SOFTWARE REUSE AND ITS EFFECT ON SOFTWARE QUALITY
FOR REAL-TIME GEOMETRIC MEASUREMENT

A Thesis

Submitted to the Faculty of Graduate Studies and Research

In Partial Fulfilment of the Requirements

For the Degree of

Master of Applied Science

in

Software Systems Engineering

University of Regina

by

Seang Buan Cau

Regina, Saskatchewan, Canada

September, 2016

Copyright © 2016: S. B. Cau
Seang Buan Cau, candidate for the degree of Master of Applied Science in Software Systems Engineering, has presented a thesis titled, Software Reuse and Its Effect on Software Quality for Real-Time Geometric Measurement, in an oral examination held on March 14, 2016. The following committee members have found the thesis acceptable in form and content, and that the candidate demonstrated satisfactory knowledge of the subject material.

External Examiner: Dr. Mehran Mehrandezh, Industrial Systems Engineering

Supervisor: Dr. Luigi Benedicenti, Software Systems Engineering

Co-Supervisor: Dr. Raman Paranjape, Electronic Systems Engineering

Committee Member: Dr. Craig Gelowitz, Software Systems Engineering

Committee Member: *Dr. Paitoon Tontiwachwuthikul, Industrial Systems Engineering

Chair of Defense: Dr. Yee-Chung Jin, Environmental Systems Engineering

*Not present at defense
ABSTRACT

The purpose of this thesis is to examine and understand the positive and negative effects of software quality with respect to software reuse during development. This work takes the concept of code reuse and applies it to the development of a complete software package prototype. This prototype will allow the real-time geometric measurement of images during flight mode of an unmanned aerial vehicle (UAV). The prototype is based on the AR.Drone SDK navigation example and incorporates software code from a Physical Measurement Calculator. This case study examines and compares the software quality of the software components prior to reuse and software quality of the prototype after reuse. The ISO 25000 standard model was applied to examine the software quality. Results obtained in this experiment supports the claim that software reuse increases software quality. More importantly, it notes that systematic software reuse and updates to the software that is being reused is required for future software development to improve software quality.
ACKNOWLEDGEMENTS

I sincerely acknowledge Dr. Raman Paranjape and Dr. Luigi Benedicenti, the supervisors for their invaluable advice, supervision, and support. Without their help, faith, and patience, this research would not have been possible.

I am very grateful to Mr. Rui Zhou, Mr. Chayatat Ratanasawanya, and Mr. Mehrdad Bakhtiari for sharing their knowledge and expertise. I have learned a great deal from my fellow graduate students. I would like to thank Mr. Ferguson Earnshaw and Ms. Jiapei Chen for their support and encouragement.
DEDICATION

My deepest gratitude goes to my parents, Mea Nee Cau and Tan Shu Tien, and my siblings for their unconditional love and support.
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABSTRACT</td>
<td>I</td>
</tr>
<tr>
<td>ACKNOWLEDGEMENTS</td>
<td>II</td>
</tr>
<tr>
<td>DEDICATION</td>
<td>III</td>
</tr>
<tr>
<td>TABLE OF CONTENTS</td>
<td>IV</td>
</tr>
<tr>
<td>LIST OF TABLES</td>
<td>VIII</td>
</tr>
<tr>
<td>LIST OF FIGURES</td>
<td>IX</td>
</tr>
<tr>
<td>ACRONYMS</td>
<td>X</td>
</tr>
<tr>
<td>CHAPTER 1: INTRODUCTION</td>
<td>11</td>
</tr>
<tr>
<td>1.1 Motivation for Research</td>
<td>11</td>
</tr>
<tr>
<td>1.2 Research Objectives</td>
<td>12</td>
</tr>
<tr>
<td>1.3 Thesis Contributions</td>
<td>12</td>
</tr>
<tr>
<td>1.4 Thesis Organization</td>
<td>13</td>
</tr>
<tr>
<td>CHAPTER 2: LITERATURE REVIEW</td>
<td>14</td>
</tr>
<tr>
<td>2.1 Software Reuse</td>
<td>14</td>
</tr>
<tr>
<td>2.1.1 Reuse Libraries and Components</td>
<td>14</td>
</tr>
<tr>
<td>2.1.2 Various approaches to software reuse</td>
<td>17</td>
</tr>
<tr>
<td>2.1.3 Domain Engineering</td>
<td>18</td>
</tr>
<tr>
<td>2.1.4 Advantages of Software Reuse</td>
<td>18</td>
</tr>
<tr>
<td>2.1.5 Disadvantages of Software Reuse</td>
<td>19</td>
</tr>
</tbody>
</table>
2.1.6 Reuse Metrics.............................................................................................................. 22

2.2 SOFTWARE QUALITY ...................................................................................................... 22

CHAPTER 3: PROTOTYPE DESIGN DECISIONS ................................................................. 26

3.1 PURPOSE .......................................................................................................................... 26

3.2 REASONS FOR SOFTWARE REUSE METHODOLOGY .............................................. 26

3.3 UNMANNED AERIAL VEHICLE SELECTION ............................................................. 27

3.4 CHOOSING THE CONTROL PROJECT ......................................................................... 27

3.4.1 Review of the different programming languages..................................................... 28

3.5 IMAGE PROCESSING ..................................................................................................... 30

3.5.1 Computer Vision Library Selection ....................................................................... 30

3.5.2 Design Choice for Computer Vision ....................................................................... 31

3.6 OPERATING SYSTEM ................................................................................................. 33

3.6.1 Design Choice for the Operating System............................................................... 34

CHAPTER 4: AR.DRONE ................................................................................................. 36

4.1 AR.DRONE AND AR.DRONE 2.0.................................................................................. 36

4.1.1 AR.Drone Hardware .............................................................................................. 36

4.1.2 AR.Drone Software ............................................................................................... 38

4.1.3 Control Architecture .............................................................................................. 40

4.2 AR.DRONE SDK .......................................................................................................... 45

4.2.1 Video Stream ........................................................................................................... 48
6.2 The Impact of Software Reuse on Software Quality ............................... 77

CHAPTER 7: CONCLUSION AND FUTURE WORK ................................. 79

7.1 Conclusion............................................................................................... 79

7.2 Research Achievements........................................................................... 80

7.3 Future Work.............................................................................................. 80

REFERENCES.................................................................................................. 82

APPENDIX A: EVALUATION REQUIREMENTS ............................................. 88

A.1 Purpose of Evaluation ........................................................................... 88

A.2 Software Product Quality Requirements ............................................... 88

A.3 Software Products.................................................................................... 89

A.4 Stringency of Evaluation ....................................................................... 89

A.5 Quality Measure ..................................................................................... 90

A.5.1 Software Quality Measure Elements ............................................... 91

A.5.2 Mapping between QME and Quality Characteristics ....................... 97
LIST OF TABLES

TABLE 4.1: HARDWARE COMPARISON BETWEEN AR.DRONE AND AR.DRONE 2.0 .......... 37
TABLE 4.2: SOFTWARE COMPARISON BETWEEN AR.DRONE AND AR.DRONE 2.0 ........... 39
TABLE 6.1: METRIC FOR QME1 LINES OF CODE (LOC) USING CLOC [82] ............... 69
TABLE 6.2: METRIC FOR QME2: OPERATIONAL TASK ........................................ 70
TABLE 6.3: BREAKDOWN OF METRIC FOR QME2: OPERATIONAL TASK ................. 71
TABLE 6.4: METRIC FOR QME 4 & QME 5: DEFECT DENSITY .................................. 74
TABLE 6.5: METRIC FOR QME6: NUMBER OF ERROR MESSAGES ....................... 74
TABLE A.1: INITIAL SET OF QMES ................................................................. 90
TABLE A.2: QME 1 – LINES OF CODE .......................................................... 91
TABLE A.3: QME 2 – OPERATION TASKS ...................................................... 92
TABLE A.4: QME 3 – NUMBER OF NON-FATAL DEFECTS .................................. 94
TABLE A.5: QME 4 – NUMBER OF FATAL DEFECTS ......................................... 95
TABLE A.6: QME 5 – NUMBER OF ERROR MESSAGES .................................... 96
TABLE A.7: MAPPING BETWEEN QME AND QUALITY CHARACTERISTICS ............ 97
LIST OF FIGURES

FIGURE 2.1: ISO 25010 SOFTWARE QUALITY CHARACTERISTICS [39] ......................... 24
FIGURE 4.1: BLOCK DIAGRAM OF AR.DRONE CONTROL PRINCIPLE [79] .................. 42
FIGURE 4.2: STATE MACHINE GOVERNING THE FLIGHT AND CONTROL MODES [79] ........ 44
FIGURE 5.1: PHYSICAL MEASUREMENT CALCULATOR ........................................... 54
FIGURE 5.2: EXTRACTED CORNERS USING PHYSICAL MEASUREMENT CALCULATOR ...... 54
FIGURE 5.3: AR.DRONE SDK NAVIGATION EXAMPLE MAIN USER INTERFACE ............. 56
FIGURE 5.4: ORIGINAL VIDEO WINDOW OF AR.DRONE SDK NAVIGATION EXAMPLE ...... 58
FIGURE 5.5: PROTOTYPE’S MODIFIED VIDEO WINDOW ........................................... 59
FIGURE 5.6: ORIGINAL DISTORTED IMAGE .......................................................... 60
FIGURE 5.7: PROTOTYPE’S UNDISTORTED IMAGE .................................................. 61
FIGURE 5.8: REUSED COMPONENTS WITHIN THE Prototype ................................. 64
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>Application Program Interface</td>
</tr>
<tr>
<td>AR.Drone</td>
<td>Augmented Reality Drone</td>
</tr>
<tr>
<td>IHM</td>
<td>Interface Homme Machine (Human-Machine Interface)</td>
</tr>
<tr>
<td>IP</td>
<td>Internet Protocol</td>
</tr>
<tr>
<td>ISO</td>
<td>International Organization for Standardization</td>
</tr>
<tr>
<td>KLOC</td>
<td>Thousand Lines of Codes</td>
</tr>
<tr>
<td>LOC</td>
<td>Lines of Code</td>
</tr>
<tr>
<td>QM</td>
<td>Quality Measure</td>
</tr>
<tr>
<td>QME</td>
<td>Quality Measure Element</td>
</tr>
<tr>
<td>RBG</td>
<td>Red-Green-Blue color space</td>
</tr>
<tr>
<td>SDK</td>
<td>Software Development Kit</td>
</tr>
<tr>
<td>TCP</td>
<td>Transmission Control Protocol</td>
</tr>
<tr>
<td>UAV</td>
<td>Unmanned Aerial Vehicle</td>
</tr>
<tr>
<td>UDP</td>
<td>User Datagram Protocol</td>
</tr>
<tr>
<td>USB</td>
<td>Universal Serial Bus</td>
</tr>
</tbody>
</table>
1.1 Motivation for Research

This thesis examines the creation of a complete software package, referred as the Prototype for the remainder of this thesis, by reusing code from two separate software programs and combining them. The development is to be done using the Physical Measurement Calculator, which is written in MATLAB® and the navigation example from the AR.Drone Software Development Kit (SDK), referred to as the SDK Navigation for the remainder of this thesis. The advantages and disadvantages of code reuse needs to be examined for practicality in the software industry. A study will be conducted on the effectiveness of code reuse and the quality of the AR.Drone SDK navigation, Physical Measurement Calculator, and this Prototype.

