5.4 | Flussdiagramm Callback-Funktion | 34 |

Abkürzungsverzeichnis

| | |
|-------------|-------------------------------------|
| API | Application Programming Interface |
| EMVA | European Machine Vision Association |
| FPS | Frames per Second |
| GUI | Graphical User Interface |
| IPC | Industrial PC |
| PAT | Personal Access Token |
| REST | Representational State Transfer |
| SDK | Software Development Kit |
| STL | Standard Template Library |

1 Einführung

1.1 Problemstellung

Die Entwicklung eines industriellen 3D-Bildgebungssystems auf Basis von Radartechnologie (RadarImager) erfordert umfangreiche Tests zur Gewährleistung der korrekten Funktionalität und Datenübertragung. Hierfür existiert bereits ein provisorisches Testprogramm, welches jedoch nicht dem gewünschten Funktionsumfang entspricht. Es soll eine einfache Parametrisierung und Konfiguration des Tests möglich sein, um das Einstellen von verschiedenen Testbedingungen zu ermöglichen. Zudem müssen das Verhalten sowie mögliche Fehler des RadarImagers beziehungsweise der Datenübertragung aufgezeichnet werden und nachvollziehbar sein. Das Testkonzept sowie das dazugehörige Programm dürfen keine weiteren Fehler und somit Unsicherheiten erzeugen.

1.2 Zielsetzung

Ziel dieser Arbeit ist die Entwicklung eines Konzepts für die Konfiguration und Parametrisierung von Tests mittels einer Konfigurationsdatei und die automatisierte Generierung von Log-Dateien. Dieses Konzept soll in C++ implementiert und gründlich auf seine Funktionalität getestet werden. Am Ende des Projekts sollen die Testparameter in der Konfigurationsdatei eingestellt werden, sodass der Test automatisiert durchgeführt und mögliche Fehler über die Log-Dateien identifiziert werden können. Dadurch sollen Fehler und Auffälligkeiten des RadarImagers beziehungsweise in der Datenübertragung erkannt werden.

1.3 Vorgehensweise

Die Arbeit beginnt mit einer gründlichen Einarbeitung in die Thematik, einschließlich der GenICam-Technologie, der C++-Programmiersprache und des Verhaltens des RadarImagers. Die provisorische Version des Testprogramms wird dabei ebenfalls berücksichtigt.

Die Anforderungen werden mit dem zuständigen Testingenieur abgestimmt. Anschließend wird das Testkonzept unter Berücksichtigung der gesammelten Anforderungen an den Test ausgearbeitet und in C++ implementiert. Zusätzlich wird der Aufbau der Konfigurationsdatei und der Log-Dateien festgelegt. Dabei steht die Automatisierung des Testvorgangs im Fokus. Um die Funktionalität sicherzustellen und Fehler durch das Testprogramm zu vermeiden, werden nach jeder Erweiterung des Funktionsumfangs weitere Modultests implementiert.

2 Theoretische Grundlagen

2.1 Grundlagen des RadarImagers

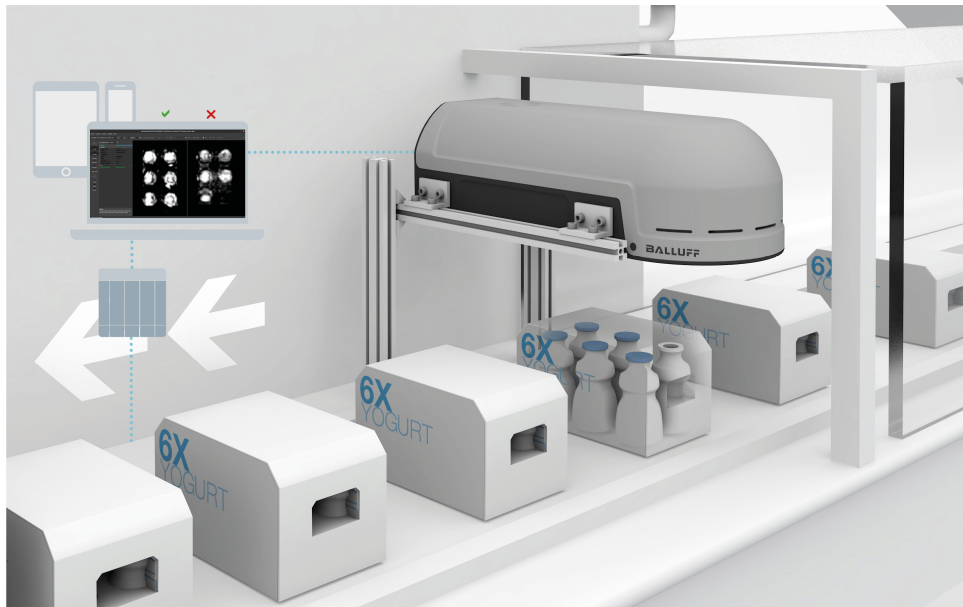


Abbildung 2.1: RadarImager (Balluff GmbH, 2024a)

Der RadarImager von Balluff ist ein industrielles 3D-Bildgebungssystem, welches Objekte hinter nicht-leitfähigen Materialien sichtbar macht. Dieses System findet Anwendung in der Qualitätssicherung, insbesondere in Branchen wie der Verpackungs- und Pharmaindustrie. Es ermöglicht eine zerstörungsfreie Kontrolle von Verpackungen auf Vollständigkeit, die Überprüfung der Unversehrtheit des Produkts sowie die Identifizierung von Fremdkörpern. Darüber hinaus können Verschlüsse geprüft und Füllstände erkannt werden (siehe Abbildung 2.1). Die Auswertung der Bilder erlaubt die zuverlässige Identifikation von Fehlern.

Der RadarImager basiert auf der Radartechnologie, wobei Radar für „Radio Detection And Ranging“ steht. Diese Technologie nutzt elektromagnetische Wellen, um Informationen über die Position, Bewegung und Eigenschaften von Objekten zu erhalten. Das System sendet elektromagnetische Wellen aus, die von den Objekten reflektiert oder absorbiert werden. Der Frequenzbereich der verwendeten Radarwellen liegt im elektromagnetischen

Spektrum zwischen Mikrowelle und Infrarot, somit im unteren Gigahertz-Bereich. In diesem Frequenzbereich sind die Wellen nicht ionisierend, sodass keine gesundheitlichen Risiken bestehen. Die Wellenenergie wird je nach Material spezifisch absorbiert, was zu einer Reduzierung der Amplitude führt. Die Reflektion an den Grenzflächen der Objekte resultiert in Laufzeitdifferenzen zwischen der ursprünglichen und der reflektierten Welle. Die reflektierten Wellen werden vom RadarImager empfangen und anschließend als Signale aufbereitet, verarbeitet und analysiert. Eine Software wandelt diese Signale im Anschluss in Bilder um, die für das menschliche Auge erkennbar und somit auswertbar sind. Der RadarImager kann sämtliche dielektrische Materialien wie Folien, Kartonagen und Kunststoffe durchleuchten. Leitfähige Gegenstände, Metall und Flüssigkeiten werden erkannt, aber nicht durchdrungen. Daher können auch metallische Objekte aufgefunden und Füllstände gemessen werden.

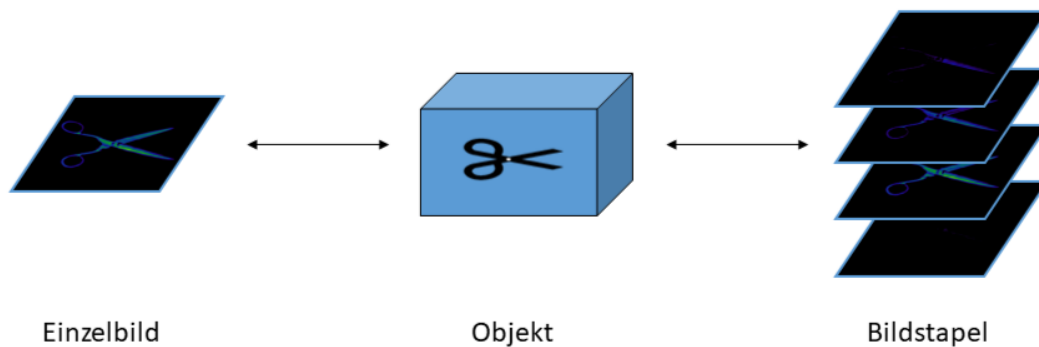


Abbildung 2.2: Einzelbild und Bildstapel (eigene Darstellung)

Der RadarImager ermöglicht es, zwischen Einzelbildern oder mehreren Bildern einer Messung (sogenannten Bildstapeln) zu wählen (siehe Abbildung 2.2). Ein Bildstapel ist eine Sammlung von Schichtaufnahmen, ähnlich den Ergebnissen einer MRT-Untersuchung in der Medizin. Dies ermöglicht es, später gezielt die Schichten mit dem höchsten Informationsgehalt auszuwählen oder mehrere Schichten gleichzeitig zu analysieren. Die Bilder werden mithilfe von GenICam über Gigabit-Ethernet übertragen. Dabei wird das Gige Vision Protokoll für die Kommunikation und den Datentransfer verwendet. Der RadarImager verfügt über eine 12V-Spannungsversorgung sowie zwei Eingänge für Trigger. Der erste Eingang ist für die Objekterkennung und die Erstellung des Bildes vorgesehen. Der zweite Eingang kann zur Messung der Bewegungsrichtung und Geschwindigkeit des Objekts verwendet werden. Des Weiteren ist ein Ethernet-Anschluss (RJ45) für die Übertragung der Bilder vorhanden und LEDs, die zur Anzeige des Zustands dienen.

(Balluff GmbH, 2024a)

2.2 Grundlagen des GenICam-Standards

GenICam, kurz für „Generic Interface for Cameras“, ist ein Standard von der European Machine Vision Association (EMVA), der eine einheitliche Programmierschnittstelle für verschiedene Arten von Kameras in der industriellen Bildverarbeitung definiert. Der Standard umfasst mehrere Module, darunter GenApi, GenTL, GenCP und GenDC, welche verschiedene Aspekte der Kameraschnittstellen und -kommunikation abdecken.

Das **GenApi** (Generic Application Programming Interface) definiert eine XML-basierte Beschreibungssprache, die die Funktionen und Parameter einer Kamera beschreibt. Durch diese Beschreibung können Anwendungen die Kameraeinstellungen dynamisch auslesen und konfigurieren, ohne auf herstellerspezifische Treiber angewiesen zu sein.

Der **GenTL** (Generic Transport Layer) legt eine standardisierte Schnittstelle für die Datenübertragung zwischen der Kamera und dem Host-System fest. Dies ermöglicht die Entwicklung von Anwendungen, die unabhängig vom verwendeten Übertragungsprotokoll (z.B. GigE Vision, Camera Link) sind. Ein GenTL Producer stellt die erforderlichen Funktionen bereit, um Datenströme von der Kamera zu empfangen und zu verwalten.

Das **GenCP** (Generic Control Protocol) definiert ein standardisiertes Protokoll für die Steuerung und Konfiguration von Kameras. Es erlaubt die Kommunikation zwischen der Kamera und der Anwendung, sodass Konfigurationsparameter gesetzt oder abgefragt sowie Befehle an die Kamera gesendet werden können.

Der **GenDC** (Generic Data Container) ist ein Format für die standardisierte Beschreibung und Übertragung von Bild- und Metadaten. Die Speicherung komplexer Datenstrukturen, wie beispielsweise Bildstapel (3D-Bilddaten), erfolgt in einem einheitlichen Containerformat. Dies erlaubt eine erleichterte Interpretation und Verarbeitung der Daten durch unterschiedliche Anwendungen und Systeme.

Die vom RadarImager-System erfassten Daten werden in Form eines Bildstapels verarbeitet. Diese Bildstapel setzen sich aus mehreren einzelnen Bildlagen zusammen, welche in einem GenICam GenDC bereitgestellt werden. Der Einsatz des GenDC gewährleistet eine einheitliche und standardisierte Darstellung der Bilddaten, wodurch eine erleichterte Weiterverarbeitung und Analyse möglich ist. Die Übertragung der Bilddaten erfolgt über ein GenICam GenTL Producer Interface, welches die Daten vom Kamerasystem über Gigabit-Ethernet an das Host-System sendet.

(Balluff GmbH, 2024b) (European Machine Vision Association, 2024)

2.3 Grundlagen von C++

C++ ist eine weit verbreitete und leistungsstarke Programmiersprache, die in verschiedenen Anwendungsbereichen der Softwareentwicklung eingesetzt wird. Die Programmiersprache ist eine Erweiterung von C und bietet zusätzliche Funktionen und Möglichkeiten, die über die von C hinausgehen. Ein wesentliches Merkmal von C++ ist die Fähigkeit, sowohl prozedurale als auch objektorientierte Programmierung zu unterstützen. Dies erlaubt es Entwicklern sowohl strukturierten Code, der auf Funktionen und Prozeduren basiert, als auch objektorientierten Code, der auf Klassen und Objekten basiert, zu schreiben. Die Flexibilität der Sprache ermöglicht es, den Code gemäß den jeweiligen Anforderungen und Präferenzen zu strukturieren.

Des Weiteren zeichnet sich C++ durch seine Effizienz und die Kontrolle über die Hardware aus. Die Sprache ermöglicht es Entwicklern, direkt auf den Speicher und die Ressourcen eines Systems zuzugreifen. Dies ist insbesondere in Anwendungsbereichen wie in eingebetteten Systemen von Vorteil, da dort eine hohe Leistung sowie eine effiziente Nutzung der Ressourcen erforderlich sind. Zur Unterstützung der Softwareentwicklung bietet C++ eine Vielzahl von Funktionen und Bibliotheken. Zu den wichtigsten Bibliotheken gehört die Standard Template Library (STL), die eine Vielzahl von Datenstrukturen und Algorithmen bereitstellt. Die Verwendung dieser Bibliotheken erlaubt es Entwicklern auf bewährte Lösungen zurückzugreifen und Zeit bei der Implementierung grundlegender Funktionen einzusparen.

Die Syntax von C++ ähnelt der von C, erweitert diese jedoch um zusätzliche Konstrukte und Schlüsselwörter, die speziell für die objektorientierte Programmierung konzipiert sind. Darüber hinaus unterstützt C++ fortgeschrittene Konzepte wie Vererbung, Polymorphismus, Templates und Ausnahmebehandlung, wodurch die Wiederverwendbarkeit und Flexibilität des Codes verbessert werden. Die genannten Konzepte ermöglichen es Entwicklern, komplexe Softwarearchitekturen zu entwerfen und den Code besser zu organisieren. C++ findet Anwendung in einer Vielzahl von Bereichen, darunter Desktop-Anwendungen, Spieleentwicklung, eingebettete Systeme, Datenbanken und künstliche Intelligenz. Diese Vielseitigkeit macht C++ zu einer wichtigen Programmiersprache, welche die Entwicklung von leistungsstarken und effizienten Softwarelösungen erlaubt.

(Stroustrup, 2015)

2.4 Grundlagen des Testings

Testing ist ein wichtiger Prozess bei der Entwicklung und Wartung von Software- und Hardwareprodukten. Das Hauptziel besteht darin, die Qualität, Sicherheit und Funktionalität des Produkts zu gewährleisten, indem Fehler und Probleme identifiziert werden, bevor das Produkt in Produktion geht oder an den Endbenutzer ausgeliefert wird. Zu den Hauptzielen des Testens gehören die Überprüfung der Einhaltung spezifizierter Anforderungen, die Validierung der Benutzererwartungen und die Identifizierung von Verbesserungsmöglichkeiten.

Testprozess Der Testprozess gliedert sich in mehrere Phasen, die eng miteinander verknüpft sind, um eine effiziente Überprüfung des Systems zu ermöglichen:

- **Testplanung:** Diese Phase umfasst die Erstellung des Testkonzepts und des detaillierten Testplans. Hierbei werden das Testobjekt definiert, die erforderliche Testumgebung beschrieben, die Konfiguration des Testsystems festgelegt und die benötigten Testressourcen bestimmt. Die Testplanung legt den Grundstein für alle nachfolgenden Testaktivitäten und definiert den Umfang sowie die Werkzeuge, die für die Tests verwendet werden sollen.
- **Testdesign:** In dieser Phase werden die Testanforderungen verfeinert und spezifiziert. Es werden Testszenarien entwickelt und Kriterien für den Abschluss der Tests festgelegt. Je nach Umfang und Komplexität des Projekts kann diese Phase eng mit der Testplanung verbunden sein.
- **Testspezifikation:** Hier erfolgt die detaillierte Beschreibung der einzelnen Testfälle, einschließlich der Festlegung der Testvoraussetzungen, der Definition der Eingaben und der Spezifikation der erwarteten Ausgaben.
- **Testdurchführung:** Die Tests können manuell oder automatisiert durchgeführt werden, wobei häufig eine Kombination beider Methoden zum Einsatz kommt. Diese Phase ist typischerweise iterativ, um das System kontinuierlich zu testen und sicherzustellen, dass neue Funktionen und behobene Fehler keine unerwünschten Nebeneffekte verursachen (Regressionstests).

