AN INVESTIGATION INTO BAD SMELLS IN MODEL-BASED SYSTEMS ENGINEERING

by

XIN ZHAO

JEFF GRAY, COMMITTEE CHAIR
JEFFREY CARVER
ZHE JIANG
ROBERT PETTIT
RANDY SMITH

A DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy
in the Department of Computer Science
in the Graduate School of
The University of Alabama

TUSCALOOSA, ALABAMA

2021

ABSTRACT

Systems engineering is a multi-disciplinary approach to design, realize, manage and operate a system, which consists of hardware, software, process and personnel. Engineers and scientists from different domains often create domain-specific software artifacts - systems models to describe phenomena in the process of system development. Systems models are frequently tied to external instrumentation and devices that coordinate experimentation and observation. The methodologies and tools that support systems modeling often lack the capabilities that are found in software engineering environments and practice, limiting the potential analysis capabilities that can be realized by the software adopted in the system. Moreover, due to the different focus of interest, systems engineers may lack systematic software engineering knowledge compared with software engineers, creating a knowledge gap between systems engineers and software engineers.

To assist engineers in developing systems models, this dissertation first mined systems engineers' questions they post on the discussion forum to understand the challenges and issues they face during the development of systems models. The examination results show that systems engineers have a great number of questions and problems related to bad smells in systems models. Motivated by this observation, the goal of my research is to assist systems engineers with a better understanding of bad smells in systems models from three aspects: 1) the summarization of bad smells in systems models; 2) the evaluation of bad smells from systems engineers; and 3) the identification of prominent bad smells in systems models. The work presented in this dissertation has informed the systems engineering community by an empirical investigation of bad smells in systems models.

DEDICATION

To my parents whose unyielding love and support have enriched my soul, and to my sister who has shown me that *Where there is a will, there is a way.*

LIST OF ABBREVIATIONS AND SYMBOLS

ADL	Architecture Description Language
AI	Artificial Intelligence
BCS	Basic Control Structure
BoW	Bag-of-Words
BPM	Business Process Model
CFG	Control Flow Graph
CNN	Convolutional Neural Network
CoC	Cognition Complexity
CSS	Cascading Style Sheet
CW	Cognitive Weight
CyC	Cyclomatic Complexity
CyPhyML	Cyber Physical Modeling Language
DSM	Domain-Specific Modeling
DSML	Domain-Specific Modeling Language
ECJ	Evolutionary Computation Library in Java
FN	False Negative
FP	False Positive
GME	Generic Modeling Environment
GPM	General-Purpose Modeling
HTML	Hyper-Text Markup Language
ID	Interface Density
IDE	Integrated Development Environment
IDF	Inverse Document Frequency

LabVIEW	Laboratory Virtual Instrument Engineering Workbench
LDA	Latent Dirichlet Allocation
MBSE	Model-Based Systems Engineering
MDE	Model-Driven Engineering
ML	Machine Learning
NLP	Natural Language Processing
OOP	Object-Oriented Programming
RBF	Radial Basis Function
RNN	Recurrent Neural Network
ROC	Receiver Operating Characteristic
SDLC	Software Development Life Cycle
SDPT	Smell-Driven Performance Tuning
SFL	Spectrum-based Fault Localization
SMO	Sequential Minimal Optimization
SOA	Service Oriented Architecture
SRS	Software Requirements Specification
SVM	Support Vector Machine
TF	Term Frequency
TN	True Negative
TP	True Positive
UML	Unified Modeling Language
VI	Virtual Instrument
VPL	Visual Programming Language

ACKNOWLEDGMENTS

First and foremost, I would like to express my deepest gratitude and thanks to my advisor, Dr. Jeff Gray, for his guidance and advice throughout my Ph.D. career. His incredible mentorship and unwavering support made this dissertation possible. Beyond this, his empathy towards his students as a professor, his enthusiasm for work as an educator, and his kindness to others as a person have given me a perfect example in actions and love. I am so lucky, blessed and proud to have him as my Ph.D. advisor, in every sense of the word.

I would also like to thank my committee members, Dr. Jeffrey Carver, Dr. Zhe Jiang, Dr. Randy Smith and Dr. Rob Pettit for their advice and support. In addition, I want to extend my thanks to Dr. Jiang. He has been helpful in providing advice and co-authoring several papers during my Ph.D. study.

I also thank my friends for being with me over the past six years. They have been my listeners, party members, travel buddies, and video game teammates. The time spent with all of them has been, and will continue to be, some of the best times of my life. I would also like to thank my colleagues in the Software Engineering Group at the University of Alabama for the paper collaborations and valuable feedback on my talks and presentations.

I want to thank the Department of Computer Science at the University of Alabama for providing me the opportunity for being a Teaching Assistant for 9 semesters. I would also like to thank National Science Foundation (STEM+C grant) and the Department of Education (Education Innovation and Research grant) for their support during this research work.

Last but not least, I thank my mom, my dad and my sister. My parents have provided unconditional support, care and love to me. I would not have made it this far without them. I love them forever.

CONTENTS

ABSTRACT ii

DEDICATION iii

LIST OF ABBREVIATIONS AND SYMBOLS iv

ACKNOWLEDGMENTS vi

LIST OF TABLES xiii

LIST OF FIGURES xiv

CHAPTER 1 INTRODUCTION 1

1.1 Systems Engineering 1

1.2 Software Engineering 3

1.3 The Interweave between Systems Engineering and Software Engineering . . . 5

1.4 Key Challenges with Software Engineering for Systems Engineering 7

1.5 Scope of the Research 7

1.5.1 Understanding Systems Engineers' Challenges 8

1.5.2 Bad Smells Summarization from End-Users 8

1.5.3 Empirical Evaluation of Bad Smells in Systems 9

1.5.4 Identification of Complicated Systems Models 9

1.6 Main Contributions of this Dissertation 10

1.7 Structure of the Dissertation 11

CHAPTER 2 BACKGROUND 12

2.1 Model-based Systems Engineering 12

2.2 LabVIEW 13

2.2.1 Front Panel 14
2.2.2 Block Diagram 15
2.2.3 Simulink 16
2.3 Bad Smells 18
CHAPTER 3 MINING FORUM POSTS TO UNDERSTAND MODELING CHALLENGES 23
3.1 Introduction 23
3.2 Text Classification 25
3.2.1 Data Pre-Processing 26
3.2.2 Feature Extraction 27
3.2.3 Model Training 27
3.2.4 Model Evaluation 33
3.2.5 Text Classification for LabVIEW Modeling Community 36
3.3 Topic Modeling 40
3.3.1 Latent Dirichlet Allocation 41
3.3.2 Topic Modeling for LabVIEW Modeling Community 43
3.4 Discussion and Implication 45
CHAPTER 4 BAD SMELL SUMMARIZATION FROM END-USERS 47
4.1 Introduction 47
4.2 Related Work 49
4.3 Methodology 51
4.3.1 Level 1 Summarization 52
4.3.2 Level 2 Summarization 53
4.3.3 Level 3 Summarization 54
4.4 Experimental Results 56

4.4.1 Bad Smells Summarized from Level 1 56
4.4.2 Bad Smells Summarized from Level 2 61
4.4.3 Bad Smells Summarized from Level 3 63
4.5 Discussion and Insights 66
4.5.1 Inconsistent Term Usage 66
4.5.2 OOP in LabVIEW 67
4.5.3 Lack of Documents/Examples 67
4.6 Threats to Validity 68
4.6.1 Threats to Construct Validity 68
4.6.2 Threats to Internal Validity 69
4.6.3 Threats to External Validity 69
4.7 Conclusion 69
CHAPTER 5 AN EMPIRICAL EVALUATION OF BAD SMELLS IN SYSTEMS MODELS 71
5.1 Introduction 71
5.2 Related Work 75
5.2.1 Empirical Analyses of Code Smells 75
5.2.2 Bad Smells in Systems Models 76
5.3 Selected Bad Smells in LabVIEW Systems Models 77
5.3.1 VI Structure 79
5.3.2 Loops and Arrays 80
5.3.3 Functions 81
5.4 Empirical Study: A Survey of LabVIEW Users 82
5.4.1 Survey Design 82
5.4.2 Participant Recruitment 83

5.4.3 Data Collection 84
5.5 Experimental Results 84
5.5.1 Demographics 84
5.5.2 Systems Engineer's Perception 86
5.5.3 The Impact of Experience on Smell Perception 87
5.5.4 The Impact of Domain on Smell Perception 89
5.5.5 Other LabVIEW Model Smells 90
5.6 Recommendations and Lessons Learned 91
5.6.1 Recommendations 92
5.6.2 Lessons Learned 93
5.7 Threats to Validity 94
5.7.1 Threats to Construct Validity 94
5.7.2 Threats to Internal Validity 94
5.7.3 Threats to External Validity 94
5.8 Conclusion 95
CHAPTER 6 A COMPLEXITY METRICS SUITE FOR LABVIEW MODELS . . 96
6.1 Introduction 96
6.2 Related Work 98
6.3 Proposed Metrics Suite 99
6.3.1 Cyclomatic Complexity 100
6.3.2 Halstead Complexity 101
6.3.3 Information Flow Complexity 105
6.3.4 Card & Glass Complexity 107
6.3.5 Cognition Complexity 108
6.3.6 Front Panel Complexity 111

6.4 Theoretical Validation . . . 113
6.5 Empirical Evaluation . . . 118
6.5.1 Model Selection . . . 118
6.5.2 Data Collection . . . 118
6.5.3 Results and Discussion . . . 120
6.6 Threats to Validity . . . 123
6.6.1 Threats to Construct Threats . . . 124
6.6.2 Threats to Internal Threats . . . 124
6.6.3 Threats to External Validity . . . 125
6.7 Conclusion . . . 125
CHAPTER 7 FUTURE WORK . . . 127
7.1 An Interview-based Empirical Study on Bad Smells in Systems Models . . . 127
7.1.1 Research Questions . . . 127
7.1.2 Potential Approach . . . 128
7.2 System Model Refactoring Based on Examples . . . 130
7.2.1 Background: Genetic Programming . . . 130
7.2.2 Research Questions . . . 131
7.2.3 Potential Approach . . . 132
7.3 Exploring the Relationship between Model Metrics and Model Smells . . . 134
7.3.1 Background: Regression Analysis . . . 134
7.3.2 Research Questions . . . 136
7.3.3 Potential Approach . . . 136
CHAPTER 8 CONCLUSION . . . 138
REFERENCES . . . 140
APPENDIX A IRB CERTIFICATE . . . 158

LIST OF TABLES

1.1 Lines of Code in Some Computer-Aided Systems 6

3.1 Confusion Matrix 35

3.2 Unsupervised Text Classification Evaluation Results 39

3.3 Unique Term Frequency Using LDA 45

4.1 LabVIEW Bad Smells Summarized by Chambers and Scaffidi 53

4.2 Simulink Bad Smells Summarized by Gerlitz 54

5.1 Model Smells Included in the Survey 78

5.2 Mean Perception Value of Model Smells 88

5.3 P-Value Matrix 88

6.1 Cyclomatic Complexity Computation for Figure 6.1 101

6.2 Cognitive Weights for LabVIEW Blocks 109

6.3 Combination of Decision Blocks (Cyclomatic Complexity = 5) 114

6.4 An Overview of Models Adopted For Empirical Evaluation 119

6.5 Empirical Evaluation Results for 10 LabVIEW Models 120

7.1 Potential Interview Questions 129

7.2 Metrics Provided by the VI Analyzer Toolkit 137

LIST OF FIGURES

1.1 Systems Engineering Process 2

1.2 Software Engineering Life Cycle 4

2.1 An Example of a VI's Front Panel 15

2.2 An Example of a VI's Block Diagram 17

2.3 An Example of a Simulink Model 18

2.4 A LabVIEW Systems Model Without a Wait Function 21

2.5 A LabVIEW Systems Model With a Wait Function 21

3.1 Two Categories Containing Different Data Points 30

3.2 Possible Separations of Two Categories 31

3.3 Convert Inseparable Data in 2D Space to 3D Space 31

3.4 An Example of Classification Strategy Based On Decision Tree 33

3.5 Post Distribution Prediction for LabVIEW Discussion Forum 40

4.1 Overview of BESMER Workflow 52

5.1 Flat and Stacked Sequence Structure in LabVIEW 81

5.2 Survey Questions (Partial) 83

5.3 Participants' Geographical Distribution 85

5.4 Participants' Role and Knowledge Level 86

5.5 Mean Perception Value Based on Experience 89

5.6 Mean Perception Value Based on Domain 90

6.1 Code Fragments with Different Number of Decision Points and Their Corresponding CFGs 102

6.2 A LabVIEW Block Diagram to Calculate $\frac{1}{x^2} + \frac{1}{y^2}(x \neq 0; y \neq 0)$ 104

6.3 The Icon for the SubVI *Square_Reciprocal_Sum* 106

6.4 The Modularization of VI by the Adoption of SubVI 107

6.5 A more complicated example for cognition complexity calculation 111

6.6 The Front Panel for Model 2 122

6.7 The Front Panel for Model 6 123

7.1 Overview of the Flowchart of Genetic Algorithms 131

7.2 Refactoring As a Learning Process 133

CHAPTER 1

INTRODUCTION

Modern systems are composed of a set of hardware, software, data, procedures and personnel. All of these elements work together to produce quality systems. Among of all these elements, software now plays a major role in all enterprise systems. This chapter first discusses the basic concept and major phases in systems engineering and software engineering and their relationship. Following this is the examination of key challenges with software engineering for systems engineering and briefly introduce how this dissertation solves these challenges. Finally, this chapter outlines the main contributions of this dissertation and present the structure of this dissertation.

1.1 Systems Engineering

Systems engineering is a multi-disciplinary and integrative approach to design, implement and operate systems. It guides "the engineering process so that the system can be described in a way that can be implemented, and along the way, ensuring that it is meeting the needs of the project" [52]. There are several groups of people involved in this process: stakeholders who drive the high-level objectives of the system; project managers who schedule the project, manage the risk and control the budget; and engineering specialists who have specific domain knowledge (such as software engineers, mechanical engineers and electrical engineers) responsible for designing and implementing the system.

The National Aeronautics and Space Administration (NASA) defined five phases in a systems engineering life-cycle [162]. Figure 1.1 illustrates an overview of these five phases in system development.

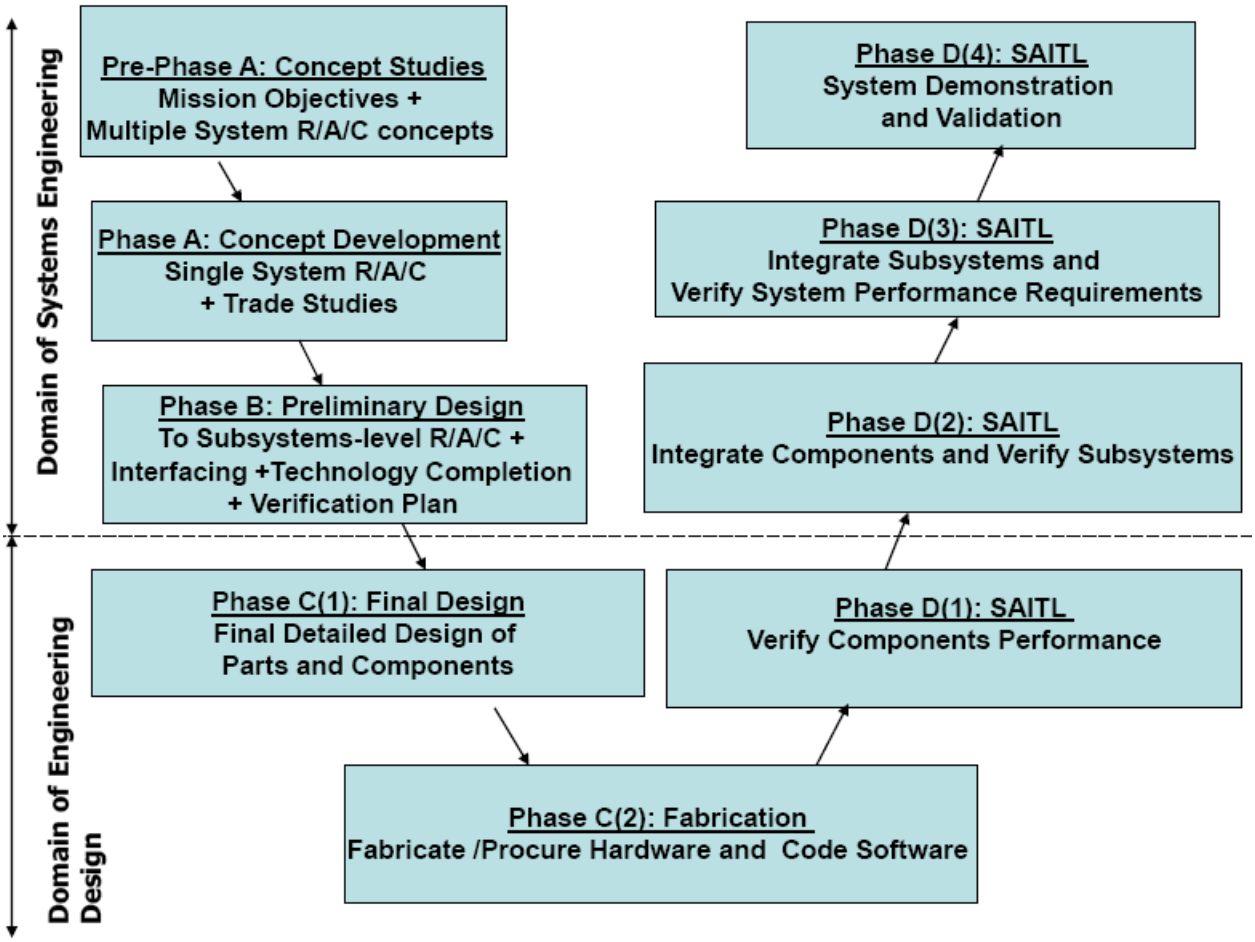


Figure 1.1: Systems Engineering Process (Adopted from [162]). R/A/C refers to Requirements, Architecture Design, Concept of Operation. SAITL refers to System Assembly, Integration, Test and Launch.

1. **Pre-Phase A**. This phase is named as *Concept Studies*. The goal is to establish the objectives of the system to be developed.

2. **Phase A**. Phase A is called *Concept and Technology Development*. In Phase A, systems engineers define systems requirements, architecture and manage the project from different perspectives, such as project schedule, risk, budget and personnel.

3. **Phase B**. The name of Phase B is *Preliminary Design and Technology Completion*. As the name suggests, in this step, systems engineers divide the system into several smaller subsystems and establish a preliminary design with the specific requirements for each subsystem.

4. **Phase C**. Phase C is named *Final Design and Fabrication*. This phase implements all the subsystems defined in the previous phase. Domain specialists, such as software engineers, commonly plays an active role in this phase.

5. **Phase D**. Phase D is also known as *System Assembly, Integration, Test and Launch*. In this phase, systems engineers and domain specialists work together to assemble and integrate subsystems to create, test, verify, validate and deploy the final system.

1.2 Software Engineering

Software engineering is the application of systematic engineering approaches to software development [95]. The goal of software engineering is to ensure the software built is correct, reliable, and efficient.

A Software Development Life Cycle (SDLC, as shown in Figure 1.2) is a systematic process to create software. The goal is to provide a structured flow of steps that assists the production of high-quality, well-tested and ready-to-deploy software products. The SDLC commonly consists of six stages:

1. The first stage is *Requirements Analysis*. Similar to the first stage in systems engineering, requirements analysis (also known as *requirements engineering*) is to establish and manage requirements with stakeholders for the software to be built or modified.

2. The second stage is *Planning*. After the first step, the next phase is to document the requirements clearly for stakeholder's approval. This document is frequently referred to as the Software Requirements Specification (SRS), which acts as a communication bridge between stakeholders and software designers. The SRS plays an important role in the software development practice and has been investigated heavily over the past several decades [23] [87] [181].

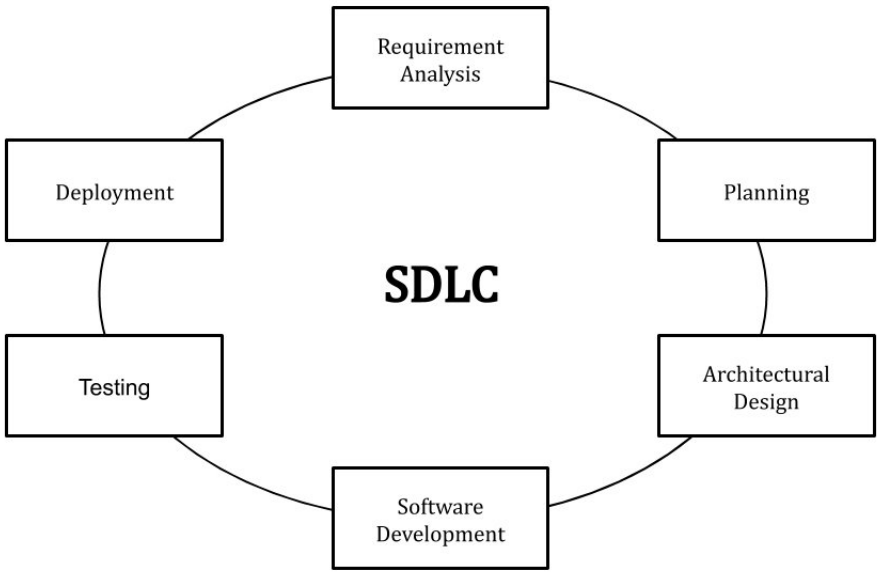


Figure 1.2: Software Engineering Life Cycle [8]

3. *Architectural Design* is the third step in a software development life cycle. This step describe a software project from a high level of abstraction to establish the framework for the development in the following phases. The architectural design includes:

   - Architecture, such as overall design, industry practices;
   - User Interface, such as input controls, navigation components and informational components;
   - Programming, such as programming paradigm and programming language;
   - Security, such as the measures need to be taken to secure the application, storage security.

4. The next step is *Software Development*. This is the actual implementation phase in a software development life cycle. Based on the design documents from *Architectural Design*, software developers code the software product.

5. *Testing* is the fifth step in a software development life cycle. Software testing is vital to verify and validate both functional and non-functional features scoped in a software project. Common functional testing include unit testing, integration testing, system testing and acceptance testing. Non-functional testing approaches include security testing, performance testing and usability testing.

6. The last step in a software development life cycle is *Software Deployment and Maintenance*. Once the software is implemented and tested, the software deployment and maintenance follow. In this stage, the software product is released formally into the market. Maintenance and updates may be required as soon as the software deployment starts.

1.3 The Interweave between Systems Engineering and Software Engineering

Systems engineering and software engineering are highly intertwined. Software is "fundamental to the performance, features, and value of most modern engineering systems. It is not merely part of the system, but often shapes the system architecture; drives much of its complexity and emergent behavior; strains its verification; and drives much of the cost and schedule of its development" [138]. Software is central to modern computer-aided systems (particularly cyber-physical systems), such as automotive, robotics, aeronautics and astronautics. Table 1.1 presents lines of code in some computer-aided systems[1].

One of the most common paradigms used in systems engineering is Model-Based Systems Engineering (MBSE). It is an approach that uses models as a structured representation of the system. The creation of systems models is frequently based on modeling languages, which are used to "express information or knowledge or systems in a structure

[1]Data Source:
https://www.computerworld.com/article/2725085/curiosity-about-lines-of-code.html
https://www.visualcapitalist.com/millions-lines-of-code/
https://codeinstitute.net/blog/much-code-cars/

Table 1.1: Lines of Code in Some Computer-Aided Systems

System	Domain	Estimated Lines of Code	Year
Apollo 11	Spaceflight	145,000	1969
Hubble	Space Telescope	1,800,000	1990
F-22 Raptor	Fight Jet	2,000,000	2005
Curiosity	Mars Rover	2,450,000	2011
Flight Software in Boeing 787 Dreamliner	Commercial Jet Airliner	13,500,000	2011
F-35 Raptor	Fight Jet	24,000,000	2013
Ford F-150 Model 2016	Pick-up Truck	150,000,000	2016

that is defined by a consistent set of rules"[2] to interpret the components in a system. A modeling language can be either domain-specific or general-purpose.

Domain-Specific Modeling (DSM) uses Domain-specific Modeling Languages (DSMLs) to build software components for a system in a systematic manner. With a DSML, users frequently build models with a range of abstraction levels and generate the code automatically from the models. A Domain-Specific Modeling Language is commonly used in domain-specific language environments, such as the LabVIEW modeling language [90] developed by National Instruments[3] in industry, CyPhyML (Cyber Physical Modeling Language) [172] from Vanderbilt University used in the GME (Generic Modeling Environment)[4]/WebGME[5] in academia.

Geneal-Purpose Modeling (GPM), on the other hand, uses a general-purpose modeling language to repersent the software components in a system. One of the most popular general-purpose modeling languages is the Unified Modeling Language (UML) [149]. UML adopts several diagrams to represent different views of a system, such as structural view (such as class diagram which describes the attribute, methods and relationships among

[2] https://en.wikipedia.org/wiki/Modeling_language
[3] https://www.ni.com/en-us.html
[4] https://www.isis.vanderbilt.edu/Projects/gme/
[5] https://webgme.org/

system's classes) and behavioral view (such as sequence diagrams that depict the interactions between objects, and an activity diagram that models the process and data flows in each activity).

1.4 Key Challenges with Software Engineering for Systems Engineering

Although software engineering is central to systems engineering [24], researchers have shown that there exists a knowledge gap between systems engineering and software engineering [62] [165]. The very first challenge this dissertation pays attention to is *what software engineering related challenges do systems engineers face in the practice of systems engineering.*

To answer this question, systems engineers' questions posted on an online discussion forum were analyzed. With the combination of text classification and topic modeling techniques, over 100,000 forum posts were examined and the results show that systems engineers are concerned about bad smells (surface indications of potential problems) in systems models. Motivated by this observation, my dissertation focuses on the following questions:

- What bad smells do systems engineers face during the development of software artifacts when practicing systems models development?
- How do systems engineers evaluate the bad smells summarized in existing literature and development practice?
- How to assist systems engineers in identifying the prominent bad smells during the systems models development?

1.5 Scope of the Research

To address the existing challenges and answer the above-mentioned research questions, this section presents the scope of this dissertation with four subsections: under-

standing systems engineers' challenges, bad smells summarization from end-users, bad smells empirical evaluation and the identification of complicated systems models.

1.5.1 Understanding Systems Engineers' Challenges

Online discussion forums play an important role in building and sharing domain knowledge. An extensive amount of information can be found in online forums, covering every aspect of life and professional discourse. Online forums provide a unique way to crowdsource current topics of interest within a community. A deep analysis of online discussion forums has benefits for individuals and society overall. This dissertation introduces the application of supervised and unsupervised machine learning techniques to analyze forum questions. The investigation starts with supervised machine learning techniques to classify forum posts into pre-defined topic categories. After this, this subsection introduces unsupervised learning techniques to identify latent topics in documents. The combination of supervised and unsupervised machine learning approaches offers deeper insights of the data obtained from online forums to understand systems engineers' challenges during the development of systems models.

1.5.2 Bad Smells Summarization from End-Users

Bad smells are indications of potential problems with source code quality and have been investigated deeply within the purview of Object-Oriented Programming (OOP). However, there has not been much research conducted to understand bad smells within the context of systems models. Moreover, the majority of bad smells in existing literature have been suggested by experienced developers and researchers who may view "smells" differently from inexperienced developers. To this end, this dissertation proposes BESMER, a project that categorizes bad smells in systems models by mining discussion forum posts of end-users of a specific systems modeling tool. Specifically, BESMER describes how its three-level discovery mechanism based on machine learning techniques assisted this study in finding bad smells from the LabVIEW online discussion forum. The experimental results not only confirm that end-users also encounter the bad smells proposed by experts,

but also reveal new bad smells for LabVIEW models. This subsection also presents some implications discovered from the examination of user posts and list areas of future work based on current findings. This study is the first attempt to investigate bad smells from an end-user's perspective within the context of systems models.

1.5.3 Empirical Evaluation of Bad Smells in Systems

Compared with extensive research on bad smells in the context of OOP (code smells), bad smells in systems models (model smells) need much more investigation. Although some works have proposed several model smells in a few modeling domains, the understanding and perception of different types of model smells may vary due to depth of knowledge and area of expertise. To fill this gap, I conducted an empirical study to evaluate the model smells summarized in the existing literature within the context of LabVIEW systems models through an anonymous online survey. Based on the 45 complete responses received from a diverse group of systems modelers, I observed that there exist differences regarding the perception of various model smells. Furthermore, depth of knowledge (experienced and inexperienced users) was observed as a factor that affects a user's understanding of different model smells. However, area of expertise (academia/industry, as well as domain of focus) did not show significant difference in model smells perception. Moreover, this study identified additional model smells from this empirical study. This work also provides several recommendations to avoid common model smells and the lessons learned from this investigation. This dissertation provides empirical evidence that drives deeper insights into model smells and lays out recommendations to practitioners on how to avoid some of the prominent smells, thus improving the quality of software artifacts in systems models.

1.5.4 Identification of Complicated Systems Models

LabVIEW is a leading modeling tool in the realm of systems engineering. Although LabVIEW is widely used in various fields, such as industrial design, academic research, and engineering education, rare attention has been paid to the quality of the systems model

it builds. Few researchers have analyzed the quality of LabVIEW systems models. To fill this gap, this dissertation introduces a metrics suite for LabVIEW systems models to assist the characterization of model complexities from different aspects. The metrics in the proposed suite ware theoretically validated using Weyuker's validation [184]. A proof-of-concept prototype was implemented and the metrics suite was applied to 10 LabVIEW models from GitHub to empirically evaluate its suitability to support the description of complexity analysis of LabVIEW models. This examination leads to a better understanding of the complexity residing in systems models via metrics analysis, thus making a contribution to improving the quality of systems models built within the LabVIEW environment.

1.6 Main Contributions of this Dissertation

The main outcome of this dissertation is the investigation of bad smells in systems models. To achieve this goal, I analyzed user forum posts using machine learning techniques, empirically evaluated the bad smells summarized from user posts and proposed methodologies to assist in the identification of the most severe bad smells based on the empirical evaluation. To be more specific, the contributions of this dissertation are:

1. Bad Smells Summarization from Mining End-User's Posts.
    - Discovery of new bad smells from end-user's perspective;
    - Understanding the similarities and differences between bad smells in systems models and bad smells in OOP languages.
2. Empirical Evaluation of Bad Smells in Systems Models.
    - Understanding bad smells in systems models based on the end-user's perception;
    - Investigation of the perceived differences of bad smells between participants with diverse experience and areas of expertise;

- Identification of additional bad smells in systems models from systems engineers.

3. The proposal of a metrics suite to identify complicated systems models.

   - Proposing a complexity metrics suite for the understanding of the complexity of LabVIEW systems models;
   - A theoretical validation of the proposed metrics suite;
   - An empirical evaluation of the proposed metrics suite.

1.7 Structure of the Dissertation

This chapter introduced the concepts of systems engineering, software engineering and the connection between systems engineering and software engineering. The following were the key challenges related to software engineering in the development practice of systems engineering. This chapter then discussed the scope of research in this dissertation and the main contributions of this research.

The remainder of this dissertation is organized as follows: Chapter 2 introduces the background knowledge, including Model-Based Systems Engineering, LabVIEW systems models and bad smells. Chapter 3 describes the empirical study of mining end-user forum posts from the modeling community with the help of machine learning techniques and how the results of this empirical study motivated the following research. Chapter 4 describes the methodology of bad smell summarization from mining forum posts. Chapter 5 outlines the empirical evaluation of bad smells through an online survey, and Chapter 6 describes the identification of the most prominent bad smell based on the result of previous examinations. Chapter 7 discusses future research directions, and Chapter 8 concludes this dissertation.

CHAPTER 2

BACKGROUND

This chapter provides some basic background for understanding the research conducted in this dissertation. The introduction starts with the concept of Model-Based Systems Engineering (MBSE), as well as two popular tools used in MBSE: LabVIEW and Simulink. This chapter also provides an introduction of bad smells, which is a main research focus in this dissertation.

2.1 Model-based Systems Engineering

Model-Based Systems Engineering (MBSE) is "the formalized application of modeling to support system requirements, design, analysis, verification and validation activities beginning in the conceptual design phase and continuing throughout development and later life cycle phases" [130]. Model-based systems enable engineering teams to "understand design change impacts, communicate design intent and analyze a system design before it is built"[1].

Systems models play an important role in the context of MBSE. Systems modeling takes an active part not only in domain analysis, but also in other phases of software development, such as requirements analysis and product validation. Constructing a concise and accurate system model is central to software development in many domains, particularly cyber-physical systems that are tied to instruments and external devices.

There are different types of system models. A function model is a "structured representation of the functions (activities, actions, processes, operations) within the modeled system or subject area" [155]. The purpose of function models is to describe the function

[1] `https://www.jrcisi.com/model-based-systems-engineering-mbse-programs`

and process in a system. An architecture model is a conceptual model that describes the structure and views of a system. Compared with a function model, the architecture model focuses on the external presentation of the system. The formalization of system architecture models has been proposed by several researchers. These languages are called Architecture Description Languages (ADLs) [42] [112]. A Business Process Model (BPM) is a system model used in business process management. It is the graphical representation of a business process or workflow. A BPM model usually consists of two states: the current state of the process and the state after making changes or improvements. A Gantt chart [60] is the first business process model dating back to 1910s. UML activity diagrams are also widely used as BPMs for business management [53]. An activity diagram could model both computational and organizational processes, as well as data flows in these activities. This section discusses two popular systems modeling tools: LabVIEW and Simulink.

2.2 LabVIEW

Laboratory Virtual Instrument Engineering Workbench (LabVIEW), developed by National Instruments, is a systems development environment using a visual programming language[2]. It follows a graphical programming paradigm which enables systems engineers to visualize every aspect of the system development, such as hardware configuration, and data measurement. LabVIEW also enables the integration of measurement hardware from any vendor while representing complex logic on the system model diagram. A systems engineer can also develop domain-specific algorithms and design customized user interfaces in LabVIEW.

There are three reasons why LabVIEW systems models were selected as the research focus:

1) Extensive Usage. As one of the leading industrial modeling tools, more than 35,000 companies/organizations worldwide use LabVIEW today [54]. It is widely used in

[2] `https://www.ni.com/en-us/shop/labview.html`

different areas (such as systems design and production test) across various fields, including aerospace, automotive, energy and semiconductor industries.

2) Limited Research. Although there is a large user base, the research literature reporting investigations into LabVIEW systems modeling is limited. Moreover, most LabVIEW users are engineers who may lack knowledge in software engineering, thus increasing the possibility of introducing poor designs during the systems development.

3) National Instruments Collaboration. My lab is currently collaborating with National Instruments, which helps to contact experts to gain more insights from LabVIEW professionals when conducting this research.

LabVIEW models are also referred to as Virtual Instruments (VIs). Each VI consists of two main components: a front panel (front-end) acting as the user interface for a model and a block diagram (back-end) containing the graphical source code for a model.

2.2.1 Front Panel

A front panel is the user interface for a VI. It consists of **Controls** and **Indicators**. Controls are analogous to instrument input - they provide data to the block diagram of a VI. Common controls include strings, arrays, and numeric numbers. Similarly, indicators simulate the output of a system - they display results generated by the block diagram. Indicators are typically charts/graphs, numeric numbers, and LEDs. In LabVIEW, all the controls and indicators are provided in the **Control Palette**. A full supply list of controls and indicators in the control palette can be found in the LabVIEW online manual[3]. Figure 2.1 shows an example of the front panel in a VI.

