Perceived relevance of automatic code inspection in end-user development
A study on VBA

Roy, Sohon; Van Deursen, Arie; Hermans, Felienne

DOI
10.1145/3319008.3319028

Publication date
2019

Document Version
Accepted author manuscript

Published in
Proceedings of EASE 2019 - Evaluation and Assessment in Software Engineering

Perceived Relevance of Automatic Code Inspection in End-User Development: A Study on VBA

Sohon Roy
Delft University of Technology
Delft, The Netherlands
S.Roy-1@tudelft.nl

Arie van Deursen
Delft University of Technology
Delft, The Netherlands
Arie.vanDeursen@tudelft.nl

Felienne Hermans
Leiden University
Leiden, The Netherlands
f.f.j.hermans@liacs.leidenuniv.nl

ABSTRACT

Microsoft VBA (Visual Basic for Applications) is a programming language widely used by end-user programmers, often alongside the popular spreadsheet software Excel. Together they form the popular Excel-VBA application ecosystem. Despite being popular, spreadsheets are known to be fault-prone, and to minimize risk of faults in the overall Excel-VBA ecosystem, it is important to support end-user programmers in improving the code quality of their VBA programs also, in addition to improving spreadsheet technology and practices. In traditional software development, automatic code inspection using static analysis tools has been found effective in improving code quality, but the practical relevance of this technique in an end-user development context remains unexplored. With the aim of popularizing it in the end-user community, in this paper we examine the relevance of automatic code inspection in terms of how inspection rules are perceived by VBA programmers. We conduct a qualitative study consisting of interviews with 14 VBA programmers, who share their perceptions about 20 inspection rules that most frequently detected code quality issues in an industrial dataset of 25 VBA applications, obtained from a financial services company. Results show that the 20 studied inspection rules can be grouped into three categories of user perceptions based on the type of issues they warn about: i) 11 rules that warn about serious problems which need fixing, ii) 7 rules that warn about bad practices which do not mandate fixing, and iii) 2 rules that warn about purposeful code elements rather than issues. Based on these perceptions, we conclude that automatic code inspection is considerably relevant in an end-user development context such as VBA. The perceptions also indicate which inspection rules deserve the most attention from interested researchers and tool developers. Lastly, our results also reveal 3 additional issue types that are not covered by the existing inspection rules, and are therefore impetus for creating new rules.

CCS CONCEPTS

• **Software and its engineering → Software verification and validation**.

EASE '19, April 15–17, 2019, Copenhagen, Denmark

ACM ISBN 978-1-4503-7145-2/19/04…$15.00
https://doi.org/10.1145/3319008.3319028

KEYWORDS

static analysis, end-user development, VBA, developer perceptions, code quality

ACM Reference Format:
Sohon Roy, Arie van Deursen, and Felienne Hermans. 2019. Perceived Relevance of Automatic Code Inspection in End-User Development: A Study on VBA. In *Evaluation and Assessment in Software Engineering (EASE '19), April 15–17, 2019, Copenhagen, Denmark*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3319008.3319028

1 INTRODUCTION

Next to the practice of professional software development, there exists a software development paradigm referred to as *End-User Development (EUD)* [17]. In EUD, programming is performed by *end-user programmers*—individuals who usually do not have background or formal training in software engineering disciplines [17]. In recent times, due to increased dependence on computers in every sphere of life, the practice of EUD has grown considerably. The increasing popularity of several scripting languages has contributed generously towards this growth. According to estimates, end-user programmers largely outnumber professional programmers [25].

Among end-user development languages, Microsoft VBA (*Visual Basic for Applications*) enjoys a particularly high popularity, and is most commonly used with another popular end-user tool, the spreadsheet software Microsoft Excel [9]. Excel spreadsheets, and VBA programs, together form the widely popular Excel-VBA ecosystem.

Despite their popularity, however, spreadsheets are known to be fault-prone [20–22], and they lack reliable testing practices [23]. Furthermore, spreadsheets in the industry have average lifespans of five years [10] and are used by up to dozen users within organizations [10]; these spreadsheets evolve, growing in size and functionality [12]. Consequently, the maintenance of both the spreadsheets, and the corresponding VBA programs, is not trivial. Yet, results obtained from spreadsheets are commonly used even in critical business contexts, occasionally resulting in serious incidents due to spreadsheet faults [7]. Therefore we believe, to minimize the risk of faults, and to support maintainability in the overall Excel-VBA ecosystem, it is important to support end-users in improving the code quality of their VBA programs also, in addition to the ongoing efforts in improving spreadsheet technology and practices [9, 11].