It is proposed that code from the Physical Measurement Calculator, which uses geometric triangulation to calculate distance measurements in a planar surface, will be reused in the SDK Navigation software that controls the AR.Drone. The advantages would be a complete software package where users can use the AR.Drone to measure distance in real-time while in-flight.
1.2 Research Objectives

The objective of this research is to provide a practical application of software code reuse and observe its effects on software quality. It will detail the development of a prototype using software reuse as a microcosm of the methodology used in larger software development. The prototype will combine various components to produce an all-in-one platform for unmanned aerial vehicle control and geometric distance measurement.

In addition, a software quality analysis of the prototype will be conducted to evaluate it. This analysis will use a standardized method that is currently used in the industry.

1.3 Thesis Contributions

This thesis covers the primary research on the AR.Drone, AR.Drone 2.0, AR.Drone SDK 1.8 and the AR.Drone SDK 2.0, which was conducted by the UAV research group at the University of Regina. The examination of various projects utilizing the AR.Drone for development purposes provides the foundation to utilize the AR.Drone software as an academic research tool.

By combining two different applications, it is possible to provide a complete software solution that allows the user to use the solution as a standalone product. This will streamline the user process and experience.
The contributions made by this research are:

1) A study of software code reuse strategies and its benefits,

2) A study of software quality metrics, and the

3) Creation of a prototype for measuring distances during flight.

1.4 Thesis Organization

This thesis is organized as follows: Chapter 2 reviews the existing literature on software reuse and its effects on software quality. Chapter 3 details preliminary studies for the development of a prototype that allows the measurement of distances in real-time flight. Further discussion is provided on the various software tools, previously existing control projects for the AR.Drone, and design decisions. Chapter 4 examines the on-board architecture, and open-source software development kit (SDK) for the AR.Drone and AR.Drone 2.0. Chapter 5 details the combination and software reuse of two separate software systems into one complete software package. The development of the Prototype was conducted by the author throughout this research. Chapter 6 evaluates the previously existing software and the resulting software system as a whole. It illustrates the effects of software quality before and after code reuse. It will use the measures as described in ISO 25010. Chapter 7 summarizes the conclusion of this thesis work with possible future work.
CHAPTER 2: LITERATURE REVIEW

This chapter presents a literature review of software reuse and its effects on software quality. Software reuse is not a new concept. It has been widely used and examined. Numerous researchers support the idea that reusing software can significantly reduce software cost and increase software quality. Thus, software reuse should be considered standard practice during software development because it increases software quality by decreasing the number of defects. Section 2.1 presents a literature review of software reuse techniques. Section 2.2 highlights the effects of software reuse on quality.

2.1 Software Reuse

Software reuse, by definition, is the use of existing software or software knowledge to construct new software. Software reuse has been extensively researched and is often traced back to McIlroy’s paper, which proposed that software industry should be based on reusable components. Lin states that software reuse covers the whole process of identification, representation, retrieval, adaptation and integration of reusable components [1] [2].

2.1.1 Reuse Libraries and Components

Prieto-Diaz and Freeman identify three steps that are involved in code reuse: (1) accessing the code, (2) understanding it, and (3) adapting it. They also state that there are
two levels of reuse to consider: the reuse of ideas and knowledge, and the reuse of particular artifacts and components. Haefliger claims that code reuse, a form of knowledge reuse, is fundamental to innovation. Both levels are important in academia and industry. Reusing knowledge and ideas allow knowledge to expand, and adapt previous knowledge and ideas into solving new problems. Reusing particular artifacts and components allows for the application of specific code into solving new problems [3] [4].

In order to create and maintain repositories and libraries, classification is required for software components to ensure quick and easy access to resources. Prieto-Diaz and Freeman have derived a schedule for classifying code fragments based on functionality and environment. The functionality of the code is what it does. The environment is where the code does it. This approach to classification allows the user to quickly search for components based on its use [5] [6] [2] [7].

The Gertrude model, presented by Succi et al., approaches the reuse model by categorizing objects into entities. There are four entities: people, roles, processes, and infrastructures. Processes are a collection of related activities. People are the real employees who perform the activities. The people then play roles to perform the activities. The infrastructures are physical objects, such as equipment and facilities, which are used by the people to perform the activities. Approaches presented by Prieto-Diaz and Freeman, and Succi et al. define methodologies to classify software components for reuse and how to effectively access them for reuse [8].
Xu and Qiang proposed an agent-based system, which uses a component reuse method. Their findings suggest that the design methodologies, models, and components are different based on the different reuse technologies used [9].

In 2011, Niranjan and Guru Rao proposed an intelligent classification and retrieval technique. Their methodology classifies multimedia software components using a genetic algorithm which sorts and categorizes the components based on 9 characteristics: component name, functionality, domain, operating system, algorithm, implementation language, developer, time complexity, and price. However, their intelligent classification and retrieval has only been tested on a small sample of generated data. Further work is required to access the usefulness of their methodology [10].

Xu et al. approaches software reuse using object orientation. They proposed a method of extracting desired components to be stored for future reuse. The first two steps were to find and identify the reusable components. The next steps are to generate the general component, and verify its correctness, integrity, and consistency. The final steps are to create the documentation, and store the component along with its documentation in a repository or library [11].

Recent literature looks at software reuse within an agile development organizations. The findings of Spoelstra et al. show that while organizational software reuse is informal, organizations are able to capitalize on software reuse where applicable. Their proposed management tools gives insight into the adoption of software reuse within agile
development organizations [12] [13].

2.1.2 Various approaches to software reuse

Since software is such a broad term, there many different approaches to software reuse. Some areas that these approaches focus on are embedded systems, functional integration, and domain engineering. Lin took a functional integration approach and proposed a functional integration model (FIM). FIM is a form of black box component reuse. It only requires an understanding of the external operations: the input requirements, and the output results. The internal details of the component are not required [2] [7].

BTOPP (Business, Technology, Organizations, Process, and People) model proposed by Morton et al. looks at software reuse from a business perspective. This acronym, named for the five factors within its model, describes and identifies the five factors that are related reuse including both technical and non-technical aspects [14] [15].

Smith et al. attempts to apply reuse across different robotic platforms. Smith et al. views software reuse as a means to limit the diverse robotic platforms [16].

Hidetoshi et al. attempts to improve software reuse within embedded software by visualizing inter-module relations. Ha et al. presents an empirical study on the effects of embedded software reuse within small and medium enterprises [17] [18].
2.1.3 Domain Engineering

Domain engineering (product line engineering) is a key concept in software reuse, where software systems are not new, and are variants of previously existing systems. These software systems are often part of a business product line, called Domains. The ultimate purpose of domain engineering and software reuse, according Frakes et al., is to improve product quality and maximize profits. Some domain engineering approaches include DARE, FAST, and FORM. DARE (domain analysis and reuse environment) method is used to explore the automation of repeatable process within domain analysis. FAST (family-oriented abstraction, specification, and translation) method, introduced by Lucent Technologies, focuses on the processes for product line engineering. FORM (feature-oriented reuse method) searches for commonalities within features and uses them to design a common framework or architecture across the product line. Domain Engineering is field that has been extensively researched [19] [20] [21] [1].

2.1.4 Advantages of Software Reuse

A major reason for reusing software is the added benefit of increased quality. As early as 1994, Frakes and Isoda state that software reuse is a technology for improving software quality and productivity. They claim that improved quality is a function of the reuse level achieved. The reuse level is defined as the ratio of reused components to total components developed for the system (internal reuse) and outside the system (external reuse). Other benefits include increased development productivity, shortened time-to-
market, and consistent application functionality. Basili et al. concludes that in order to improve software quality, reuse of corporate capabilities such as knowledge and experience needs to exist. These beneficial claims are supported by Lim, where he examines two reuse programs at Hewlett-Packard (HP®) and documents the improved quality, increased productivity and shortened time-to-market. His research concludes that reuse allows the accumulation of defect fixes to improve quality [5] [6] [22] [23].

An empirical study of software reuse versus defect density and stability was undertaken by Mohagheghi et al. to support Lim in proving that reuse correlates to improved quality. In their published paper, they determined that their reused code had almost 50% less defect density than their non-reused code counterpart. Succi et al. discovered a direct correlation that higher levels of reuse correlate to significantly lower defect density. Gupta et al. took a deeper look at the different types of defect density by comparing the reused software and the software that reuses it. All three of these paper draw the same conclusion that there exists a direct correlation between the amount of reuse and the amount of defect density, such that an increase in reuse level causes a decrease in defect density leading to an increase product quality [24] [25] [26].

2.1.5 Disadvantages of Software Reuse

Systematic software reuse in software development firms is not free and requires years of investment according to Frakes and Isoda. Firms need to create and maintain reusable code, populate repositories and libraries. Cost analysis is very important to
determine the success and sustainability of any software reuse program within a firm. The main comparison is the cost of reuse compared to the cost of new software development from scratch. One problem of cost is the scalability of software reuse. As an organization grow, so too does the cost of maintaining a reuse infrastructure. Increasing maintenance costs would make systematic software reuse undesirable. Reusable artifacts and components only has value if it is reused, and only repeated reuse would lower its cost [1] [3] [4] [5] [27] [28].

Jalender et al. notes that there are technical and non-technical impediments to general acceptance of software reuses. The non-technical issues include the lack of support, commitment, encouragement, training, and rewards for software reuse within organizational firms. The difficulty of measuring the gains from reuse will also deter organizations from committing resources to promote reuse. Jalendar et al. note that there are legal issues, such as intellectual property rights, which could impede software reuse. Lynex and Layzell support this claim by stating that intellectual property on software modules restricts the distribution and reuse of these modules [29] [30].