The front panel shown in Figure 2.1 represents the result of the comparison of two strings. In Figure 2.1, there are four controls (i.e., x, y, m and n, marked in purple boxes in Figure 2.1). These four controls are four string inputs to be compared. There are two indicators (marked in blue boxes in Figure 2.1). These indicators are two Boolean LED lights which a green light indicates two strings are the same while a red light indicates

[3] `http://zone.ni.com/reference/en-XX/help/371361R-01/lvconcepts/fp_controls_indicators`

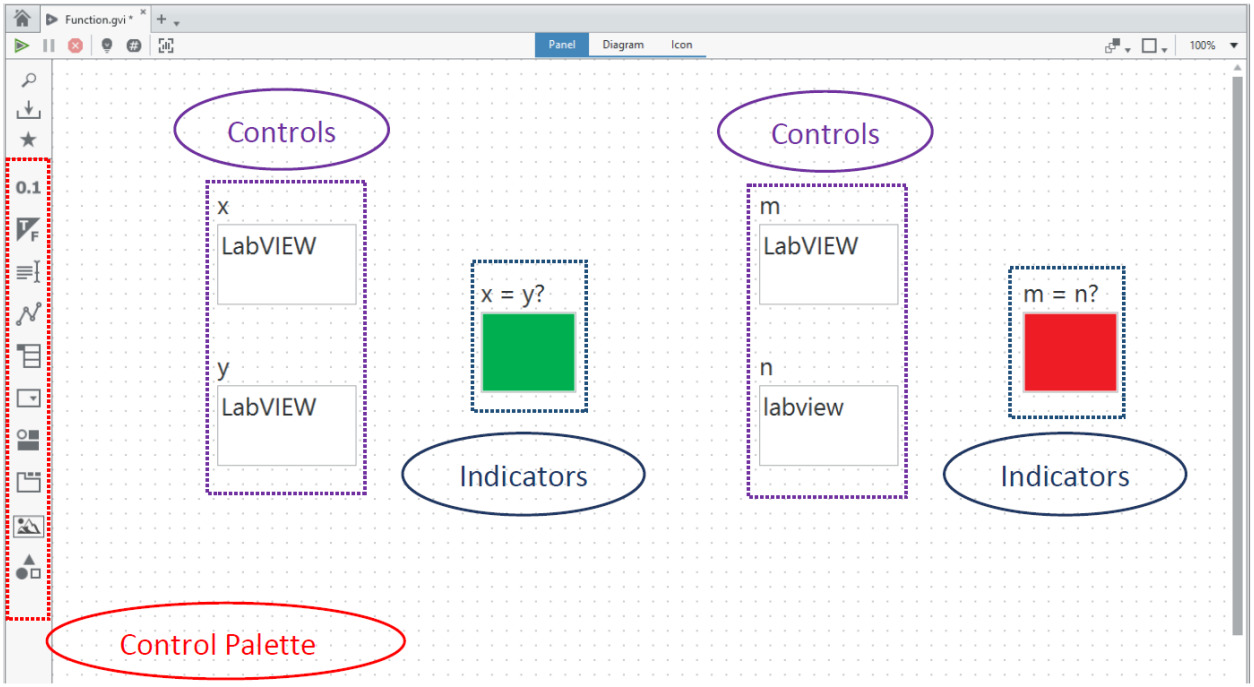


Figure 2.1: An Example of a VI's Front Panel

they are not. All the controls and indicators used in a front panel are pre-defined by LabVIEW and stored in the Controls Palette (marked in the red box in Figure 2.1). LabVIEW enables users to build a front panel by simply dragging and dropping different controls and indicators from the controls palette.

2.2.2 Block Diagram

A block diagram acts as the back-end of a VI. It contains the graphical source code of a LabVIEW program. Indicators and controls from the front panel are presented as **terminals** in the block diagram. To process system inputs, LabVIEW uses different **blocks**. There are three main types of blocks:

1. *Function Blocks.* A function block shows what the code does. The **function palette** provided by LabVIEW offers various functions arranged in groups based on the function type. For example, in a Math function palette, users can find functions from basic mathematical operations such as summation, as well as complicated calculations such as solving ordinary differential equations.

2. *Structure Blocks.* A structure block controls how and when the code runs. The structures in LabVIEW share the same concept with other programming languages. For example, a *loop structure* repeats the code inside of it until some condition is met; a *case structure* indicates a decision needs to be made to execute different pieces of code. Similar to function palettes, structure palettes are also provided within the LabVIEW environment to help users control their program flows.

3. *SubVIs.* A SubVI is a function defined by end-users instead of pre-defined functions by the LabVIEW environment. It is similar to a subroutine in textual programming languages. SubVIs are also VIs, which means that they also contain front panels and block diagrams.

LabVIEW programming follows a data-flow programming [180] style. The execution of a block diagram happens after receiving all the required inputs. After processing, the output data is transferred to the next block. The flow of the data determines the order of program execution. In a block diagram, **wires** are used to transfer the data and specify the direction of the data flow. Figure 2.2 shows the block diagram corresponding to the example shown in Figure 2.1.

In Figure 2.2, there are six terminals (marked in blue boxes): four string terminals and two Boolean terminals which correspond to the four controls and two indicators in Figure 2.1, respectively. The yellow box in Figure 2.2 shows the compare function provided by the **function and structure palette** (marked in the red box). The dotted magenta and green lines in the block diagram are wires that are used to transfer data between different blocks.

2.2.3 Simulink

Simulink is a system modeling tool developed by MathWorks[4]. It is a graphical Integrated Development Environment (IDE) for modeling, simulating and analyzing systems.

[4] `https://www.mathworks.com/products/simulink.html`

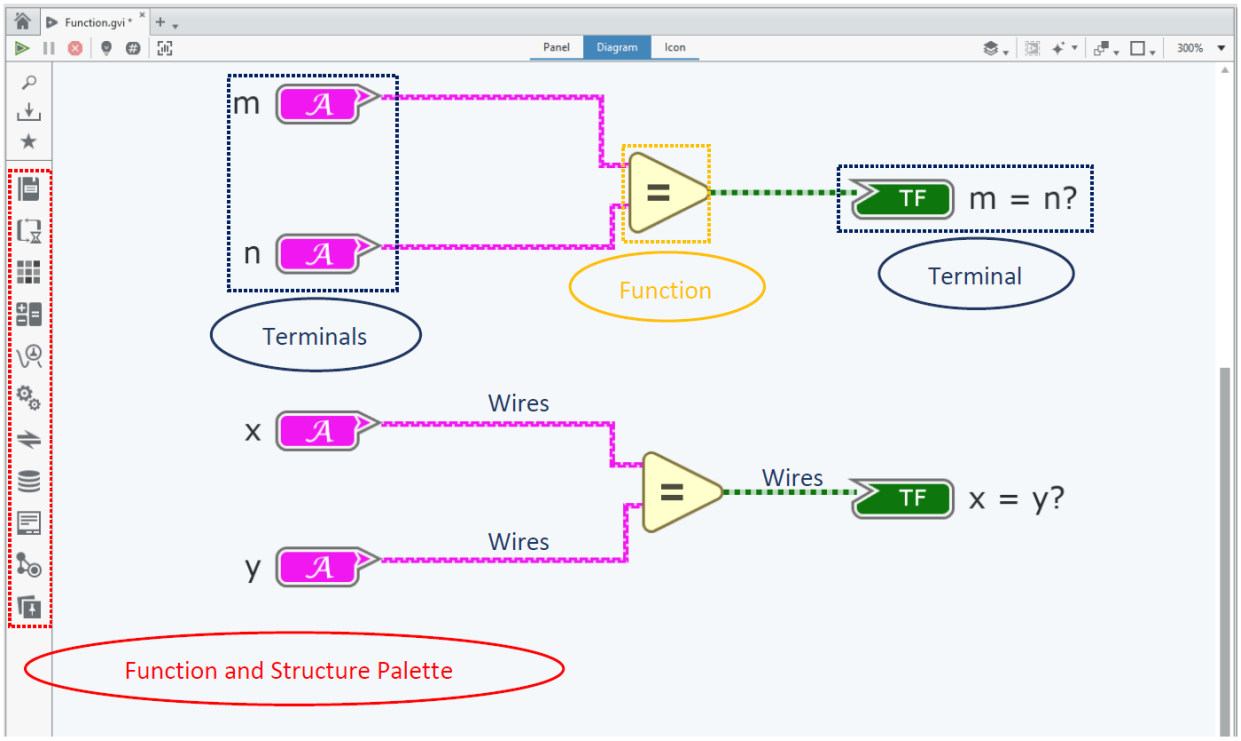


Figure 2.2: An Example of a VI's Block Diagram

Simulink is used in many scientific disciplines, such as automatic control [97], signal processing [82] and model-driven engineering [25]. Beyond academic research, Simulink is also widely adopted in industry (e.g., Boeing, Lockheed Martin, and General Motors).

Simulink adopts a graphical block diagramming interface. The Simulink library offers many systems modeling functions, such as math operations, string operations and control operations. A full list of functions integrated in this library can be found at the Simulink official website[5]. Figure 2.3 shows a simple example Simulink model which compares two strings. A Simulink model consists of four components: input (source), function block, output (sink) and connector. An input port is the input to the system, it can be a number, string, signal, or a wave. In Figure 2.3, the two constant strings "UserName" and "username" are the system input. A function block is the block where users process data. In Figure 2.3, the middle block which is shown as "Abc" =="Abc" is the comparison function block.

[5]`https://www.mathworks.com/help/simulink/block-libraries.html`

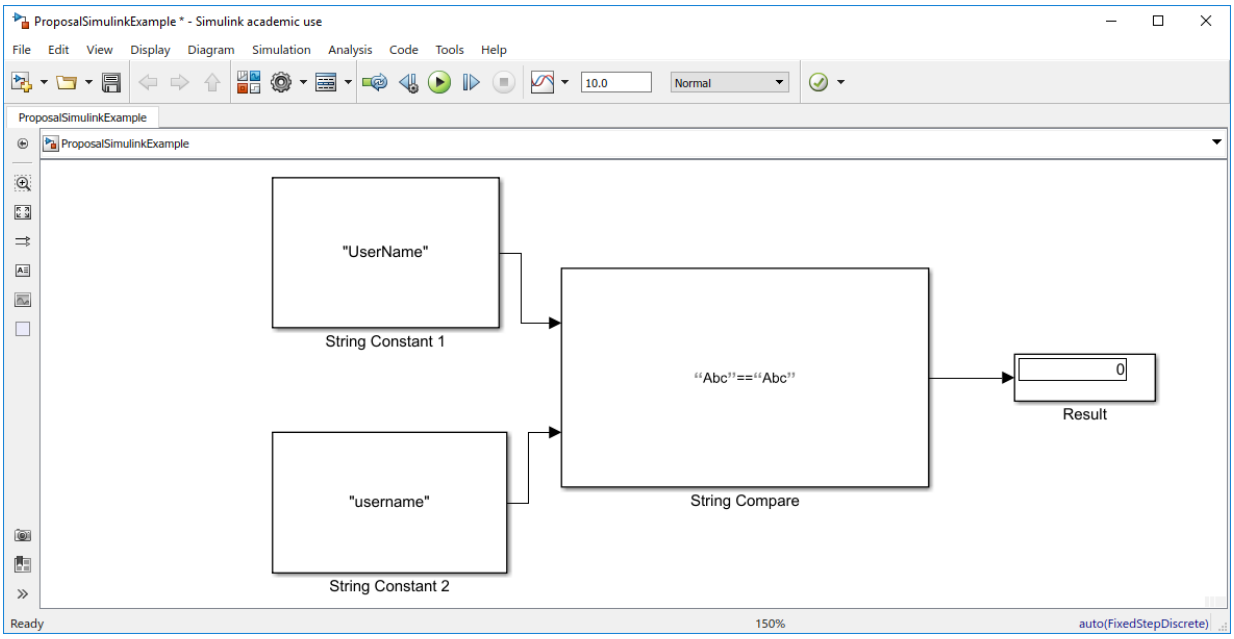


Figure 2.3: An Example of a Simulink Model

Integrated function blocks act as a "black box" in a Simulink model - users do not need to consider the implementation detail in function blocks, such as the String comparison function block shown in Figure 2.3. Simulink provides more than 300 function blocks in its library for different data processing needs. Simulink also supports user-defined function blocks. An output port is the result of the system. The result could be a number, a string, a Boolean value (like the example shown in Figure 2.3), an expression or a signal. For simulation systems with time constraints, Simulink provides the observation of system simulation at run-time to users. The connector is the component that connects all the blocks together in a model. The arrows represent the data flow direction.

2.3 Bad Smells

A bad smell (also known as *code smell*) is any characteristic in the source code of a program that possibly indicates a deeper problem [57]. Researchers have shown that bad smells cause deeper frustration to programmers [189] and take 32% longer to debug [190]. Since the term was first coined in the context of OOP [57], numerous studies have been

conducted to analyze bad smells from different perspectives, such as their causes [178], effects [153], summarization [159], and refactoring [1].

Bad smells are common in software development practice. For example, **Duplicated Code** is a bad smell which represents a portion of code that is duplicated, or cloned, in many different locations across source files. There are multiple reasons leading to the occurrence of duplicated code, such as copy-and-paste programming [89], or "add new features or to fix bugs" [15]. Listing 2.1 shows a piece of Java code which contains duplicated code.

```
1  public class MyClass{
2      public int addition(){
3          int a = 0;
4          int b = 1;
5          a++;
6          b++;
7          return a+b;
8      }
9      public void print(){
10         int a = 0;
11         int b = 1;
12         a++;
13         b++;
14         System.out.println(a+b);
15     }
16 }
```

Listing 2.1: Duplicated Code Example in Java

The above code snippet shows that Line 3 to Line 6 and Line 10 to Line 13 repeat the same piece of code executing the same functionality. Code that is duplicated may in-

crease the time and effort during maintenance due to the implicit coupling of the source code. Another common bad smell is **Dead Code** which is defined as "a section in the source code of a program which is executed but whose result is never used in any other computation" [48]. Listing 2.2 shows an example of this bad smell. In Listing 2.2, Line 2 is executed but not used in the function. The execution of dead code increases the program computation complexity and wastes computer memory. Moreover, even the result of the execution of a piece of dead code may never be used, it may lead to program exceptions and introduce bugs.

```
1 public int add{
2     int z = x * y;
3     return x + y;
4 }
```

Listing 2.2: Dead Code Example in Java

Bad smells in models (also know as "model smells") have been introduced by Arendt [12] to detect design flaws in models created within the Eclipse Modeling Framework (EMF) [169]. Model smells share some similarities with bad smells in OOP, but some smells apply to models only. For example, a counterpart of duplicated code is the model smell called **Duplicated Model Part**, which indicates a repetition of a model element across multiple locations in a model. On the other hand, there are some smells that only exist in the context of models. In textual languages, the execution follows a top-to-bottom order. In graphical models, however, execution flow often needs to be specified explicitly, using directional wires and arrows. Model smell **Unorganized Wires** [195] suggests that the wire used to connect different model components is untidy and unordered, thus increasing the difficulty to understand a model. This smell is an example of a model smell that does not have a corresponding bad smell in OOP.

Another LabVIEW-specific bad smell is **No Wait In a Loop**. As the name suggests, this bad smell means that there is not a Wait function in a loop, which may lead

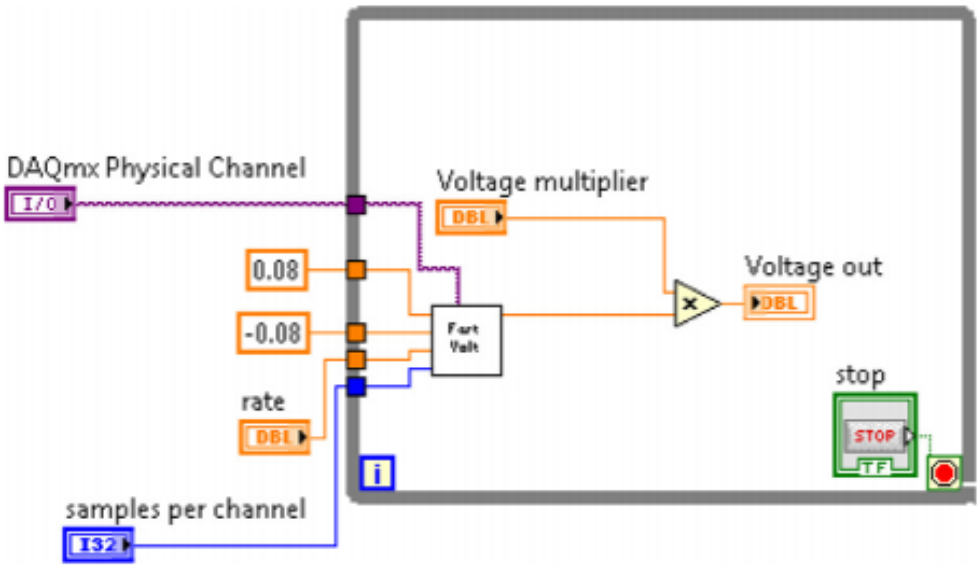


Figure 2.4: A LabVIEW Systems Model Without a Wait Function [35]

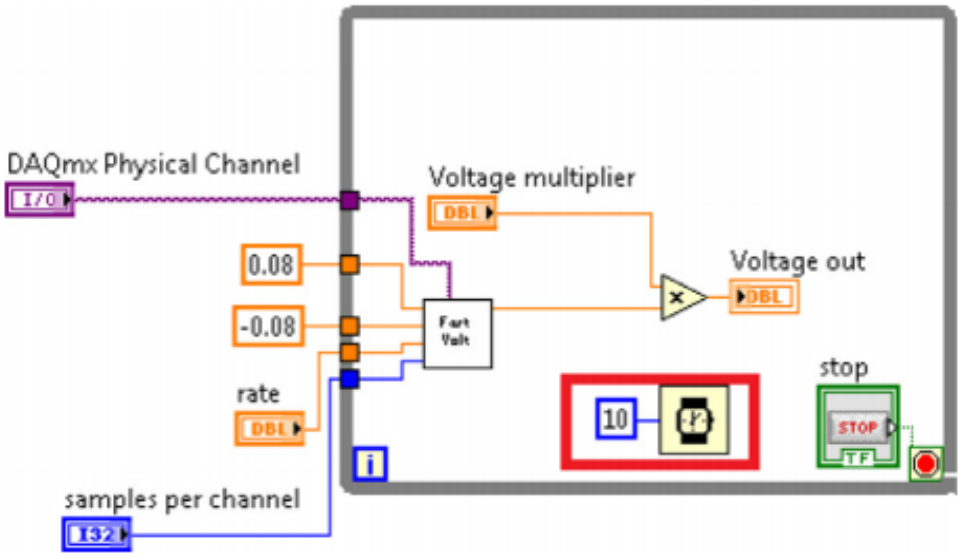


Figure 2.5: A LabVIEW Systems Model With a Wait Function [35]

to synchronization issues and unresponsive front panel elements [35]. Figure 2.4 shows a block diagram without a wait node and Figure 2.5 shows the same block diagram after adding a Wait function (red box in Figure 2.5).

Compared with numerous investigations of bad smells in OOP, limited work has been conducted to address the bad smells in systems models in the context of MBSE, es-

pecially for LabVIEW system models. Chambers and Scaffidi first identified bad smells in LabVIEW models through an interview with 13 LabVIEW application engineer specialists who had about five years of LabVIEW experience [34] .

Although Chambers and Scaffidi identified several bad smells in LabVIEW systems models, there are several limitations to their work. First, all of the bad smells summarized in their work are related to performance only. Other bad smell types, such as structural bad smells in a systems model, are not considered in their work. Second, the study was conducted among 13 experts who had many years' of experience with using LabVIEW. It is difficult to generalize their findings. For example, do other end-users (such as college students who have zero experience using LabVIEW) have the same challenges as experienced users? The paper failed to provide an evaluation of their summarization from a general perspective.

CHAPTER 3

MINING FORUM POSTS TO UNDERSTAND MODELING CHALLENGES

This chapter introduces the application of supervised and unsupervised machine learning techniques to analyze discussion forum questions. This chapter starts with supervised machine learning techniques to classify forum posts into pre-defined topic categories. As a supporting technique, web scraping is also discussed in this chapter to gather data from an online forum. After this, unsupervised learning techniques to identify latent topics in documents are discussed. Motivated by the results of the combination of supervised and unsupervised machine learning approaches, this chapter also presents the research questions that lead to the investigations in this dissertation.

3.1 Introduction

An online discussion forum is a website where various individuals from different backgrounds can discuss common topics of interest in the form of posted messages. Online discussion forums are useful resources for sharing domain knowledge. The discussion forums can be used for many purposes, such as sharing challenges and ideas, promoting the development of community, and giving/receiving support from peers and experts. Several researchers have identified benefits of online discussion forums from different aspects, such as education [86], individual and society development [133], and socialization [7].

To understand the issues and challenges faced by systems engineers during the development of a systems model, user posts from the LabVIEW online discussion forum were collected. The LabVIEW discussion forum has very rich resources for text summarization because most of the user-generated content in the forums is text-based. This study applied

text classification based on supervised machine learning techniques and topic modeling based on unsupervised machine learning techniques to the large collection of LabVIEW forum posts [197]. After downloading post questions through web scraping, supervised machine learning techniques were used to classify all the questions into four categories (i.e., "program," "hardware," "tools and support" and "others"). I compared three popular methods, including Multinomial Naïve Bayes [110], Support Vector Machine [176] and Random forest [26]. After this, this dissertation applied unsupervised machine learning technique - Latent Dirichlet Allocation (LDA) to delve into the largest category ("program") to find subtopics. The results show that users are most concerned about bad smells in systems models.

These findings serve as motivation for the following research. By applying text summarization techniques to extract online discussion forum key information, I discovered the challenges and problems faced by systems engineers based on the analyses of summarization results, which give direction toward future areas of investigation for the modeling community.

This chapter is structured as follows: Section 3.2 first introduces the process and technical details of supervised machine learning techniques in the context of text classification. After this, the LabVIEW discussion forum is used as a case study to introduce the empirical experiment that demonstrates how to apply supervised machine learning techniques to classify posts with predefined categories for the LabVIEW forum. In Section 3.3, unsupervised machine learning algorithms are first presented, followed by the empirical application of unsupervised machine learning techniques in gaining subtopics of LabVIEW posts in a specific category. The implications of this investigation are offered in Section 3.4.

3.2 Text Classification

Text classification is the process of assigning a set of predefined categories to text. Text classification is the heart of many applications, such as spam filtering [93], sentiment analysis [30] and readability assessment [115].

Text classification has been a popular research topic over the past decade. Nenkova and Mckeown [127] surveyed several text summarization techniques from the perspective of different phases involved in the summarization process. A different summarization approach is described by Altınel and Ganiz [9]. In their work, they divided text summarization into two major categories: traditional text summarization and semantic text summarization.

Traditional text classification: Traditional text summarization is based on the concept of Bag-of-Words (BoW) [151]. This classification technique separates documents into individual words and it only cares about their corresponding frequencies in a document. The information about word locations is discarded during the process of classification.

Semantic text classification: Semantic text classification tries to overcome the shortcomings in traditional text classification techniques by including semantic relations between words. Altinel and Ganiz [9] summarized five approaches for semantic text summarization:

- *Domain knowledge based approaches*: techniques that classify documents based on common knowledge bases (such as Wikipedia);
- *Corpus-based approaches*: similar to domain knowledge based approaches, but based on a training corpus instead of knowledge;
- *Deep learning based approaches*: text classification based on deep learning methodologies, such as Convolutional Neural Networks (CNNs) [96] and Recurrent Neural Networks (RNNs) [150];

- *Word/character sequence enhanced approaches*: string-matching techniques are applied to find string sequences in a document;

- *Linguistic enriched approaches*: lexical and syntactic rules are applied to extract key information compared with other approaches.

This chapter focuses on text classification analysis based on an existing corpus. This study applied machine learning techniques to analyze a corpus to predict new documents. In general, text classification based on machine learning techniques includes four steps:

Step 1: Data Pre-processing

Step 2: Feature Extraction

Step 3: Model Training

Step 4: Model Evaluation

Using the model obtained from Step 3, predictions on new data can be achieved. This section first describes these four steps in detail. At the end of this section, the case study from the LabVIEW modeling community is introduced as an example application of how these techniques are applied in a real scenario.

3.2.1 Data Pre-Processing

The first step is to refine the text in the dataset to remove noise and errors. In the text that is mined from a discussion forum, there are several characters (such as '['and '/') and words (such as 'I' and 'we') that are meaningless and not important for extracting the key information. To remove these characters, a common approach is to convert these characters into their corresponding Unicode characters. These Unicode characters will be removed when they are recognized. After removing meaningless characters, tokenization, a technique that breaks sentences into pieces (these pieces are called tokens) is applied to the text data. Tokens can be individual words or phrases. Usually, tokens are individual words. In this step, two operations are needed: stemming (e.g., converting words into com-

mon bases, such as 'windows' to 'window') and filtering (e.g., dropping stop words like 'a', 'an' and 'the').

3.2.2 Feature Extraction

After data cleaning, each document consists of a sequence of tokens (or symbols). In order to apply machine learning algorithms to these documents, the first step is to represent the sequence of tokens in each document as a numeric feature vector, which is recognizable for machines. One simple way to extract features is to use a word-document frequency matrix [194]. In this approach, the frequency of occurrence of each word is used as a feature for training a classifier. A more prevalent approach is TF-IDF [85], which is a measure that adopts two statistical methods - Term Frequency (TF) and Inverse Document Frequency (IDF). TF is the total number of times a given term appears in the text, and IDF measures the weight of a given word in the entire text. It is a measure of how much information the word provides. TF-IDF is the product of term frequency and inverse document frequency. There are several schemes for calculating TF and IDF [106]. Recently, Word2Vec [100] based on deep learning has become increasingly popular.

3.2.3 Model Training

The third step of the technique is to apply machine learning algorithms to train the machine to build the classification model. The training process starts with a manually labeled training set, whereby each document is manually labeled into predefined categories. The machine then learns a classification model that can automatically predict the class label of a new text document. A test set of documents are often used to verify the accuracy of the classifier. There are various techniques that can be used for text classification. This section introduces three widely used algorithms for text classification: Multinomial Naïve Bayes [110], Support Vector Machines [176] and Random Forest [26]. The reason why these three algorithms were chosen is because they are easy to implement and demonstrate relatively good classification results.

Multinomial Naïve Bayes

Multinomial Naïve Bayes classifier is a probabilistic algorithm based on Bayes' Theorem [18]. It is a widely adopted algorithm in Natural Language Processing (NLP), such as text classification, spam email detection, and sentiment analysis. Because of the nature of Bayes' Theorem, one precondition is assumed when applying Naïve Bayes classifiers: All features are conditionally independent given the underlying class category. Bayes' Theorem states that:

$$P(C \mid X) = \frac{P(X \mid C) \times P(C)}{P(X)} \tag{3.1}$$

where C represents one category of the text and X is one feature of the text. $P(C|X)$ is called **posterior probability**, which is the statistical probability of the category for a given document based on its numeric feature. $P(X|C)$ is called **likelihood**. It means that for a given category, the probability of feature X occurs. $P(C)$ and $P(X)$ are called **prior probability**. $P(C)$ is the probability of the occurrence of category C in the documents and $P(X)$ is the probability of the occurrence of feature X in the document. Based on this theorem, given an instance with feature X_1, X_2 ... X_n, the possibility of this instance belonging to category C_k is defined as:

$$P(C_k \mid X_1, X_2, \cdots X_n) = P(C_k) \times P(X_1 \mid C_k) \times P(X_2 \mid C_k) \times \cdots \times P(X_n \mid C_k) \tag{3.2}$$

Equation 3.2 is known as a **Bayes Classifier**. Multinomial Naïve Bayes means that in the above equation, $P(X_n \mid C_k)$ is a **multinomial distribution**. Multinomial distribution is a generalization of the **binomial distribution**. The difference between binomial distribution and multinomial distribution is that in binomial distribution, one event only has two outcomes (for example, tossing a coin only produces two possible results: heads or tails) while in multinomial distribution, one event has several possible out-

comes (such as rolling a die). The formula of multinomial distribution is defined as:

$$p = \frac{n!}{n_1! \times n_2! \times \cdots \times n_k!} \times (p_1^{n_1} \times p_2^{n_2} \times \cdots \times p_k^{n_k}) \tag{3.3}$$

where $n_1 + n_2 + \cdots + n_k = n$. For example, suppose there is a question like this: Tossing a die 3 times, what is the possibility of getting the number 3 one time and number 4 two times? In this example, a die is tossed three times, thus $n = 3$. Number 3 is expected to show up one time and number 4 two times, thus $n_1 = 1, n_2 = 2$. In each toss, the possibility of getting any number (1 to 6) is equal, thus $p_1 = p_2 = \frac{1}{6}$. Following the formula defined in 3.3, the possibility of getting the number 3 one time and number 4 two times is: $p = \frac{3!}{1! \times 2!} \times (\frac{1}{6}^1 \times \frac{1}{6}^2) \approx 0.0138$.

In Multinomial Naïve Bayes, the algorithm configuration is related to the setting of the **smoothing parameter** (which is usually called α). When α is set to 1 (also known as **Laplace Smoothing** [94]), it means 1 is added to each word count. When α is set to 0 (also known as **Lidstone Smoothing** [99]), 0 is added to each word count. In the following discussion throughout this chapter, $\alpha = 1$.

Support Vector Machine

A Support Vector Machine (SVM) [176] is a supervised machine learning algorithm that is widely used in classification problems. Similar to Multinomial Naïve Bayes, data preprocessing techniques, such as data cleaning and feature extraction, are executed as the first step in SVM. However, unlike calculating the frequency of each feature in the document in Multinomial Naïve Bayes, SVM plots each feature vector as an n-dimensional point (where n is the number of total features) with the value of each feature corresponding to the value of a particular coordinate. Then, SVM performs classification by finding the hyperplane that separates points from the different classes ("support vectors" are the points near the boundary hyperplane). Thus, the core problem of learning an SVM classifier is to find the best separation hyperplane.

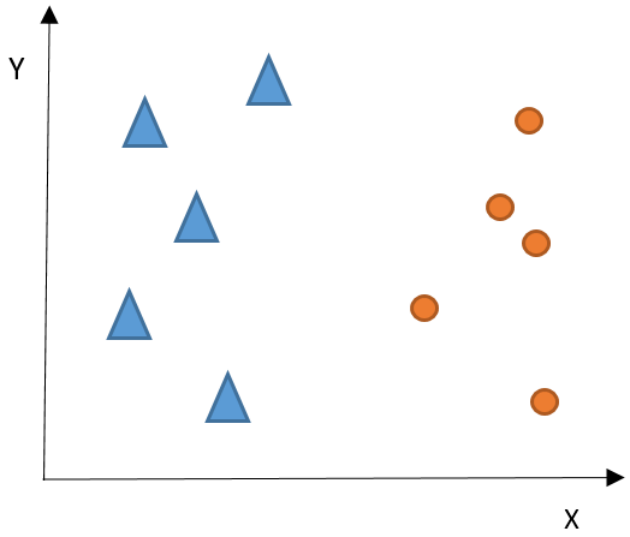


Figure 3.1: Two Categories Containing Different Data Points (Adapted from [49])

This section will illustrate SVM with a simple example (adapted from [49]). Suppose there are two categories (represented by a blue triangle and orange circle) and the data has two features (x and y). Figure 3.1 shows this example. There are several possible hyperplanes that can be used to separate points from these two categories (such as the dotted black line and bold black line in Figure 3.2). Research has shown that the hyperplane with largest margin (shown in the bold black line) is the best [83]. For a dataset that inseparable in 3.2 dimensional space, SVM transforms the data to a higher dimension to do separation. Figure 3.3 shows such a situation.

For a dataset with n features (in other words, a dataset in an n-dimension), SVM finds an $n - 1$ dimension hyperplane to separate it. When data is not linearly separable, SVM transforms an initial dataset to a higher dimensional dataset where it becomes separable. However, it is almost impossible to try out all of the transformations from a lower dimension to a higher dimension. To solve this issue, SVM computes the pair-wise dot products. For a given pair of vectors and a transformation into a higher-dimensional space, there exists a function (which is known as a **kernel function**) that can compute the dot product in a higher-dimensional space without explicit transformation. This is

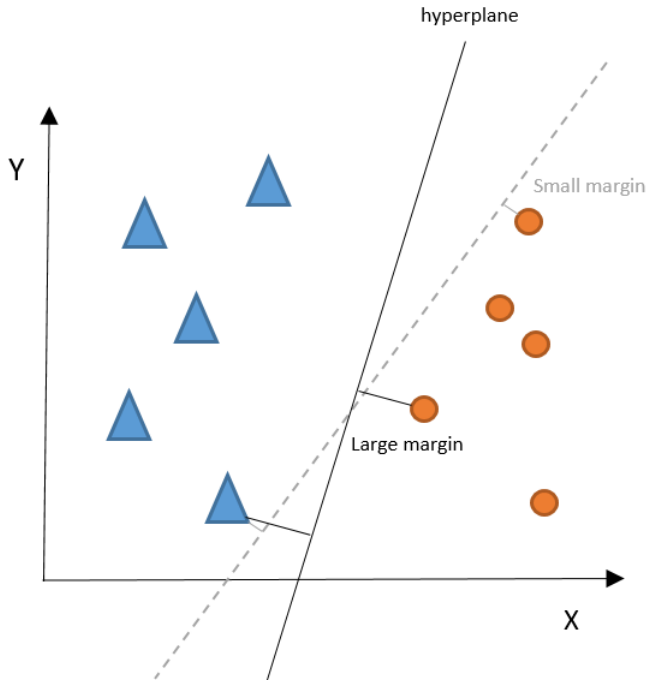


Figure 3.2: Possible Separations of Two Categories (Adapted from [49])

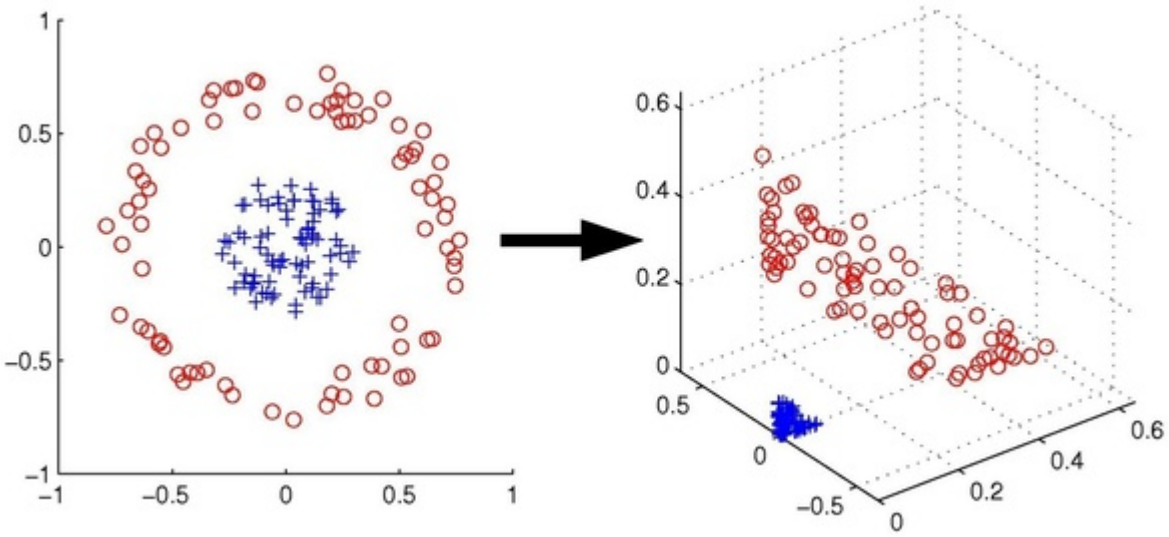


Figure 3.3: Convert Inseparable Data in 2D Space to 3D Space (Adapted from [50])

called **kernel tricks**, which reduce computational complexity and makes SVM feasible for use with high dimension datasets [77].

The algorithm configuration for SVM is more complicated than Multinomial Naïve Bayes. There are several parameters used in SVM, such as a penalty parameter, kernel function type, shrinking heuristic, and kernel size. There are two important parameters for SVM: penalty parameter and kernel function type.

Penalty parameter (or complexity parameter) shows the tolerance of error. The larger the value of the penalty parameter, the less error tolerance for the algorithm. Over-fitting is more likely to happen with higher values of a penalty parameter. On the contrary, a smaller penalty parameter value offers more tolerance of error and under-fitting is more likely to happen. The kernel function type refers to which kernel is to be used by an SVM engine. Common kernel functions include linear kernel, poly kernel and RBF (Radial Basis Function) kernel. The penalty parameter was set to 1 and linear kernel was adopted as the kernel function in the experiment in this dissertation.

Random Forest

Random Forest [26] is a supervised machine learning technique based on decision trees. A decision tree is a tree-like structure where internal nodes represent a test condition on a feature attribute. The answer for the test is binary (only "yes" or "no") or multi-way. Each leaf is a final predicted class. When using a decision tree to classify text, it starts from the root, and selects a sub-branch based on the test result on the root node. This decision process is repeated until a leaf node is reached, where a class is predicted. Figure 3.4 is a decision tree showing a simple post classification strategy (the number below each category indicates the prediction probability).