Der gesamte Testprozess ist dynamisch und passt sich den Gegebenheiten des jeweiligen Entwicklungsprojekts an. In agilen Umgebungen wird der Testprozess flexibel gestaltet, um schnell auf Änderungen reagieren zu können und eine gleichbleibend hohe Qualität

zu gewährleisten. Das Testmanagement spielt dabei eine zentrale Rolle, indem es die verschiedenen Testaktivitäten koordiniert, Ressourcen verwaltet und die Kommunikation zwischen den Projektbeteiligten steuert.

Grundsätze Im Rahmen von Tests sind einige Grundsätze zu berücksichtigen, die zu einem gemeinsamen Verständnis beitragen und eine korrekte Einordnung der Testaktivitäten ermöglichen. Im Folgenden erfolgt eine kurze Darstellung und Erläuterung ausgewählter Grundsätze:

Durch die Testaktivitäten werden Fehler erzeugt und somit erkannt. Je höher die Testabdeckung ist, desto geringer ist daher das Risiko, dass noch unentdeckte Fehler vorhanden sind. Testen kann jedoch nicht beweisen, dass ein System fehlerfrei ist. Ziel des Testens ist es daher, mit den gegebenen Ressourcen so viel Qualität wie möglich sicherzustellen.

Tests sind immer Stichproben. Es werden nicht alle möglichen Eingabemöglichkeiten mit allen möglichen Umgebungsbedingungen getestet, sondern häufig Grenzfälle oder Kombinationen, die am ehesten praxisrelevant sind. Der Aufwand hängt von der Priorität und dem Risiko ab und muss gegen den Nutzen des Tests abgewogen werden. Darüber hinaus dient ein Test dazu, festzustellen, ob ein System so funktioniert, wie es die Spezifikation vorsieht. Inwieweit diese Spezifikation sinnvoll ist, kann im Test hinterfragt werden, ist aber grundsätzlich nicht Aufgabe des Tests.

Es ist sinnvoll, frühzeitig mit dem Testen zu beginnen. Dadurch können Fehler bereits in der Entwicklungsphase erkannt werden, in der sie entstehen. Somit wird die Korrekturzeit verkürzt und der Integrationsaufwand verringert.

Häufig sind Fehler nicht gleichmäßig über das gesamte System verteilt, sondern treten gehäuft in bestimmten Komponenten auf. Daher sollte man beim Testen flexibel auf solche Häufungen eingehen und diese Bereiche genau überprüfen. Es ist also wichtig, neben einer genauen Testplanung auch kreativ auf solche Herausforderungen zu reagieren.

Neue Anforderungen, veränderte Umgebungsbedingungen und erweiterte Szenarien erfordern neue Testfälle. Die Testspezifikation muss ständig kritisch überprüft und gegebenenfalls ergänzt oder aktualisiert werden. Fehlende Tests führen zu einer unzureichenden Testabdeckung und damit zu einem erhöhten Fehlerrisiko.

Die Tests müssen an das Einsatzgebiet und die Umgebung des Systems angepasst werden. Die Testabdeckung und der Testumfang sind für jedes System individuell zu bewerten

und festzulegen. Sicherheitskritische Systeme oder Anwendungen in der Medizin erfordern umfangreichere und detailliertere Tests als eine Spielsoftware oder eine grafische Benutzeroberfläche. Je nach Art der Anwendung unterscheiden sich auch die zu tolerierenden Fehler.

Grundsätzlich ist es sinnvoll, das Testen von der Entwicklung personell zu trennen, da eine andere Sichtweise auf das System erforderlich ist. Zudem bewertet ein Entwickler sein eigenes Produkt naturgemäß milder als ein neutraler Tester. So findet er tendenziell weniger Fehler und kann auch nicht korrigieren, wenn ein Fehler durch eine falsch verstandene Anforderung entstanden ist.

Testarten Die Überprüfung eines komplexen Systems gliedert sich in mehrere Testarten. Die Einordnung der Testarten orientiert sich an dem Entwicklungsstand des Systems. Dabei wird angestrebt, die Tests bestimmter Funktionalitäten möglichst frühzeitig durchzuführen. Abbildung 2.3 zeigt eine Übersicht der verschiedenen Testarten sowie deren Eigenschaften in Bezug auf den Aufwand und die Anzahl der Testfälle:

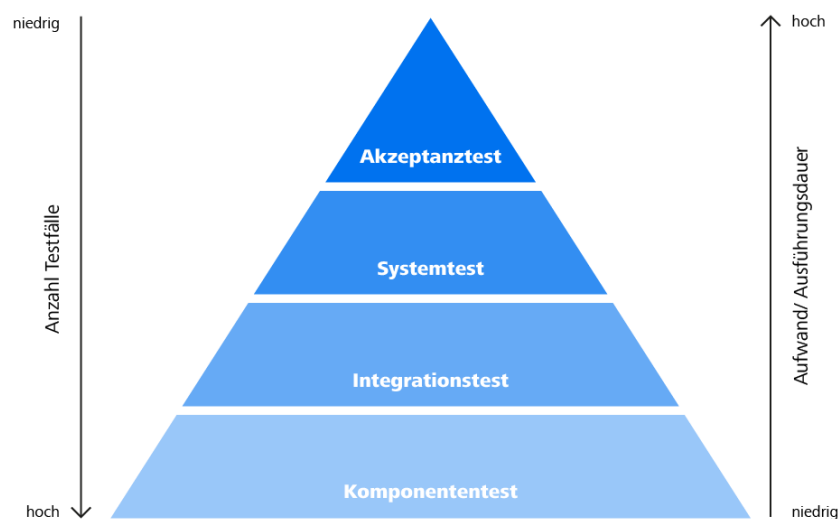


Abbildung 2.3: Testpyramide (Carl Zeiss Digital Innovation GmbH, 2024)

- **Komponententests** (auch Modultests oder Unittests) werden durchgeführt, um die einzelnen Komponenten separat zu überprüfen.
- **Integrationstests** untersuchen, wie die verschiedenen Komponenten zusammenarbeiten und prüfen die Schnittstellen zwischen unterschiedlichen Systemen und Anwendungen.

- **Systemtests** erfolgen, wenn alle Komponenten zusammengeführt sind. Dabei wird die gesamte Anwendung auf die spezifizierten Anforderungen getestet.
- **Akzeptanztests** prüfen, ob die entwickelte Lösung die Anforderungen der Nutzer erfüllt. Sie werden in Zusammenarbeit mit dem Kunden durchgeführt und basieren auf zentralen Testfällen aus den Systemtests.

(Witte, 2019) (Witte, 2023)

2.5 Grundlagen der Testautomatisierung

Die Testautomatisierung ist ein wesentlicher Bestandteil der modernen Entwicklung, um den Testprozess effizienter und zuverlässiger zu gestalten. Durch den Einsatz spezieller Softwaretools werden Tests automatisch gesteuert, Ergebnisse verglichen und detaillierte Berichte erstellt. Diese Automatisierung trägt wesentlich zur Steigerung der Effizienz und Qualität des Testprozesses bei.

Ziele Das primäre Ziel der Testautomatisierung ist es, manuelle Testverfahren zu ersetzen oder zu ergänzen, um die Geschwindigkeit und Genauigkeit der Testausführung zu erhöhen. Zu den Hauptvorteilen gehören die Steigerung der Testeffizienz und die Verbesserung der Testqualität. Automatisierte Tests liefern konsistente Ergebnisse und erhöhen die Testabdeckung, was die Fehlererkennung verbessert und die Softwarequalität insgesamt erhöht.

Vorteile Ein wichtiger Vorteil ist die Kosten- und Zeitersparnis. Die Einführung und Pflege von Testautomatisierung erfordert zwar Aufwand, führt aber langfristig zu erheblichen Einsparungen durch die Reduzierung manueller Tätigkeiten und die Minimierung von Fehlern. In agilen Entwicklungsumgebungen ermöglicht das schnelle Feedback automatisierter Tests eine schnellere Reaktion auf Änderungen und trägt zur kontinuierlichen Verbesserung bei.

Herausforderungen Trotz der vielen Vorteile bringt die Testautomatisierung auch Herausforderungen mit sich. Der anfängliche Aufwand für die Einrichtung und die laufende Wartung automatisierter Tests erfordert Fachwissen und eine sorgfältige Planung. Die Tests müssen regelmäßig aktualisiert werden, um den sich ändernden Anforderungen gerecht zu werden. Ein weiteres Problem ist der menschliche Faktor: Kreatives Testdesign und exploratives Testen sind nach wie vor Aufgaben, die menschliches Urteilsvermögen erfordern und nicht vollständig automatisiert werden können.

Integration Die Integration der Testautomatisierung in den Entwicklungsprozess erfordert eine sorgfältige Planung und Zusammenarbeit zwischen Testern, Entwicklern und anderen Beteiligten. Die Qualität der automatisierten Tests sollte im Vordergrund stehen, da stabile und aussagekräftige Tests wichtiger sind als eine hohe Anzahl automatisierter Testfälle. Darüber hinaus ist es wichtig, einen kontinuierlichen Verbesserungsprozess zu implementieren, um sicherzustellen, dass die Testautomatisierung effizient und effektiv bleibt und an sich ändernde Anforderungen angepasst werden kann.

(Baumgartner et al., 2021) (Witte, 2023)

2.6 Aufbau der Testumgebung

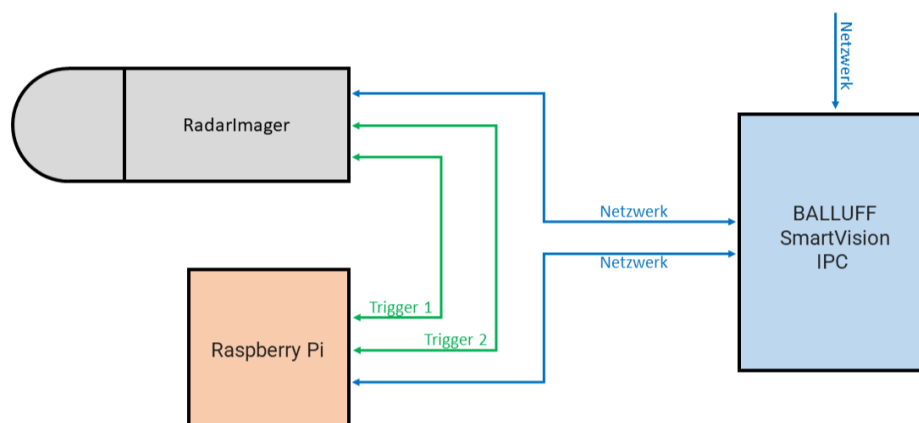


Abbildung 2.4: Testumgebung (eigene Darstellung)

Abbildung 2.4 zeigt den schematischen Aufbau der Testumgebung. Der RadarImager ist über ein Netzwerk mit einem Industrial PC (IPC) verbunden, auf dem Ubuntu als Linux-Distribution läuft. Zusätzlich ist der IPC über ein weiteres Netzwerk mit einem

Raspberry Pi verbunden, der einen Hardware-Trigger simuliert. Somit wird kein komplexer Aufbau mit Lichtschranken und einem Förderband benötigt, um Triggersignale zu erzeugen. Das Intervall und der Abstand zwischen den einzelnen Signalen ist einstellbar. Bei Implementierungen werden die Trigger durch externe Mechanismen wie beispielsweise eine Lichtschranke oder durch eine individuelle Lösung des Kunden erzeugt. Die Ansteuerung des Raspberry Pi erfolgt über eine REST-API vom IPC aus. Eine REST-API (Representational State Transfer Application Programming Interface) ist eine Programmierschnittstelle, die den Datenaustausch zwischen verschiedenen Softwareanwendungen über das Internet ermöglicht. Sie verwendet standardisierte HTTP-Methoden, um Ressourcen abzurufen, zu erstellen, zu bearbeiten oder zu löschen. Die Übertragung der Bilder vom RadarImager zum IPC erfolgt über das Netzwerk unter Verwendung des GenICam-Standards. Zum Empfangen und Verarbeiten der Bilder auf dem IPC wird der GenICam-Client verwendet. Dieser wird im Laufe dieser Arbeit für die Tests verbessert und weiterentwickelt. Um das Senden der Bilder bzw. Bildstapel zu simulieren, wird während der Weiterentwicklung ein NVIDIA Jetson Orin NX verwendet, welcher auch im RadarImager verbaut ist. Dieser sendet synthetische Bilder auf die gleiche Weise, wie der RadarImager reale Bilder sendet. Der Einsatz des NVIDIA Jetson Orin NX ermöglicht eine erhebliche Kosteneinsparung in der Entwicklungsphase, da die Kosten für dieses Gerät nur wenige hundert Euro betragen. Im Gegensatz dazu kostet der komplette RadarImager mit Radarmodulen und anderen Hardwarekomponenten etwa 30.000 Euro. Darüber hinaus bietet dies die Möglichkeit, vorhandene RadarImager für andere Zwecke zu nutzen, beispielsweise für Demonstrationen bei potenziellen Kunden.

3 Stand der Technik

3.1 Impact Acquire SDK

Impact Acquire bezeichnet eine Software-Schnittstelle, die speziell für die Steuerung von Bildverarbeitungskomponenten, insbesondere Kameras, konzipiert ist. Die Schnittstelle ermöglicht Entwicklern den Zugriff auf eine Vielzahl von Funktionen und Datenströmen der angeschlossenen Bildverarbeitungskomponente. Ihre Anwendung erfolgt insbesondere in Bereichen wie der Industrieautomatisierung und Qualitätskontrolle sowie in anderen bildverarbeitungs-basierten Systemen.

Das Impact Acquire SDK (Software Development Kit) für C++ stellt Entwicklern ein umfassendes Toolkit zur Verfügung, welches die erforderlichen Bibliotheken, Werkzeuge und Application Programming Interfaces (APIs) beinhaltet, um maßgeschneiderte Softwarelösungen zu entwickeln. Dies erlaubt eine unmittelbare Interaktion mit der Hardware. Die Unterstützung der Programmierung in C++ ermöglicht eine effiziente Integration von Kamerafunktionen in komplexe Systeme. Es umfasst eine Vielzahl von Funktionen, die von einfachen Bildaufnahmen bis hin zu fortschrittlichen Bildverarbeitungs- und Analyseoperationen reichen.

Als Beispiel im Impact Acquire SDK ist die Datei „ContinuousCapture.cpp“ gegeben, welche den kontinuierlichen Aufnahmemodus demonstriert. Hierbei werden Bilder bis zum manuellen Stoppen des Prozesses erfasst. Jedoch erfolgt lediglich eine Ausgabe einer Information im Terminal, sobald ein Bild empfangen wird. Dieses Beispiel dient als Grundlage für die Implementierung des Testkonzepts. Des Weiteren umfasst das SDK eine ausführliche Dokumentation zu den Klassen und Methoden sowie eine Vielzahl an weiteren Beispielen, welche ebenfalls für die Entwicklung des Testprogramms verwendet werden.

(Balluff GmbH, 2024c) (Balluff GmbH, 2024d)

3.2 Methoden der Testkonfiguration

Zur Konfiguration von Tests stehen verschiedene Methoden zur Verfügung. Diese Methoden werden im Folgenden kurz dargestellt, um eine fundierte Entscheidung über die am besten geeignete Konfigurationsmethode treffen zu können.

Die erste Methode ist die direkte Konfiguration der Testparameter im Quellcode. Dieser Ansatz kann in C++ durch die Definition von Konstanten, statischen Variablen oder speziellen Konfigurationsklassen realisiert werden. Dies bietet eine genaue Kontrolle über die Testausführung, schränkt jedoch die Flexibilität ein, da Änderungen an der Konfiguration eine Neukompilierung des Codes erfordern.