More importantly, technical issues concerning the development of truly reusable components, where the functionality is crystal-clear, is nearly impossible. Software components may not be generic and may not be developed for reuse. It may take more energy for a programmer to learn to use a specific component than it does for the programmer to write new software from scratch. The difficulties of understanding source
code may not be worth the effort to reuse a particular artifact or component. The cost of searching for reusable software could be higher than what it is worth [29] [30] [31].
2.1.6 Reuse Metrics

In order to implement systematic software reuse programs to improve quality, there needs to be measures of progress and effectiveness. Most papers apply a form of cost benefit analysis to justify the effectiveness of software reuse, such as Gaffney and Durek, Barnes and Bollinger, and Poulin et al. Cost benefit analysis, for the business model, helps determine the cost incentive for systematically implementing and reusing software. However, there also needs to be measurements on the models of reuse in terms of its functionality, usability, and storage capabilities. This is where the studies from Beiman and Karunanithi, and Frakes and Nejmeh, become important in measuring and evaluating the internal attributes and assets of software reuse. Beiman and Karunanithi proposed reuse metric for object-oriented systems through the definition of measurable system reuse attributes. Frakes and Nejmeh proposed a metric for measuring the quality of assets within a reuse library. These papers show that not only is software reuse is important, the quality of the reused component and the quality of their classification is just as important for software reuse to be truly effective [4] [7] [32] [33] [34] [35] [36].

2.2 Software Quality

The majority of literature examines defect-density as a measure to define the perceived increase in quality as a direct result of reuse as researched by Mohaghedhi et al., Succi et al., and Lim. Mohagheghi et al. also examines the stability of the software
with respect to the amount of reuse in the software. They conclude that their examination is on the impact of the quality attributes in large industrial projects. Succi et al. notes that the decrease in defect-density and increase in customer satisfaction is associated with the adoption of a corporate software reuse policy. Lim notes that reuse also leads to decreased defect-density and decreased time to market. The commonality of these three papers is that the metrics are different aspects of software quality. These three papers support the concept that systematic software reuse leads to improved quality as a whole [6] [25] [26].

Software Quality is defined by the ISO 25000 standard as the capability of software products to satisfy stated and implied needs when used under specified conditions. Software quality, while being a well sought after goal by many software companies, deals with a wide variety of aspects within software. Traditional software quality improvement requires experience reuse and organizational sharing. These aspects include items from the basic code defects to the user experience. Asili and Caldiera note that software quality is not consistent. Software Quality is a qualitative measure and is mostly subjective. Therefore, the same metric given to two different people can provide different results. As such, qualitative studies of software quality should be considered an average over a large sample size. However, quantitative analysis can be applied to distinct measures [23] [37] [38].
Software product Quality Requirements and Evaluation (SQuaRE) model is the basis of the International Software Organization ISO/IEC 25010 standard model, which replaces the previous ISO/IEC 9126 standard. SQuaRE contains 8 quality characteristics as shown in Fig. 2.1. This international standard contains an explanation of how to apply software and computer system quality measures, and a basic set of quality measures for each characteristic. However, it falls short of providing the range of values for these quality measures. The standard states that the values need to be defined for each product or a part of the product. As such, specific values may only be valid in specific instances, and not generally applicable for all products [39] [40] [41].

The conclusion of Singh and Kannojia's assessment, after the comparison of McCall's, Boehm's, Dromey's, and ISO 9126, indicates that quality models can result in optimal quality software when the software product model satisfy their need within specified environment. Their research of various software quality models provided only five consistent software qualities: reliability, usability, performance efficiency, maintainability, and portability. Although these five qualities are a commons to all models, the other qualities should not be ignored. The ISO standard states that the quality
measurements defined within it are guidelines only. There is no definitive best quality measurement given that software is designed for a specific goal in a specified environment. Khoshgoftaar and Gao observe that the objective of software quality modeling is to seek the underlying relationship between the software metrics and the program faults. Thus, the ultimate goal of any software quality metric is use the quality analysis of a software product to determine its weaknesses and strengths. The weaknesses within a program can then be examined and improved upon [42] [43].
CHAPTER 3: PROTOTYPE DESIGN DECISIONS

3.1 Purpose

The purpose of this experiment is to develop a prototype as part of an examination on software reuse and its effects on software quality. The prototype is an unmanned aerial vehicle (UAV), which can be deployed and simultaneously provide geometric measurements during flight mode. Current methodology is to obtain the images and then apply photogrammetry after the initial flight has already taken the desired images [44] [45].

Analysis of the prototype will be conducted as a study of component reuse and its effect on software quality.

3.2 Reasons for Software Reuse Methodology

The reason for reuse application in this project is to streamline the development process. By utilizing components from other projects, and examples, it would allow for faster prototyping. The complexities of designing a control system component as well as a photogrammetry component for the sake of a proof of concept is unrealistic. After a proof of concept, code reuse may or may not be relevant in the development of a final product. This exercise will examine the advantages and disadvantages software reuse will have on software development and software quality. Given the rich history of software
reuse, it is relevant to re-examine whether this software process is still valid. And if it can still be applied in specific situations.

3.3 Unmanned Aerial Vehicle Selection

Other quadruple rotor helicopters were evaluated and tested before settling on the use of the Parrot® AR.Drone. The Q-Ball, developed by Quansar, was too expensive and unreliable. It would randomly reset itself, and maintenance became time-consuming. The Dragonflyer, developed by Dragonfly Innovations, was a closed system with inaccessible code and sensor data. Lastly, the AR.Drone was an inexpensive alternative with a relatively open operating system.

3.4 Choosing the Control Project

Different projects were considered for this project. They are the Helisimple project, the Autopylot project, and the navigation example from the AR.Drone Software Development Kit (SDK).

The Helisimple project was developed by Juan Rodriguez on 2 Nov 2011. It was written in C++ using the AR.Drone SDK libraries. It is a standalone project that utilizes the keyboard or a joystick for its control system. Helisimple uses the AR.Drone SDK library, which features the basic functionality of communication protocols, keyboard mapped command structure for flying the AR.Drone, and a GUI to display a live feed from the onboard cameras. Helisimple is very simplistic in its code. It can process the
navigational data from AR.Drone feedback [46] [47].

Autopilot is written by Levy and Stough. They used Python for the source code and supports both MATLAB® and C. The project is supported on a 64-bit Ubuntu operating system. There exists source code for Autopilot to track a green ball in 3D space. An example video of the tracking system is available on YouTube. This project is available under the GNU Lesser General Public License [48] [49].

The AR.Drone SDK navigation example was the third control platform reviewed. The navigation example provides examples on the use of the library and sensor data. The navigation example is written in C and provides different threads for handling the sensor data, video streams, user interface, and control.

Helisimple uses C++ code which would require modifications to the build process when incorporating the C based OpenCV. Similarly, Autopilot would be using SimpleCV as both are based on Python. The Autopilot was not used for development because it was written in python, with would have been incompatible with the C based OpenCV.

### 3.4.1 Review of the different programming languages

An important factor in the decision to use the SDK navigation example was the programming language the control project was written in. The three projects reviewed were written in three different languages.

The Autopilot project was developed by Levy using the Python programming
language. Python has simpler syntax and designed for both programmers and non-programmers alike [50].

The Helisimple project was written in C++. C++ was written by Stroustrup at Bell Labs during 1983-1985. Stroustrup added class features to C and combined Simula's use of classes and object-oriented features with the power and efficiency of C [51] [52].

The C programming language, used in the SDK navigation example, is a minimalistic programming language that has low-level access to memory. It could also be compiled by a relatively simple compiler, and does not require extensive run-time support. This structured oriented programming language focuses on the procedural programming paradigm, making it ideal for small scale programs like the ones used for this thesis [51] [53] [52].

Although C++ has more features than C (such as encapsulation, multiple inheritance, and polymorphism), C++ is prone to data type related errors because it does not offer strong type-checking, and not a pure object-oriented programming language as it doesn't have the feature of garbage collection. C++ is more suited to large systems compared with C language. C++ is not platform independent, and can't run on all platforms. Accord the Hao, C is more suited to systems-programming applications, and industrial automation whereas C++ is more suited for application software, and device drivers [51] [52].

For the purposes for this prototype, the C programming language along with the
SDK navigation example was chosen. The Python based Autopylot was dropped due to the combined need to know both Python and C as it also used the AR.Drone SDK library, which is based in C. Similarly, The C++ based Helisimple program also uses the AR.Drone SDK library. Since all three project incorporates the AR.Drone SDK library. The design to utilize only one programming language should increase the simplicity of the design and decrease the difficulty of combining different reusable components.

3.5 Image Processing

3.5.1 Computer Vision Library Selection

OpenCV, an open-source library released in 1999, was conceived as a way to make computer vision infrastructure universally available. One of OpenCV’s goals is to provide a simple-to-use computer vision infrastructure that helps people build fairly sophisticated vision applications quickly. OpenCV, consisted of several hundreds of computer vision algorithms, was used to parse and manipulate the image stream coming in from the AR.Drone. Since OpenCV is BSD-licensed, it can be freely download and used in an academic environment. The modular structure of OpenCV means that the package includes several share or static libraries [54] [55].

SimpleCV and Accord.Net libraries were considered as an alternatives to OpenCV. SimpleCV was developed to provide simple to use computer vision. The development of SimpleCV is largely based on OpenCV. As the installation process, OpenCV is a major
dependency. SimpleCV was created using the Python Framework. Python was chosen for SimpleCV because of its simplicity and easy learning curve. As an open sourced library like OpenCV, SimpleCV is available to anyone who wishes to use it. It is simple and easy use. It also contains libraries for higher level functions. SimpleCV allows the user to use computer vision without having to know about bit depths, file formats, colour spaces, buffer management, eigenvalues, or matrix versus bitmap storage [56] [57] [58] [59] [60] [61].

The Accord.NET framework is a generalized framework created for scientific computing. It is used for building machine learning, computer vision, computer audition, signal processing, and statistical applications. The main aspect focused on Accord.NET is its computer vision library. The image processing libraries are built upon the AForge.NET framework [62] [63] [64].

3.5.2 Design Choice for Computer Vision

There are multiple reasons OpenCV was chosen as the framework for computer vision. The first reason OpenCV was chosen is because the majority of the UAV research group at the University of Regina was already using OpenCV. The second reason OpenCV was chosen was because the framework has interfaces to C++, C, Python, Java, and MATLAB®. This provided multiple platforms for which OpenCV could be used. Since the AR.Drone SDK was written in C, the OpenCV libraries were easily implemented into the system.
An advantage of SimpleCV is that the user does not have to worry about the lower level programming. The disadvantage is that this prevents the user from optimizing the code, and delving deeper into computer vision. Developed using the Python programming language, SimpleCV aims at new users who are unfamiliar with computer vision and/or programming. The disadvantage of SimpleCV is there is very little documentation available since it is relatively new.

One of the advantages is that there exist multiple academic publications using the Accord.NET framework. The .NET framework is designed for Windows®. However, the source code is available for Linux users willing to run the framework on Mono. Accord.NET is licensed under the GNU Lesser General Public License. The assemblies are automatically registered to be available under Visual Studio's Add Reference Dialog. It is designed to work very well within Visual Studio's .NET framework. The main disadvantage of Accord.NET is that it runs in the .NET framework and requires it to work optimally. There is only limited support for MonoDevelop, a cross platform IDE, on Linux [65] [66].