A random forest classifier is an algorithm based on the ensemble of decision trees. It involves learning a collection of decision trees to make a final prediction based on voting. The random forest algorithm first creates several decision trees randomly to evaluate

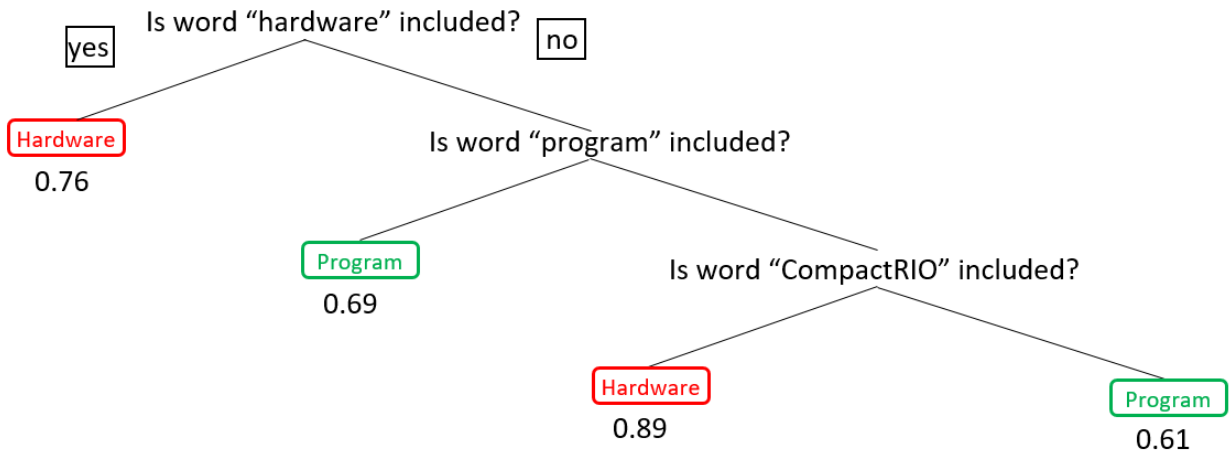


Figure 3.4: An Example of Classification Strategy Based On Decision Tree

the performance of each decision tree constructed and then averages the results to improve the predictive accuracy.

The algorithm configuration for random forest is closely related to the setting of random trees. In general, two parameters are crucial to the performance of random forest. First, the depth of each tree is important. A deep tree may increase the prediction accuracy, on the other hand, it may increase the calculation time. Second, the number of max features is critical, which decides how many attributes a tree uses every time a subtree is created. The higher value this parameter holds, the more attributes a tree has. Usually, increasing this value could improve the performance of a random forest. Similarly, it also increases the running time. In the experiment conducted in this dissertation, unlimited tree depth and $\log_{10} \frac{Number\ of\ Feature}{2}$ as the max number were assigned to each tree in the forest.

3.2.4 Model Evaluation

Evaluation Approaches

The most dominant way to evaluate the model built from a machine learning algorithm in the context of text classification is to apply **K-fold Cross Validation** [91], which divides the training data into K parts equally. Suppose $K = 10$ and these 10 parts are labelled

as A, B, C, D, E, F, G, H, I, J. In the first iteration, part A through part I are used as training datasets (nine parts in total) and use part J as a testing dataset. In the second iteration, part B through part J become training datasets and part A becomes a testing dataset. Following this pattern, there will be 10 iterations in total. Evaluators calculate the average value for these 10 iterations as the final evaluation result. The reason why K-fold cross validation is applied is because cross-validation is a good evaluation technique when there are no sufficiently large training and testing sets [170].

Evaluation Metrics

There are several evaluation metrics in the context of text classification problems. The most basic and simple one is **Accuracy**, which can be defined as:

$$\textbf{Accuracy} = \frac{Number\ of\ Correct\ Predictions}{Total\ Predictions} \tag{3.4}$$

Another metric is **Receiver Operating Characteristic** (ROC) curve, which is a plot that is used in a binary classification. The ROC curve represents a model's ability to discriminate between positive and negative classes. In an ROC curve, an area of 1.0 means a perfect prediction and 0.5 represents a model that is equivalent to a random guess.

Another frequently used metric is called the **Confusion Matrix** (also known as **Error Matrix**). It is an evaluation metric that presents the accuracy of a model with two or more categories. A confusion matrix takes the following form:

True Positive (TP) means that an instance is predicted as positive and the prediction is true (e.g., a post related to "program" is predicted as "program" correctly). Similarly, **True Negative** (TN) means that an instance is predicted negative and the prediction is true (e.g., a post not related to "program" is not predicted as "program"). **False Positive** (FP, also known as **Type 1 Error**) indicates that an instance is predicted positive, but it is negative. For example, a post is related to "hardware," but it is predicted as "program." **False Negative** (FN, also known as **Type 2 Error**) indicates that an in-

Table 3.1: Confusion Matrix

		Prediction Outcome		
		p	**n**	**total**
Actual Value	**p′**	True Positive (TP)	False Negative (FN)	P′
	n′	False Positive (FP)	True Negative (TN)	N′
	total	P	N	

stance is predicted negative, but it is positive. For example, a post related to "program," but it is predicted as "hardware."

Precision, recall and F-measure are widely used metrics in performance evaluation. **Precision** (also called **positive predictive value**) is the "fraction of relevant instances among the retrieved instances" [10]. It usually refers to the accuracy of the result. It is defined as:

$$\textbf{Precision} = \frac{TP}{TP + FP} \tag{3.5}$$

Recall (also known as **sensitivity**) is the "fraction of relevant instances that have been retrieved over the total amount of relevant instances" [10]. It usually refers to the completeness of the prediction result. Recall is defined as:

$$\textbf{Recall} = \frac{TP}{TP + FN} \tag{3.6}$$

Combining the measurement of Precision and Recall, a metric that evaluates the correctness and completeness at the same time can be obtained. This metric is called the **F-measure** (also known **as F-score**), which is defined as:

$$\textbf{F-measure} = \frac{2 \times Precision \times Recall}{Precision + Recall} \tag{3.7}$$

3.2.5 Text Classification for LabVIEW Modeling Community

This section presents the empirical text classification based on the posts mined from the LabVIEW discussion forum. The section first introduces web scraping – a technique used in this experiment to obtain the textual data from the forum. Following this, a discussion is given on how supervised machine learning techniques are applied to find post distribution among predefined categories. The experimental result is given based on the empirical investigation in the end of this section.

Data Source and Pre-Processing

Web Scraping

Web scraping is the process of automatically mining data to collect information from a rendered web page. There are several techniques for web scraping, ranging from manual examination to fully automated approaches [119] [125]. The most naive approach for web scraping is human copy-and-paste. However, this approach requires much human effort when the dataset is large. One of the most popular approaches for web scraping is based on parsing text represented as **HTML** (**H**yper-**T**ext **M**arkup **L**anguage). This section focuses on HTML parsing. In general, web scraping based on HTML parsing for an online discussion forum consists of three main parts:

- **Data Acquisition**. This step builds a connection between a client and a website. It uses an HTML request library or a headless browser (i.e., a web browser without a graphical user interface). A popular method for data acquisition is to use a Node.js request-promise module [164]. There are several tools supporting this method, such as Puppeteer [46] and PhantomJS [20]. By sending a request to the URL of a website, this method receives a promise that specifies the format of the returned object. A client will receive the raw HTML format from the web page if the connection is built successfully.

- **Source Inspection**. This step finds the exact pattern/tag a client is looking for in the HTML code. Most current web browsers support the inspection of the source code of a web page. An HTML selector is often used to locate the target element and extract HTML and text within the selected element.

- **Data Parsing**. The last step is to parse the HTML by calling handler methods when start tags, end tags, text, comments, and other markup elements are encountered. Clients can customize the output to export and save the results to different file formats, such as a simple text file, an Excel file, or a CSV file.

Dataset

The dataset for this experiment was collected from the official LabVIEW discussion forum. With the web scraping approach mentioned above, I implemented a JavaScript web crawler to automatically retrieve all the user questions posted on the LabVIEW discussion forum. I collected 184,203 questions which were asked by users from May 18^{th}, 1999 to February 15^{th}. Among these posts, the following were removed: 1) Posts that are not written in English (4,116 posts in total); 2) Meaningless posts (such as empty posts, posts with characters only, etc., 2,592 posts in total). After filtering, 177,495 posts remained in the dataset.

The first step is to preprocess text data in the posts through stemming and lemmatization (i.e., find the root of a word, such as converting 'playing' to 'play', 'books' to 'book'), and removed stop words (i.e., common words such as 'the,' 'an,' 'a'). The second step is to represent each post with a numeric vector based on TF-IDF.

Machine Training

To train the machine, 1,800 posts randomly chosen from the dataset were labelled manually. Based on the exploration of the dataset, each post was labelled with one of the four categories: **program**, **hardware**, **tools and support**, and **others**. The **program** cate-

gory includes any posts related to program problems, such as program design/debugging, graph processing, coding/language, function/control, program performance. For example:

> *" . . . Does LabVIEW 7 have an easy way allow the user at runtime to change the default values of controls? . . . "*

> *" . . . Hello. I am new in the LabVIEW, and I have to do a task. When I will get lowercase it has to be display as an uppercase and uppercase to lowercase . . . Any ideas how to fix it?"*

Any question related to hardware issues is classified into the **hardware** category, such as hardware driver, computer hardware, controller, chassis, sensor and any external hardware supported/developed by National Instruments (a complete hardware list can be found at the National Instruments official website). Examples of hardware related posts are:

> *" . . . Does anybody have experience booting LabVIEW RT disklessly over the internet using PXE on a PXI Embedded slot 1 controller? . . . "*

> *" ... Hello has somebody developed a LabView driver for the Diamond-MM-16 board. If yes, I'm interested ..."*

A post was labelled as **tools and support** if the question is asked about LabVIEW itself (such as LabVIEW version, license and installation) or any existing external tools/architectures/libraries supported by LabVIEW. For example:

> *" . . . How can I download all NI software and packages that our academic license covers? I know that the license covers many tools, and I wish to download and install them all . . . "*

> *" ... I want to know if there is any easy way that I can update LabVIEW 11.0 to 16.0. Do I have to uninstall the old one and download the new one? . . . "*

Table 3.2: Unsupervised Text Classification Evaluation Results

	TermFreq-1	**TermFreq-2**	**TermFreq-3**	**TermFreq-5**
Multinomial Naïve Bayes	Pre:0.766 Rec:0.753 F-1:0.758	Pre:0.753 Rec:0.728 F-1:0.740	Pre:0.761 Rec:0.733 F-1:0.745	Pre:0.761 Rec:0.733 F-1:0.745
SMO	Pre:0.701 Rec:0.726 F-1:0.711	Pre:0.709 Rec:0.727 F-1:0.715	Pre:0.704 Rec:0.719 F-1:0.710	Pre:0.702 Rec:0.714 F-1:0.707
Random Forest	Pre:0.704 Rec:0.721 F-1:0.648	Pre:0.699 Rec:0.714 F-1:0.641	Pre:0.701 Rec:0.724 F-1:0.656	Pre:0.701 Rec:0.724 F-1:0.656

Any posts that do not belong to the above categories will be labelled as **others**. For example:

> *"Is it the word "vi" pronounced like "vee" or "v-i" (each letter separately). All opinions welcomed."*

> *"Happy New Year to all the LabVIEW users."*

Evaluation Results

In this work, 10-fold cross validation is used to evaluate the classification performance. The evaluation metrics are Precision, Recall and F-measure. The performance result of applying Multinomial Naïve Bayes [88], Sequential Minimal Optimization (SMO) [14] (a fast implementation of SVM) and Random Forest [186] on the training set is shown in Table 3.2. In Table 3.2, TermFreq-N means keeping words that appeared at least N times in the dataset. Pre refers to Precision and Rec refers to Recall.

Prediction

From Table 3.2, it is clear that Multinomial Naïve Bayes (minTermFreq = 1) has the best performance (marked in red in Table 3.2). Therefore, I chose this model as the prediction model. After applying Multinomial Naïve Bayes (minTermFreq = 1) to predict all the

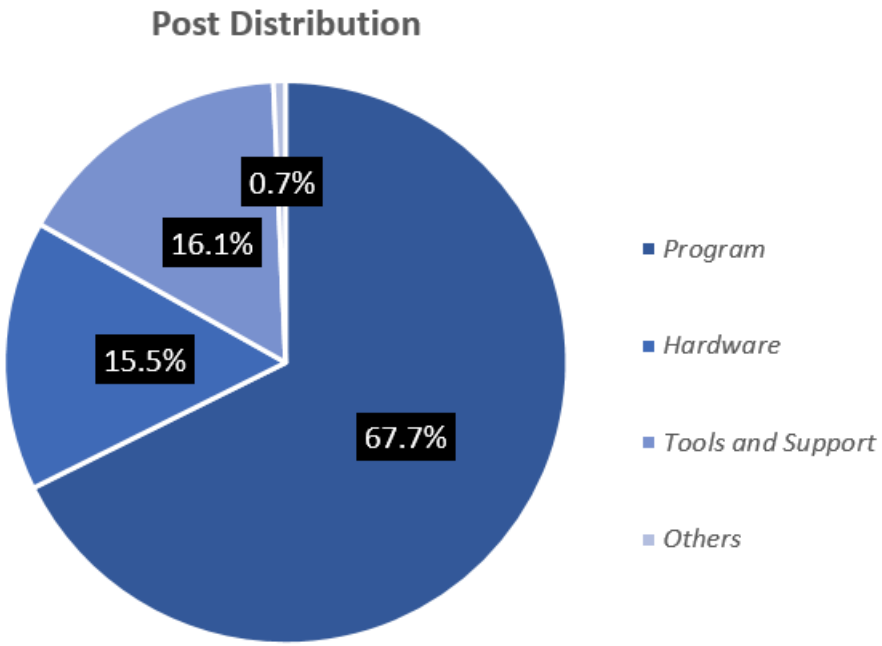


Figure 3.5: Post Distribution Prediction for LabVIEW Discussion Forum

posts, a post distribution for all the posts in the dataset was obtained. The post distribution is shown in Figure 3.5. Figure 3.5 shows that posts related to program have the largest portion (120,085 posts are related to program out of 177,495 posts). There are 28,636 posts that are predicted as tools and support, which is a little more than posts predicted as hardware (27,541 posts). Only 1,233 posts are predicted as others.

The prediction result shows that most LabVIEW discussion forum users are interested in program-related topics. Therefore, it was beneficial in this analysis to understand what subtopics are discussed most among these posts. However, with a wide range of topics, it is difficult to suggest predefined subcategories. To this end, I applied unsupervised machine learning techniques to delve into subtopic clustering among different program categories.

3.3 Topic Modeling

Applying unsupervised machine learning techniques to find topics in the text is referred to as **topic modeling** or **topic clustering** [78]. The task of topic modeling is to

find groups (clusters) of similar topics in the dataset. The similarity is computed through a similarity function. The term "unsupervised" indicates this process is done automatically without any human seeding or initiation of the process. The cluster can be at different levels of granularity, such as sentences, phrases and words. Conventional unsupervised ML techniques for topic clustering include k-means clustering [185], hierarchical clustering [84], and LDA [21]. A comparison of these approaches is summarized by Karypis et al. [168]. Unsupervised machine learning for topic modeling mainly consists of four steps:

Step 1: Data Pre-processing

Step 2: Feature Extraction

Step 3: Unsupervised Clustering

Step 4: Results Interpretation

The approaches used in Step 1 and Step 2 are similar to supervised machine learning techniques for text classification. Therefore, this section mainly discusses the most popular technique used in Step 3 - Latent Dirichlet Allocation. The empirical example from the LabVIEW online discussion forum is introduced at the end of this section to show how unsupervised clustering is adopted in practice.

3.3.1 Latent Dirichlet Allocation

Latent Dirichlet Allocation (LDA) is a statistical topic modeling approach that is widely used in NLP. It automatically discovers topics in the documents. The topic inference for LDA is built upon Dirichlet-multinomial distribution [147], a family of discrete multivariate probability distributions from finite non-negative integers. The number of clusters needs to be specified manually in LDA at the beginning of the process. There are two principles in LDA:

- Each document contains multiple topics.
- Each topic contains multiple words.

For example, given three documents (e.g., posts from the LabVIEW discussion forum):

Post 1: "LabVIEW program is hard to debug."

Post 2: "CompactRIO cannot connect to LabVIEW."

Post 3: "Share your example program with others in the NI community."

Suppose the goal is to find two topics among these posts. After applying LDA to these three posts, it may produce a result like:

Building topics for documents:

Post 1: 100% belongs to Topic A;

Post 2: 100% belongs to Topic B;

Post 3: 80% belongs to Topic A, 20% belongs to Topic B.

Building words for topics:

For Topic A: 95% program, 70% debug, 30% share ...

For Topic B: 90% CompactRIO, 60% connect ...

At this point, LDA cannot specify what is Topic A and what is Topic B. The result of LDA needs human (usually domain expert) interpretation. Based on existing domain knowledge, Topic A could be interpreted as "program" and Topic B could be interpreted as "hardware."

In LDA, there exist 3 important parameters:

- The number of topics K: the number of requested latent topics to be extracted from the training corpus.
- Prior estimator α: the average frequency that each topic occurs within a given document.
- Topic-word estimator β: a collection of k topics where each topic is given a probability distribution over the vocabulary used in a document corpus.

In general, higher α values mean documents contain more similar topic contents. The same is true for β – a higher β value means a topic contains more similar topic words. In this experiment, the value of K was three. The algorithm was implemented based on the LDA Python library – Gensim[1]. In Gensim, the value of α are β are tailored to the structure of the dataset it uses. This automatic adjustment is known as **asymmetric prior**.

3.3.2 Topic Modeling for LabVIEW Modeling Community

This section discusses topic modeling using the case study from the LabVIEW discussion forum. Section 3.2 in this chapter introduced post classifications based on the LabVIEW discussion forum and the results suggested that LabVIEW users are more interested in program related questions. Based on this observation, this dissertation focuses on program-related posts. Because the posts in the forum spanned a very wide range of topics, it is difficult to suggest predefined subcategories. To this end, unsupervised machine learning techniques were adopted to identify several important sub-topics. Because LDA is a powerful tool for discovering and exploiting hidden topics [103], this section only discusses LDA for topic clustering.

Dataset

As mentioned in Section 3.2, there are 120,085 posts (out of 177,495) that are related to the topic of program. Among these 120,085 posts which were predicted as program-related, this experiment chose posts with a prediction probability value equal to one, which means the prediction model is very confident that the selected posts are related to the program category. Based on this threshold, 92,655 posts remained after filtering.

[1] `https://pypi.org/project/gensim/`

Clustering Results

I developed a Python program to model the dataset based on pyLDAvis [163], a library for helping users interpret the results in a topic model.

The first step for LDA is to choose a clustering number. In this experiment, three different cluster numbers were examined: three, five and 10. Experimental results suggest that when using five ($K = 5$) and 10 clusters ($K = 10$), there are many overlapping concepts across each cluster. Therefore, this study used three topic clusters in the following empirical discussion. The result for one iteration is as follows:

(1, 0.014×"control" + 0.014×"array" + 0.009×"button" + 0.009×"event" + 0.008×"panel" + 0.008×"problem" + 0.007×"change" + 0.007×"try" + 0.007×"front" + 0.007×"value"')

(2, 0.008×"try" + 0.008×"array" + 0.008×"write" + 0.008×"example" + 0.007×"signal" + 0.007×"error" + 0.007×"problem" + 0.007×"graph" + 0.006×"input" + 0.006×"sample"')

(3, 0.014×"graph" + 0.009×"problem" + 0.008×"control" + 0.007×"try" + 0.007×"display" + 0.007×"output" + 0.007×"waveform" + 0.006×"array" + 0.006×"input" + 0.005×"could")

Because LDA is a probabilistic model, when running multiple times, different results may be produced. This experiment applied LDA to the same dataset five times. In order to find uniqueness in each cluster, the algorithm eliminated repeated words across different clusters. Table 3.3 shows the frequency of unique terms after five applications. Combined with Table 3.3 and manual examination based on existing domain knowledge, three clusters can be interpreted as: program change, file operation and graph processing.

Table 3.3: Unique Term Frequency Using LDA

Unique term frequency	5	4	3	2	1
Unique terms in Cluster 1		change	event, button	graph, panel, event	example, control, display, front
Unique terms in Cluster 2			output, value, function	write, number, file	attach, example
Unique terms in Cluster 3		graph	waveform, sample	point, display	create, value, however, input, signal, control, output, could

3.4 Discussion and Implication

The combination of supervised and unsupervised ML techniques revealed several interesting insights. For example, the results suggest that most LabVIEW users are concerned about program-related questions compared with other categories. Additionally, in the program-related posts, many posts focus on program change, file operation and graph processing.

This empirical investigation also reveals several interesting research implications. From the results in Section 3.2, the word "change" always appears in the largest cluster. This study was particularly interested in understanding posts containing the keyword "change," in which many of the posts implied an intention of "refactoring," as used in software engineering. By examining posts belonging to the first cluster, many posts that discussed the effects of poor designs for LabVIEW systems models were discovered. For example,

> *"...we have a big LV-project with some conditional disabled or with a switch unused VI's. But LabVIEW still tries to load them when building an application (even when they are not used). Something like ignore unused/missing VI's during built?..."*

Some posts contain bad smells that are common in OOP. For example,

"... A lot of duplicated code. Most of it is the same except selected cluster element changes..."

Based on this observation, this research was particularly interested in bad smells in LabVIEW systems models. Therefore, the general objective of this dissertation is *a better understanding of bad smells in systems models*. To be more specific, the examination focuses on the following three aspects:

1. What bad smells do systems engineers face in the practice of systems model (i.e., program) development?

2. What do systems engineers perceive about the bad smells summarized in existing literature?

3. How to identify prominent bad smells to help systems engineers to improve the quality of systems models?

The following three chapters are developed from these research questions. Chapter 3 presents a novel way to summarize bad smells by mining systems engineers' forum posts. Motivated by the results that are discovered from Chapter 3, Chapter 4 introduces an empirical evaluation of bad smells in systems models. Driven by the results from Chapter 4, Chapter 5 suggests approaches to identify prominent bad smells in systems models.

CHAPTER 4

BAD SMELL SUMMARIZATION FROM END-USERS

This chapter presents an overview of BESMER, my tool to summarize bad smells from end-users. This chapter first discusses the major steps in BESMER, followed by a presentation of the results this study obtained from applying BESMER to the LabVIEW discussion forum along with some posts mined from the forum. This chapter concludes with insights gained from mining forum posts.

4.1 Introduction

Bad smells in Object-Oriented Programming (OOP) have been studied by many researchers over the past two decades [193] [141]. However, compared with OOP bad smells, investigation of bad smells for systems models is very limited. Systems models are central to Model-Based Systems Engineering (MBSE) - they provide a way to explore, update, and communicate system aspects to stakeholders, while significantly reducing or eliminating dependence on traditional documents. Systems models also provide essential knowledge during system implementation and help the analysts and designers to understand and communicate the functionality of a system. Burmark [28] has shown that the human mind works better with visual elements. Applying models to visualize systems enables end-users to view all of the system components and how they work together [38]. Systems models developed by tools such as Simulink and LabVIEW are of great importance in the context of Model-Driven Engineering (MDE).

To understand the bad smells in LabVIEW systems models, this dissertation introduces **BESMER** (**B**ad sm**E**ll Summarization fro**M** **E**nd use**R**s), a bad smell summarization process based on mining end-user discussion forum posts (the project name is in-

spired by the **Bessemer process**[1] in steel making to remove impurities from raw iron). BESMER collected user posts from the LabVIEW online discussion forum as the dataset (a JavaScript web crawler was implemented to retrieve all of the post threads from May 1999 to February 2019).

BESMER first finds bad smells in LabVIEW systems models based on bad smells already described in literature. This process is similar to searching papers using keywords in an electronic library: BESMER searches bad smells documented in existing literature that appear in user posts. After collecting the posts that contain bad smells from this step, BESMER applies a data mining technique (Bag of Words [151]) to analyze which words are frequently mentioned by end-users when introducing posts containing bad smells. BESMER then merges these words with the common bad smells that are known in OOP [57] to find any post that contains bad smells based on the concept of OOP smells. BESMER then searches of all of the posts again using the newly combined keywords. The results after these two steps represent the following types of posts: 1) user posts containing bad smells related to systems models as described in literature; and 2) user posts containing bad smells related to OOP. However, different end-users may choose different words/phrases when describing the same problem. In order to address this difference in word usage, BESMER uses machine learning techniques as a third level of bad smell summarization. Posts obtained from previous steps are labelled as positive training samples and randomly selected posts from the dataset that did not express any bad smells are labelled as negative training samples. BESMER then trains the machine to make it learn and recognize whether a new post contains a bad smell. The research questions addressed in this chapter include:

- **Research Question 4.1 (RQ 4.1)**: Are the bad smells in existing literature also mentioned by end-users from the LabVIEW online discussion forum?

[1] `https://en.wikipedia.org/wiki/Bessemer_process`

- **Research Question 4.2 (RQ 4.2)**: Are there any new bad smells mentioned by end-users from the online discussion forum that are not mentioned in existing literature?

- **Research Question 4.3 (RQ 4.3)**: What are the possible inferences that can be made through the examination of end-user posts, in relation to software quality and bad smells?

The contribution of this work is twofold: first, BESMER confirmed that some bad smells discussed in existing literature are also "bad" from a systems engineer's perspective. Second, BESMER discovered bad smells mentioned by end-users that are unique to the LabVIEW and the systems modeling context. BESMER can also be applicable to other domains and contexts, including the investigation of data mining for other modeling tools and user forums, as well as more general application to other online discussions among domain experts.

4.2 Related Work

Although bad smells have been a popular research topic for over two decades in OOP since the term gained mainstream usage when popularized in the book by Martin Fowler in 1997 [57], few research has been conducted within the context of Visual Programming Languages (VPLs). This section first discusses bad smells in the context of systems models built in LabVIEW and Simulink, followed by a summary of bad smells in other visual programming languages.

The research on bad smells in LabVIEW is very limited. Chambers and Scaffidi first identified bad smells in LabVIEW models by conducting an interview with 13 LabVIEW specialists who had an average of five years of LabVIEW experience [34]. Although the authors identified several bad smells, there are several limitations to their work. First, all of the bad smells summarized in their work are only related to performance. Other bad

smells, such as structural bad smells in a systems model, are not considered in their work. Second, the study was conducted only among 13 experts who had many years' experience with using LabVIEW. It is difficult to generalize their findings. For example, it is still not clear if new developers (such as college students who have zero experience using LabVIEW) have the same understanding of bad smells as the experienced users interviewed in their study.

Another popular tool for systems modeling is Simulink developed by MathWorks. One way to check the existence of bad smells in Simulink models is to follow the pre-defined model design guidelines [22] by the MathWorks Automotive Advisory Board (MAAB, an independent board that develops guidelines for using MATLAB, Simulink, Stateflow and other products), which was originally established to coordinate feature requests from several key customers in the automotive industry. The guideline summarizes the recommended stylistic and architectural constraints within a Simulink model. Gerlitz et al. proposed a catalog for the detection and handling of Simulink models [64]. In their work, they summarized bad smells in Simulink based on their own experience with creating and maintaining Simulink models in the automotive domain.

Both Simulink and LabVIEW models are examples of Visual Programming Languages (VPLs). A VPL is a programming language that "lets users create programs by manipulating program elements graphically rather than by specifying them textually" [29]. VPLs allow programmers to describe a system with visual expressions, spatial arrangements of text and graphic symbols. Many ideas of VPLs are based on the concept of "Drag and drop boxes and arrows," where boxes represent entities and arrows represent relations between different entities. VPLs are applied in different areas in computer science, such as computer science education (such as block languages Snap![2]), and system simulation (such as MATLAB/Simulink and LabVIEW).

[2] `https://snap.berkeley.edu/`

Block-based and Visual Programming Languages are widely used in K-12 computer science education [17] [45] [74]. Aivaloglou and Hermans's investigation summarized several bad smells in Scratch programs and proposed bad smell detection approaches [6]. In their work, the summarization of bad smells includes dead code, large class, large method, duplicated code, and incomplete scripts. Similarly, Hermans et al. investigated bad smells in two other educational VPLs - Lego Mindstorms EV3 and Microsoft's Kodu. Compared with Aivaloglou's work [75], Hermans's work presented a more complete list of bad smells - dead code, deprecated interface, duplicate code, feature envy, inappropriate intimacy, lazy class, long method, many parameters, message chain, useless block and unused field.

Although game engines like Unity[3] and Blender[4] (or similar engine tools) apply visual blocks for program implementation, these tools are highly dependent on textual OOP (such as C# for Unity and Python for Blender) to fulfill dynamic behaviors in the system. The use of visual blocks are mostly adopted to the creation of static scenes. Therefore, if there exist code smells in these tools, the smells are typically related to code smells in OOP languages [37] [160].

4.3 Methodology

This section discusses BESMER in detail. BESMER includes three levels of summarization. First, BESMER refers to bad smells in related work for systems models to find out what bad smells discovered by existing literature also are also mentioned in online discussion forum posts by end-users (Level 1 summarization). Second, BESMER analyzes the posts collected from Level 1 and finds the most frequently mentioned keywords. After obtaining these keywords, BESMER combines these keywords and the concept of bad smells in OOP, repeating the first step to seek any bad smells in systems models related to OOP bad smells. An extended set of user posts related to bad smells is produced after the second level of summarization. Finally, BESMER uses the posts obtained from Level 1

[3] `https://unity.com/`
[4] `https://www.blender.org/`

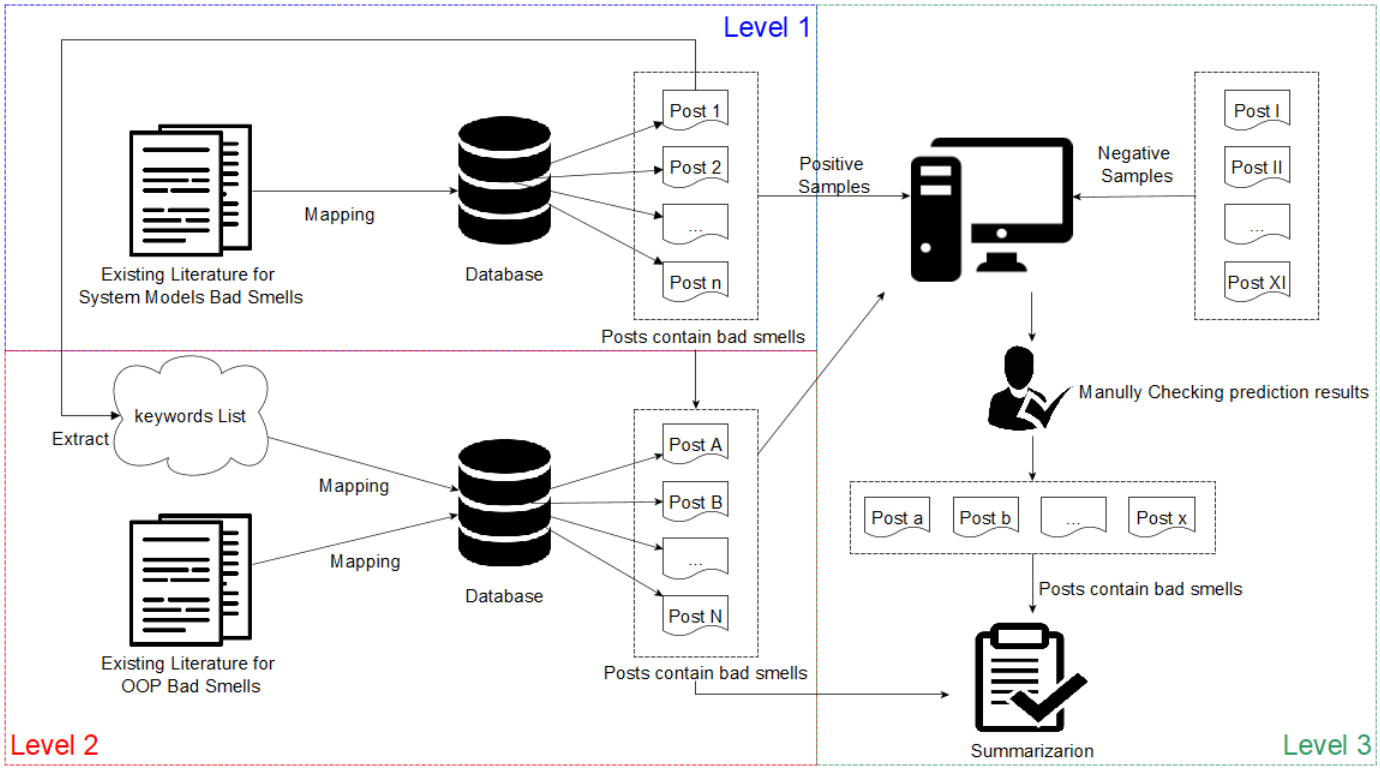


Figure 4.1: Overview of BESMER Workflow

and Level 2 as positive samples and randomly chooses posts that do not contain any bad smells as negative samples for machine learning training. The goal is to let the machine identify whether a new post contains bad smells or not. Figure 4.1 shows an overview of the workflow in BESMER.

4.3.1 Level 1 Summarization

The first level of bad smell summarization is to answer Research Question 4.1 (RQ 4.1): Are the bad smells in existing literature also mentioned by end-users from the LabVIEW online discussion forum? To this end, BESMER constructed a mapping between bad smells in existing literature and the online forum posts in the database. As mentioned earlier, the research on bad smells in systems models is very limited. As far as is known, the work analyzing bad smells in LabVIEW models is only found previously in [34] [35] [36]. In Chambers' work [34], the authors listed 13 bad smells related to LabVIEW model performance. Table 4.1 lists these bad smells. In order to enlarge the search scope, BE-

Table 4.1: LabVIEW Bad Smells Summarized by Chambers and Scaffidi [34]

Bad Smell	Results
Too many variables	race condition
Build array inside a loop	slow performance and memory waste
Multiple array copies	memory waste
No wait in a loop	synchronization issues and unresponsive front panel elements
Unconnected front panel elements	unexpected behavior in VIs or confusing VIs
Redundant operations	waste time and memory
No Constraint in queues	performance problems and loop synchronization issues
Infinite while loop	Execution time issues
Non reentrant VIs	poor execution speed
String Concatenation in loop	slow performance and memory issues
Sequence instead of state machine	reduce readability
Multiple nested loops	reduce readability
Only loop once	reduce readability

SMER also incorporated bad smell summarization literature related to Simulink models. Although LabVIEW models and Simulink models have many differences, both LabVIEW and Simulink are graphical modeling tools and systems models built in these tools share some commonalities. In this dissertation, Gerlitz's work [64] is adopted in BESMER to extend the scope of this search.

The work conducted by Gerlitz summarized 21 bad smells, which is shown in Table 4.2. In total, 34 bad smells were considered to confirm whether the bad smells in existing literature are mentioned by end-users from online discussion forum posts. The input to Level 1 summarization of BESMER is represented by these 34 bad smells. The output of Level 1 are the posts that contain these bad smells (The upper-left rectangle shown in Figure 1 illustrates the workflow of this level).

4.3.2 Level 2 Summarization

This level of summarization answers the second research question (RQ 4.2): Are there any new bad smells mentioned by end-users from the online discussion forum that are not mentioned in existing literature? To answer this question, BESMER first extracted

Table 4.2: Simulink Bad Smells Summarized by Gerlitz [64]

Category	Bad Smells
Name	Vague Name; Inconsistent port name
Partition	Subsystem with multiple functions; Superfluous subsystem; Deeply nested subsystem hierarchy; duplicate model part
Interface	Subsystem interface incongruence; Inconsistent interface definition; Long port list; Non-optimal port order
Signal flow	Redundant signal paths; Cyclic signal path; Mismatch between visual and effective flow; Independent local signal paths; Unused signal; Pass-through signal; Hidden signal flow
Signal Structure	Superfluous bus signal; Non-optimal signal grouping; Unnamed signal entering signal bus; Duplicate signal in bus

hot words (i.e., words that are mentioned more frequently than others) from the posts collected from Level 1. To this end, the Bag-of-Words (BoW [151]) model was applied to analyze these posts. A BoW model simply counts the frequency of occurrences of each word. Although a BoW model cannot provide much semantic information, it can observe which words are more active when describing bad smells using a BoW model.

In order to further enlarge the search scope, BESMER introduces the concept of bad smells in OOP at this level. This introduction is helpful in understanding whether bad smells in OOP also exist in systems models. When the concept of OOP bad smells is adopted, BESMER uses hot words extracted from user posts to replace OOP terms. For example, in OOP, there is a bad smell called **Large Class** (a class that contains too many functions). In the LabVIEW context, instead of using the word "class," end-users use the word "VI" (LabVIEW programs-subroutines are termed virtual instruments, or VIs), which is unique to LabVIEW systems models. The input of this level summarization is adapted bad smells and the output is the posts containing these adapted bad smells (the lower-left rectangle shown in Figure 4.1 illustrates the workflow of this level).

4.3.3 Level 3 Summarization

This level of summarization continues answering the second research question (RQ 4.2). The reason why BESMER needs this level is based on the fact that when a user de-

scribes a bad smell in a systems model, they may use different words from others who may describe a similar smell. For example, when describing a large VI, one engineer may use the word "large" while someone else may use the word "big" or "oversized." However, even if several users have different word choices, their posts reflect the same meaning: that is, a VI contains too much functionality. Moreover, by training the machine learning engine to understand the pattern behind the description of forum posts related to bad smells, BESMER could find more new bad smells that are not mentioned in existing literature. For example, **Too Many Variables** is a bad smell in LabVIEW systems models. It refers to a program that uses excessive variables during programming. Similarly, there may be LabVIEW user forum posts containing bad smells related to the excessive usage of other properties/entities in LabVIEW (such as too many wires). This level of summarization is based on the understanding of the whole sentence/paragraph, rather than individual words, to find more bad smells in LabVIEW models. This helps to spot any bad smells mentioned by end-users due to:

- Different semantics when describing same smell;
- Same semantics when describing different smells.