For traditional software, automatic code inspection by static analysis tools has been found effective in improving code quality [28]. Automatic code inspection helps by warning developers about issues related to best practices and coding styles. The types of issues that are warned about, are governed by the *inspection rules* configured within the static analysis tools. As such, the inspection rules

constitute the essence of automatic code inspection, determining which code quality issues are detected and reported to developers.

Due to its benefits, automatic code inspection is being increasingly applied in traditional software development [3, 24, 29]. Hence, adoption of such tools in the VBA community may also help ensure better code quality of VBA programs. However, the VBA end-user paradigm is substantially different from traditional software development, and VBA programmers are also different compared to traditional developers, mostly in terms of background and training. Hence empirical evidence supporting the applicability and relevance of automatic code inspection in the VBA context, helps the cause of adoption of the technique. Therefore, with the aim of popularizing automatic code inspection in the VBA context, our goal in this paper is **to assess the relevance of automatic code inspection according to VBA programmers**.

To address our goal, we conducted an exploratory qualitative [5] study comprising of interviews with 14 VBA developers. Since inspection rules constitute the essence of automatic code inspection, we were specifically interested in learning how VBA programmers perceive automatic code inspection rules; do they view the warnings generated by them as relevant issues, or they do not, and why? To answer such questions, we conducted the interviews, focusing on 1) perceptions of automatic code inspection rules, and 2) discovery of code quality issue types that may not yet be covered by existing rules, creating the need for new rules.

During the interviews we used a set of 20 VBA-specific automatic code inspection rules to discuss with the participants. We arrived at this set of rules by applying automatic code inspection on an industrial dataset of 25 operational VBA applications, obtained from a prominent Dutch financial services company. From this analysis we were able to identify these top 20 inspection rules that most frequently detected code quality issues in a dataset of real-world VBA applications.

Our results reveal:

- The 20 studied inspection rules can be grouped into three categories of user perceptions based on the type of issues they warn about: i) 11 rules that warn about serious problems which need fixing, ii) 7 rules that warn about bad practices which do not mandate fixing, and iii) 2 rules that warn about purposeful code elements rather than issues.
- 3 additional issue types are not covered by currently existing rules, but could be captured by creating new rules for them.

Based on our results, we conclude that developers perceive automatic code inspection to be relevant in an end-user development context such as VBA. 18 of the 20 inspection rules studied are capable of revealing code quality issues that VBA programmers consider as problems, and only 2 detect issues that they consider irrelevant. As such VBA programmers can adopt automatic code inspection to ensure better code quality of their VBA programs. Moreover, VBA programmers who are inexperienced, or uninitiated to the concept of code quality, can now be warned and instructed to fix the serious issues detected by the most relevant 11 rules, irrespective of their understanding of the issues, or their potential dangers. Researchers and developers of static analysis tools can focus their efforts in refining the most relevant 11 inspection rules. They should also consider creation of new inspection rules, to detect the additional 3 issue types revealed in our study.

2 BACKGROUND AND MOTIVATION

Code inspection is defined as a "*static analysis technique that relies on visual examination of development products to detect errors, violations of development standards, and other problems*" [1]. With the emergence of *Automatic Static Analysis*, (ASA) and *Automatic Static Analysis Tools* (ASATs) [3, 28], automatic code inspection has been a regular feature of ASATs, and their variants such as linters [6, 27].

Automatic code inspection warns developers about issues related to best practices and coding styles. In some popular ASATs such as IntelliJ IDEA [14, 31] and ReSharper [15], 'warnings' generated by automatic code inspection are referred to as *inspections*. For example, the warning "*Method* 'Foobar()' *is too long. Consider breaking up into smaller methods*" is an *inspection*. The types of issues that are warned about, are governed by the *inspection rules*, which in this case is the 'Long Method' rule. Typically, the tools offer configurable lists of such inspection rules, in which developers can choose which rules they want enabled. The tools only generate warnings pertaining to the enabled inspection rules.

Warnings generated by automatic code inspection in ASATs help developers find issues and refactoring opportunities early on in the development process, when it requires less effort, and is less expensive to apply fixes [13]. They also help detect problems that affect maintainability of code [2, 3, 30, 31]. Hence, we hypothesize that automatic code inspection can be helpful to address code quality and maintainability related challenges in the VBA programming context.