OpenCV was chosen for this project because of its set of prebuilt functions and memory allocation capability. It handles all the memory allocation and de-allocation automatically most of the time. Memory allocation is very important factor for UAVs, especially if we wish to move the control platform onto the Drone itself. Even when the drone is controlled by the base station as in this case, improper memory allocation and
de-allocation will cause segmentation faults. Proper de-allocation of memory space will prevent the segmentation faults. OpenCV was designed for computational efficiency and with a strong focus on real-time applications OpenCV was written in optimized C and can take advantage of multi-core processors. Real-time applications, such as manipulating images while the drone is in flight, require efficient computation so that latency does not affect the control system. Since OpenCV was written in C, it blends naturally into the C environment of the AR.Drone SDK. Since OpenCV deals with image processing, Code utilizing OpenCV libraries exists primarily in the video thread of the SDK. This makes OpenCV ideal for fast computations during flight mode of the UAV [55].

Finally, the documentation on using OpenCV with the AR.Drone SDK were abundant and readily available. Naturally, it became clear that OpenCV was the framework to use for this project [67][68].

3.6 Operating System

Two operating systems were experimented with during this project, Microsoft Windows® XP and Ubuntu. Windows® is a commercially available product by Microsoft with a multitude of commercially available products for any program and purpose. At the time of this project, the dominate operating system is still Windows® XP, with Windows® 7 being commercially available. A stripped down version of Visual Studio,
Microsoft Visual Studio Express 2013, is freeware and available for download from the Microsoft website [69] [70] [71].

Ubuntu is a flavour of Linux that is quite popular. The version of Ubuntu used for the experiments is 12.04.5 LTS. LTS stands for Long Term Support [72] [73]. Ubuntu is a free operating system where the commercial and community teams collaborate to produce a single, high-quality release, which receives regular maintenance [74].

3.6.1 Design Choice for the Operating System

There were a numerous reasons why Ubuntu, a flavour of Linux, was chosen over Windows®. These reasons include developer familiarity with Ubuntu, and native gcc compiler for AR.Drone Linux Examples.

The example that came with AR.Drone SDK 1.8 had multiple errors during compilation. The AR.Drone SDK 1.8 had a Windows® example. In order to get the Windows® example working, extensive modifications were made to the program. File links needed to be created within Microsoft Visual Studio Express 2008 for the AR.Drone SDK 1.8 to finally work in Microsoft Visual Studio. Upon the release of AR.Drone SDK 2.0 for AR.Drone 2.0 development, the Windows® example was removed [75].

AR.Drone SDK versions 1.8 and 2.0 both had Linux examples that worked as long as the required dependencies were installed correctly. Therefore, Ubuntu seemed the logical choice.
CHAPTER 4: AR.Drone

4.1 AR.Drone and AR.Drone 2.0

The AR.Drone was the first model released commercially. The second model released was AR.Drone 2.0. Although both drones share many similarities, there are slight changes in the hardware, and consequently the software API changed as well. The AR.Drone was chosen as a research tool because it was a commercially available, cost effective, and easily replaceable platform for UAV development.

4.1.1 AR.Drone Hardware

The AR.Drone has two built-in mono-cameras which can be used for vision-based navigation. The camera sizes on the drones are different for AR.Drone and AR.Drone 2.0 models. Both drones consists of a carbon-fibre support structure, plastic body, four high-efficiency brush-less motors, sensor and control board, two cameras, and indoor and outdoor removable hulls [76] [77]. The firmware of both models contains sensor fusion and control technology for stability of the drone. The motor speeds are automatically adjusted by the control board based on input of yaw, pitch, roll and vertical speeds. The control board also provides access to preprocessed sensory measurements and images from the on-board cameras.

The drone sensory equipment consists of a 6 degree-of-freedom (DOF), micro-
electromechanical system (MEMS) based, miniaturized inertial measurement unit (IMU), and a sonar-based altimeter. Data from the 2-axis gyros, IDF-400 or Invensense IDG500, and a Bosch BMA150 3-axis accelerometer is fused to provide accurate pitch and roll, the yaw motion is measured by a 1-axis high precision gyro, the XB-3500CV or Epson XV3700. The sonar-based altimeter is an ultrasound telemeter which provides altitude measures for automatic altitude stabilization and assisted vertical speed control.

**Table 4.1: Hardware Comparison between AR.Drone and AR.Drone 2.0.**

<table>
<thead>
<tr>
<th></th>
<th>AR.Drone</th>
<th>AR.Drone 2.0</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Front Camera</strong></td>
<td>- 93 wide-angle diagonal lens CMOS sensor</td>
<td>- 90 wide-angle diagonal lens CMOS sensor</td>
</tr>
<tr>
<td></td>
<td>- 15 fps video frequency</td>
<td>- 15 or 30 fps video frequency</td>
</tr>
<tr>
<td></td>
<td>- 640x480 pixels resolution (VGA)</td>
<td>- 1280x720 pixels resolution (720p)</td>
</tr>
<tr>
<td><strong>Bottom Camera</strong></td>
<td>- 64 wide-angle diagonal lens, CMOS sensor</td>
<td>- 64 wide-angle diagonal lens, CMOS sensor</td>
</tr>
<tr>
<td></td>
<td>- 60 fps video frequency</td>
<td>- 60 fps video frequency</td>
</tr>
<tr>
<td></td>
<td>- 176x144 pixels resolution (QCIF)</td>
<td>- 320x240 pixels resolution (QVGA)</td>
</tr>
<tr>
<td><strong>Embedded Computer System</strong></td>
<td>- Processor ARM9 RISC 32 bits – 468MHz</td>
<td>- CPU OMAP 3630 1GHz Arm cortex A8</td>
</tr>
<tr>
<td></td>
<td>- DDR RAM 128 MB – 200 MHz</td>
<td>- DDR SDRAM 128 MB</td>
</tr>
<tr>
<td></td>
<td>- Wi-Fi b/g</td>
<td>- NAND Flash memory 128MB</td>
</tr>
<tr>
<td></td>
<td>- Linux OS</td>
<td>- Wi-Fi b/g/n</td>
</tr>
<tr>
<td><strong>Ultrasound Altimeter</strong></td>
<td>- 40kHz emission frequency</td>
<td>- 40kHz emission frequency</td>
</tr>
<tr>
<td></td>
<td>- 6 meters range</td>
<td>- 6 meters range</td>
</tr>
</tbody>
</table>
Weight
| indoor hull: 420g | indoor hull: 436g |
| outdoor hull: 380g | outdoor hull: 400g |

The bottom camera provides ground speed measurements for automatic hovering and trimming. AR.Drone 2.0 adds a 3 DOF to the IMU with a 3-axis magnetometer, which is necessary for “Absolute Control” mode, and adds a pressure sensor for altitude measurement at any height [76] [78] [75] [79].

Upgrading to the AR.Drone 2.0 was necessary and practical. Upon the release of AR.Drone 2.0, the original AR.Drone became obsolete. Although replacement parts were still available, it was more cost effective at times to purchase a new replacement machine rather than to purchase individual parts. The primary advantage of AR.Drone 2.0 is the quality of the camera. The camera onboard the AR.Drone 2.0 had higher resolution than its predecessor. The second advantage of the AR.Drone 2.0 was a redesigned battery pack. Although still compatible with the original batteries for the original AR.Drone, it contained one less wire for charging and a more compact charger.

4.1.2 AR.Drone Software

In order to handle the changes in the hardware between AR.Drone and AR.Drone 2.0, changes also were required to change on the software side as well. The Linux kernel was updated and the image processing code were also revamped to incorporate the HD camera as shown in Table 4.2.
The wireless network now supports Wireless N as part of the Wi-Fi connection to AR.Drone 2.0. The status and sensory data on the AR.Drone are updated at a rate of 30Hz. The communication between the drones and the computer station has not changed.

Table 4.2: Software Comparison between AR.Drone and AR.Drone 2.0

<table>
<thead>
<tr>
<th></th>
<th>AR.Drone</th>
<th>AR.Drone 2.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>Busy Box GNU/Linux Distribution</td>
<td>- Linux 2.6.27</td>
<td>- Linux 2.6.32</td>
</tr>
<tr>
<td>AR.Drone SDK Versions</td>
<td>1.x</td>
<td>2.x</td>
</tr>
<tr>
<td>Front Camera Video Stream</td>
<td>- QVGA 320x240 images</td>
<td>- 360p (640x360) or 720p (1280x720) images</td>
</tr>
<tr>
<td>Bottom Camera Video Stream</td>
<td>- streamed at full resolution of 176x144.</td>
<td>- Up-scaled images to 360p or 720p.</td>
</tr>
</tbody>
</table>

The video stream provided by the AR.Drone and AR.Drone 2.0 firmware only provides images from either the front camera or the bottom camera. It does not provide simultaneous streams from both cameras at the same time. Both drones allow for the video channel to switch between the front camera and bottom camera. However, AR.Drone has two options available in the video channel which the AR.Drone 2.0 does not support. AR.Drone provides two picture-in-picture options: the front camera image in the top left corner overlaying the bottom camera image or the bottom camera image in the top left corner overlaying the front camera image [75].
The onboard server for both AR.Drone versions can be accessed using telnet. This allows the user to change the settings of the on-board operating system and manually adjust configuration files of the drone internal controllers. Therefore, it is possible to cross-compile an application for the ARM processor and run it directly on the AR.Drone control board. Custom on-board applications will need to take into account the limited memory and computational limits of the control board when developing low-level control architecture. The pre-existing control architecture is discussed in Section 1.1.3 [76].

4.1.3 Control Architecture

Although the control architecture for AR.Drone and AR.Drone 2.0 is obscured from the end user, there exists documentation for the original AR.Drone architecture. The on-board architecture of the AR.Drone contains 3 nested loops: hovering control loop, attitude control loop, and the angular rate control loop as illustrated in Fig. 4.1: Block Diagram of AR.Drone Control Principal. The angular rate control loop regulates the motors through a simple proportional integral (PI) controller. This is the base loop for the drone. The intermediate loop is the attitude control loop which governs the height of the drone. The outer loop, hovering control loop, keeps the drone in static position when no inputs are received by the end user (Pilot) [79].

The user’s input (Pilot in Fig. 4.2) to control the AR.Drone prompts reference computations for the angle (yaw, roll, and pitch), the altitude, and the vertical velocity (Vz). These computations allow the AR.Drone maintain its stability. The computations
are offset using the hover control loop which provides the feedback to correct any over or under computations of the required control signal. Two main components allow for the feedback correction: video camera and ultrasonic sensor. The ultrasonic sensor provides the estimation of the current height of the AR.Drone and allows it to maintain its height. The Video Camera uses optical flow to provide feedback regarding the horizontal drift of the AR.Drone. Any drift will result in optical (vision) feedback allowing the AR.Drone perform corrections to horizontal drift.