BESMER trained the machine learning engine to identify what is a post related to a bad smell (positive samples) and what is not (negative samples). All of the posts collected from Level 1 and Level 2 were labelled as positive samples and randomly choose posts in the database that are not related to bad smells (manually checking). Because the performance of machine learning is closely related to the number of positive samples and negative samples, as well as the chosen machine learning algorithm, different ratios of positive and negative samples were tested on different algorithms to see which combination yielded the best performance.

After the prediction phase, all of the predicted posts that are related to bad smells were manually checked. The manual check helped to find new bad smells and confirm bad

smells found in Level 1 summarization and Level 2 summarization. The input of this step is the posts collected from Level 1 and Level 2 and the output is the bad smells in the posts discovered by the machine. (the right rectangle shown in Figure 4.1 illustrates the workflow of this level).

4.4 Experimental Results

As discussed in Section 4.2, BESMER adopts a three-level summarization mechanism to find bad smells from online discussion forums. This section presents the summarization results obtained from each level in BESMER.

4.4.1 Bad Smells Summarized from Level 1

In Level 1, BESMER focuses on RQ 4.1. To this end, 34 bad smells mentioned in existing literature were searched in the database of collected forum posts. In the end, BESMER found nine bad smells that were discussed among end-users in the online forum. Six bad smells are from Chambers' work and three bad smells are discussed in Gerlitz's work. Overall, 16 posts containing nine bad smells were discovered (all the posts listed in this dissertation are from the LabVIEW discussion forum). The following subsections discuss each of these nine bad smells with their potential causes and negative effects on the systems models. This section also presents some of the forum posts containing bad smells from end-users in the following discussions.

a. Too Many Variables

Too Many Variables refers to a VI that contains too many variables. In LabVIEW, there are two kinds of variables: a global variable which is used to access and transfer data among different VIs and a local variable which is used to access and pass data within a VI. Users complained in the online forum that too many variables makes a VI hard to read and may lead to race conditions. For example,

> *"... I currently use shared and local variables to accomplish this. I've been told however that local variables are bad due to their memory..."*

"... too many variables declared in the function, which use up the allocated memory or CPU time ..."

By the examination of user posts related to this bad smell, this study found that the reason mentioned by end-users for this bad smell is the overall bad design of their VI. The results for too many variables in a VI include wasted memory space, race conditions, low readability and comprehension. Based on the interview with LabVIEW experts, two to five variables per VI are recommended [35]. There is not a fixed answer for "how many variables" is sufficient. The answer may vary based on specific program design and purpose.

b. Stacked Sequence Structure

In LabVIEW, a sequence structure is a data processing structure that forces two portions of LabVIEW code to run one after the other. There are two types of sequence structure: flat sequence structure and stacked sequence structure. Through manual examination, a stacked sequence structure can often be confusing and greatly reduces readability for many end-users. For example,

"... I am refactoring code of a state machine and one of the states has a lot of loops in it. A lot of True/False states with stacked sequence structures nested in each other. It looked really messy ..."

" ... Remove once and for all the stacked sequence structure from labVIEW. It's an absolute horror..."

By the examination of posts related to this bad smell, systems engineers expressed their concern about program maintainability when applying stacked sequence structures.

c. Duplicated Model Part

Duplicated Model Part is a smell mentioned by Simulink bad smell investigations. As the name suggests, this smell refers to the duplication of the model (or parts of the model) into multiple instances that have the same functionality. For example,

" . . . Is there a way to find duplicate/un-used code in a project? Are there any utilities/toolkits or anything that might facilitate in this respect? Have a pleasant day! Thanks for your time . . . "

". . . First, there is a lot of shared code between the applications, but the shared code isn't actually shared code on disk - Project 1 uses class A, and Project 2 also uses class A, however they both have their own, totally separate copy of class A. So, if I find a bug or need to add a feature to class A, I need to make that change in both the Project 1 and Project 2 copies . . . "

The results of duplicated model part may lead to high coupling, which decreases the maintainability during model evolution.

d. No Wait in Loop

This bad smell refers to when a wait primitive (such as Wait Until Next ms Multiple node, which is used to synchronize separate loops in a VI) is missing in a loop structure. The potential issues caused by this bad smell include program synchronization issues and response issues (such as program freezing). For example,

". . . It is always a good idea to put a Wait(0) to allow in loops that don't provide some other ability for LabVIEW to perform a task switch. . . "

e. Deeply Nested Subsystem Hierarchy

This smell originates from bad smell investigation for Simulink systems models. A deeply nested subsystem hierarchy in a LabVIEW systems model suggests that there are too many layers of a SubVI (a SubVI is a VI that can be called by another VI. A SubVI is similar to a user-defined function in OOP). This bad smell means that a VI contains a SubVI and this SubVI contains another SubVI. If there are many deeply nested subsystems, it will make the program hard to maintain. For example,

". . . There is a scantling of TestStand on top of some deeply nested VIs. If the nesting were shallow it would be easy to pass new variables as parameters but

as it is I would need to modify many layers of VIs to get the new variable down to where needed... "

f. Unused Model Part

Unused Model Part originated from an existing bad smell in the literature called "Unused Signal," which was investigated by Gerlitz within the context of Simulink systems models. Because "signal" is a term used in Simulink but not LabVIEW, this research modified the smell to Unused Model Part. This bad smell refers to when there exists functions/code in a VI that are useless and can never be invoked within the control flow of the model. For example,

" ... Here is an instruction I received from my senior: Avoid making multiple copies and leave unused code on disk ... "

Although LabVIEW offers a tool called VI Analyzer[5] (a commercial toolkit developed by National Instruments) that could help to detect unused model part, LabVIEW users complain about its performance. For example,

" ... So when using the VI Analyzer I'm being told that I have a segment of unused code and am wondering if this is not the way to perform the operation I'm wanting. I am pretty sure that the segment of code is indeed used, but perhaps the issue lies within the way that I'm attempting to accomplish this ... "

Unused model part could increase the size of a VI, thus decreasing its readability while offering no practical use. Any unused model part should be deleted within a LabVIEW model.

g. Build Array in Loop

Build array in loop, as its name suggests, means an array is built inside of a loop structure. When a build array node is inside a loop, every time the loop starts a new it-

[5] `https://www.ni.com/pdf/manuals/375339g.html`

eration, a new copy of the array is constructed, thus slowing down the execution speed greatly due to increased memory consumption. For example,

> *" . . . I've read many times (like in here, for instance) about how using Build Array in loops is a Very Bad Thing and that you should always start off by initializing an array once and replacing the values therein to improve performance. My understanding is that using Array Subset in a loop is also bad, because it once again causes LabVIEW to reallocate memory for the array in each iteration . . . "*

h. Non-reentrant VI

Reentrant VIs are used when several instances of the same VI are running simultaneously. When a VI is not reentrant, there is one data space for the VI. This means this VI has a data space shared between multiple calls. A **Non-Reentrant VI** in a parallelized structure that could introduce synchronization issues, thus greatly decreasing the execution speed by waiting and blocking. For example,

> *" . . . Multiple instances of reentrant VIs cannot run in parallel if that reentrant VI calls a non-reentrant subVI. One reentrant instance will have to wait on the execution of that subVI if it is currently running in the other instance . . . "*

i. Redundant Operation

Redundant Operation refers to a VI that has unnecessary operations. This bad smell could waste time and memory resources. For example,

> *" . . . Avoid putting a calculation in a loop if the calculation produces the same value for every iteration. Instead, move the calculation out of the loop and pass the result into the loop. The result of the division is the same every time through the loop; therefore you can increase performance by moving the division out of the loop . . . "*

4.4.2 Bad Smells Summarized from Level 2

This level of summarization focuses on the bad smells related to OOP concepts in LabVIEW systems models. To map bad smells in systems models to bad smells in OOP, BESMER first collected keywords that are always mentioned by end-users in the posts previously collected. A BoW model was implemented in Java to calculate the frequency of occurrence for each word. BESMER also deleted common stop words (such as "we" "this" "the." The stop words used in this study are based on Natural Language Toolkit's list of English stop words[6]). The experimental result shows that the top-three frequently mentioned keywords by systems engineers in the posts are *VI*, *loop* and *data*.

Next, BESMER merged these keywords into the concept of bad smells discussed in OOP. In this study, Fowler's categorization of bad smells [57] was chosen as a reference. These bad smells include: Duplicated Code; Long Method; Large Class; Long Parameter List; Divergent Change; Shotgun Summary; Feature Envy; Data Clumps; Primitive Obsession; Switch Statement; Parallel Inheritance Hierarchies; Lazy Class; Speculative Generality; Temporary Field; Message Chain; Middle Man; Inappropriate Intimacy; Alternative Classes with Different Interfaces; Incomplete Class Library; Data Class; Refused Bequest; and Comments. In the end, 18 posts containing three new bad smells were discovered in this level of bad smell summarization. These new bad smells are **Large VI**, **Data Clumps** and **Lazy SubVI**.

a. Large VI

This bad smell is largely discussed by the end-users from the observation. Large VI refers to a VI that contains too many functions. It corresponds to Large Class in OOP. A large VI greatly decreases the maintainability of a program and makes the program hard to debug when something is wrong. For example,

> *" . . . I have a VERY large VI that finally crashed due to the large size. What is the best way to create 'Sub VIS'? . . . "*

[6] `https://www.nltk.org/`

" . . . I've now created a monster. I'm sure I've not done things in the most efficient way but I'd really like to break it up into smaller VIs, so troubleshooting or modification isn't such a pain . . . "

" . . . The LabView I am wading through involves multiple VIs, many of which are large files that are difficult to decipher. I would like the programmer to consider reducing the VI file sizes by breaking the large VIs up into sets of smaller Vis . . . "

From the users' responses to the large VI bad smell, it was observed that the best way to get rid of this smell is to break a large VI into several smaller SubVIs. This break down also confirms the benefits of cohesion and modularization - separating the functionality of a program into independent, interchangeable modules, thus increasing the readability and maintainability.

b. Data Clumps

Data Clumps refers to many data copies in various places across a program. This may lead to memory waste for a LabVIEW program. For example,

" . . . A common practice to reduce the memory footprint of a program, is to pass data value references, instead of full blown copies (Storing Data and Reducing Data Copies with Data Value References) . . . "

" . . . I would appreciate tips on how to stop labview from making extra data copies, as I think that might be a problem . . . "

c. Lazy SubVI

Lazy SubVI corresponds to bad smell Lazy Class mentioned in OOP bad smells. It refers to a SubVI that does not exhibit much useful functionality. Usually, this bad smell occurs when trying to reduce the diagram size of systems models. However, too many unnecessary SubVIs can slow down program execution. For example,

" . . . If your idea of 'simplification' is just to select random areas of the diagram followed by 'create subVI', you are only curing the symptoms of a messy, overloaded diagram by creating many 'single-use', messy subVIs. This is like sweeping all the dirt under a rug. The mess is still there, but buried under an additional layer. SubVIs need to be functional and reusable and with a clear and useful purpose . . . "

However, a good VI should exhibit a hierarchical structure where VIs are constructed on top of other lower-level VIs. A good VI often executes one function.

4.4.3 Bad Smells Summarized from Level 3

After Level 1 and Level 2 summarization, BESMER obtained posts related to bad smells that are already described in existing literature. In this level of summarization, BESMER further extended bad smell summarization by applying machine learning techniques. The goal of this level is to recognize potential bad smells hidden in posts from the examples that are provided to the machine learning engine.

Machine learning techniques provide an opportunity to use algorithms to automatically analyze a large number of forum posts. This is also called text data mining. There are two general categories of techniques: supervised mining and unsupervised mining. Supervised techniques first learn a classification model based on a small set of manually labeled posts and then apply the model to predict the class categories for the vast majority of unlabeled posts. In this experiment, BESMER used three different supervised machine learning algorithms for post classification: Multinomial Naïve Bayes [88], Sequential Minimal Optimization (SMO, a fast implementation of Support Vector Machine) [132] and Random Forest [186]. Similar to the work conducted in Chapter 3, BESMER also used **Precision**, **Recall** and **F-Measure** metrics to evaluate classification performance. In this section, Precision is defined as:

$$Precision = \frac{Number\ of\ correctly\ predicted\ posts\ into\ a\ class}{Total\ number\ of\ posts\ predicted\ into\ a\ class} \tag{4.1}$$

In this section, Recall refers to the completeness of the posts predicted into a class. It is defined as:

$$Recall = \frac{Number\ of\ correctly\ predicted\ posts\ into\ a\ class}{Total\ number\ of\ true\ posts\ in\ a\ class} \tag{4.2}$$

F-measure has the same formula as the definition in Equation 3.7 in Chapter 3. Moreover, a 10-fold cross validation discussed in Chapter 3 was adopted as the model performance evaluation approach.

The last question is how many posts related to bad smells (positive training samples) and how many posts not related to bad smells (negative training sample) to train the model. There is not a standard answer for this question because there are many factors affecting model performance (e.g., the quality of training samples, the algorithm chosen for the model). In order to obtain the best performance, BESMER tried different combinations of these three algorithms. The result shows that choosing the ratio of positive training samples and negative samples at 1:4 (34 positive samples and 136 negative samples) with Multinomial Naïve Bayes could produce the best prediction result. The Precision, Recall and F-Measure are 0.946, 0.947, and 0.946, respectively.

Applying this model to predict all of the posts in the database, there were 5,434 posts predicted as posts related to bad smells out of 117,495 posts in the database. These 5,434 posts were manually checked and three new bad smells were discovered.

a. Excessive Property Nodes

In LabVIEW, a property node is a function used to read and write properties of a reference. It is used to get or set properties and methods on local or remote application instances, VIs, and objects. Users can also use the Property Node to access the private data of a LabVIEW class. However, excessive usage of property nodes may decrease the performance of a program. For example,

> *" . . . Avoid the excessive use of property nodes everywhere. This will slow your application down. Each property node access will result in a context switch to*

the UI thread. Use shift registers and wires (wire are similar to a variable in a text based language) to pass your data through your program. Avoiding all of the property nodes may help to get more consist timing . . . "

b. Constant Array Resizing

Constant array resizing means adjusting the array size in a static manner but not allocating new space for the array dynamically. Because the increase of the length of an array always involves allocating memory space, resizing it in a static manner may waste memory. For example,

" . . . Your problem is not with the initialization, but with the constant array resizing. There are no memory leaks, put potential for memory fragmentation . . . "

c. Unorganized Wires

LabVIEW transfers data among block diagrams through wires, which are one of the most important parts in many visual programming languages because they connect different blocks together. However, too many unorganized wires may make the model and diagram extremely hard to read. For example,

". . . If I understand it right, wires are always always always the most efficient way to transfer the data, but stretching a bunch of wires across my program, well.....it's gonna look really ugly . . . "

" . . . Sometimes there are too many wires. Much of my state machines have a dozen or more wires just going from input to output, doing nothing, just because one or two states in the machine need that variable in some state. Yeah, I could spend a lot of time bundling and unbundling and rebundling those values, but I don't think that would improve things much . . . "

4.5 Discussion and Insights

This section discusses the results from previous sections to give a detailed analysis of the bad smells summarized from end-users by mining forum posts. This section will also present some insights from the examination of user posts.

BESMER summarized 15 bad smells in total. There are nine bad smells that are from existing literature and six are newly discovered. Among nine bad smells from existing literature, six of them originated from LabVIEW bad smell research and three of them originated from Simulink bad smell investigation. This suggests that the two popular systems modeling tools, LabVIEW and Simulink, share similarities – some program issues exist in both platforms. Yet, they also have unique traits.

From the perspective of systems modeling and OOP, some of the bad smells are both applicable to these two areas, such as Duplicated Code/Model Part, Unused Code/-Model Part, Large Class/VI. This intersection of application also shows that common knowledge is adopted in model-based systems engineering and software engineering.

To find possible inferences that can be made through the examination of end-user posts (RQ 4.3), all of the user posts obtained from three levels of summarization were carefully examined. Most of the posts did not contain any bad smells, but still provided many useful and practical insights. Next section will discuss three insights from the examination of user posts.

4.5.1 Inconsistent Term Usage

Inconsistent term usage means there is a gap between new developers and experts when describing the same problem. For example, refactoring is closely related to bad smells. It refers to the process of restructuring existing program model/code without changing its external behavior. Refactoring seeks to improve functional attributes of the software. However, many end-users use the word "refactoring" to imply other meanings. For example,

" . . . I have two input arrays and two output arrays. Need to refactor the input arrays to listbox or table . . . "

where the user's intention is to convert an array to a listbox or a table. Another reason for inconsistencies is because of the knowledge discipline difference. For example, in LabVIEW, a graphical block is equivalent to textual code in OOP.

4.5.2 OOP in LabVIEW

Through the examination of user posts related to bad smells, it was observed that OOP in LabVIEW is difficult for most users. For example,

" . . . I am finding it very time consuming and error prone to refactor LVOOP code . . . "

" . . . I'm bit stuck within class inheritances in my current project . . . I have to admit, I am quite confused.... especially because within another class this exact methodology works..."

Part of the reason for this is because many LabVIEW developers are from a systems engineering background. LabVIEW is their first tool to start their project. They do not have much experience in using an OOP language (such as Java, C++).

4.5.3 Lack of Documents/Examples

Another insight observed in this study is the lack of documentation for LabVIEW end-users. For example,

" . . . We have already scoped out the effort but would desire guidance on VI layout, descriptions and ways to give a 'LabVIEW' experience and not a disjointed VI that looks unprofessional..."

" . . . Unfortunately I was unable to find any documentation on how to create external data value references. The best thing I could up was creating a data

value reference constant, which can be set to 'External' in its properties. However that alone is pretty much useless. . . "

" . . . Since I'm new to OOP, it would help me to understand this by having an example. I can't find one and the link commits the famous error of assuming that we are all experts and so the pattern is too simple to merit an included example. The article talks about them being all over the place, so obviously I don't understand the pattern. Would someone point me to a specific example of this pattern? . . . "

In order to solve this problem, LabVIEW developers (especially new developers) are encouraged to spend more time on LabVIEW tutorials. There are many questions asked by end-users that are answered in the tutorials. There are also other resources that introduce more advanced tutorials, such as LabVIEW Wiki[7] and Community Nuggets[8].

4.6 Threats to Validity

This section discusses the main threats to validity, as well as the means that have been undertaken to mitigate these threats. This section is organized by introducing three types of threats to validity: construct threats, internal threats, and external threats to validity.

4.6.1 Threats to Construct Validity

In this work, in order to solve the term usage difference between bad smells in OOP and bad smells in systems modeling, in Level 2 bad smell detection, BESMER only replaced hot keywords mentioned in posts collected in Level 1. However, for bad smells that could derive from OOP bad smells but do not contain hot keywords found in Level 1, the proposed approach could not include these smells.

[7] `http://labviewwiki.org`
[8] `https://forums.ni.com/t5/tag/community_nugget/tg-p`

4.6.2 Threats to Internal Validity

In this work, BESMER downloaded all of the user posts ranging from 1999 to 2019 from the LabVIEW online discussion forum. Although the dataset consists of more than 117,000 posts, more user-based posts from other online sources are needed to discover additional bad smells.

Moreover, BESMER applied three supervised machine learning techniques to discover new posts related to bad smells. Although these three techniques have been applied to the area of text classification widely, more advanced approaches, such as text classification methods based on deep learning, are needed to improve the prediction results.

4.6.3 Threats to External Validity

One key threat to external validity in this study is to what extent LabVIEW experts agree with the bad smells mined from end-users. Because the majority of users who post questions in the forum are inexperienced developers, they may have different views and understanding about bad smells compared with experts.

4.7 Conclusion

This chapter presented an approach for bad smell summarization for LabVIEW systems models from an end-user's perspective. The proposed work, BESMER, consists of three levels of discovery: existing literature confirmation, OOP extension and machine learning investigation. This investigation confirmed six bad smells from existing literature that were discussed by LabVIEW end-users, and three bad smells from the Simulink bad smell literature. BESMER discovered six new bad smells from end-user's online forum posts – three of them are derived from OOP bad smells and three of them are recognized by a machine learning technique. BESMER helped to answer the first and second research question: The bad smells in existing literature are also mentioned by end-users and there exist unique bad smells from an end-user's perspective that are not mentioned in existing literature.

Moreover, during the examination, this dissertation sought the insights of mining user posts. The investigation proposed several implications based on the examination of user posts. These implications not only could help end-users design better LabVIEW systems models, but also give insights into future research directions. This work is the first investigation to analyze and summarize bad smells from an end-user's perspective based on deep mining of user forum posts. BESMER is also applicable to other domains, thus paving a way to more studies that focus on end-user forum discussion.

CHAPTER 5

AN EMPIRICAL EVALUATION OF BAD SMELLS IN SYSTEMS MODELS

This chapter discusses an empirical evaluation of bad smells in LabVIEW systems models. Chapter 4 discovered several bad smells systems that engineers encounter during their development practice. Following this discovery, this chapter focuses on the discussion of an online survey sent to systems engineers to gain deeper insights into their perception on the bad smells that previous investigations have summarized. This chapter first gives a detailed explanation on bad smells that are scoped in this study. Then, the survey design, participant recruitment, and data collection process are described. This empirical study shows that systems engineers give different perceptions on different smells and the perception also varies due to different levels of experience and working domains. This chapter also offers several recommendations to systems engineers and summarizes the lessons learned from this empirical study.

5.1 Introduction

Previous examinations have revealed that model smells (i.e., bad smells in systems models) share some similarities with code smells (i.e., bad smells in textual programming languages); however, some bad smells apply to systems models only. For example, Duplicated Code [57] is a code smell that represents a portion of code that is duplicated or cloned, in many different locations across source files. Code that is duplicated may increase the time and effort during maintenance due to implicit coupling of the source representation. A counterpart for duplicated code is the **model smell** (bad smell in systems models) called Duplicated Model Part [34], which indicates a repetition of a model element across multiple locations in a model. There are some smells that only exist in the

context of models. In textual languages, the execution follows a top-to-bottom order. In graphical models, however, execution flow often needs to be specified explicitly, using directional wires and arrows. Unorganized Wires suggests that the wire used to connect different model components is untidy and unordered [195], thus increasing the difficulty to understand a model. This smell is an example of a model smell that does not have a corresponding code smell.

Compared with a large number of studies on textual code smells, less research has been presented on model smells. To fill this gap and better understand bad smells in systems models, I conducted an empirical study to evaluate model smells through a user survey [196]. This study consists of an online survey that focused on respondents' perception of model smells. The survey was sent to online LabVIEW discussion forums, developers at National Instruments, academic users from different institutions, and social media (e.g., Twitter and Reddit). The goal of this study is to understand model smells based on user community feedback, and how users' experience and area of expertise affect their perception of smells in LabVIEW systems models. To this end, the research questions in this study are defined as follows:

1) Perception of Specific Smells

The perception of specific smells may vary from engineer to engineer. For example, **Dead Code** is reported as a bad smell that decreases the program quality in both textual (such as C++ [57], Python [143]) and graphical programming languages (such as Scratch [174], LabVIEW [34]). Dead Code wastes computation time and memory; furthermore, it may also lead to program failure. Due to these reasons, Dead Code has been heavily studied by researchers over the past several decades [145] [146]. However, other smells have not received as much attention as Dead Code, such as **Multiple Nested Loops** and **Excessive Property Nodes** that have been discovered from previous examination.

Moreover, engineers agree that one specific model smell may negatively affect the model quality, but they may have different opinions on the extent. Some may strongly

agree that a smell decreases the model quality while some may slightly disagree. To better understand the perceived difference of bad smells summarized in the literature for LabVIEW models, the first research question is:

> **Research Question 5.1 (RQ 5.1)**: To what extent do systems engineers agree with the bad smells summarized in existing literature for LabVIEW systems models?

2) Development Experience and Domain of Expertise

Studies have shown that personal experience affects the perception and understanding of domain knowledge [140] [148]. This study sought to understand how the level of experience in using LabVIEW influenced the degree of perception ascribed to various model smells. This idea led to the second research question:

> **Research Question 5.2 (RQ 5.2)**: To what extent does LabVIEW usage experience influence a systems engineer's perception of specific smells?

Similar to experience, the particular domain of expertise (or employment sector) may also affect the perception for specific model smells. This study focused this question broadly on two general sectors: academia and industry. Each of these two sectors could be defined at a finer granularity (e.g., avionics or automotive for industry; faculty, graduate students, or undergraduates for academia). This study sought to understand the perceived differences between academic researchers and industrial developers on bad smell understanding in systems models, which led to the third research question:

> **Research Question 5.3 (RQ 5.3)**: Do academic and industrial users have the same opinion on bad smells in LabVIEW systems models?

3) Additional Smells

Smells summarized in this empirical study are from two existing previous works that studied LabVIEW systems models [34] [195]. There may exist other model smells

that survey respondents have identified in their own work, but not listed in current literature. Therefore, the fourth research question is:

> **Research Question 5.4 (RQ 5.4)**: What other smells have survey respondents observed in LabVIEW systems models?

4) Recommendations and Lessons

Finally, providing specific recommendations to end-users to help them avoid prominent model smells during development practice is of great interest to my research. This investigation could also offer potential research directions to both the research community and practitioners. To this end, this empirical study is also interested in the following question:

> What recommendations can this dissertation provide for end-users to help them avoid model smells and what lessons can researchers learn to lead potential research from the investigation of this empirical study?

The primary contributions of this study are:

- Understanding bad smells in systems models based on the end-user's perception;
- Investigation of the perceived differences of model smells between participants with diverse experience and areas of expertise;
- Identification of additional model smells, as a contribution to research literature related to systems modeling practice;
- Practitioner recommendations to avoid prominent model smells from a summarization of lessons that are learned from this empirical study.

The rest of this chapter is organized as follows. Section 5.2 introduces the bad smells in LabVIEW models analyzed in this chapter. Section 5.3 describes the research methodology. Section 5.4 provides the results and discussions. In Section 5.5, several specific recommendations are provided to address prominent model smells and summarize the lessons

from this empirical investigation. Section 5.6 examines the threats to validity and Section 5.7 presents related work in existing literature related to this study. Finally, Section 5.8 concludes this chapter.

5.2 Related Work

In this section, existing literature is examined. The investigation of human factors in code smells analysis is first presented. Following this, existing literature on bad smells in systems models is discussed.

5.2.1 Empirical Analyses of Code Smells

A survey-based empirical analysis for bad smell evaluation is rarely seen in literature in the context of Model-Driven Engineering. Yamashita and Moonen conducted an empirical study on code smells within the authors' organization among professional developers [187]. Similarly, Mäntylä and Lassenius [107] conducted research in the domain of software evolvability using code smells by sending a survey to the author's company. Compared with my dissertation, these two efforts only scoped professional developers in industry, which lack the evaluation from other fields with different knowledge depth.

Some empirical studies focus on code smell summarization and identification instead of code smell evaluation. Most of these studies are based on the author's own experience using a specific tool, for example, the summarization of LabVIEW smells [34], Simulink smells [64] and block languages like Kodu [75]. As to the identification of code smells, De Mello et al. [47] analyzed what human factors affect reviewer's recognition of code smells. Hozano et al. [79] focused on how similar developers detect code smells through an empirical study. Instead of code smell evaluation, these studies paid attention to smell summarization and identification.

Other empirical studies related to code smells seek to understand the relationship between smells and software maintainability. For example, to understand the relationship between code smells and software development and evolution, Tufano et al. [178] con-

ducted a large empirical study using over 200 open source software ecosystems to investigate when code smells are introduced by developers and why they are introduced. The result shows code smells are introduced from the creation of the project and the smells become more severe as the project evolves over time. Hecht et al. [71] focused on the individual and combined performance impacts of code smells in Android systems available from the Google Play store[1]. This empirical examination shows that correcting Android code smells effectively improved the user interface and memory performance by 12.4 % and 3.6%, respectively.

5.2.2 Bad Smells in Systems Models

Chambers and Scaffidi first identified bad smells in LabVIEW models [34]. They conducted an interview with nine LabVIEW specialists who had an average of five years of LabVIEW development experience. They proposed an approach - Smell-Driven Performance Tuning (SDPT) to help engineers remove smells and improve system performance [35] [36]. Some other related work focused on bad smells of LabVIEW from a hardware perspective, such as energy consumption during the execution of LabVIEW models [31]. The investigation of LabVIEW systems models is also seen in the work conducted by Zhao and Gray [195], which mined an online discussion forum to find model smells with the help of machine learning techniques from an end-user's perspective. They found six new bad smells in LabVIEW systems models compared with Chambers' work.

Gerlitz et al. [64] first proposed a catalog for bad smells in Simulink systems models. The authors summarized 21 model smells based on their own experience working in the automotive domain. In their research, two tools were implemented to remove the smells: Artshop [63] and SLRefactor [177]. Another way to check the existence of bad smells in Simulink models is to follow the pre-defined model design guidelines [22] proposed by the MathWorks Automotive Advisory Board (MAAB, which was established to resolve request

[1] `https://play.google.com/store`

discrepancies in the automotive industry). The guidelines summarize the recommended stylistic and architectural constraints for Simulink models.

The research of UML model smells focuses on the different diagrams available in UML. There are several studies conducted to understand bad smells in UML models. Arendt and Taentzer [13] presented a catalog containing 17 model smells in UML class diagrams, use case diagrams and state machines. Similarly, Misbhauddin and Alshayeb [116] introduced a list that included 27 model smells in UML class diagrams, state diagrams, sequence diagrams, and component diagrams. Garcia et al. [61] proposed the concept of "architectural code smells" - a commonly adopted architectural decision that negatively impacts system quality based on their research on UML diagrams. The refactoring methods of UML models to eliminate these model smells are summarized in Mumtaz's work [123].

5.3 Selected Bad Smells in LabVIEW Systems Models

My research adopted 20 existing model smells for LabVIEW systems models based on two investigations [34] [195]. Chambers and Scaffidi first proposed model smells for LabVIEW systems models based on an interview with nine professional LabVIEW developers who had an average of five years of LabVIEW experience [34]. Although the authors summarized 13 model smells, there exist some limitations to their investigation. First, all of their described model smells are related to performance. Other smells, such as structural model smells, were not considered. Second, the study was conducted with input from only domain experts, thus limiting the feedback that informed their observations. Furthermore, the impact of model smells from the perspective of inexperienced users was not evident in their study. It was not clear from their study whether new developers have the same perception and understanding of model smells as experienced users.

To fill this gap, Zhao and Gray [195] described an effort in mining user posts from a LabVIEW discussion forum to understand what model smells are naturally described

Table 5.1: Model Smells Included in the Survey

Category	Model Smells
VI Structure	1.1. Too Many Controls
	1.2. Large VI
	1.3. Deficient SubVI
	1.4. Deeply Nested Hierarchy
	1.5. Dead Code
	1.6. Duplicated Code
	1.7. Unorganized Wires
	1.8. Unconnected Front Panel Element
Loops and Arrays	2.1. Build Array Inside a Loop
	2.2. No Wait In a While Loop
	2.3. Infinite Loop
	2.4. Loop Once
	2.5. Concatenate Strings In a Loop
	2.6. Multiple Nested Loops
	2.7. Fix-length Array Resizing
Functions	3.1. Stacked Sequence structure
	3.2. Non-reentrant VI
	3.3. Excessive Property Nodes
	3.4. Redundant Operations
	3.5. Excessive Data Copies

by end-users from a more general and practical perspective. This work obtained 15 model smells from mining over 100,000 user posts.

My research summarized the model smells identified in [34] [195] to categorize 20 distinct smells. These model smells are divided into three different groups: VI structure, Loops and Arrays, and Functions. VI Structure includes model smells related to the structure of a VI. Loops and Arrays focus on model smells in loops and arrays. Loops and arrays in the context of LabVIEW programming use graphical structures, but they share the same concepts as general-purpose programming languages, such as Java and C++. Functions concentrate on model smells related to pre-defined functions in LabVIEW. Loops, Arrays and Functions contain model smells that may impede system performance. The model smells analyzed in this study are listed in Table 5.1.

5.3.1 VI Structure

Model smells in this category affect the structural complexity of a systems model. Controls are the input for a LabVIEW model. They are similar to function parameters in OOP. The **Too Many Controls** smell is similar to the bad smell **Long Parameter List** in the context of OOP. Models containing too many controls are often hard to understand and difficult to maintain as the model evolves. **Large VI** and **Deficient VI** are a pair of opposite smells describing the size of a model in terms of functionality. Large VI refers to a VI that contains too many functions. It corresponds to code smell **Large Class** in OOP. A large VI may decrease the maintainability of a model and present challenges for debugging. A Deficient SubVI refers to a SubVI that does not do much work. Too many unnecessary SubVIs can slow down the model execution. The counterpart smell of Deficient SubVI is **Lazy Class** in the context of OOP. **Deeply Nested Hierarchy** in a LabVIEW systems model suggests that there are too many layers in the VI's dependency hierarchy. Nested hierarchy makes the model hard to maintain and increases its structural complexity.

The **Dead Code** smell refers to a section in the source code of a program that either can never be reached in the control flow, or is executed but the result is never used. In the context of systems modeling, Dead Code is equivalent to **Deal Model Parts**. **Duplicated Code** (or **Duplicated Model Parts**), as introduced earlier, refers to the duplication of model parts (or a model) into multiple instances that have the same functionality. The occurrence of this smell may lead to higher coupling, which decreases the maintainability during model evolution. **Unorganized Wires** and **Unconnected Front Panel Elements** are model smells specific to LabVIEW. In a VI, wires are used to transfer data among block diagram objects. Unorganized wires may reduce the readability of a model. However, if a block is not wired sufficiently and contains unconnected front panel elements (i.e., defined in front panels, but not used in block diagrams), a VI may suffer from unexpected behavior or confuse a systems engineer [34].

5.3.2 Loops and Arrays

This category examines smells related to Loops and Arrays in LabVIEW. There are two kinds of loops in LabVIEW: while loops and for loops. A while loop is a control flow statement to execute a block of subdiagrams repeatedly until a given Boolean condition is met. A for loop is a control flow statement to execute a block of subdiagrams for a fixed number of times. **Build Array Inside a Loop** refers to an array that is built inside of a loop structure. When a build array node is inside a loop, every time the loop starts a new iteration, a new copy of the array is constructed, thus greatly slowing down the execution speed due to increased memory consumption. In LabVIEW, when a loop finishes executing one iteration, it immediately begins running the next iteration. It is often beneficial to control how often a loop executes, or its frequency. Timing a loop also allows the processor to complete other tasks such as updating and responding to the user interface. A wait function provided by LabVIEW offers users the functionality to postpone the model's block execution for a specific amount of time. **No Wait in a While Loop**, as the name suggests, indicates a wait function is missing in a while loop. **Infinite Loop** and **Loop Once** are two smells that focus on the iteration frequency of a loop. Infinite Loop indicates that a loop never stops and may lead to a bug. Loop Once refers to a loop that only iterates one time. Most often, these two smells appear when the intention was not to loop infinitely or just once, suggesting an error that may affect the execution result. Such errors are most common among novice systems engineers. However, under certain circumstances, this is a desired behavior (e.g., round-the-clock systems may require an infinite loop).

Concatenate Strings in a Loop indicates the connection of two strings in a loop structure in a LabVIEW model. The basic idea of string concatenation is similar to its meaning in a general-purpose language, but LabVIEW adopts graphical blocks and structures instead of a textual programming language. Chambers and Scaffidi [34] suggest that this operation may lead to slow performance and cause memory issues. **Multiple Nested Loops** is similar to Deeply Nested Hierarchy discussed earlier in this section. It refers to

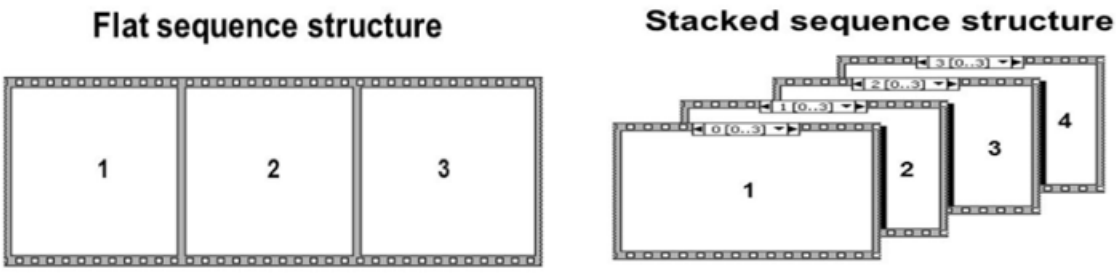


Figure 5.1: Flat and Stacked Sequence Structure in LabVIEW

a loop structure that contains another hierarchical loop structure. Multiple Nested Loops may reduce model readability and introduce maintenance challenges. The last smell examined in this category is **Fix-length Array Resizing**. This smell occurs when an array representing model state is declared to be a fixed size, rather than dynamically allocated by need. The potential result of this smell is wasted memory usage.