However, automatic inspection by ASATs traditionally suffer from the problem of *false positives* [3, 16, 31]. Moreover in the context of ASATs, a false positive can be interpreted in two different ways: 1) a wrongly generated warning about an issue that does not exist, or 2) a correct warning about an issue that is not perceived by developers as a problem, or a potential improvement to the software being analyzed [2]. The second interpretation has particularly motivated researchers to study developers' perception of ASATs and their warnings [16, 28].

Nevertheless, these studies focus on traditional software development. VBA developers are different compared to traditional software developers, in at least two aspects. One, they usually lack formal training in software disciplines, and two, they work within the boundaries of a significantly different development paradigm, with relatively limited development aids. The default IDE for VBA development does not even provide a feature as basic as automatic indentation of code. As such, we cannot conclude without an investigation, whether VBA developers perceive the benefits of automatic code inspection the same way as traditional software developers do. Therefore in this paper we conduct our qualitative study on how VBA developers perceive automatic code inspection.

3 STUDY DESIGN

3.1 Objective and Research Questions

Our goal is to assess the relevance of automatic VBA code inspection according to the perceptions of VBA programmers. Specifically, in this paper, our objective is to learn how VBA programmers perceive

automatic code inspection rules that currently feature in static analysis tools for VBA. As such we seek answers to the following research questions:

RQ1: How do VBA developers (who are end-users compared to traditional software developers) perceive automatic code inspection rules in terms of the issues they detect?

RQ2: Do VBA developers also encounter issue types that are not covered by existing automatic code inspection rules?

3.2 Method

To answer our research questions, we have conducted an exploratory qualitative study [5] comprising of semi-structured interviews with 14 VBA developers, focusing on their perceptions of automatic code inspection rules.

3.3 Setup

3.3.1 Study Objects. As objects of our study, we used a set of 20 VBA-specific automatic code inspection rules that we could feasibly discuss within the time-frame of a single interview session. To arrive at this selection of 20 rules, we applied automatic code inspection to an industrial dataset of 25 operational VBA applications, obtained from a prominent Dutch financial services company specializing in international asset management. Due to reasons of confidentiality, the identity of this organization cannot be disclosed in this paper. The VBA applications along with the spreadsheets they are embedded in, are also confidential. They can not be made publicly available either, but being directly sourced from an industrial organization, they are operational real-world applications. Further characteristics of the VBA applications are provided in Table 1.

To conduct automatic code inspection of the 25 VBA applications, we used a VBA code analysis toolkit developed by our industrial collaborators from the commercial spreadsheet analysis services company Infotron.[1] This VBA analysis toolkit, developed as part of a larger spreadsheet analysis application called PerfectXL, is an implementation of the static analysis engine of the Rubberduck[2] open-source COM add-in for the default VBA editor.

The Rubberduck add-in potentially supports VBA developers with features of modern IDEs like code inspections, refactoring, and unit testing, all of which are unavailable in the default VBA editor provided with Excel. Rubberduck is a modern open-source project with an active developer community and it is also unofficially supported by JetBrains,[3] the developer of IntelliJ and ReSharper. For all practical purposes, we presume the list of 67 inspection rules featured in Rubberduck as the current industry standard of VBA-specific automatic code inspection rules.

In the PerfectXL implementation of Rubberduck engine that we used for our study, the number of inspection rules is reduced to 44, featuring those which are adjudged as more relevant in the VBA context, according to expert opinion of Infotron's team of consultants.

From the automatic code inspection of the 25 VBA applications, we were able to identify a set of top 20 inspection rules that most frequently detected code quality issues. We have considered these 20 rules as suitable candidates for discussing in the interviews, as depicted later with results of this study in Table 4. Detailed descriptions and explanations of the rules are provided with the discussion of results in Section 4.

It is important to note that the goal of this study is not finding which issues occur most frequently in VBA applications at large. Our goal, in this paper, is to obtain a qualitative understanding of how developers perceive VBA code inspection rules, and for that we needed to start with a set of rules which we could discuss with our interviewees. The set of 20 inspection rules used in our study are not guaranteed to be the ones capturing the most prevalent issue types occurring in VBA code in general; partly because 25 VBA applications do not represent a sufficiently large population for generalization purposes, and partly because the PerfectXL VBA analysis toolkit uses a reduced set of 44 inspection rules out of the 67 listed by Rubberduck. Nevertheless, instead of being chosen arbitrarily, these 20 rules represent a set of rules methodically obtained from the analysis of real-world applications, and as such they provide us with a rational starting point for a first study of its kind.