Es gibt auch die Möglichkeit, externe Konfigurationsdateien, wie JSON- oder XML-Dateien, zu verwenden. Diese Methode bietet Flexibilität, da sie die Verwaltung von Testparametern ermöglicht, ohne dass der Code neu kompiliert werden muss. Es ist jedoch erforderlich, dass die Konfigurationsdateien korrekt in den Quellcode eingebunden und verarbeitet werden. Zusätzlich ist die Auswahl des Dateiformats entscheidend für die Umsetzung.

Eine weitere Möglichkeit ist die Verwendung von Terminaleingaben. Dies ermöglicht eine schnelle Anpassung der Testparameter, ohne den Quellcode oder externe Dateien ändern zu müssen. Ein Nachteil dieser Methode ist jedoch, dass bei einer großen Anzahl von Parametern die Handhabung umständlich wird.

Eine grafische Benutzeroberfläche (GUI) ermöglicht es auch technisch weniger versierten Benutzern die Testparameter einfach und intuitiv zu konfigurieren. Die GUI kann so gestaltet werden, dass sie eine visuelle Darstellung der verschiedenen Prüfparameter und Konfigurationsoptionen bietet. Die Entwicklung einer solchen Benutzeroberfläche erfordert jedoch zusätzliche Ressourcen und technisches Know-how in der GUI-Entwicklung. Zudem ist die Wartung und Aktualisierung aufwändig.

Die Entwicklung eines Testprogramms erfordert eine sorgfältige Planung der Testkonfiguration, um sicherzustellen, dass die Tests sowohl präzise als auch anpassungsfähig sind. Die Wahl der geeigneten Tools und Techniken hängt stark von den spezifischen Anforderungen des zu testenden Systems und der vorhandenen Infrastruktur ab.

(Witte, 2019) (Beneken et al., 2022)

3.3 Methoden des Loggings

Logging kann durch verschiedene Ansätze und Techniken realisiert werden. Dieses Kapitel gibt einen Überblick über die verschiedenen Logging-Ansätze, um die passende Lösung für die gegebenen Bedingungen zu finden.

Die Kategorisierung von Log-Nachrichten nach ihrer Priorität erfolgt durch verschiedene Log-Level. Typische Level sind „Debug“, „Info“, „Warn“, „Error“ und „Critical“. Durch die Einstellung eines bestimmten Log-Levels kann gesteuert werden, welche Nachrichten in den Log-Dateien erscheinen. Dies erhöht die Übersichtlichkeit und verbessert die Performance, indem irrelevante Informationen ausgeblendet werden.

Die Struktur einer Log-Nachricht ist entscheidend für ihren Nutzen. Eine Log-Nachricht enthält Elemente wie den Zeitstempel, das Log-Level, die betroffene Komponente oder Funktion und eine beschreibende Nachricht. Die Anpassung der Formatierung ermöglicht die Integration spezifischer Anforderungen wie beispielsweise Benutzer-IDs. Dies erleichtert die Nachverfolgung und Diagnose von Ereignissen und Fehlern.

Um die Systemleistung nicht durch übermäßig große Log-Dateien zu beeinträchtigen, werden Verfahren wie Log-Rotation und Archivierung verwendet. Log-Rotation ist eine Methode, bei der die Protokollierung zwischen mehreren Dateien wechselt, sobald eine festgelegte Zeitdauer oder Dateigröße erreicht wird. Es ist möglich, Parameter wie die maximale Dateigröße oder die Zeitdauer festzulegen, nach denen eine Rotation erfolgen soll, sowie die Anzahl der zu rotierenden Log-Dateien zu definieren. Durch die Anwendung von zeit- oder größenbasierter Rotation werden ältere Log-Dateien entweder archiviert oder gelöscht, wodurch der Speicher effizient verwaltet und die Systemwartung vereinfacht wird.

Es wird zwischen synchronem und asynchronem Logging unterschieden. Synchrones Logging stellt die Reihenfolge und Vollständigkeit der Log-Einträge sicher, kann aber die Anwendungsleistung beeinträchtigen, da es die Log-Nachrichten innerhalb der Hauptanwendung erstellt. Asynchrones Logging hingegen verbessert die Performance, indem Log-Nachrichten in eine Warteschlange gestellt werden, die unabhängig von der Hauptanwendung verarbeitet wird.

Monitoring und Alarmierung sind entscheidend für die proaktive Überwachung und schnelle Reaktion auf potenzielle Probleme. Automatische Benachrichtigungen bei kriti-

schen Log-Nachrichten unterstützen den Testingenieur dabei, den Testprozess effektiv zu überwachen.

Logs können auf Konsolen, in Dateien oder Datenbanken ausgegeben werden. Es ist dabei wichtig, sensible Daten zu schützen und Logs nur autorisierten Personen zugänglich zu machen. Dadurch wird sichergestellt, dass die Logging-Verfahren nicht nur effizient, sondern auch sicher sind und die Integrität und Vertraulichkeit der Systemdaten gewährleistet wird.

(Beneken et al., 2022) (Gu et al., 2023)

4 Konzeptentwicklung

4.1 Definition der Anforderungen

Es ist essentiell, die Anforderungen an das Testprogramm und die Testdurchführung präzise zu definieren. Zu diesem Zweck finden Gespräche mit Entwicklern und dem Testingenieur statt, um verschiedene Ansätze zu erörtern und die spezifischen Anforderungen festzulegen. Besonderes Augenmerk liegt dabei auf den Parametern, die für jeden Testdurchlauf flexibel gewählt werden. Dieses Kapitel behandelt sowohl die grundlegenden Anforderungen als auch die erforderlichen Konfigurationsmöglichkeiten und das Logging.

Die grundlegenden Anforderungen sind entscheidend für die Funktionalität und die Interaktion mit dem Benutzer. Dazu zählt vor allem die Bedienbarkeit des Testprogramms. Es muss gewährleistet sein, dass das Testprogramm einfach und intuitiv zu bedienen ist und dass die Parameter schnell und verständlich eingestellt werden können. Dafür ist auch eine umfassende Dokumentation dieser Parameter erforderlich. Ein weiterer kritischer Aspekt ist die Zuverlässigkeit des Testprogramms. Es muss sichergestellt werden, dass der Test reproduzierbar ist und dass das Testprogramm keine zusätzlichen Fehler verursacht. Eventuell auftretende Fehler müssen klar identifiziert und adressiert werden. Um Fehler während der Testdurchführung zu minimieren, ist eine umfassende Validierung des Testprogramms notwendig, einschließlich der Implementierung von Modultests.

Darüber hinaus muss der Quellcode des Testprogramms nachvollziehbar gestaltet sein, um den Anwendern ein tiefgehendes Verständnis der Funktionsweise zu ermöglichen und Anpassungen an veränderte Anforderungen zu erleichtern. Die Kompatibilität des Testprogramms mit der Testumgebung ist ebenfalls von großer Bedeutung, ebenso wie die Möglichkeit, das Testprogramm zu erweitern oder in übergeordnete Softwarelösungen zu integrieren.

In Bezug auf die Konfiguration des Testprogramms soll der Benutzer die Möglichkeit haben, die Speicherung der vom RadarImager gesendeten Bilder zu steuern. Der Benutzer soll ein Intervall zur Speicherung der Bilder festlegen können, sodass beispielsweise nur jedes x-te Bild gespeichert wird. Zudem soll sich einrichten lassen, nach wie vielen Einzelbildern ein neuer Ordner erstellt wird. Jeder Bilderstapel wird in einem eigenen Ordner gespeichert,

um eine klare Trennung zu gewährleisten. Die Benennung dieser Ordner und der einzelnen Bilder soll ebenfalls konfigurierbar sein, wobei der Benutzer zwischen Nummerierung und Zeitstempeln wählen kann.

Das Logging soll ein- und ausschaltbar sein. Zudem sollen verschiedene Log-Level zur Verfügung stehen, die der Benutzer auswählen kann. Sollte ein weiterführendes Logging-Konzept wie Log-Rotation implementiert werden, müssen auch hierfür spezifische Parameter einstellbar sein. Das Ziel des Loggings ist es, dem Testingenieur umfassende Informationen über den Testdurchlauf zu liefern, während gleichzeitig die Übersichtlichkeit der Log-Dateien gewahrt bleibt.

Als Ausgabe soll das Testprogramm einen spezifisch benannten Ausgabeordner erstellen, der in Unterverzeichnissen die Bilder sowie die Log-Dateien des Testdurchlaufs enthält. Der Speicherort dieses Ordners soll konfigurierbar sein.

Zur Realisierung dieser Anforderungen ist es sinnvoll, im nächsten Schritt detaillierte Konzepte für die Konfiguration und das Logging zu entwickeln, wobei im agilen Umfeld eine gewisse Flexibilität in der Umsetzung berücksichtigt werden muss.

4.2 Konzeptionelle Gestaltung der Konfiguration

Die Konfiguration der Tests wird mithilfe einer Konfigurationsdatei vorgenommen. Dies bietet den Vorteil der Flexibilität sowie der Trennung von Quellcode und Konfiguration. Darüber hinaus kann die Konfigurationsdatei wiederverwendet werden, wobei bei Bedarf nur einzelne Parameter angepasst werden müssen. Zudem ist die Konfiguration stets nachvollziehbar, wodurch die Datei bereits eine Art Dokumentation des Testdurchlaufs ist. Ein weiterer Vorteil besteht darin, dass eine Erweiterung, beispielsweise durch eine GUI, möglich ist. Es ist ratsam, dies klar zu trennen, sodass die GUI die Konfigurationsdatei beschreibt, um die Lösung ohne GUI selbst funktionsfähig zu halten.

Auf die ausschließliche Nutzung einer GUI wird verzichtet, da diese für notwendige kurzfristige Anpassungen zu unflexibel ist. Zudem bleibt damit die Möglichkeit bestehen, das Testprogramm in ein Softwaretool zur weiteren Automatisierung zu integrieren. Terminal-eingaben werden nicht genutzt, da sie für die wichtige Nachverfolgbarkeit und Konsistenz ungeeignet sind und keine einfache Handhabung bieten. Auch die Konfiguration der Testparameter direkt im Quellcode wird vermieden, da eine Neukompilierung für jeden Testdurchlauf nicht praktikabel ist. Dennoch werden innerhalb des Quellcodes bestimmte

Einstellungen vorgenommen oder Konstanten definiert, die für das Testprogramm relevant sind. Diese betreffen jedoch allgemeine Aspekte, die nicht direkt mit spezifischen Testdurchläufen verbunden sind, wie beispielsweise die Festlegung von Grenzwerten zur Überprüfung der Plausibilität bestimmter Parameter.

Als Dateiformat für die Konfigurationsdatei wird JSON verwendet, da dies sowohl für den Benutzer als auch in C++ gut lesbar und verständlich ist. Zudem ermöglicht JSON eine strukturierte Darstellung der einzelnen Parameter. Dies verschafft dem Benutzer einen besseren Überblick und vereinfacht das Eintragen der Parameter. Innerhalb des Projekts existiert bereits eine Konfigurationsdatei, die allgemeine Parameter des Tests sowie Einstellungen für den simulierten Trigger enthält. Diese wird nach entsprechender Anpassung weiterverwendet. In Listing 4.1 ist die angepasste Konfigurationsdatei dargestellt:

```
1 {
2   "Basic": {
3     "testID": "TestID",
4     "description": "Test Description",
5     "genICamClient": true,
6     "triggerSW": false,
7     "logStatus": true,
8     "checkImages": true,
9     "saveLogsRI": true,
10    "createAzureIssue": false,
11    "testDuration": 3
12  },
13  "GenICam-Client": {
14    "saveImages": true,
15    "numberImages": true,
16    "numberFolders": true,
17    "saveImagesInterval": 1,
18    "newFolderInterval": 10,
19    "createLog": true,
20    "logLevel": 3,
21    "maxLogFiles": 3,
22    "maxLogSize": 5,
23    "createIssue": true,
24    "outputPath": "../"
25  },
26  "objectParameter": {
27    "objectSpeedX": 50,
28    "offsetDistanceX": 0,
29    "sensorDistanceX": 10,
30    "objectLengthX": 50,
31    "offsetBetweenObj": 50,
32    "maxRandomOffsetBetweenObj": 0
33  }
34 }
```

Listing 4.1: Konfigurationsdatei

Um die verschiedenen Bereiche des Tests klar zu trennen, werden in der Konfigurationsdatei unterschiedliche Abschnitte verwendet, in denen jeweils die Parameter festgelegt werden. Somit ist die Konfigurationsdatei in drei Teile gegliedert. Der Abschnitt „Basic“ enthält grundlegende Informationen über den Testdurchlauf, wie eine eindeutige Test-ID, eine Beschreibung und die Dauer des Testdurchlaufs. Im Abschnitt „GenICam-Client“ sind Parameter für die Speicherung der Bilder und das Erstellen der Log-Dateien hinterlegt. Im letzten Teil „objectParameter“ sind Parameter enthalten, die sich auf das zu testende Objekt bzw. den simulierten Trigger beziehen.

Um Missverständnisse zu vermeiden und die korrekte Konfiguration der Testdurchläufe zu gewährleisten, existiert zu der Konfigurationsdatei eine Dokumentation. Diese beschreibt den Aufbau der Datei, erklärt die einzelnen Parameter und gibt deren Einheit sowie mögliche Werte an.

Damit die Parameter aus der Konfigurationsdatei korrekt ausgelesen und weiterverwendet werden, wird in C++ die Bibliothek „nlohmann/json“ verwendet. Diese ermöglicht die Überprüfung der Datentypen der einzelnen Eingaben und die Speicherung der Parameter. Die Bibliothek wird verwendet, da sie benutzerfreundlich ist, konform mit allen JSON-Datentypen arbeitet und zudem eine umfassende Dokumentation besitzt.

4.3 Konzeptionelle Gestaltung des Loggings

Als Ausgabeformat des Loggings werden Log-Dateien verwendet, um sicherzustellen, dass keine wichtigen Informationen verloren gehen. Log-Dateien bieten zudem die einfache Möglichkeit, Methoden wie die Log-Rotation zu implementieren. Die Erstellung einer Datenbank für das Logging des Testprogramms wäre unverhältnismäßig aufwendig und daher nicht praktikabel.

Bezüglich der Sicherheit und des Datenschutzes sind keine besonderen Maßnahmen erforderlich, da die Log-Dateien nur lokal gespeichert werden, keine personenbezogenen Daten enthalten und nur autorisierte Benutzer Zugriff auf die Log-Dateien haben.

Um die Log-Nachrichten sinnvoll zu kategorisieren, werden verschiedene Log-Level eingeführt. Es wird jedoch auf drei verschiedene Log-Level begrenzt, da dies eine ausreichende Abgrenzung der Nachrichten des Testprogramms ermöglicht und die Komplexität für den Benutzer reduziert. Die Log-Level sind wie folgt definiert:

- **Error** (Fehler): Das höchste Log-Level weist auf Fehlerzustände oder Probleme hin, die die ordnungsgemäße Funktion beeinträchtigen. Im Testprogramm kann dies beispielsweise der Fall sein, wenn Bilder oder Bildstapel während der Kommunikation verloren gehen oder die Kommunikation abbricht.
- **Warning** (Warnung): Dieses Level zeigt an, dass etwas Unerwartetes passiert ist, das einen der Prozesse stören kann, aber nicht unbedingt zu einem Funktionsausfall führt. Ein Beispiel hierfür ist eine unerwartete Veränderung der Anzahl der pro Sekunde empfangenen Bilder.
- **Info** (Information): Auf diesem Level werden Informationen des normalen Betriebs dokumentiert, wie das Empfangen der einzelnen Bilder bzw. Bildstapel sowie die Darstellung von Statistiken.