5.3.3 Functions

This category defines smells related to pre-defined LabVIEW functions and structures. In LabVIEW, a sequence structure contains one or more subdiagrams, or frames, that execute in sequential order. Sequence structures guarantee the order of execution and prohibit parallel operations. There are two types of sequence structures in LabVIEW: flat sequence structure and stacked sequence structure (shown in Figure 5.1). **Stacked Sequence Structure** is considered a model smell because only one frame is visible and local sequence state needs to be passed from frame to frame, thus increasing the model complexity while decreasing readability. Reentrant VIs are used when several instances of the same VI need to execute simultaneously. When the VI is non-reentrant, there is only one data space for the VI. Therefore, only one caller at a time is allowed to call this VI. A caller may have to "wait its turn" to use this VI. Thus, **Non-reentrant VI** may lead to poor execution speed.

In LabVIEW, a property node is a function used to get or set properties and methods on local or remote application instances, VIs, and objects. Excessive usage of property

nodes may decrease the performance of a model execution. An identified model smell is **Excessive Property Nodes**, which implies the over-usage of property nodes in a VI. **Excessive Data Copies** occurs when the definition of the same control/function/structure appears more than one time in a model. These two smells may lead to memory waste. The last smell in this category is **Redundant Operations**, which refers to a VI that contains unnecessary operations. For example, putting a calculation in a loop if the calculation produces the same value for every iteration.

5.4 Empirical Study: A Survey of LabVIEW Users

This section describes the methodology for the design of the survey, as well as how participants were recruited and how the participants' responses were collected.

5.4.1 Survey Design

The goal of this research is to empirically analyze whether model smells summarized in existing literature really "smell bad" from an evidence-based perspective in LabVIEW systems models. To reach this goal, this dissertation designed an online survey using Qualtrics[2]. There are five sections in this web-based survey. Section 1 is related to demographic questions. Section 2 through Section 4 ask participants to comment on 20 model smells listed in Table 5.1. Section 5 is an open-ended question that asks participants to identify any other smells that they have observed from their experience. All the survey responses were anonymous. The participation was voluntary and participants could skip any question that made them uncomfortable and they could stop the survey at any time. The survey used in this study was approved by the Institutional Review Board at the University of Alabama (Appendix A).

To avoid bias, only the description of each model smell was given without introducing any potential impacts this model smell may cause to the model. For each model smell evaluation, the survey asked respondents to indicate the degree to which each smell

[2] `https://www.qualtrics.com/`

#	Section	Question	Answer Choices
Q1	Demographics	How long have you been working in LabVIEW (Please respond in the format of years/months)?	Open-ended
Q3	Demographics	To what extent are you familiar with LabVIEW?	[Novice, Familiar, Competent, Expert]
Q4	VI Structure	Too many controls. Controls are the input for a LabVIEW program. They transfer data within a single VI and allow data to be passed between different functions/structures. This code smell refers to using too many controls in a program. To what extent do you agree "Too many controls" is a code smell for LabVIEW programs?	[Strongly disagree, Somewhat disagree, Neither agree nor disagree, Somewhat agree, Strongly agree, I am not confident in my knowledge to answer this question]
Q4_Extended	VI Structure	If you think "Too many controls" is a code smell for a LabVIEW program, how many controls in a VI do you think is too many?	Open-ended
Q23	Additional Code Smells	Do you think there are other code smells that negatively affect VI qualities? If yes, please list below.	Open-ended

Figure 5.2: Survey Questions (Partial)

negatively affects LabVIEW models using a 5-point Likert scale. Additionally, the option "I am not confident in my knowledge to answer this question" was added to participants in case they are not sure about a particular smell. For some questions, there are follow-up questions without specific answer choices. For example, if a respondent agrees that Too Many Controls is a smell for a LabVIEW model, the survey further ask the participant "How many controls do you think are too many?" to gain a deeper understanding of the smell. Figure 5.2 shows part of the survey. The full survey is available at `http://bit.ly/VISmells`.

5.4.2 Participant Recruitment

This empirical study recruited participants through the following contacts:

Online discussion forums. Online discussion forums are useful resources for sharing domain knowledge about a specific topic or tool, such as sharing challenges and ideas, promoting the development of community, and giving/receiving support from peers and experts. In this study, the research invitation was sent to three online discussion forums: official LabVIEW discussion forum supported by National Instruments, unofficial LabVIEW discussion forum LAVA, and the ACM SIGCSE forum for CS educators who may use LabVIEW in their classes.

National Instruments. There are several colleagues from National Instruments who assisted in broadcasting this survey invitation to professional LabVIEW developers. For feedback on the content in the survey, other experienced LabVIEW developers from National Instruments helped to review the survey and offer feedback on whether the survey questions were clear, complete and without ambiguity.

Academic institutions. To reach LabVIEW users from academia, this survey was sent to over 24 different institutions who may use LabVIEW for education or research purposes.

Social Media. To recruit additional responses, the research invitation was also posted to social media Twitter[3] and Reddit[4].

5.4.3 Data Collection

The research invitation was sent to the venues mentioned above and then followed up with a reminder six weeks later. Finally, this empirical study received 65 responses, from which 45 were completed and included in the analysis. Every response was manually examined to make sure the recorded responses were valid. Some of the respondents did not provide answers for the open response questions, but their answers to the other questions were included in the analysis.

5.5 Experimental Results

The discussion of the results is organized around the research questions posed in Section 5.1. To provide context for the discussion, this section first characterizes the demographics of the respondents.

5.5.1 Demographics

The participants come from 15 countries across North America, Europe, Oceania and Asia. Figure 5.3 shows the geographical distribution of the participants. Most respon-

[3] `https://twitter.com/?lang=en`
[4] `https://www.reddit.com/`

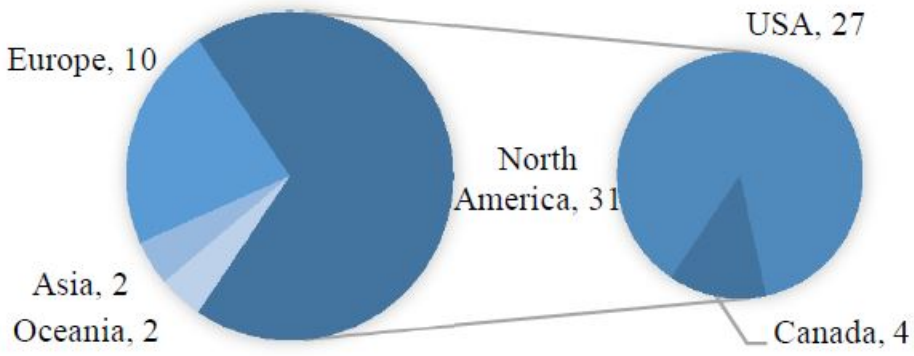


Figure 5.3: Participants' Geographical Distribution

dents (27 out of 45) are from the USA. Regarding working experience, the survey asked the respondents how long they have been working with LabVIEW. The range of experience spans from nine months to 26 years of LabVIEW usage. The average length of LabVIEW working experience is 12.7 years. The median is 13 years.

- 10 participants ($\approx 22\%$) have less than five years' experience;
- six participants ($\approx 13\%$) have five to 10 years' experience;
- nine participants ($\approx 20\%$) have 10 to 15 years' experience;
- 20 participants ($\approx 45\%$) have more than 15 years' experience.

Among the 45 respondents, their role and self-reported LabVIEW knowledge level were examined. In general, there are three categories: industrial developer, academic user (including faculty/staff, graduate students and undergraduate students), and others. The result shows that 60% of the respondents are industrial developers and 33% are from academia. The survey also asked respondents to evaluate their knowledge of LabVIEW with four levels: Expert, Competent, Familiar and Novice. The result is shown in Figure 5.4. Among 28 industrial developers, there are 23 respondents reported as experts, four competent and one familiar. Among seven academic researchers, two respondents reported themselves as experts, four competent and one familiar. Among seven graduate students, there was one

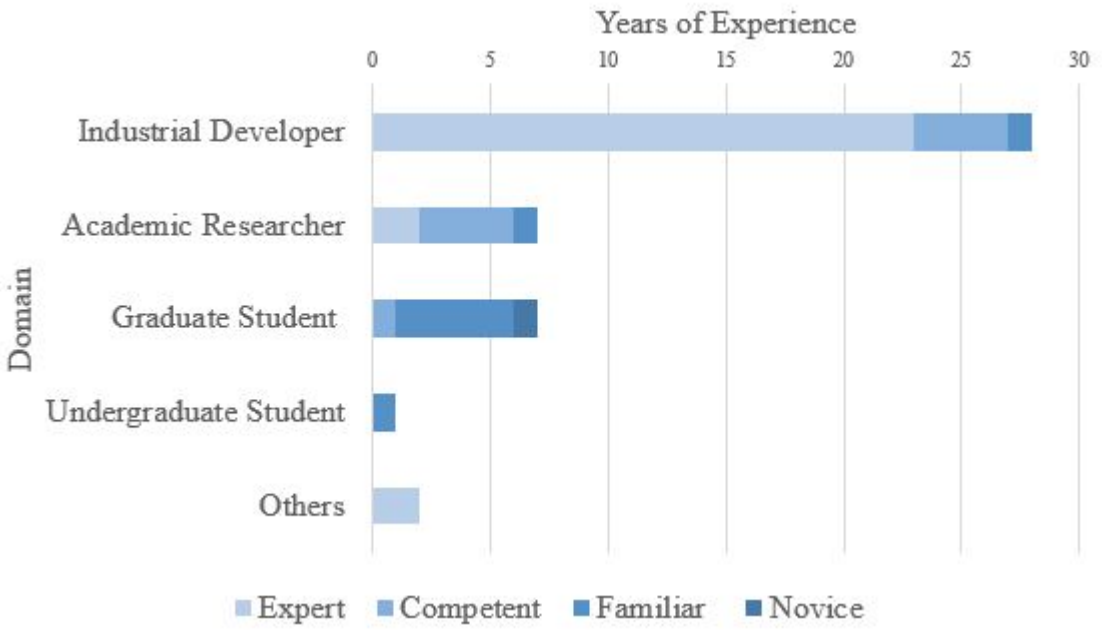


Figure 5.4: Participants' Role and Knowledge Level

respondent reported as competent, five familiar and one novice. There was one undergraduate student that chose familiar and two others reported as experts.

There are 20 model smell evaluations scoped in the survey. For 45 participants, 900 answers were expected (excluding the follow-up questions). This study received 850 answers in total (answer rate in general: 94.44%, industry participants answer rate: 98.21%, academic researchers answer rate: 95.71%, graduate students answer rate: 79.29%, undergraduate students answer rate: 75%, others: 100%). This answer rate indicates most participants understood the model smells introduced in the survey and offered a response.

Overall, these demographics suggest that the survey respondents were experienced in LabVIEW development and had a good understanding of smells in LabVIEW systems models.

5.5.2 Systems Engineer's Perception

To answer RQ 5.1, a 5-point Likert scale was adopted to convert participant's answers to numerical representations (This study defines this numerical representation as **Perception Value**) where **Strongly agree** is 2, **Somewhat agree** is 1, **Neither agree**

nor disagree is 0, **Somewhat disagree** is -1, and **Strongly disagree** is -2. If a respondent chose the option "I am not confident in my knowledge to answer this question" or did not answer the question, this question was removed from their overall response. In this study, **Mean Perception** was defined as an indicator to show the perception of each smell, which is formulated as:

$$Mean\ Perception = \frac{\sum Perception\ Value}{Total\ Number\ of\ Responses} \tag{5.1}$$

For example, for model smell Too Many Controls, there are 44 participants that answered the evaluation of this smell. Based on the mean perception value defined above, all the perception values from 44 responses were added. The sum is 23. Therefore, the mean perception of model smell Too Many Controls is $23/44 \approx 0.52$. Table 5.2 shows the results of significance analyses of all the model smells that are included in this study. The answer for RQ 5.1 is also reflected in Table 5.2.

5.5.3 The Impact of Experience on Smell Perception

To answer RQ 5.2, this study focused on two demographic attributes: user experience and domain of expertise to investigate how these attributes affect respondent's smell evaluations. First, the mean perception of model smells based on the three categories discussed in Table 5.1 was calculated. The focus of this calculation was to understand the perceived difference of contrasting groups, which is why the calculation of the mean perception of different categories instead of individual smells was performed. To this end, this study first divided the respondents into four groups based on their working experience in LabVIEW development: less than five years, 5 – 10 years, 10 – 15 years and more than 15 years working experience. Figure 5.5 shows the results of this analysis.

To test whether the difference is significant, the Mann–Whitney U test [105] was adopted. The results for Whitney U tests are shown in Table 5.3. From Table 5.3, it is clear to see that the difference between the group with more than 15 years' experience

Table 5.2: Mean Perception Value of Model Smells

Model Smells	**Mean Perception**
Large VI	1.41
Duplicated Code	1.38
Unorganized Wires	1.3
Infinite Loop	1.11
Dead Code	0.98
Redundant Operations	0.98
Stacked Sequence Structure	0.93
Excessive Data Copies	0.86
Too Many Controls	0.52
Unconnected Front Panel Elements	0.52
Multiple Nested Loops	0.48
No Wait In a While Loop	0.37
Excessive Property Nodes	0.34
Deeply Nested Hierarchy	0.3
Build Array Inside a Loop	0.29
Fix-length Array Resizing	0
Concatenate Strings In a Loop	-0.3
Loop Once	-0.6
Deficient SubVI	-0.6
Non-reentrant VI	-1

and other groups is significant, which suggests very experienced LabVIEW developers perceived model smells less prominent compared with less experienced developers.

Figure 5.5 shows that users with less than five years working experience have a smaller perception value than users who have 5–10 years working experience. The perceived perception value for each category decreases with more working experience after five years. This perception curve (inverted U-shape) seems reasonable: When beginners first learn about LabVIEW modeling, they may only use a limited set of functions to de-

Table 5.3: P-Value Matrix

P-value	<5 years	5-10 years	10-15 years	>15 years
<5 years	-			
5-10 years	0.5864	-		
10-15 years	0.2509	0.9060	-	
>15 years	0.0012	0.0067	0.000021	-

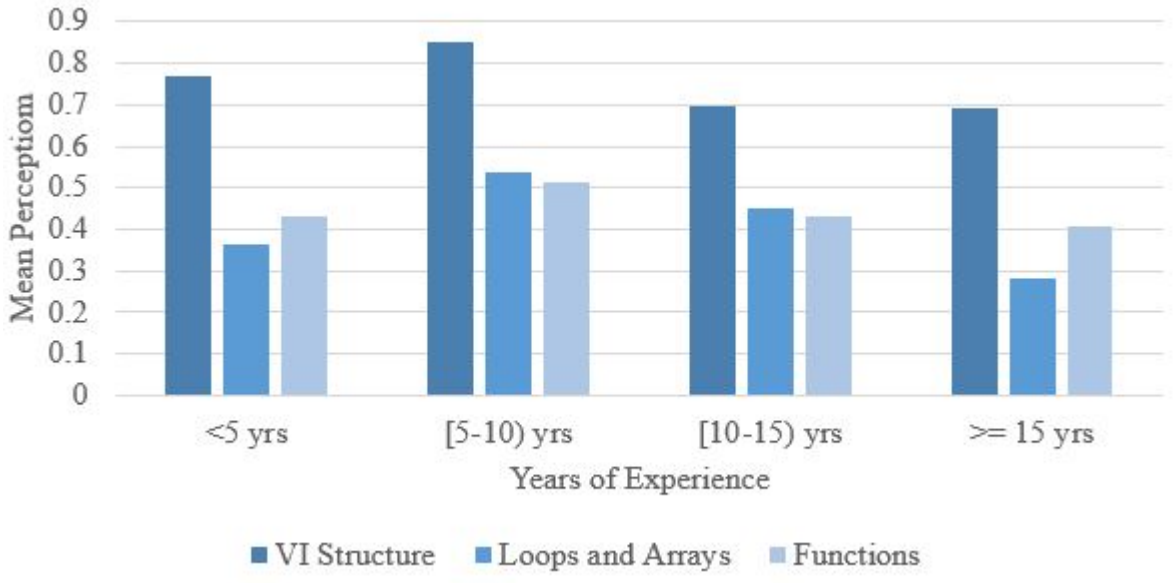


Figure 5.5: Mean Perception Value Based on Experience

sign relatively simple projects. At that point, their perception of a poor design is not well understood due to knowledge limitations and project simplicity. Later, as their knowledge accumulates and they begin to encounter more complex systems and models, they may access more design choices and modeling strategies. Meanwhile, they may have a better understanding of both good and bad designs, thus having a better ability to judge if a situation suggests a bad smell. This explains why the second group (5 -10 years' experience) agree on more model smells in each category compared to the first group (less than 5 years' experience). As the domain knowledge increases, engineers may develop more skills to fix bad designs, or avoid bad designs subconsciously. Therefore, the mean perception of model smells begins to decrease gradually as the experience accumulates.

5.5.4 The Impact of Domain on Smell Perception

RQ 5.3 focuses on the differences between industrial respondents (28 responses in total) and academic respondents (15 responses in total) regarding the perception of model smells in LabVIEW models (the other two respondents belong to group "Others"). The comparison result is presented in Figure 5.6, where it is observed that on average, industrial developers and academic developers both agree that the model smells listed in the

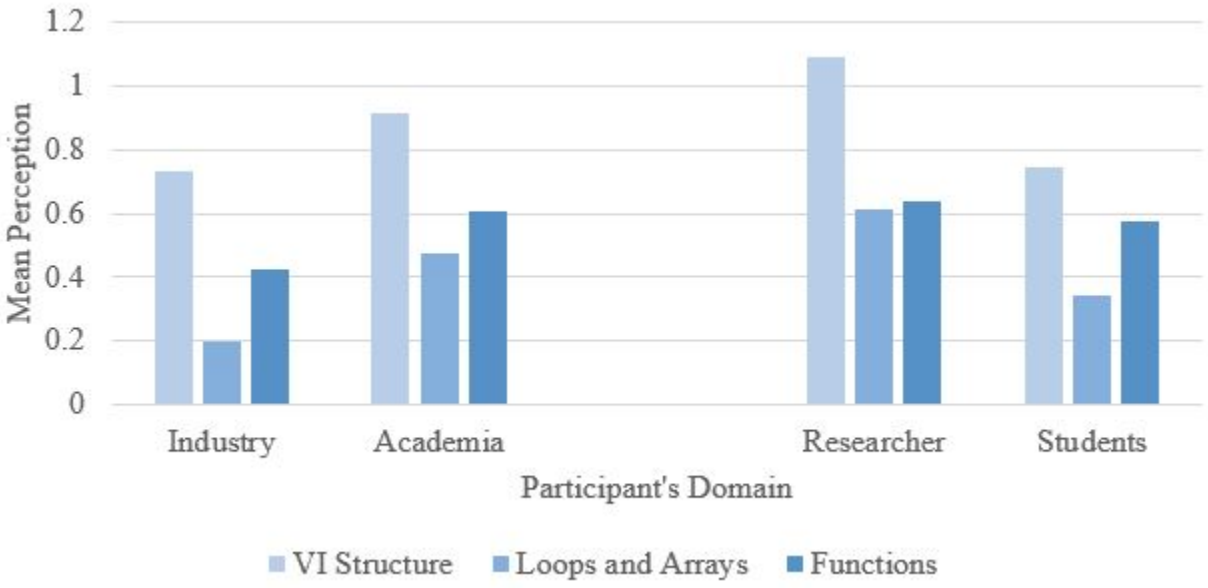


Figure 5.6: Mean Perception Value Based on Domain

survey are indications of poor model designs. Moreover, the mean perception values for three different categories of model smells obtained from academia are higher than industry respondents. However, the Mann–Whitney U test shows a significance p-value $p > 0.05$ for all of the three categories ($p = 0.7926$, 0.1282, 0.8412, respectively), which means the perception differences between industry and academia are not significant.

This study further examined the mean perception value from academic researchers and students. The results show that researchers agree with more of the model smells listed in the survey than students. However, the significance value $p = 0.9013 > 0.05$, $0.6424 > 0.05$, $0.5406 > 0.05$ for three different categories, which means that the perceived differences between researchers and students are not significant either.

5.5.5 Other LabVIEW Model Smells

To answer RQ 5.4, the last survey question was, "Do you think there are other bad smells in LabVIEW systems models that negatively affect VI qualities?" Less than half of the respondents (19 out of 45) answered this question. The recorded answers were manually examined and model smells that were already analyzed in this study (for example,

some of the respondents reported Large VI as a model smell, which was already analyzed in the survey) were eliminated. This empirical study also eliminated answers that did not match the concept of model smells, or answers that were too vague (for example, a response mentioned "Lack of scalability"). Eventually, there were seven new bad smells for LabVIEW systems models that were suggested by the respondents: **Too Many Local/-Global Variables**, **Bad Data Flow**, **Overuse of Design Template**, **Lack of Comments and Documentation**, **Lack of Error Handling**, **Inconsistent Naming and Style**, and **LVOOP** (LabVIEW Object-Oriented Programming, a design tool/language that allows users to introduce OOP concepts to LabVIEW development).

The most frequent smell suggested in the responses was Too Many Local/Global Variables, where seven out of 19 respondents answering this open question independently reported it as a model smell that they observed from their own experience, but was not listed among the 20 smells in the survey. Local variables are used to pass data between block diagram nodes. With a local variable, users can write to or read from a control or indicator on the front panel. However, introducing too many local variables may lead to memory intensive processing and race conditions may occur if writing to and reading from the same local variable at different rates. Another widely discussed bad smell in the responses is Lack of Comments and Documentation. Missing comments/documentation has been proven to accumulate technical debt [19] [144]. This study confirms that missing documentation or comments could introduce more confusion to engineers who use systems models in LabVIEW environment.

5.6 Recommendations and Lessons Learned

To offer a set of recommendations, this section first provides some specific recommendations for end-users to avoid some prominent model smells perceived based on the findings from the research questions. Second, the lessons learned from this empirical study

were summarized to steer future potential research directions for both the research community and practitioners.

5.6.1 Recommendations

The analyses of RQ 5.1 suggest that Large VI has the highest perception value among all the smells, which means respondents perceived this smell as a prominent smell for LabVIEW models. To better understand this model smell, the survey further asked the participants their own interpretation for a Large VI. There were 33 respondents that answered this open-ended question. Their responses showed that a large VI usually refers to: 1) a VI is too complicated by containing too many functions; or 2) a VI takes too much screen space. To this end, this dissertation provides three recommendations:

Recommendation 1: Follow the Single-Responsibility Principle [108], which is a programming principle indicating that every module should have responsibility over a single part of the functionality and all the related services should be narrowly aligned with that responsibility. In the context of systems modeling in LabVIEW, one VI is ideal to execute only one function/module in the system.

Recommendation 2: Restrict the size of a model within the size of the user interface window. As one respondent put in the survey,

" ... If I have to scroll, it's getting smelly ... "

Recommendation 3: Create SubVIs to increase the modularity, code reusability and program understandability.

The analysis of RQ 5.2 suggests that development experience affects the perception of model smells. Thus, for beginners who do not have too much experience, one suggestion is:

Recommendation 4: Read software documentation (user manual) before getting started to get more information on the usage of the product to save time on potential smells/bugs and lower software misuses.

The last question proposed in this study helped to identify new model smells that were never discussed before within a LabVIEW context. One of the frequently mentioned new smells is Lack of Comments. To this end, another recommendation is:

Recommendation 5: Put comments in the program during model development for maintaining, refactoring and extension.

5.6.2 Lessons Learned

The empirical evaluation (RQ 5.1) suggests that users are mostly concerned when the VI becomes overcomplicated. To this end, the analysis of VI complexity becomes necessary and important. However, current literature is silent on this topic. Similarly, Duplicated Model Part is also perceived as a prominent model smell while few investigations have been conducted within the context of graphical systems models. Therefore, the first two lessons that may lead to potential research directions for the research community are:

Lesson 1: A formal analysis of the complexity of a LabVIEW model is imperative.

Lesson 2: The research community needs to pay more attention to the investigation of duplication in systems models.

RQ 5.2 and RQ 5.3 revealed that there does not exist a significant difference in model smell perception between academia and industry, but development experience affects the understanding of model smells. From this perspective, experienced practitioners are encouraged to establish a set of guidelines based on their development experience to help unexperienced users avoid model smells in development practice. Unfortunately, such guidelines are still missing from both industry practitioners and academic researchers. Thus, the last lesson learned from this study is:

Lesson 3: Establish a set of guidelines to help inexperienced developers avoid model smells in practice.

5.7 Threats to Validity

This section discusses different categories of threats to validity in this empirical study. Following this is the examination of the means that have been undertaken to mitigate these threats. This section is organized by the introduction of three types of threats to validity: construct threats, internal threats, and external threats to validity.

5.7.1 Threats to Construct Validity

Achieving construct validity requires researchers to ensure survey questions are carefully developed based on relevant knowledge. To mitigate construct threats to validity, this study sought feedback on an initial survey design from LabVIEW experts at National Instruments. The survey was updated based on their suggestions.

5.7.2 Threats to Internal Validity

In this research, there are three internal threats to validity. In terms of study design, model smells based on three categories instead of a single smell were analyzed due to the research focus. The other internal threat to validity is the number of participants. This empirical study also did not ask why participants agree/disagree with certain designs belonging to model smells. One reason for not asking additional followup questions is that a long survey could decrease the response rate [72]. To mitigate these threats, a more comprehensive survey that combines an in-person interview in the future is planned. The in-person interview will also include additional smells obtained from the respondents (e.g., Too Many Local Variables, as mentioned in Section 5.4.5) and the justification of participant's perception. The future study will also focus on individual model smells instead of categories.

5.7.3 Threats to External Validity

In the context of this study, the external threat to validity refers to whether the survey sample is representative of all LabVIEW users. To mitigate this threat, this disser-

tation reached out to both LabVIEW users in academia and industry with different levels of experiences.

5.8 Conclusion

This chapter empirically evaluated how users perceive model smells in systems model built in LabVIEW through an online survey that was sent to various LabVIEW forums and mailing lists. The results suggest that the participants do not agree that all model smells documented in existing literature are problematic. Moreover, this study shows significant difference on model smell perception between inexperienced developers and experienced developers, but no significant difference between industry and academia. This empirical study also identified seven new model smells from the participants' responses. The novelty of this examination is the empirically evaluated confirmation of a community of experts regarding the perception of existing smells. By filling the gap between model smells summarization through a community-based survey, this examination provides new insights to the industrial and research systems modeling community about what experts consider to be smells that may impact the quality of systems models.

CHAPTER 6

A COMPLEXITY METRICS SUITE FOR LABVIEW MODELS

The previous investigation revealed that the bad smell that systems engineers are most concerned about when developing LabVIEW systems models is **Complicated Systems Models**. This observation motivated me to delve into the complexity analysis of LabVIEW models. This chapter first discusses the background of complexity metrics. Following this is the presentation of the proposed metrics suite to help systems engineers identify complicated systems models in their development practice. This chapter also offers the theoretical validation of the metrics suite using Weyuker's Property and empirical evaluation of the metrics suite using LabVIEW systems models downloaded from GitHub.

6.1 Introduction

Frederick Brooks divided software complexity into two categories: essential complexity and accidental complexity in his work *No Silver Bullet — Essence and Accidents of Software Engineering* [27]. Essential complexity is the characterization of the complexity of the problem itself, regardless of the implementation approach and the developer's experience. Accidental complexity is the non-essential complexity introduced by impoverished approaches, poor designs or inadequate complexity management. As the software evolves over time, accidental complexity can grow out of control with the introduction of new features and functionality, leaving behind an over-complicated system and higher maintenance costs [128].

One way to manage the complexity of software is to adopt software metrics, which are a set of quantifiable or countable measurements "derived from a software product, process, or resource" [199]. Software metrics are essential components of a robust and rigorous

software artifacts and play an important role in many aspects during the software development life cycle, such as cost and effort estimation [122], productivity measurements [134], performance evaluation [33] and software testing [81].

Complexity metrics, as the name suggests, are a set of assessments that measure the complexity of the code in software artifacts [152]. It is one way to evaluate the quality of the code implemented to execute certain functionality in the software. Complexity metrics also help developers to have an intuitive perception and subjective evaluation of the robustness and volatility of each line of code executed. Research on software complexity metrics has a long tradition. For example, Cyclomatic Complexity proposed by Thomas McCabe [109], was presented in 1967. Extensive studies have been conducted to survey [188], propose [56, 131], validate [67, 113], and evaluate [70, 184] software complexity metrics.

Although studies related to complexity metrics have been conducted by numerous researchers, most of the work focuses on textual programming languages. Only a few investigations in literature demonstrate complexity analysis in graphical programming languages. Moreover, complexity examinations related to graphical programming languages have focused heavily on general-purpose languages (such as Unified Modeling Language [2] and Business Process Model and Notation [135]). Limited studies have shown the complexity analysis of software artifacts built within domain-specific graphical languages.

Moreover, previous work also empirically confirmed that the complexity analysis of LabVIEW systems models is needed by LabVIEW users - the feedback from the survey shows that one of the most concerned issues during their application development is the complicated models built within LabVIEW. Based on these observations, the goal of this investigation is to *propose a complexity metrics suite for LabVIEW systems models* to help users evaluate the complexity of LabVIEW systems models in a quantifiable manner. This chapter establishes a complexity metrics suite to characterize different aspects of LabVIEW systems models by redefining some existing metrics and offering new metrics. The

metrics suite covers 11 metrics which are divided into six categories: Cyclomatic Complexity [109], Halstead Complexity [69], Information Flow Complexity [73], Card & Glass Complexity [32], Cognition Complexity [156], and Front Panel Complexity. Weyuker's validation [184] was adopted to theoretically validate the metrics suite. Finally, a proof-to-concept prototype was implemented. This study also acquired 10 open-source LabVIEW models from 10 different projects from GitHub to empirically evaluate the metrics suite using the prototype that has been implemented. The contributions of this work are briefly presented as follows:

1. The ***description*** of a complexity metrics suite for LabVIEW systems models.
2. A formal theoretical ***validation*** of the metrics suite.
3. An empirical ***evaluation*** of the metrics suite.

The rest of this chapter is organized as follows. Section 6.2 discusses the proposed metrics suite for the complexity analysis of LabVIEW systems models. Section 6.3 provides the theoretical validation of the metrics suite while Section 6.4 offers an empirical evaluation. Section 6.5 examines the threats to validity and Section 6.6 surveys related work in existing literature associated with this study. Finally, Section 6.7 concludes this chapter.

6.2 Related Work

The study on the examination of LabVIEW model complexities is barely seen in existing literature. The most similar work to this study is the complexity metric analysis for Simulink models conducted by Olszewska [129]. However, there exist several differences between LabVIEW models and Simulink models (a comparison study between these two tools was conducted by Tavsner [173]). Moreover, Olszewska's work failed to provide a validation for the metrics they proposed compared with this study.

Much of the existing research is related to complexity analysis within the context of OOP [39, 40, 98, 114, 121]. Hirzalla proposed a metrics suite for evaluating flexibility and complexity in Service Oriented Architectures (SOA) [76]. Because SOA is widely used in building distributed systems, the metrics proposed in their work share many similarities with complexity metrics for OOP. Power et al. [137] adapted some common software metrics and applied these metrics to measure the complexity of grammar-based software applications. The investigation of a complexity metrics suite is also seen in the area of Web applications, such as Web ontology [191] and Cascading Style Sheet (CSS) [3].

Other investigations focused on the analysis of the metrics themselves. As one of the most famous legacy metrics, multiple studies have been conducted to understand Cyclomatic Complexity since the term was first coined in 1976 [109]. The analyses span over several disciplines, such as program implementation [104], metric evaluation [179], and metric application [183]. Compared with numerous investigations of Cyclomatic Complexity, few studies focused on Cognition Complexity and User Interface Complexity. Misra et al. [117] proposed a metric suite to understand cognition complexity for OOP. Misra's work serves as the foundation of the cognition complexity metric proposed in this dissertation study. Similar, Riegler [142] examined the metrics affecting the user interface complexity in mobile applications.

6.3 Proposed Metrics Suite

This section introduces the proposed metrics suite for analyzing the complexity of LabVIEW systems models. The metrics suite includes six categories of metrics: Cyclomatic Complexity (CyC), Halstead complexity (HC), Information Flow Complexity (IFC), Card & Glass Complexity (CG), Cognition Complexity (CoC), and Front Panel Complexity (FP). Each metric in the suite covers a distinct aspect of the complexity characterization of LabVIEW models.

6.3.1 Cyclomatic Complexity

Cyclomatic Complexity (also known as "McCabe Complexity") was proposed by McCabe [109] in 1976. As one of the earliest and popular software metrics to measure software complexity, numerous investigations related to cyclomatic complexity have been conducted in different areas, such as its evaluation [102], application [171], and generalization [80].

The measurement of cyclomatic complexity depends on the program's Control Flow Graph (CFG). A CFG uses graph notations to model the control flow of a program. Nodes in a CFG represent execution commands while edges in a CFG indicate the direction of program execution. In McCabe's work [109], the cyclomatic complexity is originally defined as:

$$Cyclomatic\ Complexity = E - N + 2P \tag{6.1}$$

where E is the number of total edges, N is the number of total nodes, and P is the number of exit nodes in a CFG. For an individual program, there is always one exit node ($P = 1$). Thus, Equation 6.1 is simplified as:

$$Cyclomatic\ Complexity = E - N + 2 \tag{6.2}$$

McCabe proved that for a program that only has one starting node and one exit node, the cyclomatic complexity can be represented as:

$$Cyclomatic\ Complexity = D + 1 \tag{6.3}$$

where D is the number of total *decision points* in a program. A decision point is where a program produces variations to a CFG, such as *if-else* statement, *for/while* statement and *switch* statement.

Table 6.1: Cyclomatic Complexity Computation for Figure 6.1

	Code Fragment 1	Code Fragment 2	Code Fragment 3
# of Edges (E)	2	4	6
# of Nodes (N)	3	4	5
# of Decision Points (D)	0	1	2
E-N+2 (Equation 6.2)	2-3+2 = 1	4-4+2 = 2	6-5+2 = 3
D+1 (Equation 6.3)	0+1 = 1	1+1 = 2	2+1 = 3

This section will illustrate the above equations through an example. Suppose there are three code fragments, as shown in Figure 6.1. The corresponding CFGs of these code fragments are also shown in Figure 6.1. The calculation of cyclomatic complexity is listed in Table 6.1.

Mapping the concept and equation of cyclomatic complexity to LabVIEW models, this dissertation modified the number of decision points in a program to the number of decision blocks in a model. In LabVIEW, there are four types of decision blocks: *For Loop blocks* (similar to For Loop statements), *While Loop blocks* (similar to While Loop statements), *Case Structure* (similar to if...else... statements) and *Event Structure* (similar to Switch statements). Thus, in this dissertation, Cyclomatic Complexity (CyC) is redefined as:

$$CyC = D + 1 \tag{6.4}$$

where D refers to the number of decision blocks in a LabVIEW systems model.