Table 1: VBA workbook characteristics and number of inspections found in them

ID	LOC	Modules	Subs and Functions	Inspections
V01	4066	24	35	487
V02	6370	23	48	344
V03	7077	28	66	413
V04	6417	21	73	539
V05	1735	13	36	196
V06	573	12	13	110
V07	16377	26	162	1174
V08	4627	37	40	492
V09	4409	23	69	630
V10	226	5	1	3
V11	2285	11	23	112
V12	209	5	5	10
V13	181	5	5	53
V14	3187	17	36	403
V15	464	9	13	87
V16	292	1	12	15
V17	4197	22	53	476
V18	4422	26	69	630
V19	4831	25	81	791
V20	3141	18	42	434
V21	5615	20	88	423
V22	2325	22	16	299
V23	3401	13	47	410
V24	268	4	11	35
V25	406	3	5	238
Total	87101	413	1049	8804
Mean	3484	16	41	352
Median	3187	18	36	403

[1] https://infotron.nl/en/home-2/
[2] http://rubberduckvba.com/
[3] https://www.jetbrains.com/

Table 2: Participant Demography

Gender	**Male**	**Female**	
	86%	14%	
Location	**Europe, UK**	**US**	
	78%	22%	
Age	**Range**	**Mean**	**Median**
	32 - 62	47.5	48
VBA Experience			
Years	**Range**	**Mean**	**Median**
	3 - 21	15.35	16.5
Expertise	**High**	**Mid**	**Low**
	43%	43%	14%
Education Level	**Postgraduate**	**Graduate**	**Undergraduate**
	43%	43%	14%

3.3.2 Participants. In this paper, we were interested more on the qualitative aspects of different perspectives and opinions rather than generalization of conclusions. Hence, similar to past such studies [28], our participant selection strategy was to find VBA developers who could be expected to understand the consequences of code quality issues, and therefore be in a position to judge the usefulness of detecting issues through code inspections. Six of the participants are employees of the financial company owning the 25 VBA applications that we analyzed. We were introduced to them during our collaboration for analyzing their VBA spreadsheets, as described in Section 3.3.1. The remaining nine we contacted through our network of Excel and VBA professionals in LinkedIn. The participants' demographic information is shown in Table 2, and their roles, industries, lengths of VBA experience, and levels of expertise are shown in Table 3.

Our participants develop VBA applications in regular basis during the course of their primary professional activity. Three of them have background of formal training in computer science with P3 studying it as a major subject, and P2, P14 as minor. The rest all of them have non-IT formal education in diverse subjects, and of varying levels starting from high school diploma to PhD. All of them are self-taught with respect to their VBA knowledge, through books, online resources, or on-the-job learning.

In terms of level of expertise, 2 (P7, P8) are in the lowest level, using VBA programming only as a supplement to their primary functional responsibilities, and with relatively lower experience (3 - 6 years). 6 are in the middle level, either involved in VBA development as part of their main professional responsibilities, or with relatively longer experience. Remaining 6 are in the highest level, and are community acknowledged VBA/Excel experts and trainers. 3 of these 6 are also MS Excel MVP (Most Valued Professional).

3.3.3 Interviews. To understand how developers perceive the 20 inspection rules, and to leverage the chance of discovering new inspection rules, we conducted interviews with 14 developers over Skype. The interviews lasted 40 minutes in average, and were based on a guideline consisting of three parts as follows:

(1) Background - In the first part, we asked about years of experience, formal education, VBA programming experience, other programming languages used, and current professional role.
(2) Code Quality Issue Types - In the second part, first we explained the concept of code quality issue types that are typically detected by automatic code inspection rules with an example, and then asked if the participants could think of other such issue types, independently without any prompting. The aim of this section was to discover issue types that are not covered by existing inspection rules, and could be captured by creating new rules (answer to **RQ2**). By not revealing at this stage the list of 20 inspection rules being studied, we expected the participants to independently cite problems they face, which might or might not be covered by the currently existing inspection rules.
(3) Perceptions - In the last part, we discussed with the participants their perceptions about the relevance of the 20 inspection rules being studied (Section 3.3.1). While the discussions during this part of the interviews were steered to focus on the 20 inspection rules, there was no strict order of discussion across all the interviews; participants often spoke about different but related inspection rules together, resulting in switching back and forth between the items in the list. We allowed this free floating motion during the interviews, as we believed it helped the participants express their opinions best with minimum interference or interruption of their chain of thoughts. Least interference in expression of opinions was important to us, as our goal was to obtain the best understanding of the participants' perceptions. Furthermore, due to their level of expertise or usage, certain participants did not recognize or understand certain inspection rules. In such cases, the corresponding inspection rules were omitted from being discussed, and recorded as "*no comments or not familiar with*".