The altitude control loop shown in Fig. 4.2 provides the feedback loop that allows the system to correct any vertical drift. This loops uses the accelerometer to provide information regarding changes to the velocity of the AR.Drone. These changes feeds back into the estimation of the altitude and vertical speed. The accelerometer also provides feedback to the horizontal velocity estimation which is used the hover control loop.

The gyrometer allows the AR.Drone to detect any rotation that may occur. The information about the rotation feeds back into the angular rate corrector in the angular rate control loop and the altitude estimator within the altitude control loop.

Ultimately, these numerous feedback loops provides the control loops the exact values that the PWM (pulse width modular) needs to control the four motors to allow the AR.Drone to fly, hover, takeoff, or land.
Figure 4.1: Block Diagram of AR.Drone Control Principle [79]
The interaction between these loops is governed by the state machine illustrated in Fig. 4.2. Basically, the state machine determines the behaviour of the drone. Once in flight, the drone is governed by two guidance modes: flight mode and hovering mode.

The transition between these two modes are governed by the outfox motion planning technique. The motion planning technique comes into effect when the current altitude and speed leaves the flight mode and enters into the hovering mode. This technique is vital to
the drone's ability to stop in mid-air and hover without overshoot [79].

**Figure 4.2: State Machine governing the flight and control modes [79]**
Although little publication has been available on AR.Drone 2.0, there is significant improvement its stability and controls. It is noticeable that AR.Drone 2.0 maintains its hovering position in the real world better than its predecessor.

An additional control system called “Absolute Control” has been added to the AR.Drone 2.0. This control allows AR.Drone 2.0 to use the control station as a reference point rather than itself. The reference point for AR.Drone has always been itself due to the lack of the additional 3-axis magnetometer.

Vision-based navigation is built upon this control architecture and uses high level control commands which governs the speed of the pitch, roll, and yaw motion, and altitude change. High level control of the drones can be performed through the use of the SDK API.

4.2 AR.Drone SDK

The AR.Drone SDK was release by Parrot® to encourage initial software development in support of the AR.Drone. The AR.Drone SDK 2.0 was released in conjunction with the AR.Drone 2.0 due to changes in the hardware and software. Although the AR.Drone SDK 2.0 is designed for AR.Drone 2.0, it is still backwards compatible with AR.Drone. One difference is the AR.Drone SDK 1.8 is unable to render the image stream from AR.Drone 2.0 properly due to changes in the hardware configuration. Despite the enhancements made to AR.Drone 2.0, the fundamentals of the
SDK remain the same [78] [75] [46].

The contents of the SDK 2.0 libraries are for both AR.Drone and AR.Drone 2.0 unless specified. All AR.Drone SDK 1.x libraries are included in the AR.Drone SDK 2.0. [78] [75]. The libraries are:

- **Soft**: drone-specific code including:
  - **COMMON**: header (.h) files describing the communication structures used by the drone
  - **Lib/ardrone_tool**: a set of tools to easily manage the drone, including a AT command sending loop and thread, a navdata receiving thread, a ready to use video pipeline, and a ready to use main function
  - **Lib/utils**: a set of utilities for writing applications around the drone

- **VLIB**: the AR.Drone 1.0 video processing library. It contains the functions to receive and decode the video stream

- **FFMPEG**: a complete snapshot of the FFMPEG library, with associated build scripts for the AR.Drone 2.0 applications. Not present in the SDK 1.x series

- **ITTIAM**: a prebuilt, highly optimized, video decoding library for iOS and Android applications. Not present in the SDK 1.x series

- **VP_SDK**: a set of general purpose libraries, including:
The ardrone_tool part in the Soft library is very important as it contains:

- `ardrone_tool.c`: a ready-to-use `ardrone_tool_main` C function which initializes the Wi-Fi network and initiates all communications with the drone
- `UI`: contains a ready-to-use game-pad management code
- `AT`: contains all the functions you can call to control the drone
- `NAVDATA`: contains a ready-to-use data receiving and decoding system
- `ACADEMY`: contains a ready-to-use downloading and uploading system for the upcoming AR.Drone Academy. The downloading system also manages the AR.Drone photo shooting. This library is not present in the SDK 1.x series.
- `VIDEO`: contains all functions related to video receiving, decoding, and recording, for the both AR.Drone and AR.Drone 2.0.
- `CONTROL`: contains a ready-to-use AR.Drone configuration management tool [78] [75].

Since a higher level control structure is being implemented, the necessity to
understand the intricacies of the libraries is not required. The libraries are already processed and implemented in the sdk_demo and navigation programs provided by the SDK. However, an understanding of extracting and manipulating the data from with the sdk_demo is required [78] [75].

There are three main parts to the sdk_demo program. These parts, requiring modification for vision-based navigation, consists of the video stream, navigation and sensor data, and the user interface. Each of these part are in the own respective threads. The threads are declared, initialized, and relinquished in the ardrone_testing_tool.c file [78] [75].

4.2.1 Video Stream

The video stream is processed in the video_stage.c file. The required functions for the video thread are:

• output_gtk_stage_open(void *cfg, vp_api_io_data_t *in,
  vp_api_io_data_t *out);

• output_gtk_stage_transform(void *cfg, vp_api_io_data_t *in,
  vp_api_io_data_t *out);

• output_gtk_stage_close(void *cfg, vp_api_io_data_t *in,
  vp_api_io_data_t *out);
• DEFINE_THREAD_ROUTINE(video_stage, data);

The video pipeline is defined in DEFINE_THREAD_ROUTINE. This routine also contains the picture metadata as well as the video stages required to process the image information. The image is streamed from the drone as a YUV image and then converted to an RGB image through the vp_stages library. The result is an RGB image in the form of a pointer, (uint8_t*) in->buffer[0], to the memory allocation of 8bit per pixel colour to define the 256 colour scheme. The result is a list of pixels in the form of red, green, blue, red, green, blue, and so on from left to right and top to bottom. The row-stride, defined in the SDK Developer Guide, has to be taken into consideration during the processing of the image [78] [75].

Image manipulation will occur the in the output_gtk_stage_transform. Any computational result will need to pass the control commands to the user interface file, ardrone_ini.c. The control commands needs to be stored in the header file as a global variable so that the ardrone_ini loop can access them [78] [75].

4.2.2 Navigation and Sensor Data

The navigation and sensor data is processed in the navdata_ihm.c file. The required functions for the navigation and sensor data, navdata, are:

• demo_navdata_client_init( void *data);

• demo_navdata_client_process( const navdata_unpacked_t* const navdata);
• demo_navdata_client_release(void);

These functions need to be registered in the navdata_client library using the code:

BEGIN_NAVDATA_HANDLER_TABLE

NAVDATA_HANDLER_TABLE_ENTRY(demo_navdata_client_init,
demo_navdata_client_process, demo_navdata_client_release, NULL)

END_NAVDATA_HANDLER_TABLE

The navdata parameter in the demo_navdata_client_process function contains the post-processed on-board sensor information. This information is a user defined class obtained from the ardrone_navdata_client library located in the ardrone_tool folder. Further processing of this information for navigational autonomy needs to be processed in the demo_navdata_client_process function. Global variables need to be declared the header file, navdata.h, so that information can be shared with other threads [78] [75].

4.2.3 User Interface

Finally, the user interface folder contains the high level control system of the SDK. The user interface commands are processed by functions implemented by either the ardrone_ini.c or gamepad.c file. The file names for these control functions are different based on the version of SDK and the examples provided by the SDK. However the ardrone_api library for controlling the drones is the same for both AR.Drone and AR.Drone 2.0 [78] [75].
The commands are sent to the drones using the function:

```c
void ardrone_at_set_progress_cmd( int32_t flag, float32_t phi, float32_t theta,
float32_t gaz, float32_t yaw);
```

The variables phi, theta, gaz, and yaw correspond to roll, pitch, altitude, and yaw respectively. In addition to the ardrone_api library, the ardrone_input library contains two important functions.

- `ardrone_tool_set_ui_pad_start(int32_t value);`
- `ardrone_tool_set_ui_pad_select(int32_t value);`

The `ardrone_tool_set_ui_pad_start` function tells the drones to take off, 1, or land, 0. The `ardrone_tool_set_ui_pad_select` function is used for emergency landing and resetting the drones [78] [75].

### 4.2.4 Generating New Threads

New threads can be generated by modifying the ardrone_testing_tool.c file. Parrot® has incorporated its own thread managing system, which is handled by `vp_os_thread` of the VP_SDK library. The following commands are required to create a new thread in the `ardrone_tool_init_custom()` function. The follow command is added in the `ardrone_tool_shutdown_custom()` function. Threads are declared at the end of the ardrone_testing_tool.c file [78] [75].

- 51 -
START_THREAD( thread_name, null );

JOIN_THREAD( thread_name );

BEGIN_THREAD_TABLE

   THREAD_TABLE_ENTRY( thread_name, counter );

END_THREAD_TABLE
CHAPTER 5: PROTOTYPE

5.1 Background

Photogrammetric measurement using geometric triangulation allowed for the calculation of distances using a camera and a calibration board. In Zhou’s Physical Measurement Calculator software as depicted in Fig. 5.1 and Fig. 5.2, the methodology and algorithm applied is considered a block box in terms of this thesis paper. The goal of this study is not to improve upon the quality of the measurement, but to increase the value of his software code by reusing key components and combining it with the control system of the AR.Drone. By combining incorporating the code of Physical Measurement Calculator, a streamlined process of obtaining images and measuring distances became possible [44].
This chapter details the software reuse process and code integration used in creating
5.2 User Interface Modification

5.2.1 User Interface Problem

The current user interface for displaying the video stream is through the GTK library. The GTK window used in the Linux example allows for decoding the image stream and displaying said stream into an image widget. However, the GTK library is insufficient to allow user interaction with the video display. For the metric measurement to be feasible, the user must be able to select two pixel points within the image. The 2-Dimensional Geometric Triangulation algorithm will be able to calculate the metric distance between the selected two points. Another problem is that the selection of pixel points during a live video stream adds another level of unnecessary complexity.

5.2.2 Solution/Implementation

To solve the problem of user interaction with the video stream, a new type of graphic manipulation library is required. Thus, OpenCV window is incorporated because it has its own window library. Established and well-known low-level components such as graphical toolkits (GTK) have been proven useful to a large audience of users and developers. This project includes OpenCV as a component. Prior to populating the window, the image stream sent from the AR.Drone needs to be decoded [4].
The image stream is sent in packets of unsigned 8 bit integer numbers. The data packet retrieved from the Drone needs to be decoded and stored in a data variable that is able to represent the data retrieved. IplImage is an OpenCV predefined variable and uses unsigned 8 bit integer to convert into the predefined variable. IplImage contains a pointer, imageData, that points to an array of the unsigned 8-bit integer [80].