6.3.2 Halstead Complexity

Halstead Complexity was first presented by Maurice Halstead in 1977 [69]. Halstead Complexity is calculated based on two components in a program: operators, indicating what will be processed during the program execution, and operands, specifying how these operators will be processed. Let:

$n_1 = Number\ of\ distinct\ operators$

$n_2 = Number\ of\ distinct\ operands$

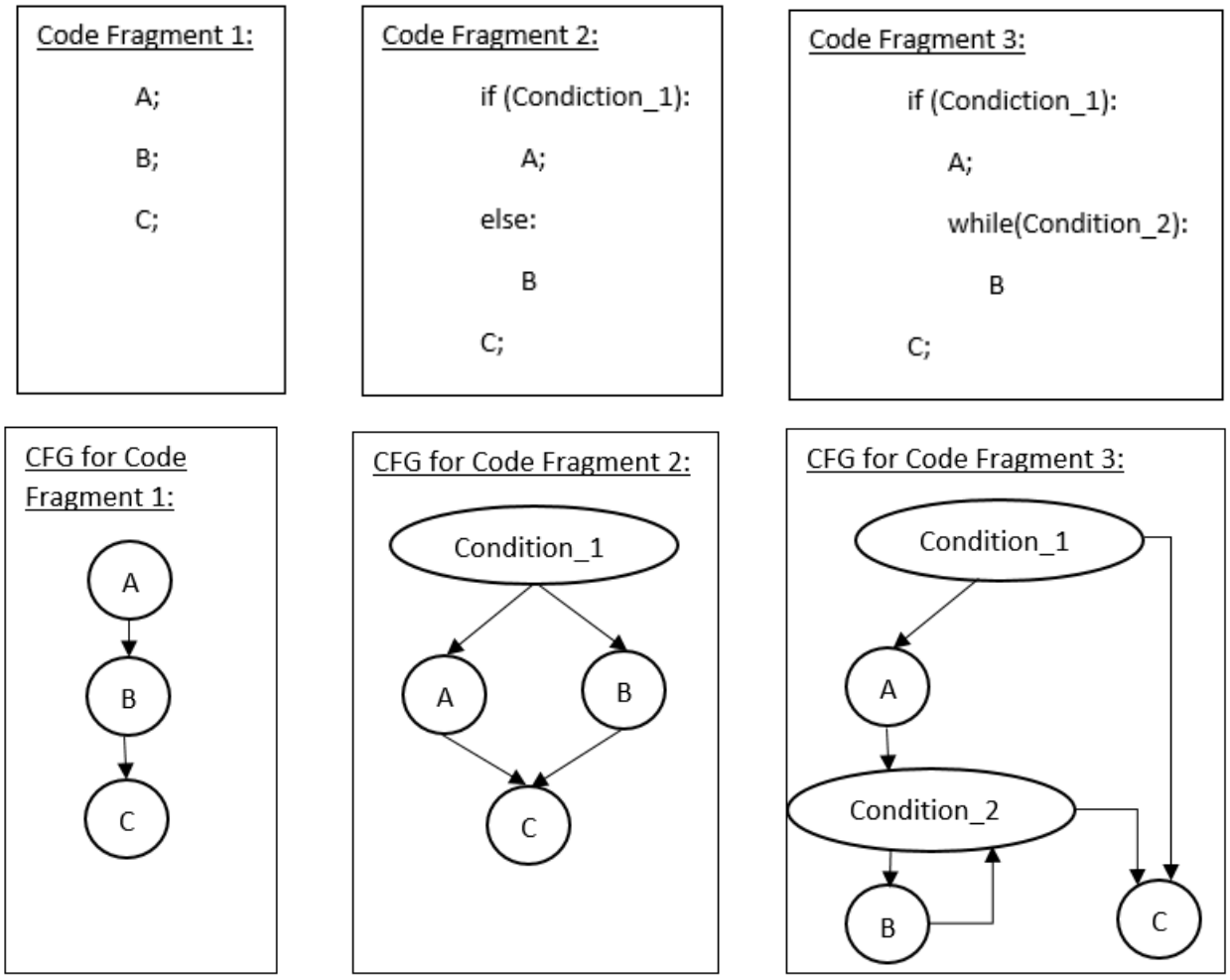


Figure 6.1: Code Fragments with Different Number of Decision Points and Their Corresponding CFGs

$N_1 = Number\ of\ total\ operators$

$N_2 = Number\ of\ total\ operands$

Halstead defined the following metrics [69] to measure the software complexity:

$$Program\ Vocabulary : n = n_1 + n_2 \tag{6.5}$$

$$Program\ Length : N = N_1 + N_2 \tag{6.6}$$

$$Program\ Volume : V = N \times \log_2 n \tag{6.7}$$

$$Program\ Difficulty : D = \frac{n_1}{2} \times \frac{N_2}{n_2} \tag{6.8}$$

$$Program\ Effort : E = V \times D \tag{6.9}$$

$$Time\ to\ implement\ the\ program : T = \frac{E}{18} \tag{6.10}$$

$$Program\ delivered\ bugs : B = \frac{E^{\frac{2}{3}}}{3000} \tag{6.11}$$

To map the concept of Halstead Complexity to LabVIEW models, this dissertation revised the original definitions. First, because LabVIEW follows a data-flow programming paradigm, data flows from one block to another block. Instead of counting how many times operators appear ($N_1 = Number\ of\ total\ operators$) in a program, my work defined $A_1 = Number\ of\ total\ operator\ flows$. In addition, blocks instead of operands were adopted to reflect the nature of LabVIEW. Therefore, the basic Halstead Complexity metrics in this study are introduced as:

$\alpha_1 = Number\ of\ distinct\ operators$

$\alpha_2 = Number\ of\ distinct\ data\ processing\ blocks$

$A_1 = Number\ of\ total\ operator\ flows$

$A_2 = Number\ of\ total\ data\ processing\ blocks$

Second, the metrics suite in this study removed two metrics: time to implement the program (defined in Equation 6.10) and program delivered bugs (defined in Equation 6.11). Muslija [126] argued that the integer values 18 and 3000 in these two equations are rigid and not strictly proved.

Finally, in Equation 6.7, when $n = 1$, it means that a model contains only one operator or one processing block. The metric $Volume$ will be computed as 0 ($V = N \times \log_2 1 = N \times 0 = 0$). However, a $Volume$ value of 0 is inconsistent with the fact when the model contains only one operator or one processing block. To this end, a revision was made to Equation 6.7 (see Equation 6.14).

After these revisions, the Halstead Complexity (HC) metrics in the metrics suite are defined as:

$$Model\ Vocabulary : \alpha = \alpha_1 + \alpha_2 \tag{6.12}$$

$$Model\ Length : A = A_1 + A_2 \tag{6.13}$$

$$Model\ Volume : V = A \times \log_2(\alpha + 1) \tag{6.14}$$

$$Model\ Difficulty : D = \frac{\alpha_1}{2} \times \frac{A_2}{\alpha_2} (D = \frac{\alpha_1}{2} \iff \alpha_2 = 0) \tag{6.15}$$

$$Model\ Effort : E = V \times D \tag{6.16}$$

For example, considering the following example: given two non-zero integers x and y, the goal is to calculate the value of $\frac{1}{x^2} + \frac{1}{y^2}$. This program can be expressed in a LabVIEW block diagram shown in Figure 6.2.

In this example, there are three distinct operators: two integer controls x and y and one double-float indicator *Result*, thus $\alpha_1 = 3$. Meanwhile, this example has three distinct data processing blocks: one block for computing a number's square; one block to calculate a number's reciprocal and one block to sum two numbers, thus $\alpha_2 = 3$. Integer x flows three times in this example: integer x to block x^2 to block $\frac{1}{x}$ to block "+". Similar, integer y also flows three times. Processing block "+" transfers the result to indicator *Result*. Therefore, the operator flows in this program is $A_1 = 3 + 3 + 1 = 7$. Finally, the number of total data processing blocks is $A_2 = 5$ (two square block, two reciprocal blocks and one

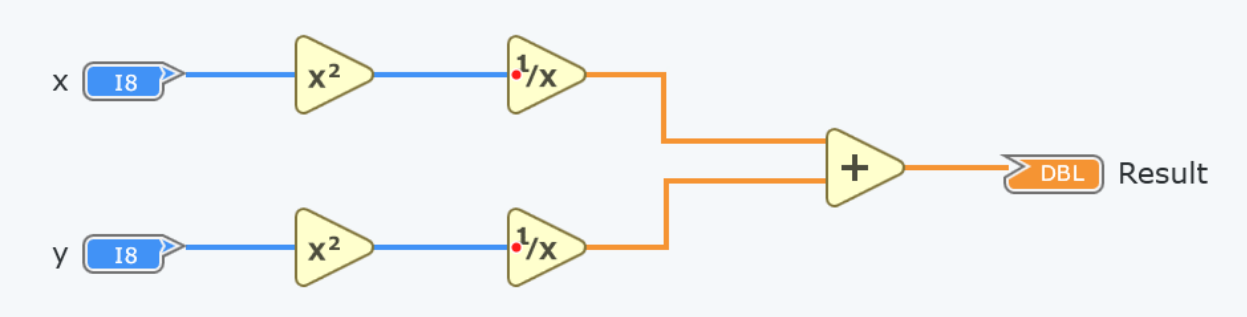


Figure 6.2: A LabVIEW Block Diagram to Calculate $\frac{1}{x^2} + \frac{1}{y^2} (x \neq 0; y \neq 0)$

addition block). Based on Equation 6.12 to Equation 6.16, the calculations of the metrics in Halstead Complexity are as follows:

- $Model\ Vocabulary : \alpha = \alpha_1 + \alpha_2 = 3 + 3 = 6;$
- $Model\ Length : A = A_1 + A_2 = 7 + 5 = 12;$
- $Model\ Volume : V = A \times \log_2(\alpha + 1) = 12 \times \log_2(6 + 1) \approx 33.69;$
- $Model\ Difficulty : D = \frac{\alpha_1}{2} \times \frac{A_2}{\alpha_2} = \frac{2}{3} \times \frac{3}{5} = 2.5;$
- $Model\ Effort : E = V \times D = 36.69 \times 2.5 = 84.225$

6.3.3 Information Flow Complexity

Information Flow Complexity, given by Henry and Kafura [73], measures the information flow between system modules. Before the discussion of information flow complexity, this section first introduces two concepts used in information flow complexity: *fan-in* and *fan-out*. Fan-in and fan-out are two terms used in a logic gate which perform logical functions in digital integrated circuits. Fan-in is the number of inputs of a gate and fan-out is the number of outputs of a gate. In the context of software complexity analysis, fan-in and fan-out are used to measure the complexity of data flows between modules. In Henry and Kafura's work [73], they defined information flow complexity as:

$$Information\ Flow\ Complexity = Length * (Fan\text{-}in * Fan\text{-}out)^2 \tag{6.17}$$

where *Length* in Equation 6.17 represents the number of lines of source code in a program.

The basic idea of information flow complexity is to understand the information flow between modules. In LabVIEW, the modularity is achieved by the implementation and adoption of SubVIs (discussed earlier in Section 2.2). To give a better illustration of the concept of SubVI, consider the example that has been introduce in Figure 6.2. To use a SubVI, users first create an *Icon* for the block diagram shown in Figure 6.2. An icon is

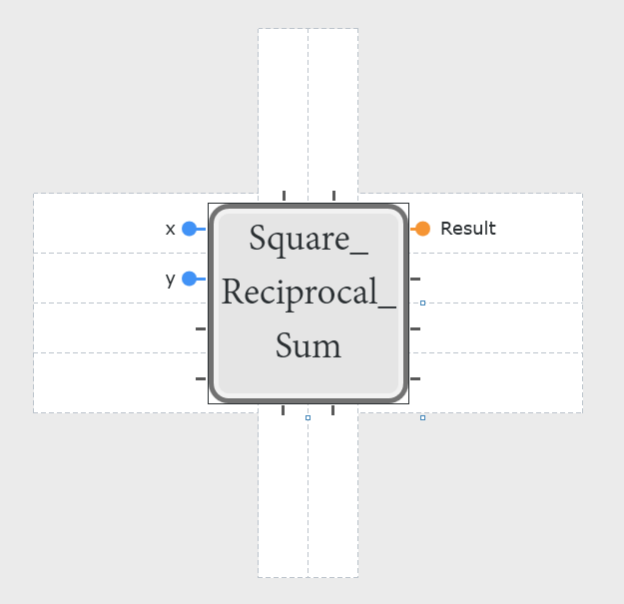


Figure 6.3: The Icon for the SubVI *Square_Reciprocal_Sum*

similar to a function signature in textual programming languages - it helps to identify the user-defined function and the function input and output. Suppose the SubVI for the function shown in Figure 6.2 is named as *Square_Reciprocal_Sum*. Figure 6.3 shows the icon for this SubVI in LabVIEW.

The x and y shown on the left of this icon indicates that this SubVI receives two inputs (the blue color shows that these two inputs are 8-bit integers). The output of this SubVI is *Result* (the orange color shows that the output is a float number). After encapsulating the block diagram into this SubVI, users just need to call the SubVI by using its icon. Figure 6.4 shows an equivalent block diagram to the block diagram in Figure 6.2 after the adoption of a SubVI.

This dissertation defines fan-out as *the data flow from one VI to its SubVI* and fan-in as *the data flow from SubVI to its caller VI*. Suppose there are N SubVIs in a VI, for each SubVI $i \in N$, the number of fan-in and fan-out is $Fan\text{-}in_i$ and $Fan\text{-}out_i$, respectively. The Information Flow Complexity (IFC) for a VI is defined as:

$$IFC = \sum_{i}^{N} (Fan\text{-}in\_i + Fan\text{-}out\_i)^2 \tag{6.18}$$

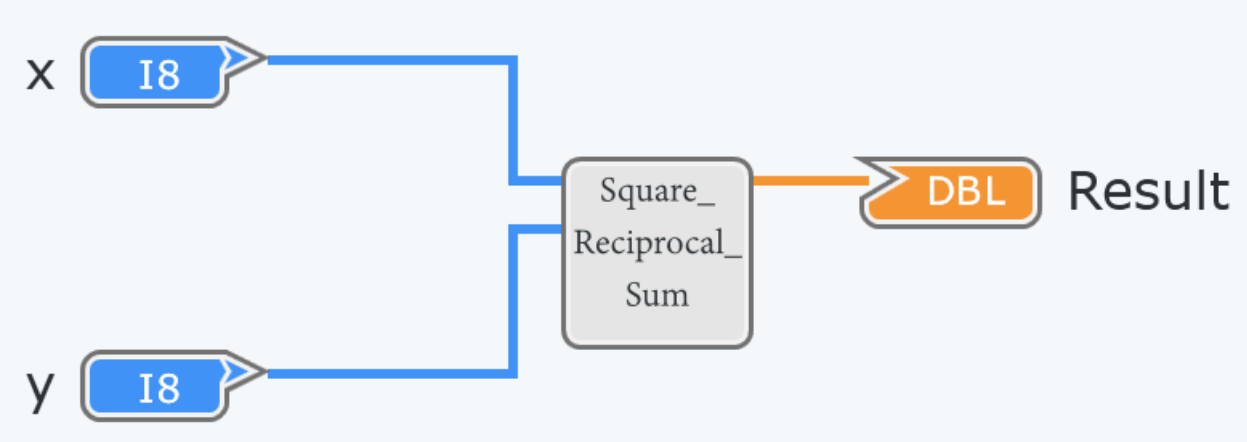


Figure 6.4: The Modularization of VI by the Adoption of SubVI

Compared with the original definition, the formula for information flow complexity proposed in this dissertation revised the operand in Equation 6.18 from $\times$ to $+$. The limitation in the original metric definition is that when there is only a fan-in but no fan-out $(Length * (Fan\text{-}in * 0)^2))$, or only a fan-out but no fan-in $(Length * (0 * Fan\text{-}out)^2)$ for a system, the information flow complexity yields to 0. However, when a model only has fan-in or fan-out, there still exists date flows. To this end, my research suggests applying $+$ instead of $*$ in the metric calculation.

6.3.4 Card & Glass Complexity

Card and Glass proposed a design complexity metric [32] based on information flow complexity. Card & Glass Complexity contains three metrics:

– Data complexity:

$$D_x = \frac{Px}{fan\text{-}out + 1} \tag{6.19}$$

– Structural Complexity:

$$S_x = fan\text{-}out^2 \tag{6.20}$$

– System Complexity:

$$C_x = D_x + S_x \tag{6.21}$$

where P_x is the summation of system input and system output. Card & Glass's metric indicates that system complexity is the summation of data complexity and structural complexity. Following the definition above, the Card & Glass complexity for LabVIEW systems models can be defined as:

- Data complexity:

$$D = \frac{P}{fan\text{-}out + 1} \tag{6.22}$$

- Structural Complexity:

$$S = fan\text{-}out^2 \tag{6.23}$$

- Model Complexity:

$$C = D + S \tag{6.24}$$

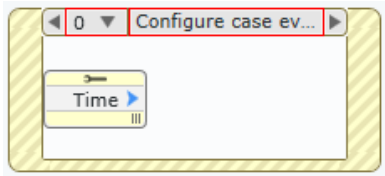
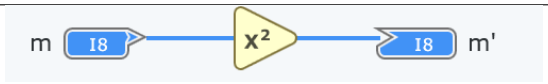
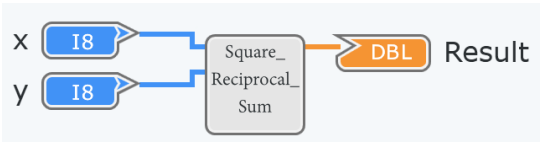
where P is the summation of the number of inputs and outputs for a LabVIEW model. Here, $fan\text{-}in$ and $fan\text{-}out$ follow the definition discussed in Section 6.2.3.

6.3.5 Cognition Complexity

Cognition plays an important role in "understanding the fundamental characteristics of software" [156]. Cognition complexity refers to the difficulty to intuitively understand a piece of code. To quantitatively assess the cognition complexity of a software program, Shao and Wang proposed the concept of Basic Control Structures (BCSs) [182], which are a set of control structures to build software in the context of OOP. Moreover, they assigned Cognitive Weights (CWs) to BCSs as a measurement to evaluate the degree of effort to comprehend a specific basic control block [156]. The cognition complexity is thus defined as the summation of cognitive weights of all the BCSs in a program.

The current literature related to cognition complexity within the context of graphical systems models is still scarce. To evaluate the cognition complexity in a LabVIEW block diagram, previous investigations were revised to fit LabVIEW. My research divides blocks in a LabVIEW block diagram into two categories: blocks with cognitive weight and

Table 6.2: Cognitive Weights for LabVIEW Blocks

Category	Blocks	Cognitive Weight	Graphical Notation Example
Iteration	For Loop	3	
	While Loop	3	
Branch	Case Structure	2	
	Event Structure	2	
Function Calls	Pre-defined Processing Blocks	1	
	User-defined Processing Blocks (SubVIs)	1	
Others	Other blocks	0	-

blocks without cognitive weight. Table 6.2 summarizes these two categories and cognitive weights for different blocks.

My research assigns cognitive weights to three different block categories. The first category is Iteration blocks, which include For Loop structures and While Loop structures. Similar to other general-purpose programming languages, a For Loop is used to execute code a fixed number of times, while a While Loop executes the code as long as a Boolean expression remains true. The cognitive weight for iteration blocks was defined as 3. The second category is Branch blocks. In LabVIEW, there are two kinds of branch blocks:

Case Structure and Event Structure. A case structure is similar to an *if.../if...else...* statement and an event structure is similar to a *switch* statement. This work defines the cognitive weight for branch category as 2.

The last block category with cognitive weight is Function Calls. Two types of blocks are included in this category: Pre-defined Processing Blocks and User-defined Processing Blocks (i.e., SubVIs). Pre-defined processing blocks provided by LabVIEW offer users a variety of functions to build a program (for example, the square block, reciprocal block and summation block shown in Figure 6.2). SubVIs, as illustrated in Figure 6.4, are also function calls but defined by users instead of LabVIEW. Both of these two function calls work like a black-box - they hide the implementation detail from users. Users just need to provide the required input and the result will be returned automatically. Therefore, the cognitive weight for function calls is 1. The rest of the blocks in LabVIEW block diagrams do not have a cognitive weight (for example, definition blocks, comment blocks and wires). The cognitive weight values defined in this investigation are consistent with previous studies on cognition complexity analysis for OOP [118, 156, 182].

Suppose there are M first-layer linear blocks in a block diagram. For each block $m \in M$, there exist N sub-layers. In each sub-layer $n \in N$, there are P linear blocks. A formal representation of Cognition Complexity (CoC) for a model is formulated as:

$$CoC = \sum_{m=1}^{M} \left\{ \prod_{n=1}^{N} \left[\sum_{p=1}^{P} Cognitive\ Weight\,(p) \right] \right\} \tag{6.25}$$

For example, the cognition complexity for the block diagram shown in Figure 6.4 is 1: there is one SubVI call (self-defined function $Square\_Reciprocal\_Sum$) which has the cognitive weight of 1. There are three terminals (two controls and one indicator) in this example. They are definition blocks that do not contribute to the cognition complexity of this block diagram. Similarly, the cognition complexity for the block diagram shown in Figure 6.2 is 5: there are five pre-defined processing blocks (two square blocks, two recip-

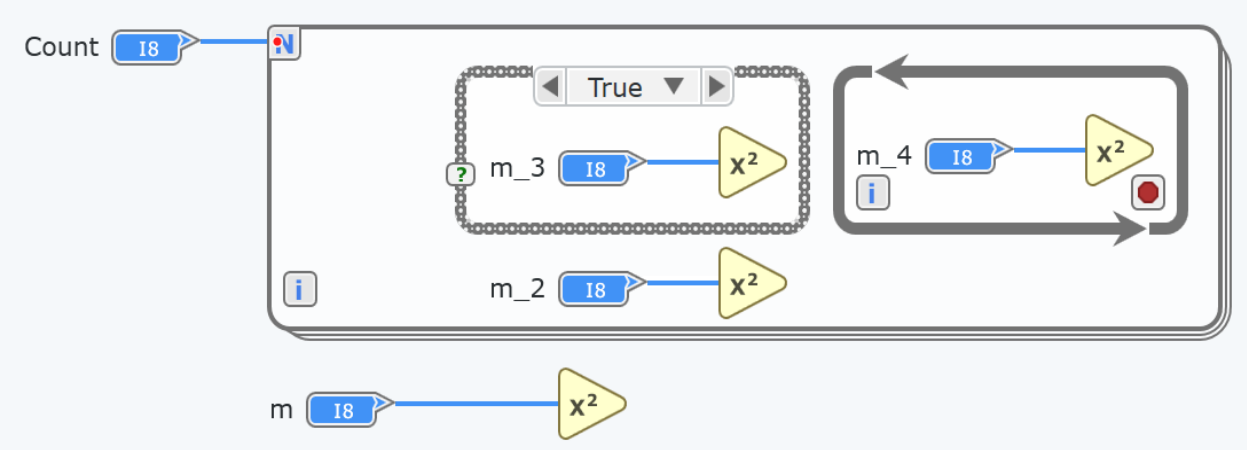


Figure 6.5: A more complicated example for cognition complexity calculation

rocal blocks and one addition block) in this diagram. Each of the blocks has the cognitive weight of 1. Thus, the cognitive complexity is $5 \times 1 = 5$. The three terminals in this block diagram do not have cognitive weights.

The block diagram represented in Figure 6.5 shows a more complicated example. In Figure 6.5, there are two first-level blocks: a For Loop structure and a Square block. In the For Loop, there are three sub-level blocks: a Case Structure, a While Loop and another Square block. Inside of the While Loop and For Loop, there is another Square block. Based on Equation 6.25, the cognition complexity of the block diagram shown in Figure 6.5 is calculated as: $1 + \{3 \times [2 \times (1)] + 3 \times [3 \times (1)] + 3 \times 1\} = 19$.

6.3.6 Front Panel Complexity

Different from other models that only have logical representations of the program (such as Simulink and UML), LabVIEW also provides a user interface - a front panel for developers, as discussed in Section 2.2.1. The front panel acts as a core link between users and systems. A good user interface design "comes down to bringing the program model and the user model in line" [59]. The inclusion of front panels in LabVIEW inspired me to scope user interface complexity analysis into the investigation, which assists developers to evaluate the complexity of interface designs for a LabVIEW model. In this dissertation,

two aspects affecting the complexity of front panels were examined: Element Smallness and Interface Density.

For a given element i on a front panel, the metrics suite first define its size as:

$$Size_i = Length_i * Height_i \tag{6.26}$$

where $Length_i$ is the length of element i and $Height_i$ is the height of element i.

The first complexity metric for user interface in this study is Element Smallness. The analysis of element smallness is inspired by the research done by Riegler and Holzmann [142]. They argued that "smaller objects tend to be harder to identify and interact with," thus increasing its understanding complexity. Therefore, Element Smallness (ES) is defined as:

$$ES = \frac{\left(\frac{Average\ Width}{Screen\ Width}\right) + \left(\frac{Average\ Height}{Screen\ Height}\right)}{2} \tag{6.27}$$

where $Average\ Width$ is the mathematical mean value of the width of all the elements, $Average\ Height$ is the mathematical mean value of the height of all the elements, $Screen\ Width$ and $Screen\ Height$ are the width and height of a front panel's screen, respectively.

The second complexity metric for front panel complexity in this study is Interface Density. It measures the degree to which elements placed on the front panel occupy the screen. Suppose there are N elements in total. Interface Density (ID) is defined as:

$$ID = \frac{\sum_i^N (size_i)}{Screen\ Size} \tag{6.28}$$

where $Screen\ Size$ is the size of the screen for the front panel in a model. This metric shows the degree to which a front panel is clustered or empty.

6.4 Theoretical Validation

In light of the suitability of the proposed metrics suite, a theory validation to the metrics was performed. Several investigations have been conducted to theoretically validate the complexity metrics for software programs within the context of OOP [55, 184, 198]. One of the benchmark works among existing studies is the investigation proposed by Weyuker [184]. As a point of reference for the examination of software complexity metrics, Weyuker's work has been adopted by many other investigations (such as [51, 117]). My research also uses Weyuker's work to theoretically validate the proposed complexity metrics. Because Weyuker's validation was original for OOP, the term "program" was used. To map Weyuker's validation to LabVIEW, my work revised "program" to "systems model." Moreover, metrics related to front panel were removed since user interface complexity was not included in Weyuker's validation.

Property 1*:* $(\exists\ P)\ (\exists\ Q)\ (|P| \neq |Q|)$*, where P and Q are two different systems models.*

This property states that a metric should not contribute to the same value when given two different systems models. From the definition of metrics equations (6.4, 6.12-6.16, 6.18, 6.22-6.25), it is clear that all of the metric computations depend on different properties of a system model, but not constant values. Thus, Property 1 holds for all of the metrics in the suite.

Property 2*: Let c be a non-negative number, then there are finite combinations to obtain complexity c.*

Cyclomatic Complexity is related to the number of decision blocks. In LabVIEW, there are four decision blocks: For Loop, While Loop, Case Structure and Event Structure. When the complexity is given as a fixed value c, it is always possible to enumerate all the combinations of these decision blocks to make a model's Cyclomatic Complexity equal to c. For example, suppose for a given model, $c = 5$, based on Equation 6.4, $c = 5 = D + 1$, thus $D = 4$. In other words, there should be four decision blocks in the block diagram of

Table 6.3: Combination of Decision Blocks (Cyclomatic Complexity = 5)

No. of Block Types	**Possible Combinations**	**No. of Combinations**
1	4 FLs or 4 WLs or 4 CSs or 4 ESs	4
2	1 FL + 3 WLs or 2 FLs + 2 WLs or 3 FLs + 1 WL (FL and WL can be placed by other decision structures)	18
3	2 FL + 1 WL + 1 CS or 1 FL + 2 WLs + 1 FL or 1 FL + 1 WL + 2 CSs (FL, WL and CS can be placed by other decision structures)	12
4	1 For Loop and 1 While Loop and 1 Case Structure and 1 Event Structure	1

FL = For Loop, WL = While Loop, CS = Case Structure, ES = Event Structure.

this model. When using only one type of decision structure, there are four possible ways: using four For Loop blocks only, or using four While Loop blocks only, or using four Case Structures only, or using four Event Structures only.

When using two types of decision blocks, there are six possible combinations: For Loop + While Loop; For Loop + Case Structure; For Loop + Event Structure; While Loop + Case Structure; While Loop + Event Structure; Case Structure + Event Structure. For each combination, in order to get four decision blocks, there are three ways to pick. For example, suppose the combination of For Loop and While Loop is chosen, the three possible ways are: one For Loop and three While Loop; or two For Loops and two While Loops; or three For Loops and one While Loop. Thus, there are $3 * 6 = 18$ combinations in total to choose four decision blocks when using two types of decision blocks.

Similarly, there are 12 combinations when choosing three types of decision blocks and there is one combination when choosing four types of decision blocks. Thus, there are $4 + 18 + 12 + 1 = 35$ combinations of decision blocks to make a model's Cyclomatic Complexity equal to 5. Table 6.3 gives a detailed explanation on this example.

Starting from this point, it can be generalized that all of the metrics in the suite hold for this property. In fact, this metric will be met by any metrics due to the fact that "the universe of discourse deals with almost a finite set of applications" [166].

***Property 3**: There are distinct models P and Q such that $|P| = |Q|$.*

This property implies that two models can have the same complexity metric values. Obviously, the metrics in this chapter satisfy this property because in LabVIEW, there are more than one type of blocks. LabVIEW users can always construct different models that have the same complexity values. For example, a model that contains only one While Loop and a model contains only one For Loop have the same Cyclomatic Complexity and Cognition Complexity.

***Property 4**: $(\exists\ P)\ (\exists\ Q)\ (P \equiv Q\ \&\ |P| \neq |Q|)$.*

This property states that there exist two models with the same input, and they will produce the same output. However, they are implemented in different ways. This property shows that the metrics should focus on internal implementation. The Cyclomatic Complexity in the metrics suite satisfies this property. For example, there are two models that do not receive any input and the output is showing a string three times. There are two ways to implement this: Model A uses a loop with three iterations and places a function inside of the loop. Model B just repeats the string code three times without using any loops. Thus, model A has a Cyclomatic Complexity of 2 (1+1, one decision block is used) while Model B has a Cyclomatic Complexity of 1 (1+0, zero decision block is used). Similarly, this example also fits Halstead Complexity because when using different structures, the model vocabulary and length change based on its implementation detail. Further, if creating a SubVI that executes the shown string function, the Information Flow Complexity and Card & Glass Complexity is different compared with a model that does not use a SubVI to produce the same result. Cognition complexity also satisfies this property as illustrated in examples shown in Figure 6.2 and Figure 6.4. In summary, the metrics proposed in the suite could reflect the implementation differences. Therefore, all the metrics in this dissertation satisfy this property.

***Property 5**: $(\forall P)\ (\forall Q)\ (|P| \leq |P;Q|\ and\ |Q| \leq |P;Q|)$.*

Suppose a model (P, Q) contains two components: P and Q. This property states that the complexity of model (P, Q) is larger than the complexity of its components P and Q. For Cyclomatic Complexity, this property is satisfied. Suppose there are m decision blocks in P ($m \geq 0$), n decision blocks in Q ($n \geq 0$), when combining P and Q, the new Cyclomatic Complexity is $m + n + 1$. Obviously, this value is always bigger or equal to $m + 1$ and $n + 1$. For Halstead Complexity, this property is also satisfied because any model component always has less or equal operators and operands than the whole model, thus making all the metrics in Halstead Complexity satisfy Property 5.

For Information Flow Complexity and Card & Glass Complexity which measure the data flow between a VI and its SubVI, if P and/or Q are SubVIs and there exist other SubVIs, $|P| < |P;Q|$ *and/or* $|Q| < |P;Q|$. If P and Q are the only two SubVIs with Fan-in and Fan-out, then $|P| < |P;Q|$ *and* $|Q| < |P;Q|$. If P (or Q) is the only SubVI with Fan-in and Fan-out, then $|P| = |P;Q|$ (or $|Q| = |P;Q|$). If P and/or Q are not SubVI components and the model contains other SubVIs, then $|P| < |P;Q|$ *and* $|Q| < |P;Q|$. If P and/or Q are not SubVI components and the model does not contain other SubVIs, then $|P| = |P;Q| = 0$ *and* $|Q| = |P;Q| = 0$. In summary, Property 5 holds for Information Flow Complexity and Card & Glass Complexity metrics.

Property 5 also holds for Cognition Complexity because the calculation of Cognition Complexity is based on the number of blocks with cognitive weight. Any model component always has less or equal blocks with cognitive weight compared with the entire model. Based on the above analyses, Property 5 holds for all the complexity metrics proposed in the suite.

Property 6: $(\exists P)\ (\exists Q)\ (\exists R)\ (|P| = |Q|\ \&\ |P;R| \neq |Q;R|)\ \&\ (|P| = |Q|\ \&\ |R;P| \neq |R;Q|)$.

This property asserts that if a model R is combined with two models P and Q that have the same complexity metric values, the combined models $R+P$ and $R+Q$ have different complexity metrics. This property does not hold for Cyclomatic Complexity. Suppose

model R has a Cyclomatic Complexity value of r, P has a Cyclomatic Complexity value of p, model Q has a Cyclomatic Complexity value of q. Based on the precondition, it is observed that $p = q$. The new models $R + P$ and $R + Q$ have Cyclomatic Complexity values of $r + p$ and $r + q$, respectively. Because of $p = q$, it is clear that $r + p = r + q$, which means the two new models $R + P$ and $R + Q$ have the same Cyclomatic Complexity. The Halstead Complexity is more complicated to analyze: if R shares some of the operators or operands with P/Q, but does not share any of the operators or operands with Q/P, then this property holds; otherwise not. Similarly, Information Flow Complexity and Card & Glass Complexity may also satisfy or not satisfy this property based on SubVI calls before and after model combination. Cognition Complexity does not satisfy this property because the model combination does not change the original model structure, thus resulting in no influence on cognition complexity. In summary, this property partially holds the metrics proposed in the suite.

Property 7*: There are program bodies P and Q such that Q is formed by permuting the order of the statements of P,and ($|P| \neq |Q|$).*

This property concerns the effect of changing the order of statements in a program. Though changing the statement order may affect the program complexity in some programming languages, it will not affect LabVIEW models. This is due to the nature of data flow programming - LabVIEW models are built based on blocks which are connected through wires - the location of blocks do not affect a model's functionality. Thus, all the metrics in the suite do not satisfy this property.

Property 8*: If P is a renaming of Q, then $|P| = |Q|$.*

This property states that the metrics should not be affected by the renaming of blocks. This property holds for all the metrics in the suite because all the metric calculations are irrelevant to any naming or renaming of blocks.

Property 9*:* $(\exists P)\ (\exists Q)\ (|P|+|Q| < |P;Q|)$.

This property states that when combining two models into a new one, the complexity of a newly created model is larger than the summation of individual models. This property has been criticized by several investigations [5, 40, 157, 167, 192]. Therefore, this section does not discuss this property within the context of LabVIEW systems models.

6.5 Empirical Evaluation

To empirically evaluate the metrics suite, a tool that can automatically retrieve different complexity metrics defined in Section 6.2 was developed. This tool supports the complexity analysis for systems models built in LabVIEW NXG[1] and LabVIEW NXG Web Module[2] (a software add-on to LabVIEW NXG to build web-based LabVIEW NXG applications).

6.5.1 Model Selection

I chose 10 LabVIEW systems models from 10 different repositories from GitHub. Popoola's work [136] has shown that these 10 repositories contain models with different size and functionality. The selection of these models help to better evaluate the metrics. Table 6.4 gives an overview of these models. Among these 10 models, four are LabVIEW NXG models (Model 3, 5, 7 and 8) and the rest (Model 1, 2, 4, 6, 9 and 10) are NXG web-based models.

6.5.2 Data Collection

The complexity analysis tool takes one model (either with file extension *.giv* for LabVIEW NXG models or with file extension *.gviweb* for LabVIEW NXG Web models) each time as input and returns the numeric values for different complexity metrics as output. For each model, the tool collected 11 values corresponding to the 11 metrics proposed in this study.

[1] `https://www.ni.com/en-us/shop/labview/labview-nxg.html`
[2] `https://www.ni.com/en-us/shop/software/products/labview-nxg-web-module.html`

Table 6.4: An Overview of Models Adopted For Empirical Evaluation

Model	URL for the Model	Model Size	SLOC*
Model 1	https://github.com/rajsite/webvihack/blob/master/webvihack_components/webvihack.gcomp/webvihack_version.gviweb	6.18 KB	66
Model 2	https://github.com/navinsubramani/develop-and-deploy-WebVI/blob/master/source/SimpleWebGame/WebGame.gcomp/Main.gviweb	22.1 KB	132
Model 3	https://github.com/ni/labview-nxg-jenkins-build/blob/master/source/ReusingCodeacrossApplications/CounterLibrary.gcomp/Counter.gvi	22.4 KB	265
Model 4	https://github.com/doczhivago/DownloadUploadAFileWebVI/blob/master/UploadAFile/Main.gcomp/FileDownloadorUpload.gviweb	34.9 KB	378
Model 5	https://github.com/JKISoftware/JKI-State-Machine-NXG/blob/master/source/JKI.Library.JKIStateMachine.gcomp/ParseStateQueue.gvi	44.1 KB	420
Model 6	https://github.com/therinoy/LabVIEW-NXG-BL1.1W-Web-App-Project/blob/master/WebApp.gcomp/Function.gviweb	130 KB	523
Model 7	https://github.com/prestwick/customizing-webvis/blob/master/monitor-test-webvi/subvis.gcomp/DequeMessage.gvi	59.7 KB	642
Model 8	https://raw.githubusercontent.com/ni/webvi-examples/master/CallSystemLinkDataServices/SubVIs.gcomp/WriteExampleTags.gvi	73.4 KB	733
Model 9	https://github.com/eyesonvis/niweek2019-webVI-hands-on/blob/master/LabVIEWNXGcode/TemperatureMonitorWebApp.gcomp/TemperatureMonitor.gviweb	110 KB	987
Model 10	https://github.com/wimtormans/LabVIEWRaspberryPI_IndoorMonitoring/blob/master/WebVI/WebApp.gcomp/IndoorMonitoring.gviweb	134 KB	1549

*SLOC: Source Line of Code.