3.4 Data Analysis

We analyzed responses pertaining to the Code Quality Issue Types parts of the interviews to obtain a list of issue types that are not covered by existing inspection rules.

We conducted the qualitative analysis of the Perceptions parts of the interviews through the process of *coding*: association of coherent keywords and excerpts with a code representing their key characteristics [5, 26]. Our aim was to assess the perceptions of the participants. Hence we grouped together related codes giving rise to categories of perceptions as described further in Section 4.

We next distributed the 20 inspection rules into the categories of perceptions, based on the popular participant response for each rule. The three categories of perceptions along with the distribution of the inspection rules into respective perception categories, represent the main outcomes of our study.

Table 3: Participants

ID	Role	Industry	VBA Experience		Educational Background
			Years	Expertise	
P1	Portfolio Manager	Financial services	18	Mid	Business Economics
P2	Portfolio Manager	Financial services	20	Mid	Computational Finance, AI
P3	Technical Project Lead	Financial services	15	Mid	Computer Science
P4	VBA Developer	Financial services	15	Mid	Business Administration
P5	Excel Consultant	Technical consultancy	15	High	Chemical Engineering
P6	Portfolio Manager	Financial services	5	Mid	Financial Economics
P7	Portfolio Manager	Financial services	3	Low	Finance
P8	Project Manager	Administration	6	Low	Mechanical Engineering
P9	Senior Management	Transportation	20	High	Architecture
P10	Excel Consultant	Technical consultancy	15	High	Secondary School Education
P11	Consultant	Technical consultancy	21	Mid	Finance, Business
P12	Excel Consultant	Technical consultancy	21	High	Metallurgy
P13	Excel Consultant	Technical consultancy	21	High	Psychology
P14	Excel Consultant	Technical consultancy	20	High	Electrical Engineering, CS

4 RESULTS

Our study resulted in three main outcomes: 1) Three broad categories of perceptions that VBA programmers hold about automatic inspection rules. 2) The grouping of the studied 20 inspection rules into the three categories of perceptions, revealing which are perceived to be most relevant. 3) Three additional issue types that are not covered by existing inspection rules.

4.1 Categories of Perceptions

We conducted 14 semi-structured interviews with VBA developers to understand their perceptions about the 20 automatic inspection rules we studied (Section 3.3.1). The *coding* of the semi-structured interviews, as explained in Section 3.4, led to three categories of perceptions that the participants hold about the 20 inspection rules based on the issue types they detect. The categories are as follows:

- A: Serious problems that need fixing - Perceptions that indicate a participant views the issue type detected as a serious problem or vulnerability and strongly recommends fixing or avoidance.
- B: Bad practices but fixing is optional - Perceptions that indicate a participant views the issue type detected as not serious, and is more related to aesthetics or coding styles; views refactoring or fixing as optional.
- C: Purposeful code elements rather than issues - Perceptions that indicate a participant views the issue type detected as a code element serving a purpose, rather than treating it as an issue.

In the following subsection we discuss the grouping of the 20 inspection rules into the above three perception categories based on the participant responses.

4.2 Perceptions of the Inspection Rules

Table 4 shows how the participants perceived the different inspection rules, in terms of the three perception categories:

- A = Serious problems that need fixing
- B = Bad practices but fixing is optional
- C = Purposeful code elements rather than issues

A "-" indicates cases where a participant did not have anything to say, or was not familiar with the programming concept or feature being examined by an inspection rule. For example, P13 is not familiar with the concept of write-only property, and as such did not have any opinion about it; hence she had no perception to share about the inspection rule *Write Only Property*.

Taking into account the most popular responses from Table 4, we have distributed the 20 inspection rules examined in our study, into the three perception categories as shown in Table 5. We discuss them in detail as follows.

4.2.1 A - Serious problems that need fixing. As seen from Table 5, the participants perceive 11 of the 20 inspection rules as highly relevant in detecting issues they consider as problems.