Die Formatierung der Log-Nachrichten basiert auf folgendem Muster:

[Zeitstempel] [Log-Level] [Nachricht/Beschreibung]

Damit wird sichergestellt, dass der Zeitpunkt und die Reihenfolge der Log-Nachrichten nachvollziehbar sind. Zudem ist das entsprechende Log-Level direkt sichtbar und eine eindeutige Beschreibung der Nachricht vorhanden. Abbildung 4.1 zeigt beispielhaft den Aufbau der Log-Dateien:

```
[2024-07-19 11:40:40.799] [info] Multipart buffer captured: 1 part
[2024-07-19 11:40:40.799] [info] Image 207 part 0: 480x480
[2024-07-19 11:40:40.895] [info] Saved image 207 to .\\Output_TestID_2024-07-19_11h-39m-08s/Images_TestID_2024-07-19_11h-39m-08s/21/image_207.png
[2024-07-19 11:40:40.895] [info] Image: 207      Timestamp_us: 1721382050394      Timestamp: 1721382040895      Difference: -9499

[2024-07-19 11:40:41.055] [info] Multipart buffer captured: 1 part
[2024-07-19 11:40:41.056] [info] Image 208 part 0: 480x480
[2024-07-19 11:40:41.149] [info] Saved image 208 to .\\Output_TestID_2024-07-19_11h-39m-08s/Images_TestID_2024-07-19_11h-39m-08s/21/image_208.png
[2024-07-19 11:40:41.149] [info] Image: 208      Timestamp_us: 1721382050649      Timestamp: 1721382041149      Difference: -9500

[2024-07-19 11:40:41.311] [info] Multipart buffer captured: 1 part
[2024-07-19 11:40:41.311] [info] Image 209 part 0: 480x480
[2024-07-19 11:40:41.435] [info] Saved image 209 to .\\Output_TestID_2024-07-19_11h-39m-08s/Images_TestID_2024-07-19_11h-39m-08s/21/image_209.png
[2024-07-19 11:40:41.435] [info] Image: 209      Timestamp_us: 1721382050905      Timestamp: 1721382041435      Difference: -9470

[2024-07-19 11:40:41.566] [info] Multipart buffer captured: 1 part
[2024-07-19 11:40:41.567] [info] Image 210 part 0: 480x480
[2024-07-19 11:40:41.670] [info] Saved image 210 to .\\Output_TestID_2024-07-19_11h-39m-08s/Images_TestID_2024-07-19_11h-39m-08s/21/image_210.png
[2024-07-19 11:40:41.670] [info] Image: 210      Timestamp_us: 1721382051161      Timestamp: 1721382041670      Difference: -9491

[2024-07-19 11:43:42.452] [error] Error: Maximum time between Images exceeded

[2024-07-19 11:43:42.452] [info] Info from GVS-M-4:
[2024-07-19 11:43:42.452] [info] AbortedRequestsCount: 0
[2024-07-19 11:43:42.452] [info] BandwidthConsumed: 2820.131
[2024-07-19 11:43:42.453] [info] CaptureTime_s: 2.465440
[2024-07-19 11:43:42.453] [info] ErrorCount: 0
[2024-07-19 11:43:42.453] [info] FormatConvertTime_s: 0.000001
[2024-07-19 11:43:42.453] [info] FrameCount: 210
[2024-07-19 11:43:42.453] [info] FramesIncompleteCount: 0
[2024-07-19 11:43:42.453] [info] FramesPerSecond: 3.909009
[2024-07-19 11:43:42.453] [info] ImageProcTime_s: 0.000021
[2024-07-19 11:43:42.453] [info] LostImagesCount: 0
[2024-07-19 11:43:42.453] [info] MissingDataAverage_pc: 0.000000
[2024-07-19 11:43:42.453] [info] QueueTime_s: 0.000020
[2024-07-19 11:43:42.453] [info] RetransmitCount: 0
[2024-07-19 11:43:42.453] [info] TimedOutRequestsCount: 0

[2024-07-19 11:43:42.453] [info] The Statistics are within acceptable limits

[2024-07-19 11:43:42.812] [info] AzureIssue created: Title: Test: TestID failed!, Description: Test failed / Time: 2024-07-19_11h-43m-42s / Images: 210.
```

Abbildung 4.1: Beispiel Log-Datei (eigene Darstellung)

Um die Performance des Systems nicht zu beeinträchtigen und das Handling der Log-Dateien zu vereinfachen, wird eine Log-Rotation implementiert. Hierbei werden Parameter für die maximale Größe einer Log-Datei und die maximale Anzahl an Log-Dateien festgelegt. Der Benutzer kann damit individuell einstellen, wie weit die Informationen der Log-Dateien zurückgehen sollen. Eine Archivierung älterer Log-Dateien wird nicht umgesetzt, da der Benutzer bei Bedarf eine entsprechend hohe Anzahl an maximalen Log-Dateien einstellen kann.

Es wird synchrones Logging eingesetzt, da auf die korrekte Reihenfolge und die Vollständigkeit der Logs großer Wert gelegt wird. Zudem ist die Performance des Testprogramms nicht in einem solchen Grad entscheidend, dass eine andere Lösung notwendig ist. Durch synchrones Logging bleibt der Quellcode des Testprogramms einfach lesbar und es ist nachvollziehbar, an welcher Stelle die jeweilige Log-Nachricht entsteht.

Bei bestimmten Fehlern oder dem erzwungenen Abbruch des Testprogramms wird ein „Azure Issue“ erstellt, um den Benutzer über das Ereignis zu informieren. Dies ist notwendig, da der Benutzer nicht ständig am Testaufbau anwesend ist und Langzeittests auch außerhalb der Arbeitszeit laufen. Diese Funktion arbeitet unabhängig von der Erstellung der Log-Dateien, um verzichtbare Abhängigkeiten zu vermeiden.

Um eine klare Trennung der Informationen des Testdurchlaufs von denen des Testprogramms sicherzustellen, werden die Informationen und gegebenenfalls Fehlermeldungen des Testprogramms im Terminal ausgegeben. Dies ist sinnvoll, da das Testprogramm direkt nach dem Start die meisten Funktionen durchläuft und anschließend hauptsächlich sich wiederholende Abschnitte ausführt. Der Benutzer ist beim Start des Tests in der Regel am Aufbau anwesend und kann daher falls notwendig eingreifen. Für den Fall, dass der Benutzer nicht anwesend ist und um die Informationen zusätzlich konsistent zu speichern, wird neben den Log-Dateien für die Datenverarbeitung eine zusätzliche Log-Datei speziell für die Überwachung der Funktionen des Testprogramms erstellt. Diese Log-Datei verwendet dieselbe Formatierung wie die anderen Log-Dateien, ist jedoch eine einfache, nicht konfigurierbare Datei mit einem festgelegten Log-Level. Dies ist ausreichend, da das Testprogramm nur einen abschätzbaren, begrenzten Umfang an Log-Nachrichten liefert. Außerdem wird dadurch vermieden, dass es bei der Konfiguration des Tests durch zu viele Parameter zu Verwirrungen kommt.

Für die Implementierung des Loggings in C++ wird die Bibliothek „spdlog“ verwendet. Diese ermöglicht eine einfache Konfiguration inklusive des Log-Levels, der Formatierung

der Log-Nachrichten und der Log-Rotation. Die manuelle Implementierung dieser Eigenschaften über standardmäßige Dateioperationen ist sehr aufwendig und schränkt die Flexibilität bei der Konfiguration der Features (z.B. der Log-Rotation) ein. Die Bibliothek „spdlog“ wird verwendet, da sie eine umfassende Dokumentation bietet, benutzerfreundlich ist und viele Konfigurationsmöglichkeiten erlaubt.

5 Implementierung des Testkonzepts in C++

5.1 Aufbau und Organisation des Projektordners

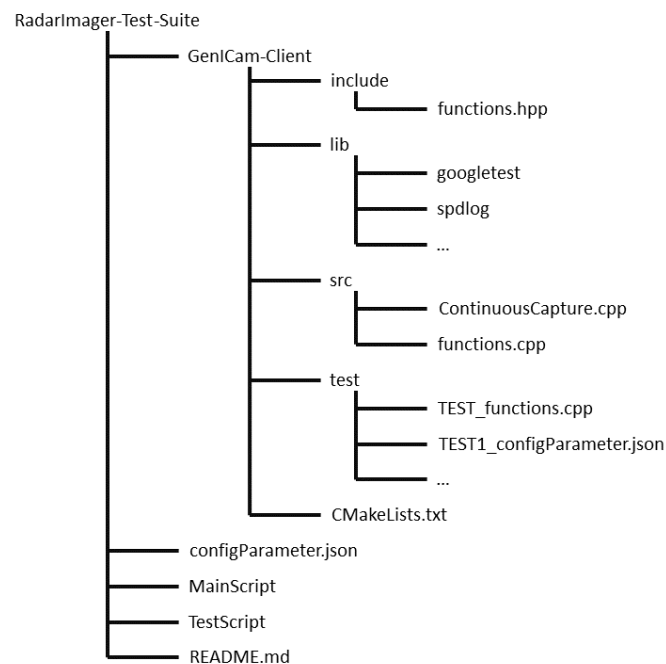


Abbildung 5.1: Verzeichnisstruktur (eigene Darstellung)

Die Struktur des Testprogramms für den RadarImager ist durch mehrere Unterverzeichnisse klar und logisch aufgebaut. Im Rahmen dieser Projektarbeit liegt der Fokus auf dem Verzeichnis „GenICam-Client“. Die Konfigurationsdatei „configParameter.json“ befindet sich im Hauptordner des Projekts, da sie codeübergreifend verwendet wird. Die Ordner „MainScript“ und „TestScript“, die unter anderem Python-Skripte zur Steuerung des Hardware-Triggers enthalten, sind für diese Arbeit von untergeordneter Bedeutung.

Die Codebasis des GenICam-Clients ist zur besseren Übersicht und Handhabbarkeit in folgende Unterverzeichnisse aufgeteilt:

- **include:** Dieser Ordner enthält die Header-Datei „functions.hpp“, welche die Schnittstellen für die in „src“ implementierten Klassen und Funktionen deklariert. Zudem sind hier verschiedene Strukturen (structs) definiert.
- **lib:** Hier sind die im Projekt verwendeten Bibliotheken, wie zum Beispiel „spdlog“, abgelegt. Diese Strukturierung unterstützt eine klare Abgrenzung zwischen eigenem Code und externen Bibliotheken.
- **src:** Dieser Ordner enthält den Quellcode des Projekts, einschließlich der Hilfsfunktionen und dem Hauptprogramm zur Verarbeitung der empfangenen Daten.
- **test:** In diesem Verzeichnis befinden sich die Modultests, die zur Überprüfung der Funktionalität des Codes dienen. Es umfasst auch zusätzliche Dateien, die in den Tests verwendet werden.

Zur Automatisierung und effizienten Verwaltung des Build-Prozesses wird CMake eingesetzt. CMake erleichtert die Integration und Verwaltung von Abhängigkeiten sowie die Konfiguration des Testframeworks, um eine konsistente und fehlerfreie Build-Umgebung zu gewährleisten. Hierzu wird eine CMakeLists.txt-Datei angelegt. Diese Datei enthält spezifische Anweisungen, die es CMake ermöglichen, die notwendigen Bibliotheken korrekt zu lokalisieren, einzubinden und mit dem Hauptprojekt zu verknüpfen. Dadurch wird sichergestellt, dass alle Komponenten des Projekts nahtlos zusammenarbeiten und die Build-Prozesse reibungslos ablaufen können.

Im Anhang dieser Projektarbeit finden sich die Quellcodedateien „ContinuousCapture.cpp“, „functions.hpp“ und „functions.cpp“.

5.2 Verarbeitung der Konfigurationsdatei

Um die Parameter aus der Konfigurationsdatei effektiv im Code zu nutzen, empfiehlt es sich, diese zunächst in Variablen zu speichern. Hierfür wird eine Struktur angelegt, die sowohl die vom Benutzer festgelegten Werte als auch daraus resultierende Werte enthält. Bevor die Parameter aus der JSON-Datei in die Variablen übernommen werden, ist eine Überprüfung der Datentypen notwendig. Dies verhindert unerwünschte implizite Datentypkonvertierungen und Fehler. Die Verarbeitung der JSON-Datei erfolgt mithilfe der „nlohmann/json“-Bibliothek, die eine robuste und effiziente Handhabung von JSON-Daten ermöglicht.

```
1 ifstream paramFile(configFilename);
2 nlohmann::json root;
3 paramFile >> root;
```

Listing 5.1: Parsen der JSON-Datei

Zunächst wird ein Input-File-Stream-Objekt erstellt, um die Konfigurationsdatei zu öffnen. Anschließend wird eine Variable „root“ definiert, die als Container für die JSON-Daten dient. Der Inhalt von „paramFile“ wird in die Variable „root“ geparsed. Dadurch wird der Inhalt der JSON-Datei in eine strukturierte Form umgewandelt, die programmatisch einfacher zu verarbeiten ist. Dann kann der Datentyp eines Parameters überprüft werden:

```
1 root["Basic"]["testID"].is_string()
```

Listing 5.2: Überprüfung des Datentyps

Mithilfe der eckigen Klammern wird durch die Struktur der JSON-Datei navigiert. Die Methode „is_string()“ überprüft, ob der Wert an der angegebenen Stelle vom Typ „string“ ist. Falls einer der Datentypen inkorrekt ist, wird eine Fehlermeldung ausgegeben und das Testprogramm abgebrochen. Sind alle Datentypen korrekt, werden die Parameter in die vorgesehene Struktur übertragen. Hierbei ist „param“ eine Instanz der Struktur für die Parameter (siehe Anhang: Seite IX ab Zeile 32).

```
1 param.testID = root["Basic"]["testID"];
```

Listing 5.3: Übernahme der Parameter

Anschließend erfolgt eine Plausibilitätsprüfung der Parameter. Dabei wird beispielsweise überprüft, dass eingegebene Strings nicht leer sind oder Pfade nur zulässige Zeichen enthalten. Bei numerischen Werten wird kontrolliert, ob diese im zugelassenen Bereich liegen, wie etwa beim Log-Level, das nur vier Werte (0 bis 3) zulässt.

Nach erfolgreicher Überprüfung werden der Ausgabeordner und die zugehörigen Unterzeichnisse erstellt, die zur besseren Nachverfolgbarkeit einen Zeitstempel im Namen enthalten. Außerdem wird dadurch vermieden, dass einer der Ordner unabsichtlich überschrieben wird. Die Klasse „TimestampProvider“ generiert diesen Zeitstempel, wobei zwischen sekunden- und millisekundengenauer Genauigkeit gewählt werden kann (siehe Anhang: Seite X ab Zeile 66 und XII ab Zeile 5). Bei Problemen während des Erstellens der

Ordner wird eine Fehlermeldung ausgegeben und in die Log-Datei für das Testprogramm geschrieben.

Die Methoden und Konstanten für die Parameterbereiche sind in der Klasse „ConfigurationHandler“ abgelegt (siehe Anhang: Seite X ab Zeile 79). Dies fördert die Kapselung von Daten und bietet eine klare Schnittstelle durch eine Hauptmethode, die interne Methoden aufruft und so die Funktionalität zentral verwaltet.

5.3 Umsetzung der Konfigurationsmöglichkeiten

Die Implementierung der verschiedenen Konfigurationsmöglichkeiten für die Speicherung von Bildern oder Bildstapeln wird in der Callback-Funktion „processRequest“ sowie in den zugehörigen Hilfsfunktionen der Klasse „ImageSavePreparer“ (siehe Anhang: Seite V ab Zeile 6, X ab Zeile 106 und XVI ab Zeile 246) realisiert. Bei Empfang eines Bildes oder Bildstapels prüft die Funktion zunächst, ob eine Speicherung gemäß der aktuellen Konfigurationseinstellungen erforderlich ist.

Anschließend wird unterschieden, ob es sich um ein einzelnes Bild oder um einen Bildstapel handelt. Diese Unterscheidung basiert auf der Anzahl der Bilder (Parts), die der empfangene Container enthält. Der Speichervorgang variiert je nachdem, ob ein einzelnes Bild oder ein Bildstapel vorliegt. Listing 5.4 zeigt diese Differenzierung und den Ablauf des Speichervorgangs:

```
1 // Save images depending on the configuration
2 if (param.saveImages && (param.saveImagesInterval == 1 || pRequest->infoFrameID.read() % param.
   ↪ saveImagesInterval == 1))
3 {
4     if (bufferPartCount == 1) // Buffer has exactly one image
5     {
6         int imageNumber = ceil((float)pRequest->infoFrameID.read()/(float)param.saveImagesInterval);
7         string filename = save.prepareImageSave(param, imageNumber);
8         pRequest->getBufferPart(0).getImageBufferDesc().save(filename);
9         loggerClient->info("Saved image {} to {}", pRequest->infoFrameID.read(), filename);
10    }
11    else // Buffer has more than one image
12    {
13        string filename = save.prepareBufferSave(param, pRequest->infoFrameID.read(), i);
14        pRequest->getBufferPart(i).getImageBufferDesc().save(filename);
15        loggerClient->info("Saved image {} part {} to {}", pRequest->infoFrameID.read(), i, filename);
16    }
17 }
```

Listing 5.4: Speichervorgang

Einzelbild Um die Nummer des aktuell zu speichernden Bildes zu bestimmen, wird die Identifikationsnummer des Bildes durch das festgelegte Speicherintervall geteilt. Diese Nummer wird für die Vorbereitung des Speichervorgangs verwendet. In diesem Schritt wird der Pfad zum Speicherort generiert und bei Bedarf ein neuer Ordner angelegt. Hierbei wird zwischen der Nummerierung der Ordner und der Benennung mittels eines Zeitstempels unterschieden.