Since the image stream does not contain the size of the image it is streaming, the data size is given to the base station, but the height and width are unknown. The system needs to read the stream to know the data size. By comparing the data-size and the known height and width dimensions according the SDK documentation, the height and width can
be correlated to the data size. This is evident in the getPicSizeFromRGB24BufferSize.

After obtaining the image stream, the system needs to be able to display the video in a

graphical window. This process reuses the external code provide by Kout [80].

The solution is to create button in the video stream window to generate a new

thread to render the selected frame and use that frame for image processing. The video

window is accessible from the button from the main interface of the SDK Navigation as

shown in Figure 5.3. The additional button, created in ihm_vision.c, allows the creation

of an OpenCV window showing the selected frame to calculate distances in the image.

Creating an OpenCV image window uses the cvCreateImage function. This function

allows the unsigned 8 bit integer image stream to be read and converted to an IplImage
data type. An image stream is an array of pixels stored using a red-green-blue colour
scheme. The CV Image is continually updating the window with new picture frames as it

is being received from the Drone unit. The endless loop that updates the image window is

the contained within its own thread. The thread is managed by the thread library of the

SDK.
Figure 5.4: Original Video Window of AR.Drone SDK navigation example
Since the image window is constantly updating, a Boolean check is incorporated for the user to stop the image window from updating and thus freeze the video frame. The frozen image frame allows the user with work with a static image to select points rather than a live stream where the points of interest are constantly in motion. The image frame

**Figure 5.5: Prototype’s modified Video Window**
is also undistorted to allow for better accuracy. This distortion from the camera can be evidenced by comparing the original image in Fig. 5.6 and the corrected undistorted image in Fig. 5.7. By incorporating the calculations of geometric distance algorithm, the distance between two points within an image is determined by clicking on the image points.

![Figure 5.6: Original distorted image](image-url)
Figure 5.7: Prototype's undistorted image
5.2.3 Reusing Geometric Triangulation Code

This software system allows the user to select two points on the image and calculates the metric distance between them. The camera’s intrinsic parameters are calculated using the MATLAB® toolbox in the Physical Measurement Calculator. The parameters are then used in the metric calculation utilizing the same equations. The equations are reused codes obtained from the MATLAB® code [44].

Since this research is conducted on a smaller scale than an industrial application of software reuse processes, accessing code is informal and often ad-hoc. The lack of a classification scheme, and existing reuse library, did not hinder the development of this prototype. However, the lack of documentation does hinder the process of understanding the code segment, or components being reused.

The implementation of the MATLAB® code required further image manipulation. Since there was no graphical interface other than the image display, the data had to be superimposed upon the image. The function (in OpenCV) that allows the display of text within an image window was used.

5.3 Complete Software Package Prototype

The incorporation of the geometric triangulation code is the final piece to providing a complete software product. This integration is hopeful for streamlining the use of drones in geometric measurement. It was eliminates the need for two separate software to
do achieve the same result. Rather than using two systems where the first is to use a UAV to fly and obtain aerial photographs, and then use a secondary application to analyze the photograph. This system allows the user to measure distances while the UAV is in flight.

5.3.1 Applications of Software Reuse

There were many types of software reuse were incorporated into the development of this prototype. Knowledge reuse was applied through transfer of knowledge between graduate students, and literary research. External and Internal Code Reuse are also evident in this project. External code includes reusing code segments from online forums and websites. Internal code reuse involve the reuse of the AR.Drone SDK source code, and source code from the physical measurement calculator. Component reuse of the OpenCV library has also been incorporated into this project.

Prototype
In Fig. 5.8, internal reuse of the physical measurement algorithm from Zhou’s Physical Calculator, and the navigation example accounts for 0.1% and 98.1%, respectively, of code in the Prototype. External reuse of online resources for decoding the video stream and displaying it in an OpenCV window accounts for 0.5% of the source code. New code accounts for 1.3% of the total code of the prototype, whereas 98.7% of the code is reused.

The majority of the code reuse falls into the white box type. The details of the reused code was accessible and modifiable. The libraries, including OpenCV, cannot be modified. The only modified component was the IHM (Human-Machine Interface) module of the navigation example. This module was where the reused components were incorporated and were the new code was implemented to utilize the reused components.

### 5.3.2 Findings from Software Reuse

After applying software reuse in this prototype development, it was found that software reuse is time consuming and requires a conscientious approach to its implementation to achieve long term benefits. Software reuse time includes the time spent on searching for the correct component to reuse, learning how the component works, and how to implement the component. Time was also spent on learning about
components that ultimately were not used or implemented. The necessity for solving the non-technical issues of software reuse such as organizational support and structure for reuse is paramount. Without the proper resources and infrastructure, reuse could become more costly than writing new software.

One aspect of software reuse that was not researched or implemented was categorizing reused components for future reuse. It would have definitely added to the time and energy spent into the reuse strategy. Applying categorizations of labeling components is useful for systematic software reuse where the repeated reuse of a particular artifact or component increases its value in the long term. However, in the short term, systematic reuse is not of benefit.

5.3.3 Testing Methods

Testing during the development of this prototype was informal. The primary testing method was to use console print lines to view the output of functions and locate the location of the faults. Tests were used on decoding the video stream, applying the OpenCV SDK, and integration of the Physical Measurement Calculator. The development of the Prototype did not use a development platform. Therefore, a debugger that allows the programmer to step through lines of code was not used.
CHAPTER 6: SOFTWARE QUALITY EVALUATION

6.1 Software Quality Characteristics

Measurements should have a clearly defined purpose. The defined purpose of this quality assessment is to evaluate and compare qualities of the AR.Drone SDK navigation, Physical Measurement Calculator, and this prototype of a complete software product, which reuses code from the previous two products, to discuss and improve future developments [42] [81].

In order to assess the positive and negative effects, the quality assessment must be standard across all three systems. To achieve a definitive assessment, it is necessary to ensure quality requirements and measures are applicable to across all three software products. It is unnecessary to examine all aspects of software quality. Measurements of software quality aspects must not be employed purely to accumulate data. Measurements should be comparable and give meaning to the data. The data should lead to understanding the proficiencies and deficiencies of software reuse. If the measurements cannot be compared, then the measurement does not provide clues to the positive or negative effects of software reuse. The applicable ISO 25010 Software Quality Characteristics examined in this research project are functional suitability, usability, performance efficiency, reliability, and maintainability.

There are three software quality characteristics of ISO 25010 that are not included
in this research. These three characteristics are compatibility, security, and portability. Compatibility is the degree to which two or more systems or components can exchange information and/or perform their required functions while sharing the same hardware or software environment. This system combines the functionality of two components into one and operates as a single complete software solution. Although compatibility applies to the Prototype, it does not pertain to the components. Security is the degree of protection of information and data so that unauthorized persons or systems cannot read or modify them and authorized persons or systems are not denied access to them. The system is not currently optimized for security. Quantifying the level of security within these software products does not add value to this quality assessment. The last characteristic, portability, is not measured as the design and development did not take portability into consideration. The object of this research is the development of a prototype to access the effects of software reuse on software quality. Therefore, portability was not among its requirements. The AR.Drone SDK 1.8 source code was successfully ran on both Ubuntu and Microsoft Windows® XP operating systems. However, the amount of modifications necessary to run the SDK example on Windows® eventually led to the removal of the Windows® example in the 2.0 SDK from the 1.8 SDK [78] [75].

6.1.1 Functional Suitability

Functional suitability determines the level at which software product provide
adequate functions when the software is used under specific conditions. In this case, the assessment will quantify the ability for the AR.Drone SDK navigation to control the UAV and obtain images from the video feed; the Physical Measurement Calculator to calculate the distance between two points with a planar surface using photogrammetry; and the Prototype, which reuse code from the two previous software, will determine the functional capability to perform both control and distance measurement simultaneously.

The Quality Measure Element (QME) 1, in Table 5.1, shows the Lines of Code (LOC) of the software components used. This table was included to show the complexities of the software systems used or analyzed during the development of this prototype. Using particular components within these systems required the learning of these systems to successfully identify the required components and reuse them within the prototype. This table is used in conjunction with QME 5 and QME 6 to determine the defect density as shown in Table 6.4.

Using QME 1, Lines of Code, there exists 7 KLOC for the SDK navigation example. This is a fair amount of code considering there are 2 fatal defects resulting in 0.29 defects/KLOC. These defects prohibit the use of the SDK navigation from obtaining single static images from the video stream. Although the video window works, the button to save a snap shot results in a complete system crash. A minor flaw in the system also prevents the user from controlling the UAV using a keyboard. A joystick or gamepad controller connected to the computer running the SDK can control of the UAV very well.
Table 6.1: Metric for QME1 Lines of Code (LOC) using CLOC [82]

<table>
<thead>
<tr>
<th></th>
<th>Prototype</th>
<th>Navigation</th>
<th>MATLAB Calibration Toolbox</th>
<th>Physical Measurement Calculator</th>
</tr>
</thead>
<tbody>
<tr>
<td>MATLAB</td>
<td>-</td>
<td>-</td>
<td>13652</td>
<td>14004</td>
</tr>
<tr>
<td>C</td>
<td>6485</td>
<td>6406</td>
<td>67</td>
<td>68</td>
</tr>
<tr>
<td>C/C++ Header</td>
<td>588</td>
<td>601</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>XML</td>
<td>75</td>
<td>75</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Total LOC</td>
<td>7148</td>
<td>7082</td>
<td>13719</td>
<td>14072</td>
</tr>
</tbody>
</table>

The Physical Measurement Calculator contains 14.1 KLOC of which the toolbox for image processing takes up 13.7 KLOC. This leaves the Physical Measurement Calculator a difference of 400 LOC for the user interface and calculations required for the measurements. The prototype contains a difference of 60 LOC in comparison to the AR.Drone SDK navigation. However, just examining the total number of lines of code does not give an accurate assessment of how many lines were added and subtracted. Comparing the capability of these two applications for measuring the distances, we notice that the Physical Measurement Calculator has more features than the prototype. The
prototype streamline the process of picking points by only allowing the selection of coordinates via mouse clicks. The features of zooming in to select the best point, and the ability to manually input coordinates were not implemented in the prototype. This allows the user to control the UAV, quickly select points, and retrieve the calculated distance measurement.