Variable Declarations (4 rules): Undeclared Variable, Variable Type Not Declared, and Option Explicit are considered problems because VBA is a weakly typed language. If a variable is not declared, VBA creates a variable on the fly of type "Variant", unless the Option Explicit declarative is added at the beginning of the code module. If a variable is declared but its type is not specified, then VBA assigns the type "Variant" to it as well, which can lead to a performance overhead in rare cases, but mostly can lead to confusion regarding what type of data the variable is holding inside at. As P12 points out "*...there is no way to tell if a number contained inside a Variant is a number or a string, like part of an address. Hence it is always better to explicitly declare the type.*" On the other hand with an Option Explicit declarative at the beginning of a module, undeclared variables throw a compilation error, and thus all participants except P7 and P10 strictly use it. However, P7 stated that they prefer not to use Option Explicit, and did not like the concept of declaring variables because "*Its annoying having to waste time thinking whether a variable is long or integer, or double... especially since I have never ran into problems ever, why would I bother to turn it (Option Explicit) on? I don't even get why people want it to be on..*" Such opinion is interesting to note the difference between VBA developers and conventional software developers. P7

Table 4: Participant Perception of Inspection Rules in Terms of Perception Categories

Inspection Rules	P1	P2	P3	P4	P5	P6	P7	P8	P9	P10	P11	P12	P13	P14	Popular Response
Variable Not Assigned	A	B	A	B	B	A	B	B	A	B	A	B	A	A	A (7/14)
Undeclared Variable	A	B	A	A	A	A	C	A	A	B	A	A	A	A	A (11/14)
Variable Not Used	B	B	A	B	B	B	A	B	B	B	B	B	B	B	B (12/14)
Parameter Can Be "ByVal"	B	A	-	A	B	-	-	-	A	-	B	B	B	B	B (6/14)
Unassigned Variable Usage	A	B	A	B	B	A	B	B	A	B	A	B	A	A	A (7/14)
Variable Type Not Declared	A	A	A	A	A	A	C	A	A	B	A	A	A	A	A (12/14)
Constant Not Used	A	C	A	B	B	B	-	B	B	B	B	B	B	B	B (10/14)
Function Return Value Not Used	A	C	A	A	B	A	A	-	B	-	A	B	A	B	A (7/14)
Procedure Not Used	A	C	A	B	B	C	C	-	C	B	A	B	A	A	A (5/14)
Long Sub Or Function	B	B	A	A	A	B	A	A	A	B	B	B	B	B	B (8/14)
Implicit Public Members	A	A	A	B	A	B	-	-	A	-	-	A	-	A	A (7/14)
Parameter Not Used	B	B	A	B	B	B	-	-	B	B	B	B	B	A	B (10/14)
Procedure Can Be Written As Function	B	B	B	A	B	-	-	-	B	B	B	B	B	B	B (10/14)
Obsolete Call Statement	C	C	C	C	B	C	C	-	B	-	C	B	A	B	C (7/14)
Option Explicit	A	A	A	A	A	A	C	A	A	B	A	A	A	A	A (12/14)
Autorun Macro	A	C	A	C	C	B	B	C	C	C	A	C	C	C	C (9/14)
Multiple Declarations	A	A	-	-	A	B	B	A	A	A	A	A	A	A	A (10/14)
Write Only Property	-	A	-	-	A	A	-	-	B	-	B	-	-	A	A (4/14)
Obsolete Global	A	B	-	-	B	B	-	B	A	-	-	B	A	B	B (6/14)
Recorded Macro	A	A	A	A	B	B	A	B	A	B	A	A	A	A	A (10/14)

A = Serious problems that need fixing, B = Bad practices but fixing is optional,
C = Purposeful code elements rather than issues, "-" = No comments or not familiar with

Table 5: Inspection Rules Grouped According to Perception Categories

A - Serious problems that need fixing
Variable Not Assigned
Undeclared Variable
Unassigned Variable Usage
Variable Type Not Declared
Function Return Value Not Used
Procedure Not Used
Implicit Public Members
Option Explicit
Multiple Declarations
Write Only Property
Recorded Macro
B - Bad practices but fixing is optional
Variable Not Used
Parameter Can Be "ByVal"
Constant Not Used
Long Sub Or Function
Parameter Not Used
Procedure Can Be Written As Function
Obsolete Global
C - Purposeful code elements rather than issues
Obsolete Call Statement
Autorun Macro

does clarify though that their views regarding Option Explicit is personal, and their organizational policy dictates usage of Option Explicit. Lastly, Multiple Declarations is a problem because in VBA, unlike in most other programming languages, 'Dim a,b,c, as integer' achieves the purpose of only declaring c as an integer, whereas a and b are typed as Variants. Due to this reason the participants favor avoiding multiple declarations in one line.