Bei Nummerierung setzt sich der Pfad aus dem Hauptverzeichnis für Bilder und einem Unterverzeichnis, das die aktuelle Ordner-Nummer trägt, zusammen. Erreicht das Speicherintervall den Punkt, an dem ein neuer Ordner benötigt wird, wird dieser mit der entsprechenden Nummer erstellt. Im Falle der Benennung durch einen Zeitstempel erhalten die Unterverzeichnisse Namen, die den Zeitstempel bis auf Millisekunden genau wiedergeben. Die Benennung auf die Millisekunde genau ist notwendig, um das unabsichtliche Überschreiben der Verzeichnisse zu verhindern, da je nach Konfiguration mehrere Verzeichnisse pro Sekunde erstellt werden. Es ist dabei zusätzlich erforderlich, den Namen des aktuellen oder vorherigen Ordners zu speichern, da der Zeitstempel nicht einfach rekonstruiert werden kann für die nächsten Bilder, die im selben Ordner abgelegt werden sollen.

Nachdem Erstellen des Pfads und gegebenenfalls dem Anlegen eines neuen Ordners, übergibt die Untermethode „createImageSubpathFolder“ den Pfad an die Methode „prepareImageSave“ (siehe Anhang: Seite XVI ab Zeile 246). Diese ist verantwortlich für die Erstellung des Bildnamens und somit für die Festlegung des endgültigen Speicherorts des Bildes. Auch hier wird zwischen Nummerierung und Zeitstempel differenziert.

Schließlich wird das Bild am festgelegten Speicherort durch den Aufruf der „save“-Methode des Impact Acquire SDK gespeichert.

Bildstapel Bei der Verarbeitung eines Bildstapels erfolgt die Vorbereitung ähnlich wie bei Einzelbildern, allerdings wird für jeden Container ein neuer Ordner angelegt. Innerhalb einer for-Schleife, in der auch der Code aus Listing 5.4 implementiert ist, wird die Hilfsvariable „i“ inkrementiert, die das aktuelle Bild im Stapel kennzeichnet. Ein neuer Ordner wird immer dann erstellt, wenn „i“ den Wert 0 annimmt, der das erste Bild eines neuen Bildstapels signalisiert.

Die Methode „prepareBufferSave“ ist anschließend dafür zuständig, den Namen der Bilddatei zu bestimmen (siehe Anhang: Seite XVIII ab Zeile 335). Abhängig von der

Konfiguration wird hierbei entweder die Nummer des Bildes innerhalb des Stapels („i“) oder ein Zeitstempel verwendet. In jedem Fall wird die Nummer des Containers in die Benennung der Datei integriert.

Das aktuelle Bild wird dann am vorbereiteten Speicherort abgelegt. Dieser Prozess wiederholt sich durch die for-Schleife für jedes Bild des Stapels.

5.4 Umsetzung des Loggings

Um einen Logger mithilfe der Bibliothek „spdlog“ zu implementieren, ist zunächst eine Initialisierung erforderlich. Diese erfolgt über eine Methode, die abhängig von den übergebenen Parametern eine Log-Rotation mit definierter maximaler Dateigröße und Anzahl an Dateien erstellt (siehe Anhang: Seite XIX ab Zeile 395). Zusätzlich wird das Log-Level durch eine Hilfsfunktion festgelegt (siehe Anhang: Seite XVIII ab Zeile 356).

Nach der Initialisierung kann der Logger in Funktionen und Methoden eingebunden werden. Dies geschieht durch folgenden Aufruf:

```
1 auto loggerClient = spdlog::get("logClient");
```

Listing 5.5: Aufruf des Loggers

Innerhalb der Callback-Funktion „processRequest“ werden relevante Informationen zur Datenübertragung protokolliert. Es wird vermerkt, wenn ein Bildstapel empfangen wird, einschließlich der Anzahl der enthaltenen Bilder und der Größe jedes einzelnen Bildes. Auch nach erfolgreichem Speichern eines Bildes wird eine entsprechende Log-Nachricht erstellt. Zudem werden die Zeiten für die Erstellung und Speicherung der Bilder sowie deren Differenz dokumentiert. Treten Fehler auf, so werden diese ebenfalls sorgfältig festgehalten. Ein typisches Beispiel für einen solchen Fehler ist der Empfang eines Bildes außerhalb des vorgesehenen Containers. Dies deutet auf eine Fehlfunktion des RadarImagers hin, da unter regulären Bedingungen jedes Bild innerhalb eines Containers übermittelt wird.

Regelmäßig werden wichtige Statistiken wie Frames per Second (FPS), Frame Count und die Anzahl verlorener Bilder in die Log-Dateien geschrieben. Diese Daten werden über das Impact Acquire SDK bezogen und auf Auffälligkeiten hin überprüft. Bei signifikanten Änderungen der FPS oder einer Zunahme verlorener Bilder im Vergleich zur letzten Überprüfung wird eine Warnung ausgegeben, um im Nachhinein mögliche Probleme

identifizieren zu können. Dies ist besonders relevant, wenn ein Fehler zum Abbruch des Tests führt, beispielsweise bei einer fehlerhaften Anfrage.

Zum Monitoring wird die „cURL“-Bibliothek genutzt, um über eine HTTP-Anfrage ein „Azure Issue“ zu erstellen. Die Methode „createAzureIssue“ in der Klasse „AzureIssueCreator“ sendet eine HTTP POST-Anfrage an die Azure DevOps REST-API, um ein „Azure Issue“ mit spezifischen Feldern wie Titel, Beschreibung und zugewiesenem Benutzer zu erstellen (siehe Anhang: Seite XI ab Zeile 146 und XXI ab Zeile 502). Die Authentifizierung erfolgt über ein in „Base64“ kodiertes Personal Access Token (PAT). Die Serverantwort wird protokolliert und relevante Informationen werden in die Log-Dateien geschrieben.

Um das Testprogramm automatisch zu beenden, falls über einen längeren Zeitraum keine Bilder empfangen werden, überwacht eine while-Schleife im Hauptprogramm die Zeit seit der letzten Anfrage. Diese Schleife endet entweder durch Überschreitung dieser Zeitgrenze oder durch das Drücken einer Taste (siehe Anhang: Seite VIII ab Zeile 155). Eine Hilfsfunktion ermöglicht dabei die parallele Abfrage der Tastatureingabe.

Zusätzlich zum Logging der Datenübertragung und Speicherung wird die separate Log-Datei zur Überwachung des korrekten Ablaufs des Testprogramms geführt. Diese Log-Datei wird unabhängig von den Hilfsfunktionen für das Logging verwaltet, um auch dort mögliche Fehlerquellen zu identifizieren.

5.5 Entwicklung und Durchführung der Modultests

Um die Funktionalität und Zuverlässigkeit des Testprogramms zu gewährleisten, werden umfassende Modultests mit dem Testframework „googletest“ durchgeführt. Für jede Klasse des Produktivcodes wird eine entsprechende Testklasse erstellt. Diese Testklassen erben von „::testing::Test“ und ermöglichen die Definition von Setup- und Teardown-Methoden, die jeweils vor und nach den Testfällen ausgeführt werden. Diese Methoden ermöglichen die Initialisierung und Bereinigung der Testumgebung. Somit wird eine konsistente Ausführung der Tests sichergestellt.

Ein spezielles Beispiel der Modultests ist der Test der Klasse „ConfigurationHandler“. In diesem Test werden verschiedene Muster-Konfigurationsdateien verwendet, um unterschiedliche Szenarien zu simulieren. Diese Dateien umfassen sowohl korrekte Konfigurationen als auch solche mit inkorrekten Datentypen oder Werten, die außerhalb des zulässigen Bereichs liegen. Listing 5.6 zeigt exemplarisch jeweils einen Modultest für diese Fälle:

```

1 TEST_F(ConfigurationHandlerTest, ValidConfigA)
2 {
3     config.configFilename = "../test/TEST1_configParameter.json";
4     EXPECT_TRUE(config.loadConfiguration(param));
5
6     EXPECT_EQ(param.saveImages, true);
7     EXPECT_EQ(param.numberImages, true);
8     EXPECT_EQ(param.numberFolders, true);
9     EXPECT_EQ(param.saveImagesInterval, 1);
10    EXPECT_EQ(param.newFolderInterval, 10);
11    EXPECT_EQ(param.createLogparam.createLog, true);
12    EXPECT_EQ(param.logLevel, 1);
13    EXPECT_EQ(param.maxLogFiles, 3);
14    EXPECT_EQ(param.maxLogSize, 5);
15    EXPECT_EQ(param.createIssue, true);
16    EXPECT_EQ(param.testID, "TestID");
17
18    EXPECT_TRUE(filesystem::exists(param.outputPath));
19    EXPECT_TRUE(filesystem::exists(param.imagesPath));
20    EXPECT_TRUE(filesystem::exists(param.logsPath));
21
22    filesystem::remove_all(param.outputPath);
23 }
24
25 TEST_F(ConfigurationHandlerTest, InvalidDatatypeBool)
26 {
27     config.configFilename = "../test/TEST3_configParameter.json";
28     EXPECT_FALSE(config.loadConfiguration(param));
29 }

```

Listing 5.6: Beispiel Modultests

Mithilfe der gültigen Muster-Konfigurationsdateien wird die korrekte Übernahme der in den Konfigurationsdateien festgelegten Werte in die entsprechenden Variablen überprüft. Ungültige Konfigurationsdateien dienen dazu, die Fähigkeit des Testprogramms zu validieren, Fehler zu erkennen und zu melden, um einen fehlerhaften Betrieb zu verhindern.

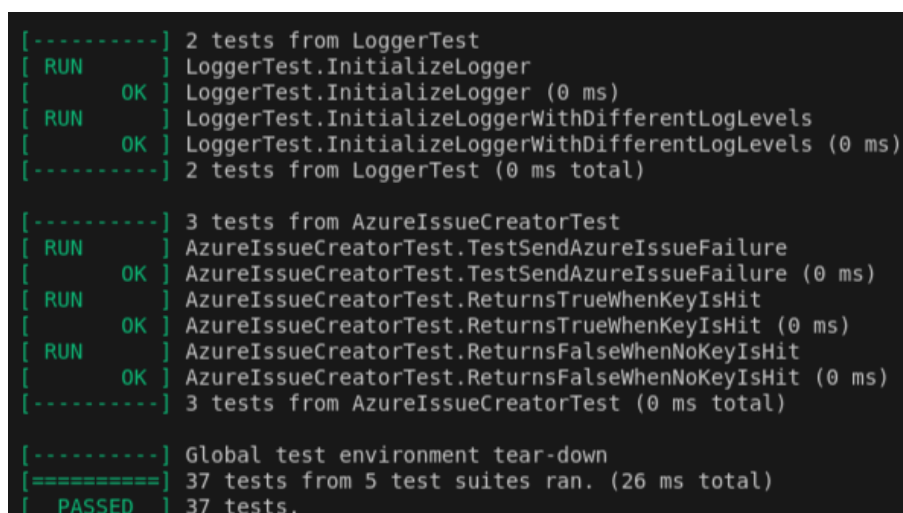
Ein Aspekt der Testimplementierung ist das Mocking externer Abhängigkeiten, um eine Isolation der Tests zu gewährleisten. Durch das Ersetzen realer Abhängigkeiten mit Mock-Objekten können Tests in einer kontrollierten Umgebung durchgeführt werden. Die Mock-Objekte simulieren dabei ein für den jeweiligen Modultest festgelegtes Verhalten der Abhängigkeiten. Dies ist insbesondere für das Testen von Komponenten, die mit externen Systemen interagieren, von großer Bedeutung.

Für eine zuverlässige Überprüfung der Methode „checkKeyboardHit“ (siehe Anhang: Seite XXIII ab Zeile 618) ist es notwendig Systemaufrufe zu simulieren. Hierfür werden mittels Mocking verschiedene Systemaufrufe angepasst. Dies ermöglicht es, das Verhalten der

Tastatureingabe ohne tatsächliche Benutzerinteraktion zu testen. Dafür sind Funktionen notwendig, die Standardbibliothekfunktionen durch die gemockten Funktionen ersetzen.

Durch die Abdeckung jeder Klasse des Testprogramms mit Modultests wird eine hohe Sicherheit des Testablaufs gewährleistet. Aktuell existieren jedoch keine Modultests für die Callback-Funktion und die Hauptfunktion. Die Herausforderung besteht darin, dass durch die Nutzung des Impact Acquire SDK zahlreiche Abhängigkeiten zu Klassen und Methoden bestehen, die potenziell gemockt werden müssen. Der Aufwand hierfür ist für ein solches Testprogramm unverhältnismäßig hoch. Dennoch decken die bestehenden Modultests viele potenzielle Fehlerquellen innerhalb der Callback-Funktion auf, beispielsweise durch Tests der Methoden zur Vorbereitung des Speicherns. Zusätzlich bietet das Impact Acquire SDK selbst ein umfangreiches Fehlermanagement, das Fehlermeldungen an das Testprogramm weiterleitet.

Die Integration des Testframeworks mittels CMake erleichtert die Ausführung der Modultests erheblich. Nach dem Start der ausführbaren Testdatei wird im Terminal eine Übersicht der Tests sowie die Ergebnisse der einzelnen Modultests angezeigt:



```
[-----] 2 tests from LoggerTest
[ RUN    ] LoggerTest.InitializeLogger
[ OK     ] LoggerTest.InitializeLogger (0 ms)
[ RUN    ] LoggerTest.InitializeLoggerWithDifferentLogLevels
[ OK     ] LoggerTest.InitializeLoggerWithDifferentLogLevels (0 ms)
[-----] 2 tests from LoggerTest (0 ms total)

[-----] 3 tests from AzureIssueCreatorTest
[ RUN    ] AzureIssueCreatorTest.TestSendAzureIssueFailure
[ OK     ] AzureIssueCreatorTest.TestSendAzureIssueFailure (0 ms)
[ RUN    ] AzureIssueCreatorTest.ReturnsTrueWhenKeyIsHit
[ OK     ] AzureIssueCreatorTest.ReturnsTrueWhenKeyIsHit (0 ms)
[ RUN    ] AzureIssueCreatorTest.ReturnsFalseWhenNoKeyIsHit
[ OK     ] AzureIssueCreatorTest.ReturnsFalseWhenNoKeyIsHit (0 ms)
[-----] 3 tests from AzureIssueCreatorTest (0 ms total)

[-----] Global test environment tear-down
[=====] 37 tests from 5 test suites ran. (26 ms total)
[ PASSED ] 37 tests.
```

Abbildung 5.2: Ergebnis Modultests (eigene Darstellung)

Neben den Modultests wird das Testprogramm auch regelmäßig durch Verhaltenstests während der Entwicklungsphase überprüft. Somit werden Fehler frühzeitig identifiziert und behoben. Obwohl diese Tests die Testabdeckung temporär erhöhen, ist es wichtig zu erwähnen, dass sie nicht als Ersatz für eine systematische und dauerhafte Erweiterung der Testabdeckung durch Modultests dienen. Vielmehr ergänzen sie diese, indem sie zusätzliche Sicherheit bieten, dass das Programm auch unter realen Bedingungen zuverlässig funktioniert.

5.6 Ablauf des Testprogramms

Um den Ablauf des Testprogramms nach dem Start eines neuen Tests zu veranschaulichen, werden im Folgenden zwei Flussdiagramme präsentiert. Diese Diagramme skizzieren den grundlegenden Ablauf, ohne dabei auf spezifische Details des Quellcodes einzugehen.