Table 6.5: Empirical Evaluation Results for 10 LabVIEW Models

Model No.	CyC	Halstead			IFC	Card & Glass			CoC	Front Panel	
		V	D	E		DC	StC	SyC		ES	ID
1	1	3.17	1	3.17	0	1	0	1	0	0.05	0.002
2	3	18.58	2	37.16	0	0	0	0	27	0.41	0.28
3	4	126	52.5	6615	0	5	0	5	53	0.04	0.02
4	4	16.25	2	32.5	8	0	2.5	2.5	15	0.13	0.14
5	5	151.6	54	8186.2	16	1	1	2	57	0.05	0.04
6	1	0	0	0	0	0	0	0	0	0.15	0.95
7	4	120.46	30	3613.8	16	2	4	6	39	0	0
8	1	63.4	6	380.4	48	0.7	9	9.7	5	0	0
9	7	165.44	2.5	413.60	55	0	3.8	3.8	21	0.27	0.43
10	4	207.74	3	623.23	169	0	2.3	2.3	21	0.26	0.29

CyC = Cyclomatic Complexity, V = Volume, D = Difficulty, E = Effort, IFC = Information Flow Complexity, DC = Data Complexity, StC = Structure Complexity, SyC = System Complexity, ES = Element Smallness, ID = Interface Density. All the values in this table are rounded to 2 decimal places.

6.5.3 Results and Discussion

The results of this empirical evaluation are shown in Table 6.5. This section will divide the discussion into three subsections: complexity metrics for block diagrams, complexity metrics for front panels, and the comparison between block diagram complexity and front panel complexity.

Complexity Metrics for Block Diagrams

In the metrics suite, there are five complexity metrics related to block diagrams: Cyclomatic Complexity, Halstead Complexity, Information Flow Complexity, Card & Glass Complexity and Cognition Complexity. For these metrics, lower values always mean lower complexity and higher understandability. The only factor affecting Cyclomatic Complexity is the number of decision blocks in a model. For example, Model 1, Model 6 and Model 8 do not contain any decision blocks, thus their Cyclomatic Complexity value equals to 1. Halstead Complexity examines the number of operators and operands. Model 5 has the highest Effort Value, however, as shown in Table 6.5, other complexity metrics for Model 5 are not always the highest. This reflects the fact that Halstead Complexity only concerns

the number of operators and operands a model contains, but not other factors, such as a model's structure (e.g., a nesting structure and a non-nesting structure have the same Halstead Complexity values as long as they have equal numbers of operators and operands) or implementation details. Because of these weaknesses, several critiques for Cyclomatic Complexity and Halstead Complexity have been proposed, such as [139, 161].

Information Flow Complexity measures the complexity of data transfer between a model and its components (a VI and its SubVI VIs). Card & Glass complexity adopts the same concept of Fan-in and Fan-out as in Information Flow Complexity and adds the number of input and output for analysis. One model can have a high Information Flow Complexity while low Card & Glass Complexity (such as Model 10), or vice versa (such as Model 8).

Cognition Complexity, as a measurement to intuitively understand a piece of graphical code in a model, indicates the model complexity from a human perspective. Unlike the metrics analyzed above, cognition complexity distinguishes "human factors from computational complexity" [175]. In this study, the cognition complexity for LabVIEW models based on the existing study related to the cognition complexity examination for OOP [156] was analyzed. When a model contains complicated blocks (such as loops and event structures), developers need more time to understand its logic and functionality, thus increasing the model's cognition complexity. There are two factors affecting the cognition complexity: the number of blocks and the cognitive weight for each block. An example to show this characterization can be found from Model 9 and Model 10: these two models have different numbers of blocks, but the same cognition complexity value.

Complexity Metrics for Front Panels

My research analyzed two metrics affecting the front panel complexity for a LabVIEW model - Element Smallness and Interface Density. Element Smallness specifies the average element width and height placed on the front panel while Interface Density indicates

Figure 6.6: The Front Panel for Model 2

how empty or clustered a front panel is. A bigger value is expected for Element Smallness because research has shown that bigger elements placed on a user interface are easier to recognize and understand. Model 2 has the highest Element Smallness value which means on average, front panel elements are bigger compared with other front panel elements (see Figure 6.6). Model 6 has the highest interface density value, which means elements placed on the user interface of Model 6 occupy most of the interface space (see Figure 6.7). Previous studies have shown that bigger elements and lower interface density help to decrease the user interface complexity [58, 142].

Block Diagram Complexity V.S. Front Panel Complexity

Block diagram complexity represents the complexity of the functionality executed by the model while front panel complexity specifies the complexity of its user interface in a model. There is not a correlation between the block diagram complexity and front panel complexity: A model could have a very complicated user interface, but not execute any function; or a model could execute very complicated functions, but with a very simple user interface (or no user interface at all). This empirical evaluation based on the proposed metrics suite is consistent with this fact: Model 6 has a very high interface density while all the other metric values are 0 (even Model's 6 Cyclomatic Complexity is 1, however, Based on Equation 5, it means there is no decision blocks in the model). This model was examined manually and the result shows that Model 6 only contains a front panel but no block diagram. However, its Interface Density is close to 1, which means the elements placed on

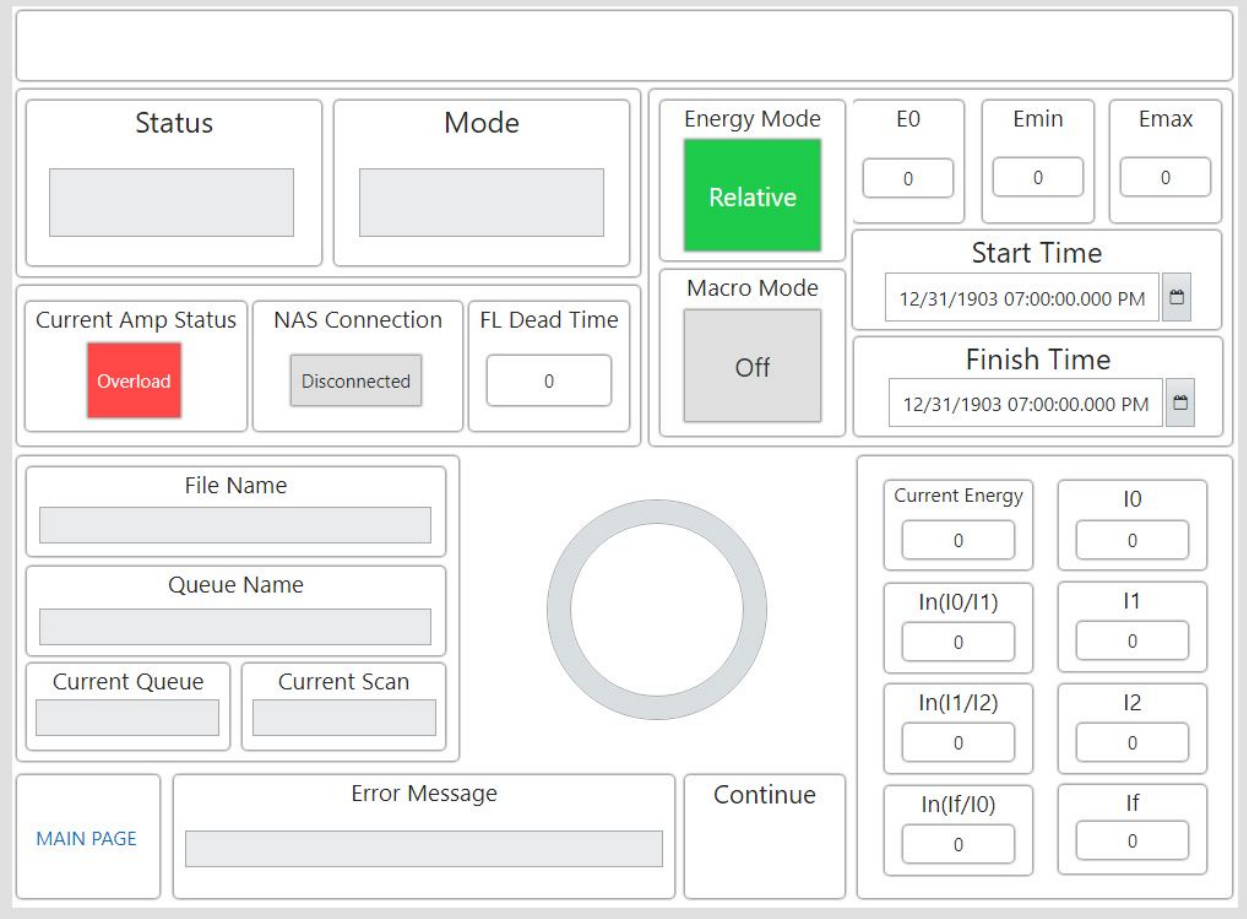


Figure 6.7: The Front Panel for Model 6

its front panel take almost all of the user interface. This is also consistent with the manual examination (shown in Figure 6.7). Model 7 and Model 8 do not have a front panel, thus their front panel complexity is 0. However, they have complicated block diagrams. Therefore, the values of Halstead complexity, Information Flow Complexity and Cognition Complexity are relatively higher compared with other models.

6.6 Threats to Validity

This section discusses three different categories of threats to validity in this study: threats to construct validity, threats to construct validity and threats to construct validity. The mitigation strategies adopted to minimize these threats are also presented.

6.6.1 Threats to Construct Threats

Construct validity demonstrates the method of measurement matches the construct of the research. Achieving construct validity requires the study design to ensure the numeric values truthfully reflect the complexity of LabVIEW models. In general, a higher numerical value indicates a higher degree of model complexity. Therefore, low metrics values are always expected. In the proposed metrics suite, only one metric - Element Smallness, is inconsistent with this expectation. All of the other metrics are in favor of a lower value. However, all of the metrics only give a general idea of model complexities - my research did not identify an exact boundary for each complexity metric. To this end, in the future, an in-person interview with LabVIEW experts is planned to conduct to gain deeper insights into each metric proposed in the suite as an extension study.

6.6.2 Threats to Internal Threats

Internal threats to validity refer to the establishment validity of the study. This dissertation proposed a metrics suite to measure LabVIEW systems model complexities. Thus, in this work, internal threats come from the limitations of the metrics that have been proposed. There are three limitations as internal threats. First, some of the metrics proposed are originally suggested in the context of OOP, but not graphical programming paradigms. To mitigate this threat, several revisions were made to some of the metrics to fit the concept of LabVIEW. Moreover, my research theoretically validated the metrics suite using Weyuker's formal validation and empirically evaluated the metrics suite using LabVIEW models downloaded from GitHub. The results show that the metrics satisfy most of the properties and could characterize LabVIEW systems model complexities from different aspects. Second, the metrics in this study are tool-specific. Only LabVIEW systems models were analyzed in this investigation. However, the concept is transferable and the definitions and equations can be adapted accordingly to match other modeling tools. Finally, there may exist other metrics that could characterize the complexity of LabVIEW

models that were not examined in this dissertation. To mitigate this threat, an extension work is scheduled by collaborating with LabVIEW experts in the future.

6.6.3 Threats to External Validity

External validity refers to the generalizability of the research result. In this study, external threats to validity refer to whether the metrics presented can characterize complexities of real-world LabVIEW models. To mitigate this threat, an empirical evaluation based on 10 open-resource LabVIEW models mined from GitHub was conducted. The evaluation result shows that the metrics suite can identify model complexity from various facets. Another external threat is the limited test cases in the examination. Only 10 models were analyzed in this study. However, the interpretation for the randomly chosen models with various size and different functions holds for other LabVIEW systems models.

6.7 Conclusion

The measurement of software complexity is a persistent problem in software engineering. Compared with a great many of the complexity analyses in the area of OOP, few studies have been conducted within the context of graphical programming languages. To fill this gap, this study defined a complexity metrics suite for LabVIEW systems models and implemented a proof-to-concept tool to automatically retrieve metric values when given a LabVIEW systems model. The metrics in the suite are based on existing metrics in the software engineering discipline. My research also revised several metrics to fit them into LabVIEW.

To characterize model complexity from different aspects, the metrics suite includes six metric categories: Cyclomatic Complexity focuses on understanding the effect of decision blocks on LabVIEW models; Halstead Complexity emphasizes the impact of model sizes from the number of operators and the number of data processing blocks. Information Flow Complexity and Card & Glass Complexity concentrate on the data transfer complexity between a model and its components. Cognition Complexity examines the

model complexity from a human's perspective based on the blocks a model contains. Front Panel Complexity tries to understand the relationship between the complexity and the elements placed on the front panel. To validate the metrics, Weyuker's theoretical validation was adopted. Furthermore, this dissertation empirically evaluated the metrics based on 10 LabVIEW models from GitHub. The validation and evaluation results show that the metrics in the suite are feasible and suitable for the characterization of LabVIEW systems model complexities from different perspectives.

CHAPTER 7

FUTURE WORK

This chapter outlines research directions as well as potential approaches that will be investigated as future work. First, motivated by the research discoveries from the empirical survey (Chapter 5), a deeper understanding of bad smells from a systems engineer's perspective is of great interest. To this end, an interview-based empirical study with systems engineers from different domains is planned. Second, previous empirical study also revealed that there exist several bad smells that systems engineers are concerned about. Although this dissertation analyzed the most prominent (Complicated LabVIEW Systems Models) in Chapter 6, other bad smells (such as Duplicated Code and Dead Code), also need research attention. Therefore, part of future work also includes the removal of duplicated code and dead code in systems models. Genetic algorithms are proposed to be applied in a future investigation. Finally, my future works include how to apply statistical analysis to understand the relationship between model metrics and model smells in systems models.

7.1 An Interview-based Empirical Study on Bad Smells in Systems Models

This section describes an interview-based empirical study as a future work. Specifically, the content will focus on two aspects: the research questions for this study and potential approaches to conduct this study.

7.1.1 Research Questions

The previous work in this dissertation suggests that systems engineers have different perceptions about bad smells in systems models. For example, Section 5.4.2 shows that systems engineers are much more concerned with complicated models and duplicated

code in LabVIEW, compared to other bad smells such as Build Array Inside a Loop and Fix-length Array Resizing. Motivated by this observation, in the future, an in-person interview to gain deeper insights into the understanding of bad smells in systems models from a systems engineer's perspective is necessary. This work will investigate the following research questions:

- **Research Question 1**: Does the developer's working domain/development tool affect their perception of bad smells in systems models?
- **Research Question 2**: Why do systems engineers agree/disagree that certain poor designs are related to bad smells in systems models?
- **Research Question 3**: *Why* do systems engineers introduce bad smells to their software models during development practice?
- **Research Question 4**: *When* do systems engineers introduce bad smells to their software models during development practice?

7.1.2 Potential Approach

To answer these research questions, a future plan is to conduct an empirical study in the future. To better understand systems engineers' perceptions, I propose to perform in-person interviews with systems engineers from different domains. Table 7.1 covers some potential interview questions in the proposed future study.

To recruit participants from different domains, future investigations will conduct the interviews with systems engineers from different companies, such as Boeing[1] (aerospace) and PACCAR[2] (automotive). To ensure the validity of the interview questions, a pilot study with experienced systems engineers from different domains will be performed. For open-ended questions, an Open Coding [68] approach will be adopted to analyze participant responses. As an extension study of the previous examinations discussed in Chapter

[1] `https://www.boeing.com/`
[2] `https://www.paccar.com/`

Table 7.1: Potential Interview Questions

#	Question Text	Answer
Q1	What is your working domain?	
Q2	What tool do you use in your modeling practice?	
Q3	Do you agree that *Complicated Systems Models* is a bad smell to systems models?	[Yes, No]
Q3'	Please explain why you agree or disagree.	
Q4	Do you agree that *Duplicated Code* is a bad smell to systems models?	[Yes, No]
Q4'	Please explain why you agree or disagree.	
Q5	Do you agree that *Unorganized Wired* is a bad smell to systems models?	[Yes, No]
Q5'	Please explain why you agree or disagree.	
Q6	Do you agree that *Dead Code* is a bad smell to systems models?	[Yes, No]
Q6'	Please explain why you agree or disagree.	
Q7	Do you agree that *Redundant Operations* is a bad smell to systems models?	[Yes, No]
Q7'	Please explain why you agree or disagree.	
Q8	Do you agree that *Excessive Data Copies* is a bad smell to systems models?	[Yes, No]
Q8'	Please explain why you agree or disagree.	
Q9	Do you agree that *Too many controls* is a bad smell to systems models?	[Yes, No]
Q9'	Please explain why you agree or disagree.	
Q10	Do you agree that *Unconnected front panel elements* is a bad smell to systems models?	[Yes, No]
Q10'	Please explain why you agree or disagree.	
Q11	What reasons do you think system engineers introduce bad smells to their software models during development practice?	
Q12	When do you think system engineers introduce bad smells to their software models during development practice?	

5, the contribution of this proposed study is to gain a deeper understanding of bad smells in systems models from a systems engineer's perspective.

7.2 System Model Refactoring Based on Examples

The previous work also suggests that Duplicated Code is a prominent bad smell for systems engineers. Duplicated Code detection and removal has been a popular research topic over the past two decades. With the fast development of ML and AI techniques, more investigations are focusing on bad smell detection and removal based on machine learning/deep learning techniques [101] [158]. However, all of these examinations are within the context of OOP. In the future, the refactoring of systems models to remove bad smells using genetic algorithms is planned to conduct as an extension research of this dissertation.

7.2.1 Background: Genetic Programming

Genetic programming was first systematically described and studied by John Koza [92]. It harnesses the mechanisms of natural evolution, including mutation, recombination, and natural selection, to automatically synthesize computer programs. It has been applied to a wide range of problems spanning several areas of science, engineering, and arts, in many cases equaling or exceeding human performance [4].

Genetic programming is a specialized version of genetic algorithms, which is outlined in Figure 7.1. It starts with random instances of answers that users are looking for, such as a bit string, or many parameters that users are trying to optimize. Users apply these random instances to successive circles of assessment (i.e., execute each solution and evaluate its fitness), derivation and update. A user determines how good each solution is and selects the most optimal one. The solutions are varied randomly, and users assess the new solutions repeatedly. This process repeats several times until a "good enough" solution is found, or the maximum iteration number is reached.

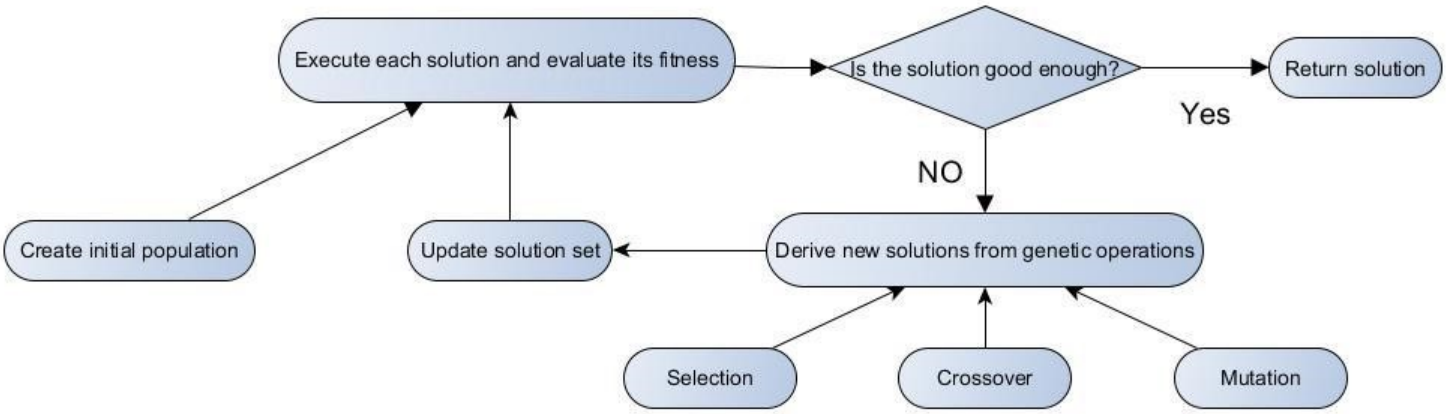


Figure 7.1: Overview of the Flowchart of Genetic Algorithms

The definition of the genetic algorithm described above shows that the determination of the fitness of an individual plays an important role in the whole process. In a genetic algorithm, the function assesses this fitness by providing a number called the fitness score. How to formulate a good fitness function is one of the most challenging steps in a genetic algorithm [11]. As a future work, different fitness functions will be applied and their results will be compared, thus obtaining the best one for the refactoring of system models.

7.2.2 Research Questions

To overcome some of the issues in semi-automatic support of model refactoring, some researchers use an example-based model transformation process. The essence of this approach is to derive refactoring rules from an initial prototypical set of interrelated source models and target models [16]. To map this concept to the future study, a source model will exhibit a bad smell and a target model will be the same model after removing the bad smell. Instead of defining the refactoring rules manually, the primary advantage of this approach is the refactoring rules are generated automatically.

The idea of refactoring by examples has been applied in several domains in the context of model-driven engineering. Ghannem et al. [66] [65] adopted genetic programming to refactor UML class diagrams and their result shows that example-based refactor-

ing may find more refactoring opportunities compared with human inspection. Based on Ghannem's work, Mokaddem et al. [120] revised the fitness function and accomplished the refactoring tasks based on examples provided by a user.

In the future, I plan to extend the current research by refactoring LabVIEW systems models based on examples. This work will investigate the following research questions:

- **Research Question 1**: Does the genetic algorithm refactor the LabVIEW systems models correctly?
- **Research Question 2**: What is the quality of refactoring rules obtained from genetic algorithms?
- **Research Question 3**: Can an example-based refactoring approach find more refactoring opportunities than a hard-coded rules-based refactoring approach for systems models?

7.2.3 Potential Approach

The potential refactoring framework consists of five steps. Figure 7.2 is an overview of the refactoring framework in the future study.

1. Initialize population. As shown in Figure 7.2, the first step of a genetic algorithm is to initialize the population. In a genetic algorithm, a population of candidate solutions (called individuals, or creatures) for an optimization problem evolves toward better solutions. The creation of an initial population will be based on existing models from the LabVIEW online discussion forum and open-source GitHub LabVIEW projects.

2. Provide refactoring examples. The second step is to provide genetic algorithm examples. These examples will be collected from existing models. All the LabVIEW model examples will be paired – in each pair, one is the model before refactoring and the other one is after refactoring.

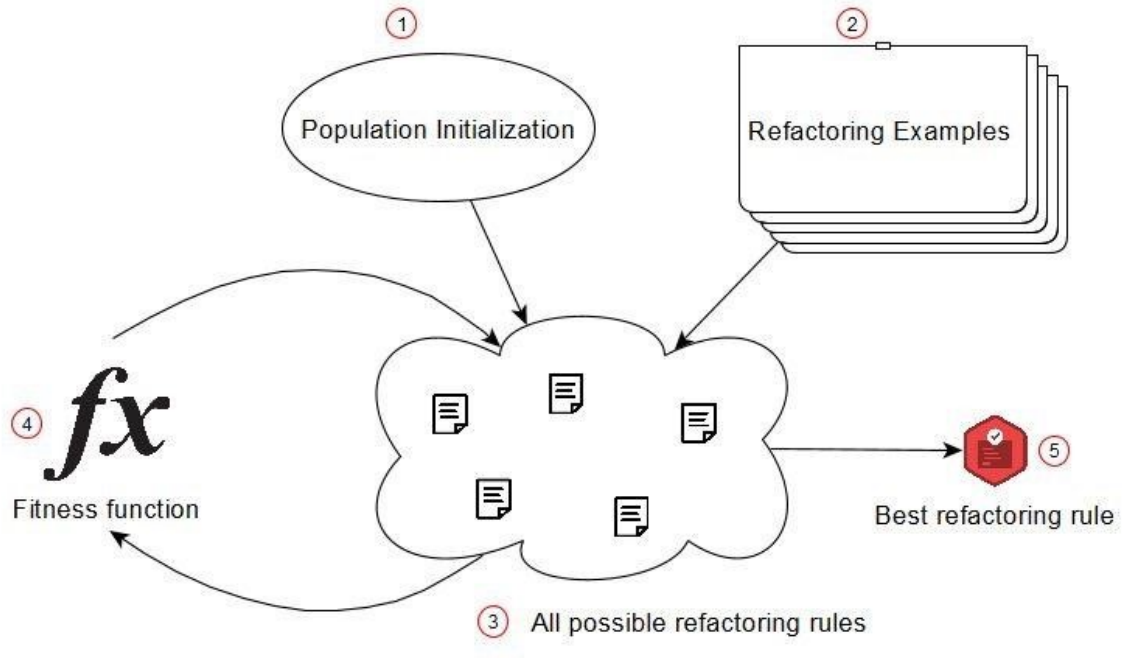


Figure 7.2: Refactoring As a Learning Process

3. Generate all possible refactoring rules. As Figure 7.2 shows, there are several operations in a genetic algorithm. A future work will apply these genetic operations to each population to obtain a space of all possible refactoring opportunities.

4. Assess each rule by fitness function. The next step is to define a fitness function and then evaluate each rule by a given fitness function. A fitness score is used to reflect how well a refactoring rule could fit the example pair.

5. Select best refactoring rule. The last step is to choose the best refactoring rule based on fitness score.

In the future, several genetic algorithms to model refactoring will be adopted, such as NSGA-II [66] and an interactive genetic algorithm [65]. The future investigation will compare the results of different genetic algorithms and provide end-users (especially non-experts) the best algorithm. To this end, the future work will adopt Evolutionary Computation Library in Java (ECJ) [111], a Java-based evolutionary computation research system to implement genetic algorithms.

The contribution of this work is the automation of model refactoring rules by inferring the rules from examples. Researchers have shown that automatic learning could handle more complex rules compared with fixed refactoring rules [66].

7.3 Exploring the Relationship between Model Metrics and Model Smells

Munro's work [124] has shown that software metrics are closely related to bad smells in Java projects. Inspired by this work, understanding the relationship between model metrics and model smells in systems models is of great interest as a future work. This section first briefly introduces the background of the proposed approach. Following this is the description of the proposed approach to conduct this investigation.

7.3.1 Background: Regression Analysis

Regression Analysis is widely used to find the relationship between different variables. Regression is a data mining technique for predicting continuous values. It estimates the conditional expectation of a dependent variable when given independent variables. There are several regression types, such as linear regression, logistic regression and non-linear regression. In the future, linear regression [41], logistic regression [43] and decision trees will be adopted to explore the relationship between model metrics and bad smells to predict bad smells in systems models.

Linear Regression

Linear regression [41] is widely used in modeling the relationship between independent variables and dependent variables. There are two types of linear regression: simple linear regression (only one independent variable) and multiple linear regression (more than one independent variable). The multiple linear regression is defined as:

$$y = \beta_0 \times 1 + \beta_1 \times x_1 + \beta_2 \times x_2 + ... + \beta_n \times x_n + \varepsilon \tag{7.1}$$

where y is the dependent variable and x_i are independent variables. β_i is coefficient and ε is an error term, which is used to formulate a linear regression model when the model cannot fully represent the actual relationship between dependent and independent variables.

Logistic Regression

Developed by David Cox [43], logistic regression uses a logistic function to model the relationship between dependent variables and independent variables. Logistic regression is used when the dependent variable is categorical, such as pass/fail, win/lose, is/is not, or healthy/sick. The logistic regression model takes the form:

$$y = \frac{exp(\beta_0 \times 1 + \beta_1 \times x_1 + \beta_2 \times x_2 + ... + \beta_n \times x_n)}{1 + exp(\beta_0 \times 1 + \beta_1 \times x_1 + \beta_2 \times x_2 + ... + \beta_n \times x_n)} \tag{7.2}$$

Linear and logistic regression are the most basic techniques in regression analyses. The core difference between these two is that linear regression is used when the dependent variable is continuous while the logistic regression is used when the dependent variable is binary. Compared with linear regression, logistic regression is more widely used in various fields for prediction, such as disease prediction [44] and machine learning [154].

Decision Tree

A decision tree is a tree-like structure where internal nodes represent a test. The answer for the test is binary (only "yes" or "no"). Each branch is the result of the test and each leaf is a decision after computing all of the tests. Each path from root to leaf is a classification rule. Decision tree learning uses a decision tree to go from observations to conclusions.

Linear regression, logistic regression and decision trees can be used as prediction approaches. The reason why the future work plans to apply these three approaches is because it is hard to know which technique will be better simply based on the input.

7.3.2 Research Questions

Metrics are important for software because they are established and objective measurements to assess the software quality. Metrics are widely used and adopted in all the phases in the software development cycle. The future work will investigate the relationship between LabVIEW model metrics and model smells through regression analyses. Specifically, the following questions will be investigated:

- What metrics in LabVIEW models are related to model smells?
- Can the proposed approach predict the existence of bad smells in LabVIEW models by analyzing model metrics?

7.3.3 Potential Approach

To answer the first research question, the future work plans to apply regression analyses to find the relationship between model metrics and bad smells in LabVIEW models. To answer the second research question, a ML approach to predict bad smells in LabVIEW models will be adopted. The future work consists of four steps:

- **Step 1: Model Collection**. Collect LabVIEW models with code smells. This is the first step in the proposed work. The data will come from two sources: 1) LabVIEW experts; 2) In the future work discussed earlier in Section 7.2, the proposed approach will investigate LabVIEW model refactoring based on examples. The model repository used in this work will also be adopted in this study.
- **Step 2: Metrics Calculation**. The purpose of this step is to calculate the metrics in LabVIEW models that contain bad smells. LabVIEW offers an analysis tool called VI Analyzer Toolkit[3]. This tool offers testing suites and static VI analyses. The metrics included in VI Analyzer are listed in Table 7.2. The future work will adopt all the metrics analyzed in VI Analyzer except one metric: Diagram Size. This

[3] https://www.ni.com/en-us/support/downloads/software-products/download.labview-vi-analyzer-toolkit.html#346221

Table 7.2: Metrics Provided by the VI Analyzer Toolkit

Metrics	Description
Connector Inputs/Outputs	The number of inputs and outputs connected to the pane
Controls and Indicators	The number of controls and indicators
Diagram Count	The number of block diagrams in a VI
Diagram Size	The width and height of a block diagram
Node Count	The number of nodes in the block diagram
Property Reads and Writes	The number of property reads and property writes by the block diagram
Shared Library Calls	The number of Call Library Function Nodes on the block diagram
Structure Count	The number of structures on the block diagram
Wire Sources	The number of wire sources on the block diagram

is because diagram size only represents the block diagram physical size occupied on the screen.

- **Step 3: Relationship Exploration**. This step will help to answer the first research question. After the collection of model metrics, different regression approaches will be used discussed in Section 7.3.1 to explore the relationship between model smells and model metrics in LabVIEW systems models.

- **Step 4: Bad Smell Prediction**. The last step of this work is to predict the existence of bad smells in LabVIEW models using model metrics as predictors. This process first train a prediction model. A training database will be established. In the database, a model will be labelled as a positive sample if it contains at least one bad smell. Otherwise, a model will be labelled as a negative sample. In this step, the proposed work will apply 10-fold cross-validation for performance testing.

The expected contribution of this future work is to employ statistical (e.g., linear regression and logistic regression) and machine learning (e.g., decision tree) methods to assess the applicability of existing LabVIEW model metrics to predict the existence of bad smells in the systems models.

CHAPTER 8

CONCLUSION

Systems modeling plays an important role in Model-based Systems Engineering. However, systems engineers (particularly, inexperienced engineers) may lack professional software engineering expertise, thus causing a knowledge gap between systems engineering and software engineering that negatively affects the quality of software artifacts (i.e., systems models) adopted in the system.

Chapter 1 introduced basic concepts of systems engineering, software engineering and the connection between systems engineering and software engineering. Following this is the presentation of some key challenges with software engineering within the context of systems engineering and the scope of this dissertation. Chapter 2 provided a detailed explanation of some key concepts investigated in the dissertation (e.g., block diagram, front panel, and bad smell) and tools used in this study.

Chapter 3 discussed the preliminary work on the examination of forum posts from systems engineers with the help of supervised machine learning techniques and unsupervised machine learning techniques. The results served as a motivation to conduct the rest of the investigations discussed in Chapter 4 to Chapter 6. The examination of forum posts shows that systems engineers are deeply concerned with bad smells in systems models during development. Based on this observation, Chapter 4 presented the research work on the summarization of bad smells from forum posts using BESMER - a bad smell summarization process based on mining end-user discussion forum posts. Following this summarization, in Chapter 5, my contribution continued previous research by sending an online survey to systems engineers to ask their perception on the bad smells that have been summarized. This empirical evaluation enables me to conclude that systems engineers do not

agree that all the bad smells in systems models are problematic, thus enabling the investigation to focus on the most problematic and prominent bad smells for systems engineers. Chapter 6 proposed a metrics suite to identify the most problematic bad smell shown in the empirical study - complicated models. This study theoretically validated the metrics suite through Weyuker's properties and empirically evaluated this metrics suite through LabVIEW models downloaded from GitHub. The results show that the proposed metrics suite can characterize the complexity of LabVIEW systems models from different aspects.

Chapter 7 outlined several future works based on investigations discussed in this dissertation. These future works include a deeper understanding of bad smells in systems models through an in-person interview with systems engineers, bad smell refactoring through examples and the exploration of the relationship between model metrics and model smells.

This research aimed to improve the quality of software components by the summarization, evaluation and identification of bad smells in systems models within the context of Model-based Systems Engineering. This dissertation contributes to bridging the knowledge gap between software engineering and systems engineering.

REFERENCES

[1] Mesfin Abebe and Cheol-Jung Yoo. Trends, opportunities and challenges of software refactoring: A systematic literature review. *International Journal of Software Engineering and its Applications*, 8(6):299–318, 2014.

[2] Amjad AbuHassan and Mohammad Alshayeb. A metrics suite for UML model stability. *Software & Systems Modeling*, 18(1):557–583, 2019.

[3] Adewole Adewumi, Sanjay Misra, and Robertas Damaševičius. A complexity metrics suite for cascading style sheets. *Computers*, 8(3):54:1–54:18, 2019.

[4] Michael Affenzeller, Stefan Wagner, Stephan Winkler, and Andreas Beham. *Genetic Algorithms and Genetic Programming: Modern Concepts and Practical Applications*. CRC Press, 2009.

[5] KK Aggarwal, Yogesh Singh, Arvinder Kaur, and Ruchika Malhotra. Software design metrics for object-oriented software. *Journal of Object Technology*, 6(1):121–138, 2007.

[6] Efthimia Aivaloglou and Felienne Hermans. How kids code and how we know: An exploratory study on the Scratch repository. In *Proceedings of the 2016 Conference on International Computing Education Research*, pages 53–61, 2016.

[7] Mete Akcaoglu and Eunbae Lee. Increasing social presence in online learning through small group discussions. *The International Review of Research in Open and Distributed Learning*, 17(3):1–17, 2016.

[8] Altvater Alexandra. What is SDLC? understand the software development life cycle. Online Resource: `https://stackify.com/what-is-sdlc/`, May 2020.

[9] Berna Altınel and Murat Can Ganiz. Semantic text classification: A survey of past and recent advances. *Information Processing & Management*, 54(6):1129–1153, 2018.

[10] Sergio A. Alvarez. An exact analytical relation among recall, precision, and classification accuracy in information retrieval. Technical Report BCCS-02-01. `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.459.9750&rep=rep1&type=pdf`, Boston College, 2002.

[11] Boukhdhir Amal, Marouane Kessentini, Slim Bechikh, Josselin Dea, and Lamjed Ben Said. On the use of machine learning and search-based software engineering for ill-defined fitness function: A case study on software refactoring. In *International Symposium on Search Based Software Engineering*, pages 31–45, 2014.

[12] Thorsten Arendt, Matthias Burhenne, and Gabriele Taentzer. Defining and checking model smells: A quality assurance task for models based on the Eclipse Modeling Framework. In *Proceedings of the 9th Belgium-Netherlands Software Evolution Workshop*, pages 1–5, 2010.

[13] Thorsten Arendt and Gabriele Taentzer. UML model smells and model refactorings in early software development phases. Technical Report `http://spes2020.informatik.tu-muenchen.de/results/AT-AP4-D-AT-4_1_c.pdf`, Philipps University of Marburg, 2010.

[14] Francis R. Bach, Gert R.G. Lanckriet, and Michael I. Jordan. Multiple kernel learning, conic duality, and the smo algorithm. In *Proceedings of the 21st International Conference on Machine Learning*, pages 6:1–6:8, 2004.

[15] Brenda S. Baker. A program for identifying duplicated code. *Computing Science and Statistics*, 24:49–57, 1993.

[16] Zoltán Balogh and Dániel Varró. Model transformation by example using inductive logic programming. *Software & Systems Modeling*, 8(3):347–364, 2009.

[17] David Bau, Jeff Gray, Caitlin Kelleher, Josh Sheldon, and Franklyn Turbak. Learnable programming: Blocks and beyond. *Communications of the ACM*, 60(6):72–80, 2017.

[18] Thomas Bayes. An essay towards solving a problem in the doctrine of chances. *Biometrika*, 45(3-4):296–315, 1958.