Variable Assignment (2 rules): Participants consider Variable Not Assigned, and Unassigned Variable Usage as problems, since VBA allocates default values to specific types of variables, and using such variables without initialization or assignment introduces unpredictability into the programs. As P9 states, "*I always initialize my variables... in a clear block of code, I declare and initialize my variables.*" Notably, for each of these rules an equal number of participants expressed their perceptions belonging to category 'B = Bad practices but fixing is optional', as that of category 'A = Serious problems that need fixing' (Table 4). Nevertheless, we chose to place these rules within group A, driven by the belief that in issues related to code quality, it is better to assume on the side of caution.

Unused Entities (2 rules): Participants view Function Return Value Not Used and Procedure Not Used as problems. In VBA, the difference between a sub-module or procedure ('Sub') and a function ('Function'), is the existence of a return value. Not using the return value defeats the purpose of a function, as it could then be simply written as a sub-module. Therefore, when a return value is not being used it could mean a bug due to a developer forgetting to use

it. Entire sub-module or procedure not in use is viewed as "sloppy", although some participants like P2, P6, P7, and P9 treat them as historical record, documentation, or candidates for reuse, lacking a better version control mechanism or library utilities in VBA. P9 uses a separate library module with several utility procedures that they include in every project, and uses only the procedures they need. Nevertheless, the popular opinion about unused procedures is unfavorable, and it is recommended not to have them in operational code.

Encapsulation (2 rules): Implicit Public Members are viewed as issues since members are public by default, and without an explicit access modifier, ambiguity is generated. Treated as a problem is also Write Only Property, which is a property with only mutator and no accessor. It is considered as a design smell, giving rise to confusion, and while not many are familiar with this relatively advanced concept in the VBA context, most of the participants who are, do agree on this point. However, P9 cautions against such a generic view citing that there can be actually a need to use such properties with only mutators—"*No... Some properties are only to be set, and that's kind of when you definitely want to use a set property (a property with only setter) instead of a variable,*" and provides an example where they want to set color of a user form, and passes the color as an object property, having only mutator.

Recorded Macro (1 rule): Using recorded macros in code is regarded as a problem, since they usually have unnecessary superfluous lines of code, and are generally considered "messy" or "ugly". According to participants, they are only suitable for demonstrating programming elements which developers cannot remember, like colors and formatting of Excel worksheets, or user forms. Typically, the participants recommend to copy the required lines of code from a recorded macro and then delete the macro.

4.2.2 B - Bad practices but fixing is optional. The participants perceive 7 of the 20 inspection rules as relevant in detecting issues they consider as bad practices, which are not critical and do not necessarily need fixing.

Unused Entities (3 rules): While unused entities like entire procedures or function return values are considered as serious problems, three other types of entities are considered okay to be left unused within code. Variable Not Used, Constant Not Used, and Parameter Not Used are thus considered rules that detect issues which most participants like P4 and P5 consider as mere "laziness" or "bad housekeeping" on developers' part. They do not believe extra variables lying around in code can be harmful from a serious point of view. P2 actually sees usefulness of leaving unused constants, for example constants that hold values representing different colors; only few of them are to be used at a time, but the rest are still useful to remain for future use. Once more, the lack of efficient library system in VBA forces users to take such crude measures ignoring the risks involved.

Bad Design (2 rules): Procedure Can Be Written As Function, and Parameter Can Be "ByVal" are two rules that are able to spot bad design on part of VBA developers, but according to participants, such design problems are not critical, and pose no immediate threat to the functioning of a VBA application.

Lengthy Sub-module (1 rule): Long sub-modules are considered by the participants as result of bad coding practice but not seriously harmful. P2, and P14 go further to state that occasionally, a sub-module can grow over time to become large, and this is not necessarily a problem as long as the sub-module is well-written with proper indentation, and comments.

Obsolete Statement (1 rule): Obsolete Global relates to the issue that the 'Global' keyword is now obsolete and 'Public' should be used in place of it [19]. Our participants did not see the use of Global instead of Public as an issue that is obligatory to fix.

4.2.3 C - Purposeful code elements rather than issues. Lastly, the participants insisted that 2 out of the 20 inspection rules detected issues that they do not perceive as issues. On the contrary, irrespective of potential or theoretical dangers, they viewed these issue types as necessary features or means to serve purposes.

Obsolete Call Statement - This issue relates to the persistent usage of obsolete keyword 'Call' [18]. P1-P4, P6, P7, and P11 insist that using the Call keyword in a statement improves code readability for them, as it explicitly communicates to a human reader that a sub-module or function is being invoked in a statement.