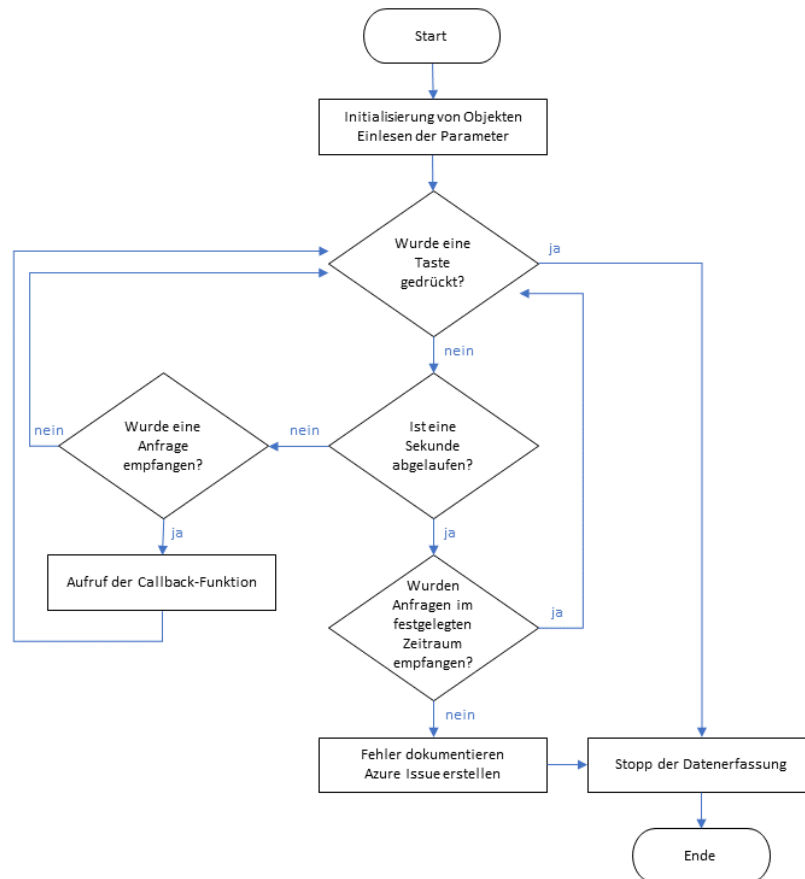


Abbildung 5.3: Flussdiagramm Hauptprogramm (eigene Darstellung)

Das Flussdiagramm in Abbildung 5.3 zeigt die Funktionsweise der Hauptfunktion, die zu Beginn des Testprogramms ausgeführt wird. Diese initialisiert am Anfang die benötigten Objekte, die für den Zugriff auf die Hilfsfunktionen notwendig sind. Anschließend wird der Logger für die Log-Datei zur Überwachung des Testprogramms erstellt, um den Prozess nachvollziehen zu können. Mithilfe der Klasse „ConfigurationHandler“ wird die Konfiguration eingelesen sowie der Ausgabeordner erstellt (siehe Kapitel 5.2). Nach erfolgreichem Abschluss dieser Schritte wird der Logger für die Datenübertragung und Speicherung konfiguriert, einschließlich der Festlegung des Log-Levels und der Implementierung der Log-Rotation. Diese Schritte setzen das vorherige Einlesen der erforderlichen Parameter

voraus, da diese für die Initialisierung erforderlich sind. Im weiteren Verlauf wird das Gerät (RadarImager) mittels des Impact Acquire SDK initialisiert und für die Kommunikation vorbereitet. Mit diesen Vorbereitungen ist das System bereit, mit der Datenerfassung zu beginnen.

Während der Datenerfassung überprüft das Programm kontinuierlich die Kommunikation. Diese Überprüfung erfolgt in einsekündigen Intervallen und stellt sicher, dass innerhalb einer festgelegten Zeitspanne Anfragen empfangen werden. Sollte keine Kommunikation feststellbar sein, wird das Programm abgebrochen, eine entsprechende Log-Nachricht erstellt und ein „Azure Issue“ generiert. Solange die Kommunikation ordnungsgemäß funktioniert, bleibt das Programm in Bereitschaft, um Daten zu empfangen und die zugehörige Callback-Funktion auszulösen. Dies gewährleistet eine effiziente und kontinuierliche Datenverarbeitung während des Testlaufs.

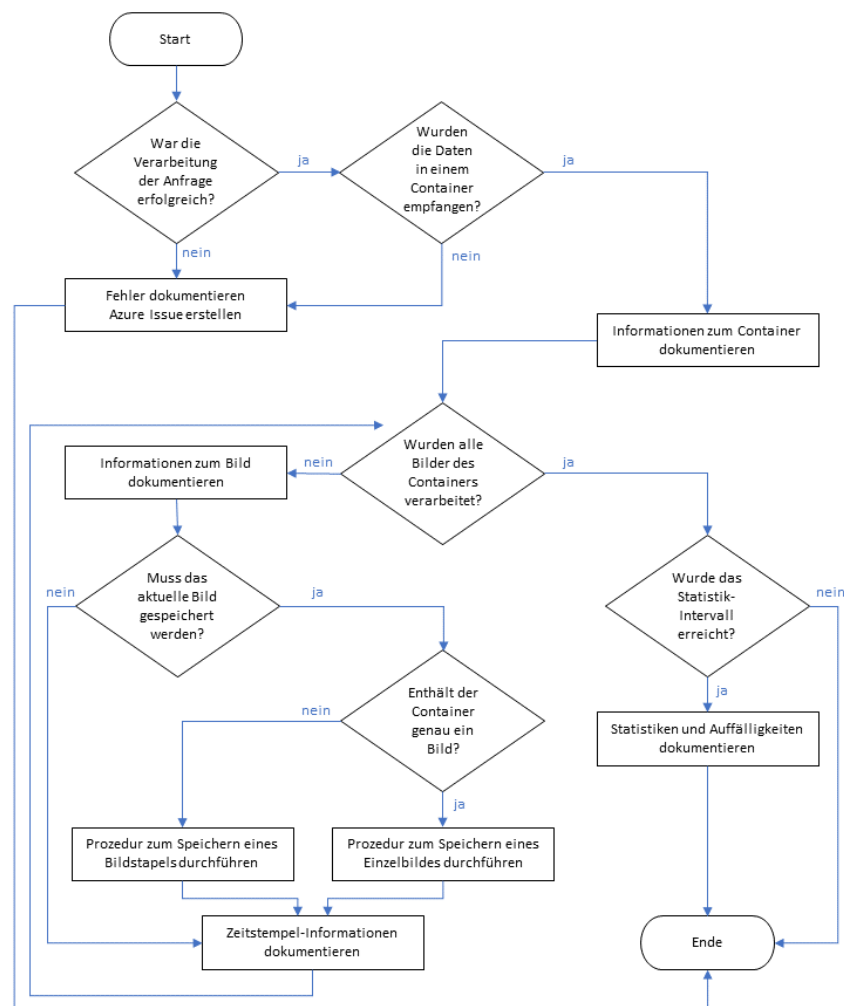


Abbildung 5.4: Flussdiagramm Callback-Funktion (eigene Darstellung)

Die Callback-Funktion gliedert sich in zwei Hauptbereiche: Fehlererkennung und Bilddatenverarbeitung (siehe Abbildung 5.4). Im Bereich der Fehlererkennung werden verschiedene Eigenschaften der empfangenen Anfrage auf Kompatibilität mit den erwarteten Werten überprüft. Dies umfasst die Bestätigung, dass die Anfrage erfolgreich verarbeitet ist und die Daten innerhalb eines Containers empfangen werden. Ein erkannter Fehler führt zur Protokollierung in einer Log-Nachricht, zur Erstellung eines „Azure Issues“ und zum Abbruch des Testprogramms.

Im zweiten Bereich, der Bilddatenverarbeitung, werden zunächst Informationen über den empfangenen Container im Terminal dargestellt und in den Log-Dateien vermerkt. Die Verarbeitung der Bilder erfolgt durch eine for-Schleife, die durch die Bilder des Containers iteriert. Für jedes Bild werden zusätzliche Informationen ausgegeben und gespeichert. Entsprechend den Anforderungen wird gegebenenfalls ein Speichervorgang für das aktuelle Bild initiiert. Unabhängig davon werden der Zeitstempel und die Verarbeitungszeit jedes Bildes in den Log-Dateien dokumentiert. Nach der Verarbeitung aller Bilder des Containers werden relevante Statistiken ausgegeben, sobald ein bestimmtes Intervall erreicht ist (siehe Kapitel 5.4 und Abbildung 4.1). Damit ist die Callback-Funktion abgeschlossen.

6 Bewertung und Fazit

In dieser Projektarbeit wird ein zuverlässiges Konzept für die Konfiguration und die automatisierte Generierung von Log-Dateien zum Testen des RadarImagers entwickelt. Das Konzept wird in C++ unter Einsatz verschiedener Bibliotheken sowie des Impact Acquire SDKs implementiert. Es ermöglicht, die Testparameter in einer Konfigurationsdatei festzulegen, den Test automatisiert durchzuführen und Auffälligkeiten in Log-Dateien zu dokumentieren. Dies erleichtert es dem Testingenieur, Fehler und andere Unregelmäßigkeiten des RadarImagers zu identifizieren.

Das entwickelte Testprogramm bietet eine einfache Konfiguration und eine zuverlässige Dokumentation der Ereignisse in Log-Dateien. Es stellt jedoch keine endgültige Lösung dar, da jedes Testprogramm im agilen Umfeld kontinuierlicher Weiterentwicklung und Anpassung an sich verändernde Bedingungen bedarf. Das aktuelle Konzept und Programm bieten eine sowohl konsistente als auch flexible Basis für zukünftige Anpassungen. Durch das Programm ist der Testablauf nun automatisierter und flexibler geworden, obwohl eine vollständige Automatisierung noch nicht erreicht ist. Die zu Beginn des Projekts gestellten Ziele werden jedoch erfolgreich erreicht.

Für die Zukunft besteht das Potenzial, das Programm weiterzuentwickeln oder zu erweitern, um den Grad der Automatisierung zu erhöhen. Beispielsweise könnte die Konfiguration durch die Entwicklung einer benutzerfreundlichen GUI intuitiver gestaltet werden. Zudem wäre die Integration des Testprogramms in ein übergeordnetes Softwaretool, beispielsweise in einen Azure DevOps Test Plan sinnvoll, um den Test weiter zu automatisieren und in bestehende Verfahren einzubinden.

Das entwickelte Konzept lässt sich auch auf andere Projekte übertragen. Obwohl jedes Projekt individuelle Anforderungen hat, sind bestimmte Ansätze wie die Verwendung und Struktur einer Konfigurationsdatei oder die Rotation der Log-Dateien oft universell anwendbar.

Ein zuverlässiges Testkonzept und -programm sowie die Automatisierung von Tests sind jedoch allein nicht ausreichend für einen erfolgreichen Testprozess. Es ist ebenso entscheidend, sinnvolle Testfälle zu erstellen, die das System umfassend beanspruchen und so dazu beitragen, mögliche Fehler effektiv aufzudecken.

Literaturverzeichnis

Balluff GmbH (2024a). Balluff Pressekit RadarImager. Abgerufen am 15.08.2024.

<https://www.balluff.com/de-de/software-und-systemloesungen/radarimager-gesteigerte-qualitaetssicherung>

Balluff GmbH (2024b). Für eine gesteigerte Qualitätssicherung: Das Unsichtbare sichtbar machen. Abgerufen am 15.08.2024.

<https://www.balluff.com/de-de/software-und-systemloesungen/radarimager-gesteigerte-qualitaetssicherung>

Balluff GmbH (2024c). Impact Acquire SDK C++. Abgerufen am 15.08.2024.

https://assets.balluff.com/documents/DRF_957352_AA_000/index.html

Balluff GmbH (2024d). Machine Vision Software. Abgerufen am 15.08.2024.

<https://www.balluff.com/de-de/products/areas/A0005/groups/G0507/products/F05701>

Baumgartner, M., Gwihs, S., Seidl, R., Steirer, T. and Wendland, M.-F. (2021). *Basiswissen Testautomatisierung*, dpunkt.verlag, Heidelberg.

<http://ebookcentral.proquest.com/lib/dhbw-stuttgart/detail.action?docID=6466725>

Beneken, G., Hummel, F. and Kucich, M. (2022). *Grundkurs agiles Software-Engineering*, Springer Vieweg, Wiesbaden.

<https://link.springer.com/book/10.1007/978-3-658-37371-9>

Carl Zeiss Digital Innovation GmbH (2024). Testautomatisierung. Abgerufen am 15.08.2024.

<https://blogs.zeiss.com/digital-innovation/de/tag/testautomatisierung/>

European Machine Vision Association (2024). GenICam Package Version 2024.04. Abgerufen am 15.08.2024.

<https://www.emva.org/standards-technology/genicam/genicam-downloads/>

Gu, S., Rong, G., Zhang, H. and Shen, H. (2023). Logging Practices in Software Engineering: A Systematic Mapping Study, *IEEE Transactions on Software Engineering*.

<https://ieeexplore.ieee.org/abstract/document/9756253>

Stroustrup, B. (2015). *Die C++-Programmiersprache*, Hanser, München.

<https://www.hanser-elibrary.com/doi/book/10.3139/9783446439818>

Witte, F. (2019). *Testmanagement und Softwaretest*, Springer Vieweg, Wiesbaden.

<https://link.springer.com/book/10.1007/978-3-658-25087-4>

Witte, F. (2023). *Konzeption und Umsetzung automatisierter Softwaretests*, Springer Vieweg, Wiesbaden.

<https://link.springer.com/book/10.1007/978-3-658-42661-3>

Anhang

ContinuousCapture.cpp

```
1 #include "functions.hpp"
2
3 using namespace std;
4 using namespace mvIMPACT::acquire;
5
6 //=====
7 //===== processRequest =====
8 //=====
9 void processRequest(shared_ptr<Request> pRequest, ThreadParameter &threadParameter,
    ↳ ConfigurationParameter &param, TimestampProvider &time, ImageSavePreparer &save,
    ↳ Logger &log, AzureIssueCreator &azureConnection)
10 {
11     auto loggerClient = spdlog::get("logClient");
12     ++threadParameter.requestsCaptured_;
13     time.timeLastRequest = time.getNowTime();
14
15     if (pRequest->isOK())
16     {
17         const unsigned int bufferPartCount = pRequest->getBufferPartCount();
18
19         if (bufferPartCount > 0) // Multi-part mode is running
20         {
21             cout << "Multipart buffer captured: " << bufferPartCount << " part" << ((
    ↳ bufferPartCount > 1) ? "s" : "") << endl;
22             loggerClient->info("Multipart buffer captured: {} part{}", bufferPartCount, (
    ↳ bufferPartCount > 1 ? "s" : ""));
23
24             for (unsigned int i = 0; i < bufferPartCount; i++)
25             {
26                 // Display information about the captured buffer
27                 const ImageBuffer *pPart = pRequest->getBufferPart(i).getImageBufferDesc().
    ↳ getBuffer();
28                 cout << "Image " << pRequest->infoFrameID.read() << " part " << i << ": " <<
    ↳ pPart->iWidth << "x" << pPart->iHeight << endl;
29                 loggerClient->info("Image {} part {}: {}x{}", pRequest->infoFrameID.read(), i,
    ↳ pPart->iWidth, pPart->iHeight);
30
31                 // Save images depending on the configuration
32                 if (param.saveImages && (param.saveImagesInterval == 1 || pRequest->infoFrameID.
    ↳ read() % param.saveImagesInterval == 1))
33                 {
34                     if (bufferPartCount == 1) // Buffer has exactly one image
35                     {
```

```

36         int imageNumber = ceil((float)pRequest->infoFrameID.read() / (float)param.
↪ saveImagesInterval);
37         string filename = save.prepareImageSave(param, imageNumber);
38         pRequest->getBufferPart(0).getImageBufferDesc().save(filename);
39         loggerClient->info("Saved image {} to {}", pRequest->infoFrameID.read(),
↪ filename);
40     }
41     else // Buffer has more than one image
42     {
43         string filename = save.prepareBufferSave(param, pRequest->infoFrameID.read()
↪ , i);
44         pRequest->getBufferPart(i).getImageBufferDesc().save(filename);
45         loggerClient->info("Saved image {} part {} to {}", pRequest->infoFrameID.
↪ read(), i, filename);
46     }
47 }
48
49 // Log timestamp information
50 int64_t timestamp_us = pRequest->infoTimeStamp_us.read() / 1000;
51 int64_t timestamp_ms = chrono::duration_cast<chrono::milliseconds>(chrono::
↪ system_clock::now().time_since_epoch()).count();
52 loggerClient->info("Image: {} \t Timestamp_us: {} \t Timestamp: {} \t Difference:
↪ {} \n", pRequest->infoFrameID.read(), timestamp_us, timestamp_ms, timestamp_ms -
↪ timestamp_us);
53 }
54 }
55 else // Multi-part mode is not running
56 {
57     cout << "Error: RadarImager is not running in multi-part mode" << endl;
58     loggerClient->error("Error: RadarImager is not running in multi-part mode \n");
59     log.logStatistics(threadParameter);
60     azureConnection.sendAzureIssue(param, threadParameter.statistics_.frameCount.read
↪ ());
61     exit(1);
62 }
63 }
64 else // Request error
65 {
66     cout << "Error: " << pRequest->requestResult.readS() << endl;
67     loggerClient->error("{} \n", pRequest->requestResult.readS());
68     log.logStatistics(threadParameter);
69     azureConnection.sendAzureIssue(param, threadParameter.statistics_.frameCount.read()
↪ );
70     exit(1);
71 }
72
73 // Log some statistics and special data every few requests
74 if (threadParameter.requestsCaptured_ % log.STATISTICS_INTERVAL == 0)
75 {
76     log.logStatistics(threadParameter);
77
78     const Statistics &stats = threadParameter.statistics_;
79     cout << "\nInfo from " << threadParameter.pDev_->serial.read()
80         << ": " << stats.framesPerSecond.name() << ": " << stats.framesPerSecond.readS()
81         << ", " << stats.errorCount.name() << ": " << stats.errorCount.readS()