[19] Woubshet Behutiye, Pertti Seppänen, Pilar Rodríguez, and Markku Oivo. Documentation of quality requirements in agile software development. In *Proceedings of the Evaluation and Assessment in Software Engineering*, pages 250–259. 2020.

[20] Aries Beltran. *Getting Started with PhantomJS*. Packt Publishing Ltd, 2013.

[21] David M. Blei, Andrew Ng, and Michael I. Jordan. Latent dirichlet allocation. *Journal of Machine Learning Research*, 3:993–1022, 2003.

[22] MathWorks Automotive Advisory Board. Control algorithm modeling guidelines using MATLAB, Simulink, and Stateflow (version 2.0). 2007.

[23] Barry W. Boehm. Verifying and validating software requirements and design specifications. *IEEE software*, 1(1):75–88, 1984.

[24] Pierre Bourque, Robert Dupuis, Alain Abran, James W Moore, and Leonard Tripp. The guide to the software engineering body of knowledge. *IEEE software*, 16(6):35–44, 1999.

[25] Marco Brambilla, Jordi Cabot, and Manuel Wimmer. *Model-Driven Software Engineering In Practice*. Morgan & Claypool Publishers, 2017.

[26] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[27] Frederick Brooks. No silver bullet - essence and accidents of software engineering. *IEEE Computer*, 20(4):10–19, 1987.

[28] Lynell Burmark. *Visual Literacy: Learn To See, See To Learn*. Education Resources Information Center, 2002.

[29] Margaret M. Burnett and David W. McIntyre. Visual programming. *Computer*, 28:14–28, 1995.

[30] Erik Cambria, Dipankar Das, Sivaji Bandyopadhyay, and Antonio Feraco. Affective computing and sentiment analysis. In Erik Cambria, Dipankar Das, Sivaji Bandyopadhyay, and Antonio Feraco, editors, *A Practical Guide to Sentiment Analysis*, chapter 1, pages 1–10. Springer, 2017.

[31] Tiago Carçao. Measuring and visualizing energy consumption within software code. In *2014 IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 181–182, 2014.

[32] David N. Card and Robert L. Glass. *Measuring Software Design Quality*. Prentice-Hall, 1990.

[33] Cagatay Catal. Performance evaluation metrics for software fault prediction studies. *Acta Polytechnica Hungarica*, 9(4):193–206, 2012.

[34] Christopher Chambers and Christopher Scaffidi. Smell-driven performance analysis for end-user programmers. In *2013 IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 159–166, 2013.

[35] Christopher Chambers and Christopher Scaffidi. Impact and utility of smell-driven performance tuning for end-user programmers. *Journal of Visual Languages & Computing*, 28:176–194, 2015.

[36] Christopher Chambers and Christopher Scaffidi. Utility and accuracy of smell-driven performance analysis for end-user programmers. *Journal of Visual Languages & Computing*, 26:1–14, 2015.

[37] Alexander Chatzigeorgiou and Anastasios Manakos. Investigating the evolution of bad smells in object-oriented code. In *2010 7th International Conference on the Quality of Information and Communications Technology*, pages 106–115, 2010.

[38] Min Chen and Amos Golan. What may visualization processes optimize? *IEEE Transactions on Visualization and Computer Graphics*, 22(12):2619–2632, 2015.

[39] Shyam R. Chidamber and Chris F. Kemerer. Towards a metrics suite for object-oriented design. In *Proceedings of the 6th Conference Proceedings on Object-Oriented Programming Systems, Languages, and Applications*, pages 197–211, 1991.

[40] Shyam R. Chidamber and Chris F. Kemerer. A metrics suite for object-oriented design. *IEEE Transactions on Software Engineering*, 20(6):476–493, 1994.

[41] Gregory C. Chow. Tests of equality between sets of coefficients in two linear regressions. *Econometrica: Journal of the Econometric Society*, 28(3):591–605, 1960.

[42] Paul C. Clements. A survey of architecture description languages. In *Proceedings of the 8th International Workshop on Software Specification and Design*, pages 16–25, 1996.

[43] David R. Cox. The regression analysis of binary sequences. *Journal of the Royal Statistical Society: Series B (Methodological)*, 20(2):215–232, 1958.

[44] Johanna A.A.G. Damen, Lotty Hooft, Ewoud Schuit, Thomas P.A. Debray, Gary S. Collins, Ioanna Tzoulaki, Camille M. Lassale, George C.M. Siontis, Virginia Chiocchia, Corran Roberts, Michael Schlüssel, Stephen Gerry, James A. Black, Pauline Heus, Yvonne T. van der Schouw, Linda M. Peelen, and Karel G. Moons. Prediction models for cardiovascular disease risk in the general population: Systematic review. *British Medical Journal*, 353(i2416):1–11, 2016.

[45] Sayamindu Dasgupta and Benjamin Mako Hill. Scratch community blocks: Supporting children as data scientists. In *Proceedings of the 2017 Conference on Human Factors in Computing Systems*, pages 3620–3631, 2017.

[46] Eyal de Lara, Willy Zwaenepoel, and Dan S. Wallach. Puppeteer: Component-based adaptation for mobile computing. In *Proceedings of the Third Usenix Symposium on Internet Technologies and Systems*, pages 159–170, 2001.

[47] Rafael Maiani de Mello, Roberto Oliveira, and Alessandro Garcia. On the influence of human factors for identifying code smells: A multi-trial empirical study. In *2017*

ACM/IEEE International Symposium on Empirical Software Engineering and Measurement, pages 68–77, 2017.

[48] Saumya K. Debray, William Evans, Robert Muth, and Bjorn De Sutter. Compiler techniques for code compaction. *ACM Transactions on Programming languages and Systems*, 22(2):378–415, 2000.

[49] Mohit Deshpande. Classification with support vector machines. `https://pythonmachinelearning.pro/classification-with-support-vector-machines`.

[50] Mohit Deshpande. Classification with support vector machines. Online Resource: `https://pythonmachinelearning.pro/classification-with-support-vector-machines/`, 2020.

[51] Ponnurangam Dhavachelvan, Narayanasamy Saravanan, and Satheesh Kumar. Validation of complexity metrics of agent-based systems using Weyuker's axioms. In *2008 International Conference on Information Technology*, pages 248–251, 2008.

[52] Brian Douglas. Systems engineering, part 1: What is systems engineering? `https://www.mathworks.com/videos/systems-engineering-part-1-what-is-systems-engineering--1602837702185.html`.

[53] Hans Eriksson and Magnus Penker. *Business Modeling with UML: Business Patterns At Work*. John-Wiley & Sons, 2000.

[54] Jeannie Falcon. Facilitating modeling and simulation of complex systems through interoperable software. In *Keynote address at ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems*, 2017.

[55] Norman Fenton. Software measurement: A necessary scientific basis. *IEEE Transactions on Software Engineering*, 20(3):199–206, 1994.

[56] Norman Fenton and James Bieman. *Software Metrics: A Rigorous and Practical Approach*. CRC press, 2014.

[57] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley Professional, 2018.

[58] Fongling Fu, Shao-Yuan Chiu, and Chiu Hung Su. Measuring the screen complexity of web pages. In *Symposium on Human Interface and the Management of Information*, pages 720–729. Springer, 2007.

[59] Wilbert O. Galitz. *The Essential Guide to User Interface Design: An Introduction to GUI Design Principles and Techniques*. John Wiley & Sons, Inc., 2007.

[60] Henry Laurence Gantt. *Work, Wages, and Profits*. Engineering Magazine Company, 1913.

[61] Joshua Garcia, Daniel Popescu, George Edwards, and Nenad Medvidovic. Identifying architectural bad smells. In *2009 13th European Conference on Software Maintenance and Reengineering*, pages 255–258. IEEE, 2009.

[62] Vahid Garousi, Gorkem Giray, and Eray Tuzun. Understanding the knowledge gaps of software engineers: An empirical analysis based on SWEBOK. *ACM Transactions on Computing Education*, 20(1):1–33, 2019.

[63] Thomas Gerlitz, Hansen Norman, Christian Dernehl, and Stefan Kowalewski. Artshop: A continuous integration and quality assessment framework for model-based software artifacts. In *Dagstuhl-Workshop: Model-Based Development of Embedded Systems*, pages 13–22, 2016.

[64] Thomas Gerlitz, Quang Minh Tran, and Dziobek Christian. Detection and handling of model smells for MATLAB/Simulink models. In *International Workshop on Modeling in Automotive System and Software Engineering*, pages 13–22, 2015.

[65] Adnane Ghannem, Ghizlane El Boussaidi, and Marouane Kessentini. Model refactoring using interactive genetic algorithm. In *International Symposium on Search Based Software Engineering*, pages 96–110, 2013.

[66] Adnane Ghannem, Marouane Kessentini, Mohammad Salah Hamdi, and Ghizlane El Boussaidi. Model refactoring by example: A multi-objective search based software engineering approach. *Journal of Software: Evolution and Process*, 30(4):e1916:1–20, 2018.

[67] Yossi Gil and Gal Lalouche. On the correlation between size and metric validity. *Empirical Software Engineering*, 22(5):2585–2611, 2017.

[68] Barney G. Glaser, Anselm L. Strauss, and Elizabeth Strutzel. The discovery of grounded theory; strategies for qualitative research. *Nursing Research*, 17(4):364, 1968.

[69] Maurice Howard Halstead. *Elements of Software Science*. Elsevier North-Holland, 1977.

[70] Thirunavukkarasu Hariprasad, G. Vidhyagaran, K. Seenu, and Chandrasegar Thirumalai. Software complexity analysis using Halstead metrics. In *2017 International Conference on Trends in Electronics and Informatics*, pages 1109–1113, 2017.

[71] Geoffrey Hecht, Naouel Moha, and Romain Rouvoy. An empirical study of the performance impacts of Android code smells. In *Proceedings of 2016 International Conference on Mobile Software Engineering and Systems*, pages 59–69, 2016.

[72] Dirk Heerwegh and Geert Loosveldt. An experimental study on the effects of personalization, survey length statements, progress indicators, and survey sponsor logos in web surveys. *Journal of Official Statistics*, 22(2):191, 2006.

[73] Sallie Henry and Dennis Kafura. Software structure metrics based on information flow. *IEEE Transactions on Software Engineering*, 7(5):510–518, 1981.

[74] Felienne Hermans and Efthimia Aivaloglou. Teaching software engineering principles to k-12 students: A MOOC on Scratch. In *Proceedings of the 39th International Conference on Software Engineering: Software Engineering Education and Training Track*, pages 13–22, 2017.

[75] Felienne Hermans, Kathryn T. Stolee, and David Hoepelman. Smells in block-based programming languages. In *2016 IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 68–72, 2016.

[76] Mamoun Hirzalla, Jane Cleland-Huang, and Ali Arsanjani. A metrics suite for evaluating flexibility and complexity in service oriented architectures. In *International Conference on Service-Oriented Computing*, pages 41–52, 2008.

[77] Martin Hofmann. Support vector machines-kernels and the kernel trick. `https://cogsys.uni-bamberg.de/teaching/ss06/hs_svm/slides/SVM_Seminarbericht_Hofmann.pdf`, 2006.

[78] Thomas Hofmann. Probabilistic latent semantic indexing. In *Proceedings of the 22nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, pages 50–57, 1999.

[79] Mário Hozano, Alessandro Garcia, Baldoino Fonseca, and Evandro Costa. Are you smelling it? Investigating how similar developers detect code smells. *Information and Software Technology*, 93:130–146, 2018.

[80] Steve Huntsman. Generalizing cyclomatic complexity via path homology. *arXiv Preprint: arXiv:2003.00944*, 2020.

[81] Marnie L. Hutcheson. *Software Testing Fundamentals: Methods and Metrics*. John Wiley & Sons, 2003.

[82] Vinay K. Ingle and John G. Proakis. *Digital Signal Processing Using MATLAB: A Problem Solving Companion*. Cengage Learning, 2016.

[83] Thorsten Joachims. Training linear svms in linear time. In *Proceedings of the 12th ACM International Conference on Knowledge Discovery and Data Mining*, pages 217–226, 2006.

[84] Stephen C Johnson. Hierarchical clustering schemes. *Psychometrika*, 32(3):241–254, 1967.

[85] Karen Sparck Jones. A statistical interpretation of term specificity and its application in retrieval. *Journal of Documentation*, 28(1):11–21, 1972.

[86] Robert Jorczak and Danielle N. Dupuis. Differences in classroom versus online exam performance due to asynchronous discussion. *Journal of Asynchronous Learning Networks*, 18(2):1–9, 2014.

[87] Mohamad Kassab, Colin Neill, and Phillip Laplante. State of practice in requirements engineering: Contemporary data. *Innovations in Systems and Software Engineering*, 10(4):235–241, 2014.

[88] Ashraf M. Kibriya, Eibe Frank, Bernhard Pfahringer, and Geoffrey Holmes. Multinomial naive bayes for text categorization revisited. In *Australasian Joint Conference on Artificial Intelligence*, pages 488–499, 2004.

[89] Miryung Kim, Lawrence Bergman, Tessa Lau, and David Notkin. An ethnographic study of copy and paste programming practices in OOPL. In *Proceedings of 2004 International Symposium on Empirical Software Engineering*, pages 83–92, 2004.

[90] Jeffrey Kodosky. LabVIEW. *Proceedings of the ACM on Programming Languages*, 4(78):1–54, 2020.

[91] Ron Kohavi. A study of cross-validation and bootstrap for accuracy estimation and model selection. In *Proceedings of the 14th International Joint Conferences on Artificial Intelligence*, volume 2, pages 1137–1143, 1995.

[92] John R Koza. Hierarchical genetic algorithms operating on populations of computer programs. In *Proceedings of International Joint Conferences on Artificial Intelligence*, volume 89, pages 768–774, 1989.

[93] Santosh Kumar, Xiaoying Gao, Ian Welch, and Masood Mansoori. A machine learning based web spam filtering approach. In *Proceedings of 30th International Conference on Advanced Information Networking and Applications*, pages 973–980, 2016.

[94] Pierre Simon Laplace. *A Philosophical Essay On Probabilities*. Wiley, 1902.

[95] Philip A Laplante. *What Every Engineer Should Know about Software Engineering*. CRC Press, 2007.

[96] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[97] Naomi Ehrich Leonard and William S Levine. *Using MATLAB to Analyze and Design Control Systems*. Benjamin-Cummings Publishing Co., Inc., 1995.

[98] Wei Li. Another metric suite for object-oriented programming. *Journal of Systems and Software*, 44(2):155–162, 1998.

[99] George James Lidstone. Note on the general case of the bayes-laplace formula for inductive or a posteriori probabilities. *Transactions of the Faculty of Actuaries*, 8:182–192, 1920.

[100] Joseph Lilleberg, Yun Zhu, and Yanqing Zhang. Support vector machines and word2vec for text classification with semantic features. In *Proceedings of 14th International Conference on Cognitive Informatics & Cognitive Computing*, pages 136–140, 2015.

[101] Hui Liu, Jiahao Jin, Zhifeng Xu, Yifan Bu, Yanzhen Zou, and Lu Zhang. Deep learning based code smell detection. *IEEE Transactions on Software Engineering*. To appear, DOI:10.1109/TSE.2019.2936376.

[102] Huihui Liu, Xufang Gong, Li Liao, and Bixin Li. Evaluate how cyclomatic complexity changes in the context of software evolution. In *2018 IEEE 42nd Annual Computer Software and Applications Conference*, volume 2, pages 756–761, 2018.

[103] Lin Liu, Lin Tang, Wen Dong, Shaowen Yao, and Wei Zhou. An overview of topic modeling and its current applications in bioinformatics. *SpringerPlus*, 5(1608):1–22, 2016.

[104] Ayman Madi, Oussama Kassem Zein, and Seifedine Kadry. On the improvement of cyclomatic complexity metric. *International Journal of Software Engineering and Its Applications*, 7(2):67–82, 2013.

[105] Henry B. Mann and Donald R. Whitney. On a test of whether one of two random variables is stochastically larger than the other. *The Annals of Mathematical Statistics*, 18(1):50–60, 1947.

[106] Christopher D Manning, Prabhakar Raghavan, and Hinrich Schütze. Scoring, term weighting and the vector space model. In Christopher D Manning, Prabhakar Raghavan, and Hinrich Schütze, editors, *Introduction to Information Retrieval*, chapter 6, pages 100–123. Cambridge University Press, 2008.

[107] Mika V. Mäntylä and Casper Lassenius. Subjective evaluation of software evolvability using code smells: An empirical study. *Empirical Software Engineering*, 11(3):395–431, 2006.

[108] Robert C. Martin. *The Principles, Patterns, and Practices of Agile Software Development*. Prentice Hall, 2002.

[109] Thomas J. McCabe. A complexity measure. *IEEE Transactions on software Engineering*, SE-2(4):308–320, 1976.

[110] Andrew McCallum and Kamal Nigam. A comparison of event models for naive bayes text classification. In *98 Workshop on Learning for Text Categorization at Association for the Advancement of Artificial Intelligence*, volume 752, pages 41–48, 1998.

[111] James McDermott, David White, Sean Luke, Luca Manzoni, Mauro Castelli, Leonardo Vanneschi, Wojciech Jaskowski, Krzysztof Krawiec, Robin Harper, Kenneth De Jong, and Una-May O'Reily. Genetic programming needs better benchmarks. In *Proceedings of the 14th Annual Conference on Genetic and Evolutionary Computation*, pages 791–798, 2012.

[112] Nenad Medvidovic and Richard N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, 2000.

[113] Andrew Meneely, Ben Smith, and Laurie Williams. Validating software metrics: A spectrum of philosophies. *ACM Transactions on Software Engineering and Methodology*, 21(4):1–28, 2013.

[114] John Michura and Miriam A.M. Capretz. Metrics suite for class complexity. In *International Conference on Information Technology: Coding and Computing*, volume 2, pages 404–409, 2005.

[115] Eleni Miltsakaki and Audrey Troutt. Real time web text classification and analysis of reading difficulty. In *Proceedings of the 3rd Workshop on Innovative Use of NLP for Building Educational Applications*, pages 89–97, 2008.

[116] Mohammed Misbhauddin and Mohammad Alshayeb. UML model refactoring: a systematic literature review. *Empirical Software Engineering*, 20(1):206–251, 2015.

[117] Sanjay Misra, Adewole Adewumi, Luis Fernandez-Sanz, and Robertas Damasevicius. A suite of object-oriented cognitive complexity metrics. *IEEE Access*, 6:8782–8796, 2018.

[118] Sanjay Misra, Murat Koyuncu, Marco Crasso, Cristian Mateos, and Alejandro Zunino. A suite of cognitive complexity metrics. In *Proceedings of the 12th Inter-*

national Conference on Computational Science and Its Applications, pages 234–247, 2012.

[119] Ryan Mitchell. *Web Scraping With Python: Collecting More Data from the Modern Web*. O'Reilly Media, Inc., 2018.

[120] Chihab Eddine Mokaddem, Houari Sahraoui, and Eugene Syriani. Recommending model refactoring rules from refactoring examples. In *Proceedings of the 21th International Conference on Model Driven Engineering Languages and Systems*, pages 257–266, 2018.

[121] Yusuf U. Mshelia, Simon T. Apeh, and Edoghogho Olaye. Towards a unified process model for comprehensive software metrics suite: An introduction. In *Proceedings of 19th International Conference on Computational Science and Its Applications*, pages 52–56, 2019.

[122] Soumyabrata Mukherjee, Bishnubrata Bhattacharya, and Suvajit Mandal. A survey on metrics, models & tools of software cost estimation. *International Journal of Advanced Research in Computer Engineering & Technology*, 2(9):2620–2625, 2013.

[123] Haris Mumtaz, Mohammad Alshayeb, Sajjad Mahmood, and Mahmood Niazi. A survey on UML model smells detection techniques for software refactoring. *Journal of Software: Evolution and Process*, 31(3):e2154:1–17, 2019.

[124] Matthew James Munro. Product metrics for automatic identification of "bad smell" design problems in Java source code. In *Proceedings of the 11th IEEE International Software Metrics Symposium*, pages 15–24, 2005.

[125] Simon Munzert, Christian Rubba, Peter Meißner, and Dominic Nyhuis. *Automated Data Collection with R: A practical Guide to Web Scraping and Text Mining*. John Wiley & Sons, 2014.

[126] Adnan Muslija and Eduard P Enoiu. On the correlation between testing effort and software complexity metrics. *PeerJ Preprints: https://doi.org/10.7287/peerj.preprints.27312v1*, 2018.

[127] Ani Nenkova and Kathleen McKeown. A survey of text summarization techniques. In Charu C. Aggarwal and Cheng Xiang Zhai, editors, *Mining Text Data*, chapter 3, pages 43–76. Springer, 2012.

[128] Edward Ogheneovo. On the relationship between software complexity and maintenance costs. *Journal of Computer and Communications*, 2(14):1–16, 2014.

[129] Marta Olszewska, Yanja Dajsuren, Harald Altinger, Alexander Serebrenik, Marina Waldén, and Mark G. J. van den Brand. Tailoring complexity metrics for Simulink

models. In *Proceedings of the 10th European Conference on Software Architecture Workshops*, pages 1–7, 2016.

[130] Christi A. Gau Pagnanelli, Barbara J. Sheeley, and Ronald S. Carson. Model-based systems engineering in an integrated environment. Online Resource: `https://pdfs.semanticscholar.org/7919/5cf9fe9c620ea3cac2c6c27c0526bb1133fc.pdf`, 2013.

[131] Dipti Pawade, Devansh J. Dave, and Aniruddha Kamath. Exploring software complexity metric from procedure-oriented to object-oriented. In *Proceedings of 6th International Conference-Cloud System and Big Data Engineering*, pages 630–634, 2016.

[132] Mengqi Pei and Xing Wu. Text classification based on SMO and fuzzy model. In *2014 IEEE 7th Joint International Information Technology and Artificial Intelligence Conference*, pages 306–310, 2014.

[133] Louise F. Pendry and Jessica Salvatore. Individual and social benefits of online discussion forums. *Computers in Human Behavior*, 50:211–220, 2015.

[134] Kai Petersen. Measuring and predicting software productivity: A systematic map and review. *Information and Software Technology*, 53(4):317–343, 2011.

[135] Gregor Polančič and Blaž Cegnar. Complexity metrics for process models–a systematic literature review. *Computer Standards & Interfaces*, 51:104–117, 2017.

[136] Saheed Popoola, Xin Zhao, and Jeff Gray. Evolution of bad smells in LabVIEW graphical models. *Journal of Object Technology*, 20(1):1–15, 2020.

[137] James F. Power and Brian A. Malloy. A metrics suite for grammar-based software. *Journal of Software Maintenance and Evolution: Research and Practice*, 16(6):405–426, 2004.

[138] Art Pyster, Rick Adcock, Mark Ardis, Rob Cloutier, Devanandham Henry, Linda Laird, Michael Pennotti, Kevin Sullivan, and Jon Wade. Exploring the relationship between systems engineering and software engineering. *Procedia Computer Science*, 44:708–717, 2015.

[139] Rafa Qutaish and Alain Abran. An analysis of the design and definitions of Halstead's metrics. In *Proceedings of 15th International Workshop on Software Measurement*, pages 337–352, 2005.

[140] Vahid Rahmani Doqaruni. Communication strategies in experienced vs. inexperienced teachers' talk: A sign of transformation in teacher cognition. *Innovation in Language Learning and Teaching*, 11(1):17–31, 2017.

[141] Ghulam Rasool and Zeeshan Arshad. A review of code smell mining techniques. *Journal of Software: Evolution and Process*, 27(11):867–895, 2015.

[142] Andreas Riegler and Clemens Holzmann. Measuring visual user interface complexity of mobile applications with metrics. *Interacting with Computers*, 30(3):207–223, 2018.

[143] Kelly Rivers and Kenneth R Koedinger. Data-driven hint generation in vast solution spaces: A self-improving python programming tutor. *International Journal of Artificial Intelligence in Education*, 27(1):37–64, 2017.

[144] Gabriela Robiolo, Ezequiel Scott, Santiago Matalonga, and Michael Felderer. Technical debt and waste in non-functional requirements documentation: An exploratory study. In *International Conference on Product-Focused Software Process Improvement*, pages 220–235, 2019.

[145] Simone Romano. Dead code. In *2018 IEEE International Conference on Software Maintenance and Evolution*, pages 737–742, 2018.

[146] Simone Romano, Christopher Vendome, Giuseppe Scanniello, and Denys Poshyvanyk. A multi-study investigation into dead code. *IEEE Transactions on Software Engineering*, 46(1):71–99, 2018.

[147] Gerd Ronning. Maximum likelihood estimation of dirichlet distributions. *Journal of Statistical Computation and Simulation*, 32(4):215–221, 1989.

[148] Margaret Rosenzweig, Joan Giblin, Marsha Mickle, Allison Morse, Patricia Sheehy, Valerie Sommer, and Bridging the Gap Working Group. Bridging the gap: A descriptive study of knowledge and skill needs in the first year of oncology nurse practitioner practice. *Oncology Nursing Forum*, 39(2):195–201, 2012.

[149] James Rumbaugh, Ivar Jacobson, and Grady Booch. *The Unified Modeling Language*. Addison-Wesley Professional, 1999.

[150] David E. Rumelhart, Geoffrey E. Hinton, and Ronald J. Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986.

[151] Gerard Salton and Chung-Shu Yang. On the specification of term values in automatic indexing. Technical Report 73-173. https://ecommons.cornell.edu/handle/1813/6016, Cornell University, 1973.

[152] Ruya Samli, Zeynep Behrin Güven Aydın, and Uğur Osman Yücel. Measurement in software engineering: The importance of software metrics. In Zeynep Altan, editor, *Applications and Approaches to Object-Oriented Software Design: Emerging Research and Opportunities*, chapter 7, pages 166–182. IGI Global, 2020.

[153] José Amancio M. Santos, João B. Rocha-Junior, Luciana Carla Lins Prates, Rogeres Santos do Nascimento, Mydiã Falcão Freitas, and Manoel Gomes de Mendonça. A systematic review on the code smell effect. *Journal of Systems and Software*, 144:450–477, 2018.

[154] Robert E. Schapire. The boosting approach to machine learning: An overview. *Nonlinear Estimation and Classification*, 171:149–171, 2003.

[155] National Technical Information Service. Integration definition for function modeling. Technical Report PUB 183. `https://nvlpubs.nist.gov/nistpubs/Legacy/FIPS/fipspub183.pdf`, National Institute of Standards and Technology, 1993.

[156] Jingqiu Shao and Yingxu Wang. A new measure of software complexity based on cognitive weights. *Canadian Journal of Electrical and Computer Engineering*, 28(2):69–74, 2003.

[157] Naveen Sharma, Padmaja Joshi, and Rushikesh K. Joshi. Applicability of Weyuker's property 9 to object-oriented metrics. *IEEE Transactions on Software Engineering*, 32(3):209–211, 2006.

[158] Tushar Sharma, Vasiliki Efstathiou, Panos Louridas, and Diomidis Spinellis. Code smell detection by deep direct-learning and transfer-learning. *Journal of Systems and Software*, 176:1–15, 2021.

[159] Tushar Sharma and Diomidis Spinellis. A survey on software smells. *Journal of Systems and Software*, 138:158–173, 2018.

[160] Raed Shatnawi and Wei Li. An investigation of bad smells in object-oriented design. In *Proceedings of the 3rd International Conference on Information Technology: New Generations*, pages 161–165, 2006.

[161] Martin Shepperd. A critique of cyclomatic complexity as a software metric. *Software Engineering Journal*, 3(2):30–36, 1988.

[162] Robert Shishko and Robert Aster. NASA systems engineering handbook. Technical Report SP-2007-6105. `https://www.nasa.gov/sites/default/files/atoms/files/nasa_systems_engineering_handbook_0.pdf`, National Aeronautics and Space Administration, 1995.

[163] Carson Sievert and Kenneth Shirley. Ldavis: A method for visualizing and interpreting topics. In *Proceedings of the Workshop on Interactive Language Learning, Visualization, and Interfaces*, pages 63–70, 2014.

[164] Kyle Simpson. *You Don't Know JS: Async & Performance*. O'Reilly Media, Inc., 2015.

[165] Alice Squires, David Olwell, Garry Roedler, and Joseph J. Ekstrom. Gaps in the body of knowledge of systems engineering. In *International Council on Systems Engineering - International Symposium*, volume 22, pages 1967–1976, 2012.

[166] K. P. Srinivasan and Devi Thirupathi. Software metrics validation methodologies in software engineering. *International Journal of Software Engineering & Applications*, 5(6):87–102, 2014.

[167] Gursaran Srivastava and Gurdev Roy. On the applicability of Weyuker property 9 to object-oriented structural inheritance complexity metrics. *IEEE Transactions on Software Engineering*, 27(4):381–384, 2001.

[168] Michael Steinbach, George Karypis, and Vipin Kumar. A comparison of document clustering techniques. In *Proceedings of the 6th International Conference on Knowledge Discovery and Data Mining Workshop on Text Mining*, pages 1–20, 2000.

[169] Dave Steinberg, Frank Budinsky, Ed Merks, and Marcelo Paternostro. *Eclipse Modeling Framework*. Pearson Education, 2008.

[170] David G. Stork, Richard O. Duda, Peter E. Hart, and D. Stork. *Pattern Classification*. Wiley Interscience Publication, 2001.

[171] Robert O. Stroud, Atila Ertas, and Susan Mengel. Application of cyclomatic complexity in enterprise architecture frameworks. *IEEE Systems Journal*, 13(3):2166–2176, 2019.

[172] Janos Sztipanovits, Ted Bapty, Sandeep Neema, Larry Howard, and Ethan Jackson. Openmeta: A model-and component-based design tool chain for cyber-physical systems. In *From Programs to Systems - The Systems Perspective in Computing Workshop*, pages 235–248, 2014.

[173] Tadej Tašner, Darko Lovrec, Frančišek Tašner, and Jörg Edler. Comparison of LabVIEW and MATLAB for scientific research. *Annals of the Faculty of Engineering Hunedoara - International Journal of Engineering*, 10(3):389–394, 2012.

[174] Peeratham Techapalokul and Eli Tilevich. Quality hound—an online code smell analyzer for scratch programs. In *2017 IEEE Symposium on Visual Languages and Human-Centric Computing*, pages 337–338, 2017.

[175] John C. Thomas and John T. Richards. Achieving psychological simplicity: methods and measures to reduce cognitive complexity. In Julie A. Jacko and Andrew Sears, editors, *The Human–Computer Interaction Handbook: Fundamentals, Evolving Technologies and Emerging Applications*, pages 498–507. 2008.

[176] Simon Tong and Daphne Koller. Support vector machine active learning with applications to text classification. *Journal of Machine Learning Research*, 2:45–66, 2001.

[177] Quang Minh Tran, Benjamin Wilmes, and Christian Dziobek. Refactoring of Simulink diagrams via composition of transformation steps. In *Proceedings of the 8th International Conference on Software Engineering Advances*, pages 140–145, 2013.

[178] Michele Tufano, Fabio Palomba, Gabriele Bavota, Rocco Oliveto, Massimiliano Di Penta, Andrea De Lucia, and Denys Poshyvanyk. When and why your code starts to smell bad (and whether the smells go away). *IEEE Transactions on Software Engineering*, 43(11):1063–1088, 2017.

[179] Rajesh Vasa and Jean-Guy Schneider. Evolution of cyclomatic complexity in object-oriented software. In *Proceedings of the 7th Workshop on Quantitative Approaches in Object-Oriented Software Engineering*, pages 1–5, 2003.

[180] William W. Wadge and Edward A. Ashcroft. *Lucid, the Dataflow Programming Language*. Academic Press, 1985.

[181] Stefan Wagner, Daniel Méndez Fernández, Michael Felderer, Antonio Vetrò, Marcos Kalinowski, Roel Wieringa, Dietmar Pfahl, Tayana Conte, Marie-Therese Christiansson, Desmond Greer, et al. Status quo in requirements engineering: A theory and a global family of surveys. *ACM Transactions on Software Engineering and Methodology*, 28(2):1–48, 2019.

[182] Yingxu Wang. On cognitive informatics. *Brain and Mind*, 4(2):151–167, 2003.

[183] Arthur Henry Watson, Dolores R. Wallace, and Thomas J. McCabe. *Structured Testing: A Testing Methodology Using the Cyclomatic Complexity Metric*. US Department of Commerce, Technology Administration, National Institute of Standards and Technology, 1996.

[184] Elaine J. Weyuker. Evaluating software complexity measures. *IEEE Transactions on Software Engineering*, 14(9):1357–1365, 1988.

[185] Manchek A. Wong and J. A. Hartigan. Algorithm AS 136: A K-Means clustering algorithm. *Journal of the Royal Statistical Society (Series C - Applied Statistics)*, 28(1):100–108, 1979.

[186] Qingyao Wu, Yunming Ye, Haijun Zhang, Michael K. Ng, and Shen-Shyang Ho. Forestexter: An efficient random forest algorithm for imbalanced text categorization. *Knowledge-Based Systems*, 67:105–116, 2014.

[187] Aiko Yamashita and Leon Moonen. Do developers care about code smells? An exploratory survey. In *Proceedings of the 20th Working Conference on Reverse Engineering*, pages 242–251, 2013.

[188] Sheng Yu and Shijie Zhou. A survey on metric of software complexity. In *Proceedings of the 2nd International Conference on Information Management and Engineering*, pages 352–356, 2010.

[189] Shahed Zaman, Bram Adams, and Ahmed E Hassan. Security versus performance bugs: A case study on firefox. In *Proceedings of the 8th International Conference on Mining Software Repositories*, pages 93–102, 2011.

[190] Shahed Zaman, Bram Adams, and Ahmed E Hassan. A qualitative study on performance bugs. In *Proceedings of the 9th International Conference on Mining Software Repositories*, pages 199–208, 2012.

[191] Hongyu Zhang, Yuan-Fang Li, and Hee Beng Kuan Tan. Measuring design complexity of semantic web ontologies. *Journal of Systems and Software*, 83(5):803–814, 2010.

[192] Lu Zhang and Dan Xie. Comments on "On the applicability of Weyuker property 9 to object-oriented structural inheritance complexity metrics". *IEEE Transactions on Software Engineering*, 28(5):526–527, 2002.

[193] Min Zhang, Tracy Hall, and Nathan Baddoo. Code bad smells: A review of current knowledge. *Journal of Software Maintenance and Evolution: Research and Practice*, 23(3):179–202, 2011.

[194] Yin Zhang, Rong Jin, and Zhi Hua Zhou. Understanding bag-of-words model: A statistical framework. *International Journal of Machine Learning and Cybernetics*, 1:43–52, 2010.

[195] Xin Zhao and Jeff Gray. BESMER: An approach for bad smells summarization in systems models. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems Companion*, pages 304–313, 2019.

[196] Xin Zhao, Jeff Gray, and Taylor Riche. A survey-based empirical evaluation of bad smells in LabVIEW systems models. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering*, pages 177–188, 2021.

[197] Xin Zhao, Zhe Jiang, and Jeff Gray. Text classification and topic modeling for online discussion forums: An empirical study from the systems modeling community. In Alessandro Fiori, editor, *Trends and Applications of Text Summarization Techniques*, chapter 6, pages 151–186. IGI Global, 2020.

[198] Horst Zuse. Properties of software measures. *Software Quality Journal*, 1(4):225–260, 1992.

[199] Horst Zuse and Peter Bollmann. Software metrics: using measurement theory to describe the properties and scales of static software complexity metrics. *ACM Sigplan Notices*, 24(8):23–33, 1989.

APPENDIX A

IRB CERTIFICATE

THE UNIVERSITY OF ALABAMA® | *Office of the Vice President for* Research & Economic Development
Office for Research Compliance

February 18, 2020

Xin Zhao
Department of Computer Science
The University of Alabama
Box 870290

Re: IRB # EX-20-CM-054: "Design Evaluation of LabVIEW Systems Models"

Dear Mr. Zhao,

The University of Alabama Institutional Review Board has granted approval for your proposed research. Your application has been given exempt approval according to 45 CFR part 46. Approval has been given under exempt review category 2 as outlined below:

(2) Research that only includes interactions involving educational tests (cognitive, diagnostic, aptitude, achievement), survey procedures, interview procedures, or observation of public behavior (including visual or auditory recording) if at least one of the following criteria is met:

(i) The information obtained is recorded by the investigator in such a manner that the identity of the human subjects cannot readily be ascertained, directly or through identifiers linked to the subjects

The approval for your application will lapse on February 17, 2021. If your research will continue beyond this date, please submit the annual report to the IRB as required by University policy before the lapse. Please note, any modifications made in research design, methodology, or procedures must be submitted to and approved by the IRB before implementation. Please submit a final report form when the study is complete.

Please use reproductions of the IRB approved informed consent form to obtain consent from your participants.

Sincerely,

[redacted]

Carpantato T. Myles, MSM, CIM, CIP
Director & Research Compliance Officer