Table 6.2: Metric for QME2: Operational Task

<table>
<thead>
<tr>
<th></th>
<th>AR.Drone SDK</th>
<th>Physical Measurement</th>
<th>Prototype</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Navigation</td>
<td>Calculator</td>
<td></td>
</tr>
<tr>
<td>Task Size</td>
<td>5</td>
<td>5</td>
<td>8</td>
</tr>
<tr>
<td>Task Passed</td>
<td>4</td>
<td>4</td>
<td>7</td>
</tr>
<tr>
<td>Functional Completeness</td>
<td>0.80</td>
<td>0.80</td>
<td>0.88</td>
</tr>
</tbody>
</table>

Functional completeness is the degree which a set of functions covers all the specified tasks and user objectives. A value from 0 to 1 is assigned, where 0 is no function implemented, and 1 where all the functions are completely implemented. While Table 6.2 gives a general overview of the completeness of the software evaluated, Table 6.3 gives meaning to the values in Table 6.2. Table 6.3 is more valuable to the developer as it points to the functionalities lacking in the software.

Although the prototype does increase the functional suitability of the system
compared to the SDK navigation, it lowers the functionality of calculating the distances in favour of a streamlined calculation process.

Table 6.3: Breakdown of Metric for QME2: Operational Task

<table>
<thead>
<tr>
<th>Software</th>
<th>Operational Task</th>
<th>Function Works</th>
</tr>
</thead>
<tbody>
<tr>
<td>AR.Drone SDK navigation</td>
<td>Take off</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Fly / Hover</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Freeze frame (video)</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Save frame (Picture)</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>Land</td>
<td>Yes</td>
</tr>
<tr>
<td>Physical Measurement Calculator</td>
<td>Perform extrinsic calibration</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Zoom in/out</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Manually input coordinates</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Undistorted Image</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td>Calculate distance</td>
<td>Yes</td>
</tr>
<tr>
<td>Prototype</td>
<td>Take off</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Fly / Hover</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Freeze frame (Video)</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Save frame (Picture)</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Land</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Zoom in/out</td>
<td>No</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>---------------------</td>
<td>-------</td>
<td></td>
</tr>
<tr>
<td>Undistorted Image</td>
<td>Yes</td>
<td></td>
</tr>
<tr>
<td>Calculate distance</td>
<td>Yes</td>
<td></td>
</tr>
</tbody>
</table>
6.1.2 Usability

Usability usually applies to the ease of use, learnability, and the aesthetics of the user interface. However, the measures of these sub-characteristics of usability are typically qualitative in nature. As a qualitative measure, values of such metrics can range depending the user or evaluator. In order to make usability metrics more quantitative, we will examine operational tasks available to the user to accomplish a certain task as the measure of how much functionality the system. Examining Table 6.2, it could be said that the system mathematically supports the prototype as more functionality suited to its design requirements than are the AR.Drone SDK navigation and the Physical Measurement Calculator software are to theirs. Table 6.3 illustrates the fact the AR.Drone SDK navigation example is unable to save a picture, which the Physical Measurement Calculator requires to make distance measurements. Through combining these two separate software packages, the functionality to save a picture, although implemented in the prototype, is not required for the prototype to function in making distance measurements and flying the UAV.
Table 6.4: Metric for QME 4 & QME 5: Defect Density

<table>
<thead>
<tr>
<th>Software</th>
<th>QME 4 Number of non-fatal defects</th>
<th>QME 5 Number of fatal defects</th>
<th>Defect Density</th>
</tr>
</thead>
<tbody>
<tr>
<td>AR.Drone SDK navigation</td>
<td>1</td>
<td>2</td>
<td>0.424</td>
</tr>
<tr>
<td>Physical Measurement Calculator</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Prototype</td>
<td>1</td>
<td>2</td>
<td>0.420</td>
</tr>
</tbody>
</table>

Table 6.5: Metric for QME6: Number of Error Messages

<table>
<thead>
<tr>
<th>Software</th>
<th>Number of Error Messages</th>
</tr>
</thead>
<tbody>
<tr>
<td>AR.Drone SDK navigation</td>
<td>2</td>
</tr>
<tr>
<td>Physical Measurement Calculator</td>
<td>1</td>
</tr>
<tr>
<td>Prototype</td>
<td>2</td>
</tr>
</tbody>
</table>

Another quantitative measure is examining the defect density in the system. Defect density is an indicator of user error protection. The system should account for user errors and handle the error in a manner that does not crash the system or notifies the user. The
error messages seen with the AR.Drone SDK navigation and the Prototype is the GTK (Gimp Toolkit) failures and assertion that is part of the GTK library. The source code of the SDK and the navigation does not contain error messages. To increase user error protection, the defect density needs to decrease while increasing the number of error messages to notify the user that they made an error and it was caught or dealt with.

6.1.3 Performance Efficiency

Performance Efficiency examines the performance relative to the amount of resources used under slated conditions. The number of Lines of Code gives context to the amount of storage space the program uses on the system. However, it is not reflective of the run time memory usage. Out of memory messages occur during times of memory leak or the system runs out of memory due to a specific task. It is noted that the Physical Measurement Calculator does run out of memory when the size of the image becomes too large for the system to process. The AR.Drone SDK navigation, and subsequently the prototype, runs into segmentation faults with the software runs out of memory. To measure these maximum performance capacity, additional tests and quality measure elements must be added into the analysis.

6.1.4 Reliability

Reliability denotes the robustness of the software. It examines when systems go down and how often. A system that is constantly crashing is not reliable. This quality
applies to all software and ours is not the exception. The defect density is 0.424 for the SDK navigation and Prototype software. The density for the Physical Measurement Calculator is 0 for known defects. Defect density identifies the amount of defects known in a system so they could be addressed. Unknown defects cannot be measured, but can be derived from the number of known defects.

Lower defect densities indicate the maturity of the software system. A mature system which has been run and tested multiple times will have a gradual decrease in the defect density as older defects are fixed within the system.

The fault tolerance of the system, where it is able to handle hardware or software faults, can be assessed by comparing the non-fatal defects and the fatal defects measure elements. Non-fatal defects are indicative of the faults that the system where able to recover. The fatal defects are of the faults where the system is unable to recover. This ability to recover from a fault is also known as the systems recoverability. It is noted the fault tolerance and recoverability of the prototype has not improved from its reused components. The same defects found from the AR.Drone SDK navigation and the Physical Software Calculator, are also present in the Prototype. However, the fault tolerance and recoverability has increased due to the added functionality and lines of code with no increase in known defects.
6.1.5 Maintainability

Maintainability is the degree of effectiveness and efficiency with which the product can be modified, maintained, or improved. This Prototype modifies the AR.Drone SDK navigation to incorporate the distance measuring functions of the Physical Measurement Calculator.

The number of lines of code indicate the size of the program. The defect density can be used to measure the reusability of the software. The defects found in the AR.Drone SDK navigation software is propagated into the Prototype. This leads to the concept that reusability of a software is based on the added functionality it brings into the system while decreasing the number of newly injected defects from new code. Ideally, reused code has been tested and assertion methods are used to ensure it has been implemented correctly.

In order to increase the reusability of the prototype, or its components, the known defects needs to be fixed, and added tests to ensure future reuse does not inject previously known defects into the new system.

6.2 The Impact of Software Reuse on Software Quality

Analysing the quantifiable characteristics of software quality using the ISO 25000 standards, it is found that the defect density of the Prototype using the reused code is marginally better than the quality of the original software. There was a notable increase in
the functional completeness of the prototype compared the functional completeness of the original software. The reused code from the Physical Measurement Calculator that was injected into the AR.Drone SDK navigation did not introduce new defects and actually lowered the defect density of the Prototype. Had the defects of the AR.Drone SDK navigation been addressed and fixed, the density defect of the Prototype would have been lower than it was.

Systematic reuse of software requires the testing of reusable components and their maintenance to realize the full potential of systematic reuse. Reusable components need to be analyzed and ensured it meets its functional requirements. By ensuring only tested and verified components are stored in the software reuse library, the possibility of injecting defects with reused code becomes reduced. This positive aspect of software reuse is the fundamental basis for using software reuse methodologies within software development processes.
CHAPTER 7: CONCLUSION AND FUTURE WORK

7.1 Conclusion

Code reuse is a recognized fundamental method of knowledge reused and system integration in software development. In this research, a prototype for a complete software package solution was created through the reuse of code from SDK navigation, Physical Measurement Calculator, and online code examples. The Prototype was able to simultaneously fly and measure distances in real-time during flight. This streamlined process, of obtaining images and calculating distances, added to the functionality of the AR.Drone. This resulted in a prototype that has been evaluated to be slighter greater quality than its components. The study of the strategies of software reuse outlines the details of the practical methodologies of categorizing, labeling, and storing previous knowledge so that it can be accessed and utilized. Moreover, this exercise enriched the understanding of the applications of the ISO/IEC 25000 standard model of software quality. The results indicate that software reuse approaches could improve and streamline the software development process. Meanwhile, the research proves the potential of allowance of arbitrary code execution, time-saving for functionality integration, as well as lower research and knowledge development cost.
7.2 Research Achievements

This thesis completed a study of the current software code reuse strategies, and software quality metrics and their relationship.

The study of software quality metrics was applied to the creation of a prototype for measuring distances during flight for evaluating the effects of reusing software on the quality of the prototype.

Most previous studies compared software which reused code, and software which did not reuse code on contrasted their quality improvements. This study took a different direction by evaluating the original software that will be reused and comparing them to the prototype software that incorporates their components.

7.3 Future Work

In this study, only small-scale applications of code reuse was conducted, which did not applying the methodologies of reuse classification. However, in real-world cases, there are much more factors and criteria should be considered. Moreover, it is possible to study the systematic reuse of other software by cataloging and applying their components in new products systematically. In the application of this study, future work could first examine and categorize component objects to evaluate and enhance the many different software reuse approaches currently available.

The metrics and measures outlined in the evaluation are key points for
improvement. However, more comprehensive and full-scale indicator system could be introduced into the evaluation and analysis process. Thus, the robustness and reliability of the results could be strengthened.

The core architecture of the AR.Drone has its own library to handle threaded processes. However, due to current multi-core CPU’s, running multiple threads does not necessarily mean the cores are being used in parallel. This brings forth the concept of parallelism were each core can be assigned to their own specific process or processes. Parallelization of the system will make full use of a systems multi-core system assigning each core to its specific task.

Further tests of the system could be created to robustly test point of failures. Implementation of a test plan would allow the developer to find bugs of components and fix them for future reuse.

Future studies and enhancements would make it is possible to improve the Prototype to a fully realized commercial product.
REFERENCES


[40] ISO/IEC 25021, Systems and software engineering - Systems and software Quality


[78] AR.Drone Developer Team, "AR.Drone SDK 1.7 Developer Guide."
[84] Parrot®, "Parrot® AR.Drone Quick Start Guide".
APPENDIX A: EVALUATION REQUIREMENTS

A.1 Purpose of Evaluation

The purpose of this software quality assessment is to evaluate and compare qualities of the AR.Drone SDK navigation, Physical Calculator Measurement, and Complete Software Package Prototype, which reuses code from the previous two products, to discuss and improve future developments.