```



```

82         << ", " << stats.captureTime_s.name() << ": " << stats.captureTime_s.readS() << "
    ↪ ↪ \n" << endl;
83     }
84 }
85
86 //=====
87 //===== main =====
88 //=====
89 int main()
90 {
91     ConfigurationParameter param;
92     TimestampProvider time;
93     ConfigurationHandler config;
94     ImageSavePreparer save;
95     Logger log;
96     AzureIssueCreator azureConnection;
97
98     config.configFilename = ".././configParameter.json";
99     spdlog::set_pattern("%^[%Y-%m-%d %H:%M:%S.%e] [%l] %v%$");
100    auto loggerScript = spdlog::basic_logger_mt("logScript", "../Logfile.txt", true);
101    loggerScript->set_level(spdlog::level::info);
102    loggerScript->flush_on(spdlog::level::trace);
103
104    if (!(config.loadConfiguration(param)))
105    {
106        cout << "Error: Reading config-file failed" << endl;
107        loggerScript->info("Reading config-file failed");
108        return 1;
109    }
110
111    log.initializeLogger(param);
112
113    if (param.saveImages == false)
114    {
115        cout << "[images will not be saved]" << endl;
116        loggerScript->info("[images will not be saved]");
117    }
118    else
119    {
120        cout << "[images will be saved - new folder every " << param.newFolderInterval << "
    ↪ ↪ images]" << endl;
121        loggerScript->info("[images will be saved - new folder every {} images]", param.
    ↪ ↪ newFolderInterval);
122    }
123
124    DeviceManager devMgr;
125    Device *pDev = getDeviceFromUserInput(devMgr);
126
127    if (pDev == nullptr)
128    {
129        cout << "Unable to continue! Press [ENTER] to end the application" << endl;
130        cin.get();
131        return 1;
132    }
133

```

```

134 cout << "Initialising the device. This might take some time..." << endl;
135 try
136 {
137     pDev->open();
138 }
139 catch (const ImpactAcquireException &e)
140 {
141     // this e.g. might happen if the same device is already opened in another process...
142     cout << "An error occurred while opening the device " << pDev->serial.read() << "(
↳ error code: " << e.getErrorCodeAsString() << ").";
143     loggerScript->error("An error occurred while opening the device {} (error code: {})"
↳ , pDev->serial.read(), e.getErrorCodeAsString());
144     return 1;
145 }
146
147 cout << "Press [ENTER] to stop the acquisition thread" << endl;
148
149 ThreadParameter threadParameter(pDev);
150 threadParameter.statistics_.reset();
151 helper::RequestProvider requestProvider(pDev);
152 time.timeLastRequest = time.getTime();
153 requestProvider.acquisitionStart(processRequest, ref(threadParameter), ref(param), ref
↳ (time), ref(save), ref(log), ref(azureConnection));
154
155 // exit the test if no more images are received
156 while (!azureConnection.checkKeyboardHit())
157 {
158     if (time.getTime() - time.timeLastRequest > time.MAX_TIME_BETWEEN_IMAGES)
159     {
160         auto loggerClient = spdlog::get("logClient");
161         cout << "Error: Maximum time between Images exceeded." << endl;
162         loggerClient->error("Error: Maximum time between Images exceeded\n");
163         log.logStatistics(threadParameter);
164         azureConnection.sendAzureIssue(param, threadParameter.statistics_.frameCount.read
↳ ());
165         exit(1);
166     }
167     this_thread::sleep_for(chrono::milliseconds(1000));
168 }
169
170 requestProvider.acquisitionStop();
171 log.logStatistics(threadParameter);
172 return 0;
173 }

```

functions.hpp

```

1 #include <cmath>
2 #include <chrono>
3 #include <string>
4 #include <thread>

```

```

5 #include <iomanip>
6 #include <sstream>
7 #include <fstream>
8 #include <fcntl.h>
9 #include <stdio.h>
10 #include <stdarg.h>
11 #include <unistd.h>
12 #include <iostream>
13 #include <termios.h>
14 #include <functional>
15 #include <filesystem>
16 #include <sys/stat.h>
17 #include <sys/types.h>
18
19 #include <apps/Common/exampleHelper.h>
20 #include <mvIMPACT_CPP/mvIMPACT_acquire_helper.h>
21
22 #include "base64.h"
23 #include "json.hpp"
24 #include "curl/curl.h"
25 #include "spdlog/spdlog.h"
26 #include "spdlog/sinks/ostream_sink.h"
27 #include "spdlog/sinks/basic_file_sink.h"
28 #include "spdlog/sinks/rotating_file_sink.h"
29
30 using namespace std;
31
32 //=====
33 //===== ConfigurationParameter =====
34 //=====
35 struct ConfigurationParameter
36 {
37     bool saveImages;
38     bool numberImages;
39     bool numberFolders;
40     int saveImagesInterval;
41     int newFolderInterval;
42     bool createLog;
43     int logLevel;
44     int maxLogFiles;
45     int maxLogSize;
46     bool createIssue;
47     string outputPath;
48     string imagesPath;
49     string logsPath;
50     string testID;
51 };
52
53 //=====
54 //===== ThreadParameter =====
55 //=====
56 struct ThreadParameter
57 {
58     Device *pDev_;
59     unsigned int requestsCaptured_;

```

```

60     Statistics statistics_;
61     explicit ThreadParameter(Device *pDev) : pDev_(pDev), requestsCaptured_(0),
        ↳ statistics_(pDev) {}
62     ThreadParameter(const ThreadParameter &src) = delete;
63     ThreadParameter &operator=(const ThreadParameter &rhs) = delete;
64 };
65
66 //=====
67 //===== TimestampProvider =====
68 //=====
69 class TimestampProvider
70 {
71 public:
72     string getTimestamp();
73     string getTimestamp_ms();
74     time_t getNowTime();
75     time_t timeLastRequest;
76     const int MAX_TIME_BETWEEN_IMAGES = 180;
77 };
78
79 //=====
80 //===== ConfigurationHandler =====
81 //=====
82 class ConfigurationHandler
83 {
84 public:
85     bool loadConfiguration(ConfigurationParameter &config);
86     string configFilename;
87
88 private:
89     bool checkPath(const string &path);
90     bool createFolder(const string &path);
91     bool readConfigurationParameter(ConfigurationParameter &config);
92     bool checkConfigurationParameter(const ConfigurationParameter &config);
93     bool createOutputDirectories(ConfigurationParameter &config);
94     const int MIN_SAVE_IMAGES_INTERVAL = 1;
95     const int MAX_SAVE_IMAGES_INTERVAL = 1000;
96     const int MIN_NEW_FOLDER_INTERVAL = 1;
97     const int MAX_NEW_FOLDER_INTERVAL = 1000;
98     const int MIN_LOG_LEVEL = 0;
99     const int MAX_LOG_LEVEL = 3;
100     const int MIN_MAX_LOG_FILES = 1;
101     const int MAX_MAX_LOG_FILES = 1000;
102     const int MIN_MAX_LOG_SIZE = 1;
103     const int MAX_MAX_LOG_SIZE = 100;
104 };
105
106 //=====
107 //===== ImageSavePreparer =====
108 //=====
109 class ImageSavePreparer
110 {
111 public:
112     string prepareImageSave(const ConfigurationParameter &config, const int imageNumber);

```

```

113     string prepareBufferSave(const ConfigurationParameter &config, const int imageNumber,
114                               ↪ const int i);
115 private:
116     string createImageSubpathFolder(const ConfigurationParameter &param, const int
117                                       ↪ imageNumber);
117     string createBufferSubpathFolder(const ConfigurationParameter &param, const int
118                                       ↪ imageNumber, const int i);
118     string imageLastSubpath;
119     string bufferLastSubpath;
120 };
121
122 //=====
123 //===== Logger =====
124 //=====
125 class Logger
126 {
127 public:
128     bool initializeLogger(const ConfigurationParameter &config);
129     void logStatistics(ThreadParameter &threadParameter);
130     const int STATISTICS_INTERVAL = 100;
131
132 private:
133     bool setLogLevel(const ConfigurationParameter &config);
134     void interpretStatistics(ThreadParameter &threadParameter);
135     const int ONE_MEGABYTE = 1048576;
136     const float FPS_CHANGE_THRESHOLD = 0.5;
137     int lastRequestCaptured = 0;
138     int lastAbortedRequestCount = 0;
139     float lastFramesPerSecond = 0;
140     int lastFramesIncompleteCount = 0;
141     int lastLostImagesCount = 0;
142     int lastRetransmitCount = 0;
143     int lastTimedOutRequestCount = 0;
144 };
145
146 //=====
147 //===== AzureIssueCreator =====
148 //=====
149 class AzureIssueCreator
150 {
151 public:
152     void sendAzureIssue(const ConfigurationParameter &param, const int frameCount);
153     bool checkKeyboardHit();
154
155 private:
156     bool createAzureIssue(const string title, const string description);
157     static size_t handleCurlResponse(void *contents, size_t size, size_t nmemb, void
158                                       ↪ *userp);
158     string organization = "your_organization";
159     string project = "your_project";
160     string user = "your_name"; // Enter your Name
161     string pat = "your_pat"; // Enter your PAT
162 };

```

functions.cpp

```
1 #include "functions.hpp"
2
3 using namespace std;
4
5 //=====
6 //===== getNowTime =====
7 //=====
8 time_t TimestampProvider::getNowTime()
9 {
10     time_t nowTime;
11     auto now = chrono::system_clock::now();
12     nowTime = chrono::system_clock::to_time_t(now);
13     return nowTime;
14 }
15
16 //=====
17 //===== getTimestamp =====
18 //=====
19 string TimestampProvider::getTimestamp()
20 {
21     time_t nowTime = getNowTime();
22
23     stringstream ss;
24     ss << put_time(localtime(&nowTime), "%Y-%m-%d_%Hh-%Mm-%Ss");
25     return ss.str();
26 }
27
28 //=====
29 //===== getTimestamp_ms =====
30 //=====
31 string TimestampProvider::getTimestamp_ms()
32 {
33     time_t nowTime = getNowTime();
34     auto now = chrono::system_clock::now();
35     auto ms = chrono::duration_cast<chrono::milliseconds>(now.time_since_epoch()) % 1000;
36
37     stringstream ss;
38     ss << put_time(localtime(&nowTime), "%Y-%m-%d_%Hh-%Mm-%Ss-") << setfill('0') << setw(3)
39         << ms.count() << "ms";
40     return ss.str();
41 }
42
43 //=====
44 //===== readConfigurationParameter =====
45 //=====
46 bool ConfigurationHandler::readConfigurationParameter(ConfigurationParameter &param)
47 {
48     auto loggerScript = spdlog::get("logScript");
49     ifstream paramFile(configFilename);
50     nlohmann::json root;
51     paramFile >> root;
```

```

52 // Check if the parameters in the json-file has the correct datatype
53 if (!root["Basic"]["testID"].is_string() ||
54     !root["GenICam-Client"]["saveImages"].is_boolean() ||
55     !root["GenICam-Client"]["numberImages"].is_boolean() ||
56     !root["GenICam-Client"]["numberFolders"].is_boolean() ||
57     !root["GenICam-Client"]["saveImagesInterval"].is_number_integer() ||
58     !root["GenICam-Client"]["newFolderInterval"].is_number_integer() ||
59     !root["GenICam-Client"]["createLog"].is_boolean() ||
60     !root["GenICam-Client"]["logLevel"].is_number_integer() ||
61     !root["GenICam-Client"]["maxLogFiles"].is_number_integer() ||
62     !root["GenICam-Client"]["maxLogSize"].is_number_integer() ||
63     !root["GenICam-Client"]["createIssue"].is_boolean() ||
64     !root["GenICam-Client"]["outputPath"].is_string())
65 {
66     loggerScript->error("Error: incompatible data type of the parameters.");
67     cout << "Error: incompatible data type of the parameters." << endl;
68     return false;
69 }
70
71 // Copy the parameters into variables
72 try
73 {
74     param.testID = root["Basic"]["testID"];
75     param.saveImages = root["GenICam-Client"]["saveImages"];
76     param.numberImages = root["GenICam-Client"]["numberImages"];
77     param.numberFolders = root["GenICam-Client"]["numberFolders"];
78     param.saveImagesInterval = root["GenICam-Client"]["saveImagesInterval"];
79     param.newFolderInterval = root["GenICam-Client"]["newFolderInterval"];
80     param.createLog = root["GenICam-Client"]["createLog"];
81     param.logLevel = root["GenICam-Client"]["logLevel"];
82     param.maxLogFiles = root["GenICam-Client"]["maxLogFiles"];
83     param.maxLogSize = root["GenICam-Client"]["maxLogSize"];
84     param.createIssue = root["GenICam-Client"]["createIssue"];
85     param.outputPath = root["GenICam-Client"]["outputPath"];
86 }
87 catch (const exception &e)
88 {
89     loggerScript->error("Error: loading the parameters failed.");
90     cout << "Error: loading the parameters failed." << endl;
91     return false;
92 }
93
94 return true;
95 }
96
97 //=====
98 //===== checkConfigurationParameter =====
99 //=====
100 // Check if the parameters are within the specified ranges or have the correct format
101 bool ConfigurationHandler::checkConfigurationParameter(const ConfigurationParameter &
102     ↪ param)
103 {
104     auto loggerScript = spdlog::get("logScript");

```

```

105 if (param.saveImagesInterval < MIN_SAVE_IMAGES_INTERVAL || param.saveImagesInterval >
    ↪ MAX_SAVE_IMAGES_INTERVAL)
106 {
107     loggerScript->error("Error: saveImagesInterval invalid.");
108     cout << "Error: saveImagesInterval invalid." << endl;
109     return false;
110 }
111
112 if (param.newFolderInterval < MIN_NEW_FOLDER_INTERVAL || param.newFolderInterval >
    ↪ MAX_NEW_FOLDER_INTERVAL)
113 {
114     loggerScript->error("Error: newFolderInterval invalid.");
115     cout << "Error: newFolderInterval invalid." << endl;
116     return false;
117 }
118
119 if (param.logLevel < MIN_LOG_LEVEL || param.logLevel > MAX_LOG_LEVEL)
120 {
121     loggerScript->error("Error: logLevel invalid.");
122     cout << "Error: logLevel invalid." << endl;
123     return false;
124 }
125
126 if (param.maxLogFiles < MIN_MAX_LOG_FILES || param.maxLogFiles > MAX_MAX_LOG_FILES)
127 {
128     loggerScript->error("Error: maxLogFiles invalid.");
129     cout << "Error: maxLogFiles invalid." << endl;
130     return false;
131 }
132
133 if (param.maxLogSize < MIN_MAX_LOG_SIZE || param.maxLogSize > MAX_MAX_LOG_SIZE)
134 {
135     loggerScript->error("Error: maxLogSize invalid.");
136     cout << "Error: maxLogSize invalid." << endl;
137     return false;
138 }
139
140 if (param.testID.empty())
141 {
142     loggerScript->error("Error: testID empty.");
143     cout << "Error: testID empty." << endl;
144     return false;
145 }
146
147 if (!checkPath(param.outputPath))
148 {
149     loggerScript->error("Error: output path invalid.");
150     cout << "Error: output path invalid." << endl;
151     return false;
152 }
153
154 return true;
155 }
156
157 //=====

```



```

158 //===== checkPath =====
159 //=====
160 bool ConfigurationHandler::checkPath(const string &path)
161 {
162     auto loggerScript = spdlog::get("logScript");
163
164     if (path.empty())
165     {
166         loggerScript->error("Error: empty output path.");
167         cout << "Error: empty output path." << endl;
168         return false;
169     }
170
171     for (char c : path)
172     {
173         if (!(isalnum(c) || c == '/' || c == '.' || c == '-' || c == '_' || c == '~'))
174         {
175             loggerScript->error("Error: invalid character in output path.");
176             cout << "Error: invalid character in output path." << endl;
177             return false;
178         }
179     }
180
181     return true;
182 }
183
184 //=====
185 //===== createFolder =====
186 //=====
187 bool ConfigurationHandler::createFolder(const string &path)
188 {
189     auto loggerScript = spdlog::get("logScript");
190
191     if (!(filesystem::create_directories(path)))
192     {
193         loggerScript->error("Error: creating the folder failed.");
194         cout << "Error: creating the folder failed." << endl;
195         return false;
196     }
197
198     return true;
199 }
200
201 //=====
202 //===== createOutputDirectories =====
203 //=====
204 bool ConfigurationHandler::createOutputDirectories(ConfigurationParameter &param)
205 {
206     auto loggerScript = spdlog::get("logScript");
207
208     TimestampProvider time;
209     string timestamp = time.getTimestamp();
210
211     param.outputPath = param.outputPath + "/Output_" + param.testID + "_" + timestamp;
212     param.imagesPath = param.outputPath + "/Images_" + param.testID + "_" + timestamp;

```

```

213 param.logsPath = param.outputPath + "/Logs_" + param.testID + "_" + timestamp;
214
215 if (!createFolder(param.outputPath) ||
216     !createFolder(param.imagesPath) ||
217     !createFolder(param.logsPath))
218 {
219     loggerScript->error("Error: creating the output directories failed.");
220     cout << "Error: creating the output directories failed." << endl;
221     return false;
222 }
223
224 return true;
225 }
226
227 //=====
228 //===== loadConfiguration =====
229 //=====
230 bool ConfigurationHandler::loadConfiguration(ConfigurationParameter &param)
231 {
232     auto loggerScript = spdlog::get("logScript");
233
234     if (!readConfigurationParameter(param) ||
235         !checkConfigurationParameter(param) ||
236         !createOutputDirectories(param))
237     {
238         loggerScript->error("Error: loading the parameters failed.");
239         cout << "Error: loading the parameters failed." << endl;
240         return false;
241     }
242
243     return true;
244 }
245
246 //=====
247 //===== createImageSubpathFolder =====
248 //=====
249 string ImageSavePreparer::createImageSubpathFolder(const ConfigurationParameter &param,
250     ↪ const int imageNumber)
251 {
252     TimestampProvider time;
253     string subpath;
254
255     if (param.numberFolders)
256     {
257         int folderNo = ceil((float)imageNumber / (float)param.newFolderInterval);
258         subpath = param.imagesPath + "/" + to_string(folderNo) + "/";
259         if (imageNumber % param.newFolderInterval == 1)
260         {
261             mkdir(subpath.c_str(), 0777);
262         }
263     }
264     else
265     {
266         if (imageNumber % param.newFolderInterval == 1)
267         {

```

```

267     string timestamp_ms = time.getTimestamp_ms();
268     subpath = param.imagesPath + "/" + timestamp_ms + "/";
269     mkdir(subpath.c_str(), 0777);
270     imageLastSubpath = subpath;
271 }
272 else
273 {
274     subpath = imageLastSubpath;
275 }
276 }
277
278 return subpath;
279 }
280
281 //=====
282 //===== prepareImageSave =====
283 //=====
284 string ImageSavePreparer::prepareImageSave(const ConfigurationParameter &param, const
    ↪ int imageNumber)
285 {
286     TimestampProvider time;
287     string filename;
288     string subpath = createImageSubpathFolder(param, imageNumber);
289
290     if (param.numberImages)
291     {
292         filename = subpath + "image_" + to_string(imageNumber) + ".png";
293     }
294     else
295     {
296         string timestamp_ms = time.getTimestamp_ms();
297         filename = subpath + "image_" + timestamp_ms + ".png";
298     }
299
300     return filename;
301 }
302
303 //=====
304 //===== createBufferSubpathFolder =====
305 //=====
306 string ImageSavePreparer::createBufferSubpathFolder(const ConfigurationParameter &param,
    ↪ const int imageNumber, const int i)
307 {
308     TimestampProvider time;
309     string subpath;
310
311     if (param.numberFolders)
312     {
313         subpath = param.imagesPath + "/" + to_string(imageNumber) + "/";
314         if (i == 0)
315         {
316             mkdir(subpath.c_str(), 0777);
317         }
318     }
319     else

```

```

320 {
321     if (i == 0)
322     {
323         subpath = param.imagesPath + "/" + time.getTimestamp_ms() + "/";
324         bufferLastSubpath = subpath;
325     }
326     else
327     {
328         subpath = bufferLastSubpath;
329     }
330 }
331
332 return subpath;
333 }
334
335 //=====
336 //===== prepareBufferSave =====
337 //=====
338 string ImageSavePreparer::prepareBufferSave(const ConfigurationParameter &param, const
    ↪ int imageNumber, const int i)
339 {
340     TimestampProvider time;
341     string filename;
342     string subpath = createBufferSubpathFolder(param, imageNumber, i);
343
344     if (param.numberImages)
345     {
346         filename = subpath + "image_" + to_string(imageNumber) + "part." + to_string(i) + ".
    ↪ png";
347     }
348     else
349     {
350         filename = subpath + "image_" + to_string(imageNumber) + "_" + time.getTimestamp_ms
    ↪ () + ".png";
351     }
352
353     return filename;
354 }
355
356 //=====
357 //===== setLogLevel =====
358 //=====
359 bool Logger::setLogLevel(const ConfigurationParameter &param)
360 {
361     auto loggerClient = spdlog::get("logClient");
362     auto loggerScript = spdlog::get("logScript");
363
364     if (param.createLog)
365     {
366         switch (param.logLevel)
367         {
368             case 0:
369                 loggerClient->set_level(spdlog::level::off);
370                 break;
371             case 1:

```

```

372     loggerClient->set_level(spdlog::level::err);
373     break;
374 case 2:
375     loggerClient->set_level(spdlog::level::warn);
376     break;
377 case 3:
378     loggerClient->set_level(spdlog::level::info);
379     break;
380 default:
381     loggerScript->error("Error: invalid Log-Level");
382     cerr << "Error: invalid Log-Level" << endl;
383     return false;
384     break;
385 }
386 }
387 else
388 {
389     loggerClient->set_level(spdlog::level::off);
390 }
391
392 return true;
393 }
394
395 //=====
396 //===== initializeLogger =====
397 //=====
398 bool Logger::initializeLogger(const ConfigurationParameter &param)
399 {
400     string logFileName = param.logsPath + "/Logfile.txt";
401
402     auto loggerClient = spdlog::rotating_logger_mt("logClient", logFileName, (ONE_MEGABYTE
403         ↳ * param.maxLogSize), (param.maxLogFiles - 1));
404     setLogLevel(param);
405     loggerClient->flush_on(spdlog::level::trace);
406
407     return true;
408 }
409
410 //=====
411 //===== logStatistics =====
412 //=====
413 void Logger::logStatistics(ThreadParameter &threadParameter)
414 {
415     auto loggerClient = spdlog::get("logClient");
416
417     const Statistics &s = threadParameter.statistics_;
418     loggerClient->info("Info from {:}", threadParameter.pDev_->serial.read());
419     loggerClient->info("{:}:", s.abortedRequestsCount.name(), s.abortedRequestsCount.
420         ↳ readS());
421     loggerClient->info("{:}:", s.bandwidthConsumed.name(), s.bandwidthConsumed.readS());
422     loggerClient->info("{:}:", s.captureTime_s.name(), s.captureTime_s.readS());
423     loggerClient->info("{:}:", s.errorCount.name(), s.errorCount.readS());
424     loggerClient->info("{:}:", s.formatConvertTime_s.name(), s.formatConvertTime_s.readS
425         ↳ ());
426     loggerClient->info("{:}:", s.frameCount.name(), s.frameCount.readS());

```

```

424 loggerClient->info("{}: {}", s.framesIncompleteCount.name(), s.framesIncompleteCount.
    ↳ readS());
425 loggerClient->info("{}: {}", s.framesPerSecond.name(), s.framesPerSecond.readS());
426 loggerClient->info("{}: {}", s.imageProcTime_s.name(), s.imageProcTime_s.readS());
427 loggerClient->info("{}: {}", s.lostImagesCount.name(), s.lostImagesCount.readS());
428 loggerClient->info("{}: {}", s.missingDataAverage_pc.name(), s.missingDataAverage_pc.
    ↳ readS());
429 loggerClient->info("{}: {}", s.queueTime_s.name(), s.queueTime_s.readS());
430 loggerClient->info("{}: {}", s.retransmitCount.name(), s.retransmitCount.readS());
431 loggerClient->info("{}: {}\\n", s.timedOutRequestsCount.name(), s.timedOutRequestsCount
    ↳ .readS());
432
433 interpretStatistics(threadParameter);
434 }
435
436 //=====
437 //===== interpretStatistics =====
438 //=====
439 // Log warnings about significant changes or special values of the statistics
440 void Logger::interpretStatistics(ThreadParameter &threadParameter)
441 {
442     auto loggerClient = spdlog::get("logClient");
443
444     const Statistics &s = threadParameter.statistics_;
445     bool anyWarningLogged = false;
446
447     int requestDifference = threadParameter.requestsCaptured_ - lastRequestCaptured;
448
449     if (s.abortedRequestsCount.read() > lastAbortedRequestCount)
450     {
451         loggerClient->warn("There were {} requests aborted over the last {} images", s.
            ↳ abortedRequestsCount.read() - lastAbortedRequestCount, requestDifference);
452         anyWarningLogged = true;
453     }
454
455     if (abs(s.framesPerSecond.read() - lastFramesPerSecond) > FPS_CHANGE_THRESHOLD * max(
        ↳ lastFramesPerSecond, 1.0f))
456     {
457         loggerClient->warn("The Frames per Second changed by more than 50% of the last value
            ↳ . Current: {}, Last: {}", s.framesPerSecond.read(), lastFramesPerSecond);
458         anyWarningLogged = true;
459     }
460
461     if (s.framesIncompleteCount.read() > lastFramesIncompleteCount)
462     {
463         loggerClient->warn("There were {} frames incomplete over the last {} images", s.
            ↳ framesIncompleteCount.read() - lastFramesIncompleteCount, requestDifference);
464         anyWarningLogged = true;
465     }
466
467     if (s.lostImagesCount.read() > lastLostImagesCount)
468     {
469         loggerClient->warn("There were {} images lost over the last {} images", s.
            ↳ lostImagesCount.read() - lastLostImagesCount, requestDifference);
470         anyWarningLogged = true;

```

```

471 }
472
473 if (s.retransmitCount.read() > lastRetransmitCount)
474 {
475     loggerClient->warn("There were {} requests retransmitted over the last {} images", s
↳ .retransmitCount.read() - lastRetransmitCount, requestDifference);
476     anyWarningLogged = true;
477 }
478
479 if (s.timedOutRequestsCount.read() > lastTimedOutRequestCount)
480 {
481     loggerClient->warn("There were {} requests timed out over the last {} images", s.
↳ timedOutRequestsCount.read() - lastTimedOutRequestCount, requestDifference);
482     anyWarningLogged = true;
483 }
484
485 if (anyWarningLogged)
486 {
487     loggerClient->info("These were the problematic statistics\n");
488 }
489 else
490 {
491     loggerClient->info("The Statistics are within acceptable limits\n");
492 }
493
494 lastRequestCaptured = threadParameter.requestsCaptured_;
495 lastAbortedRequestCount = s.abortedRequestsCount.read();
496 lastFramesPerSecond = s.framesPerSecond.read();
497 lastFramesIncompleteCount = s.framesIncompleteCount.read();
498 lastRetransmitCount = s.retransmitCount.read();
499 lastTimedOutRequestCount = s.timedOutRequestsCount.read();
500 }
501
502 //=====
503 //===== handleCurlResponse =====
504 //=====
505 size_t AzureIssueCreator::handleCurlResponse(void *contents, size_t size, size_t nmemb,
↳ void *userp)
506 {
507     ((string *)userp)->append((char *)contents, size * nmemb);
508     return size * nmemb;
509 }
510
511 //=====
512 //===== createAzureIssue =====
513 //=====
514 bool AzureIssueCreator::createAzureIssue(const string title, const string description)
515 {
516     auto loggerScript = spdlog::get("logScript");
517
518     CURL *curl;
519     CURLcode res;
520     string readBuffer;
521
522     string jsonData = R(

```

```

523 [
524     {
525         "op": "add",
526         "path": "/fields/System.Title",
527         "value": ")" + title +
528             R"(",
529     },
530     {
531         "op": "add",
532         "path": "/fields/System.Description",
533         "value": ")" + description +
534             R"(",
535     },
536     {
537         "op": "add",
538         "path": "/fields/System.AreaPath",
539         "value": "SIP Doh0 Imaging Radar\\Software",
540     },
541     {
542         "op": "add",
543         "path": "/fields/System.AssignedTo",
544         "value": ")" + user +
545             R"(",
546     },
547 ];
548
549 string url = "https://dev.azure.com/" + organization + "/" + project + "/_apis/wit/
    ↳ workitems/$Issue?api-version=6.0";
550
551 string encoded_pat = base64_encode(":" + pat);
552
553 curl = curl_easy_init();
554 if (curl)
555 {
556     struct curl_slist *headers = NULL;
557     headers = curl_slist_append(headers, "Content-Type: application/json-patch+json");
558
559     string authHeader = "Authorization: Basic " + encoded_pat;
560     headers = curl_slist_append(headers, authHeader.c_str());
561
562     curl_easy_setopt(curl, CURLOPT_URL, url.c_str());
563     curl_easy_setopt(curl, CURLOPT_CUSTOMREQUEST, "POST");
564     curl_easy_setopt(curl, CURLOPT_HTTPHEADER, headers);
565     curl_easy_setopt(curl, CURLOPT_POSTFIELDS, jsonData.c_str());
566
567     curl_easy_setopt(curl, CURLOPT_WRITEFUNCTION, handleCurlResponse);
568     curl_easy_setopt(curl, CURLOPT_WRITEDATA, &readBuffer);
569
570     res = curl_easy_perform(curl);
571
572     if (res != CURLE_OK)
573     {
574         loggerScript->error("cURL Error: {}", curl_easy_strerror(res));
575         cerr << "cURL Error: " << curl_easy_strerror(res) << endl;
576         return false;

```



```

577     }
578     else
579     {
580         loggerScript->info("cURL Response from server: {}", readBuffer);
581         cout << "cURL Response from server:" << readBuffer << endl;
582     }
583
584     curl_slist_free_all(headers);
585     curl_easy_cleanup(curl);
586 }
587
588 return true;
589 }
590
591 //=====
592 //===== sendAzureIssue =====
593 //=====
594 void AzureIssueCreator::sendAzureIssue(const ConfigurationParameter &param, const int
    ↪ frameCount)
595 {
596     TimestampProvider time;
597     auto loggerClient = spdlog::get("logClient");
598
599     if (param.createIssue)
600     {
601         string title;
602         title = "Test: " + param.testID + " failed!";
603         string description;
604         description = "Test failed / Time: " + time.getTimestamp() + " / Images: " +
            ↪ to_string(frameCount);
605
606         if (createAzureIssue(title, description))
607         {
608             loggerClient->info("\n AzureIssue created: Title: {}, Description: {}.\n", title,
                ↪ description);
609         }
610         else
611         {
612             loggerClient->info("\n Creating AzureIssue failed\n");
613         }
614     }
615 }
616 }
617
618 //=====
619 //===== checkKeyboardHit =====
620 //=====
621 bool AzureIssueCreator::checkKeyboardHit()
622 {
623     struct termios oldt, newt;
624     int ch;
625     int oldf;
626
627     tcgetattr(STDIN_FILENO, &oldt);
628     newt = oldt;

```

```
629 newt.c_lflag &= ~(ICANON | ECHO);
630 tcsetattr(STDIN_FILENO, TCSANOW, &newt);
631 oldf = fcntl(STDIN_FILENO, F_GETFL, 0);
632 fcntl(STDIN_FILENO, F_SETFL, oldf | O_NONBLOCK);
633 ch = getchar();
634 tcsetattr(STDIN_FILENO, TCSANOW, &oldt);
635 fcntl(STDIN_FILENO, F_SETFL, oldf);
636
637 if (ch != EOF)
638 {
639     ungetc(ch, stdin);
640     return true;
641 }
642
643 return false;
644 }
```