A.2 Software Product Quality Requirements

The stakeholder of the assessment is the developer to assess the positive and negative impacts of software reuse on software quality, and the feasibility of the Complete Software Package Prototype.

This evaluation will examine the following software characteristics as outlined by the ISO 25010 standard: Functional Suitability, Usability, Performance Efficiency, Reliability, and Maintainability. The characteristics will be quantified and evaluated.
A.3 Software Products

This assessment evaluates the follow products:

1. AR.Drone SDK navigation example (Version 2.0.1 for Linux)

2. Physical Measurement Calculator

3. Complete Software Package Prototype (reuses code from 1 and 2)

A.4 Stringency of Evaluation

This evaluation is to compare the components of the original software and the software containing the reused code to show that superior software has or has not been achieved.
A.5 Quality Measure

**Table A.1: Initial set of QMEs**

<table>
<thead>
<tr>
<th>NO</th>
<th>Initial set of QME</th>
<th>Definition and concepts related directly to QME</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Operational Task</td>
<td>The ability to perform a specific task.</td>
</tr>
<tr>
<td>2</td>
<td>Lines of Code</td>
<td>Number of lines of code in the software. This will provide an indication to the complexity of the software.</td>
</tr>
<tr>
<td>3</td>
<td>Number of Non-Fatal Defects</td>
<td>The number of defects found that does not cause a system crash. A defect is noted where the system does not behave as expected.</td>
</tr>
<tr>
<td>4</td>
<td>Number of Fatal Defects</td>
<td>The number of defects found resulting in a system crash. A defect is noted where the system does not behave as expected.</td>
</tr>
<tr>
<td>5</td>
<td>Number of Error Messages</td>
<td>The number of error messages seen when using the software as a result of incorrect user input.</td>
</tr>
<tr>
<td>6</td>
<td>Number of Test Cases</td>
<td>Test case is an executable part of software systems that provides either a pass or fail result.</td>
</tr>
</tbody>
</table>
## A.5.1 Software Quality Measure Elements

### Table A.2: QME 1 – Lines of Code

<table>
<thead>
<tr>
<th>QME 2</th>
<th>Lines of Code</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Target entity</strong></td>
<td>Source Code</td>
</tr>
<tr>
<td><strong>Objectives and property to Quantify</strong></td>
<td>Quantify the number of lines of code. Purpose is used for compare program sizes and calculating defect ratios</td>
</tr>
<tr>
<td></td>
<td>Size of the program is the property to quantify</td>
</tr>
<tr>
<td><strong>Relevant Quality Measures</strong></td>
<td>Performance Efficiency: Capacity</td>
</tr>
<tr>
<td></td>
<td>Maintainability: Modularity, Reusability, Analysability, Modifiability, Testability</td>
</tr>
<tr>
<td><strong>Measurement Method</strong></td>
<td>Count the number of lines of count using software</td>
</tr>
<tr>
<td><strong>Input for the QME</strong></td>
<td>Source of the information: source codes</td>
</tr>
<tr>
<td></td>
<td>Input for the measure: run CLOC (Count Line of Code) software on the source code [82]</td>
</tr>
<tr>
<td><strong>Unit of measurement for the QME</strong></td>
<td>LOC = Lines of Code</td>
</tr>
<tr>
<td></td>
<td>KLOC = 1 Thousand Lines of Code</td>
</tr>
<tr>
<td><strong>Numerical rules</strong></td>
<td>KLOC = LOC/1000</td>
</tr>
<tr>
<td><strong>Scale Type</strong></td>
<td>nominal</td>
</tr>
<tr>
<td><strong>Context of QME</strong></td>
<td>Mainly chosen to measure complexity and used a basis for other QMEs.</td>
</tr>
</tbody>
</table>
Table A.3: QME 2 – Operation Tasks

<table>
<thead>
<tr>
<th>QME</th>
<th>Operational Tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target entity</td>
<td>Interactive Software</td>
</tr>
</tbody>
</table>
| Objectives and property to Quantify | The objective is to determine whether the system is able to perform the specific tasks  
Interactive Software capability is the property to quantify  
Definition of the Property to quantify: An event in which a software or software component does or does not perform a required function  
Point of View:  
- Software Developers: Desires functionality to meet software requirements |
| Relevant Quality Measures | Functional Suitability: Functional Completeness, Functional Appropriateness  
Usability: Operability  
Reliability: Availability  
Maintainability: Analysability, Testability |
| Measurement Method   | Observe user interaction with software on system and catch events in which a software or software component does or does not perform a required function as specified in the software requirements, manuals, or source codes. |
| Input for the QME | Source of the information: software, software documents, manuals, source codes.  
Input for the measure: The actions found from the sources conclude to reactions – after an interaction by a user – that either a task passes or fails at its task. |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Unit of measurement for the QME</td>
<td>Non-ideal reaction from an action</td>
</tr>
</tbody>
</table>
| Numerical rules | TaskDoc = required tasks determined from documents  
TaskSC = required tasks obtained from source codes  
TaskPass = actions that completes a task  
TaskFail = actions that fails to perform a task  
\( \sum \) TaskSize = TaskDoc \( \cap \) TaskSC  
Functional Completeness = \( \sum ( \text{TaskPass} ) / \text{TaskSize} \) |
| Scale Type | Ratio |
| Context of QME | At the design level |
Table A.4: QME 3 – Number of non-fatal defects

<table>
<thead>
<tr>
<th>QME 3</th>
<th>Number of non-fatal defects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target entity</td>
<td>interactive software</td>
</tr>
<tr>
<td>Objectives and property to Quantify</td>
<td>To determine the number of failures that does not cause a system crash</td>
</tr>
<tr>
<td>Relevant Quality Measures</td>
<td>Functional Suitability: Functional Correctness</td>
</tr>
<tr>
<td></td>
<td>Usability: User Error Protection</td>
</tr>
<tr>
<td></td>
<td>Reliability: Maturity, Fault Tolerance, Recoverability</td>
</tr>
<tr>
<td></td>
<td>Maintainability: Reusability, Analysability, Modifiability, Testability</td>
</tr>
<tr>
<td>Measurement Method</td>
<td>Observe user interaction with software and catch events in which the system does not as expected without affecting the system.</td>
</tr>
<tr>
<td>Input for the QME</td>
<td>All available actions found in the user interface.</td>
</tr>
<tr>
<td></td>
<td>Input for the measure: The actions found that causes an error when the software fails without resulting in a system crash</td>
</tr>
<tr>
<td>Unit of measurement for the QME</td>
<td>Defect / KLOC</td>
</tr>
<tr>
<td>Numerical rules</td>
<td>NFD = non-fatal defects</td>
</tr>
<tr>
<td></td>
<td>Non-fatal Defect Density = ( \sum \text{NFD} / \text{KLOC} )</td>
</tr>
<tr>
<td>Scale Type</td>
<td>Ratio</td>
</tr>
<tr>
<td>Context of QME</td>
<td>This QME is mainly chosen to measure the defect density.</td>
</tr>
</tbody>
</table>
Table A.5: QME 4 – Number of fatal defects

<table>
<thead>
<tr>
<th>QME 4</th>
<th>Number of fatal defects</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target entity</td>
<td>interactive software</td>
</tr>
<tr>
<td>Objectives and property to Quantify</td>
<td>To determine the number of fatal failures that render the system unusable.</td>
</tr>
<tr>
<td>Relevant Quality Measures</td>
<td>Functional Suitability: Functional Correctness</td>
</tr>
<tr>
<td></td>
<td>Usability: User Error Protection</td>
</tr>
<tr>
<td></td>
<td>Reliability: Maturity, Fault Tolerance, Recoverability</td>
</tr>
<tr>
<td></td>
<td>Maintainability: Reusability, Analysability, Modifiability, Testability</td>
</tr>
<tr>
<td>Measurement Method</td>
<td>Observe user interaction with software and catch events in which the system does not as expected and causes a system failure and the software is no longer usable.</td>
</tr>
<tr>
<td>Input for the QME</td>
<td>All available actions found in the user interface.</td>
</tr>
<tr>
<td></td>
<td>Input for the measure: The actions found that causes an error when the software fails without resulting in a system crash</td>
</tr>
<tr>
<td>Unit of measurement for the QME</td>
<td>Defect / KLOC</td>
</tr>
<tr>
<td>Numerical rules</td>
<td>FD = Fatal defects</td>
</tr>
<tr>
<td></td>
<td>$\sum FD / KLOC$</td>
</tr>
<tr>
<td></td>
<td>Fatal Defect Density = $\sum FD / KLOC$</td>
</tr>
<tr>
<td></td>
<td>Defect Density = ( $\sum NFD$ (from QME3) + $\sum FD$ )/ KLOC</td>
</tr>
<tr>
<td>Scale Type</td>
<td>Ratio</td>
</tr>
<tr>
<td>Context of QME</td>
<td>This QME is mainly chosen to measure the defect density.</td>
</tr>
</tbody>
</table>
**Table A.6: QME 5 – Number of Error Messages**

<table>
<thead>
<tr>
<th>QME 5</th>
<th>Number of Error Messages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Target entity</td>
<td>interactive software</td>
</tr>
<tr>
<td>Objectives and property to Quantify</td>
<td>Observe user interaction and catch of the number of unique error messages seen during interactive use</td>
</tr>
</tbody>
</table>
| Relevant Quality Measures | Usability: Accessibility  
|                         | Performance Efficiency: Capacity  
|                         | Reliability: Availability, Fault Tolerance, |
| Measurement Method     | Observe user interaction with software and note when the software catches a failure or corrects a failure and notifies the user. |
| Input for the QME      | All available actions found in the user interface.  
| Input for the measure: The system notifies the user in the event of a failure |
| Unit of measurement for the QME | ideal response when an error has occurred |
| Numerical rules        | number of error message seen / defect found |
| Scale Type             | Ratio                                  |
| Context of QME         | This QME is mainly chosen to assess the reliability of the software. |
### A.5.2 Mapping between QME and Quality Characteristics

#### Table A.7: Mapping between QME and Quality Characteristics

<table>
<thead>
<tr>
<th>NO</th>
<th>Characteristic</th>
<th>QME Description</th>
<th>Functional Completeness</th>
<th>Functional Correctness</th>
<th>Operability</th>
<th>User Protection</th>
<th>Accessibility</th>
<th>Performance Efficiency</th>
<th>Reliability</th>
<th>Maintainability</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Operational Tasks</td>
<td><strong>X</strong> <strong>X</strong></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>2</td>
<td>Lines of code</td>
<td></td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>3</td>
<td>Number of non-fatal defects</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>4</td>
<td>Number of fatal defects</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>5</td>
<td>Number of error messages</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
</tbody>
</table>