Autorun Macro - Autorun macros are sub-modules of code that are executed based on events, rather than through user interaction. They are theoretically considered dangerous due to their unpredictability, as occasionally they may not be triggered, and if a user is relying on an operation to be automatically executed, then they can have problems unless clear feedback is provided to them of the failure of the macro. They are usually executed based on events such as 'WorkbookOpen' i.e. on opening of an Excel workbook. Our participants opine that irrespective of the potential dangers, autorun macros are necessary to perform various operations commonly required in the VBA domain. As such, not all autorun macros should be flagged as issues by default during automatic code inspection.

4.3 Additional Issue Types

The following three issue types emerged during the interviews, which are not covered by existing inspection rules:

(1) Use Of Reserved Keywords as Identifiers - This pertains to naming of variables using keywords that resemble objects in Excel, like P5 explains "*user uses something like Dim cell as Range.*" 'Cell' is an Excel object name, and using it to name variables in VBA can give rise to confusion, and errors.
(2) Hard Coded Cell References - This occurs when VBA code is referring to an Excel workbook cell on the front-end directly through its location or address. The issue with this is the fact that the VBA code remains unchanged even if the structure of the Excel worksheet on the front-end changes, e.g. addition of columns or rows. Such a change results in the original cell to be shifted to a new address, but that update does not occur automatically in the VBA code in the back-end. P5, P9, P12, and P14 point out this issue and recommend usage of named ranges, or structured table references in the Excel front-end, and referring through those names in the VBA back-end.
(3) Code In Excel Object Modules - Any VBA project associated with Excel workbook has the Excel object modules by default such as Sheet-1, Sheet-2, Sheet-3 for each of its worksheets, and other such elements. Apart from that, the project structure contains a separate project module. It is not

good practice to add code in the Excel object modules, except event-handlers. All other code should be in the project module hierarchy. The reason is that occasionally Excel behaves in an unpredictable way and undesired outcomes are observed due to wrong execution of VBA program. For example, if a breakpoint is added, and is removed afterwards, Excel might still behave as if there is a breakpoint. In such cases the only way to resolve the problem is by deleting the whole module, and restarting the application. This can be easily done for any of the project modules, but not with the Excel object modules, since that would potentially break the Excel workbook structure.

5 REVISITING THE RESEARCH QUESTIONS

In this section, we revisit our research questions in the light of results obtained.

RQ1 ***How do VBA developers (who are end-users compared to traditional software developers) perceive automatic code inspection rules in terms of the issues they detect?***

From Sections 4.1 and 4.2, we see that VBA programmers perceive automatic inspection rules as relevant in terms of detecting issues that they consider as problems. 11 of the 20 rules studied are considered capable of detecting serious code quality issues. Particularly, in case of the 7 rules under *Group B - Bad practices but fixing is optional* (Section 4.2), where VBA programmers deem the issues detected as not critical, it is interesting to note how the relevance of the rules is affected by the lack of basic assumptions such as reliable library mechanism and robust automatic version control, which are valid in case of traditional software development.

RQ2 ***Do VBA developers also encounter issue types that are not covered by existing automatic code inspection rules?***

In Section 4.3 we describe three issue types that are not covered by existing inspection rules. In this case, it is not just the 20 rules we studied that do not capture them, but also the larger set of 67 rules featured in Rubberduck, which does not address them. Therefore, we consider this as an important finding which should be used to extend the set of rules available in static analysis tools for VBA code.

6 THREATS TO VALIDITY

We treat threats to validity of our results from the perspective of qualitative research [8] as follows.

6.1 Credibility

Various factors are liable to affect credibility of our results, such as the background knowledge and experience of the participants, and external factors affecting their perceptions. To mitigate this we have selected participants having diverse backgrounds such as Finance, Economics, Engineering, Architecture, Computer Science, etc. as shown in Table 3. Another such factor could be, for example, organizational policies or culture that introduce a bias in participants' perceptions of relative importance of issues, and consequently the relevance of corresponding inspection rules that detect them. To mitigate this, we have not limited ourselves only to participants from the single financial company from where we obtained the VBA applications, but have reached out to people from various organizations, including independents. Also during the interviews we put stress on the fact that we wanted the participants' own opinions, as independently as possible.

6.2 Transferability

Threat to transferability of our results arise from the fact that our sample population of participants is not large. However, in a qualitative study of exploratory nature like ours, we chose to focus more on descriptive details rich in contextual aspects rather than generalizability of our results. As part of future work we plan to study extensibility of our results by treating larger populations and different methods such as survey.

Another threat to transferability arises from the fact that we conducted our study on VBA. Other end-user programming languages exists, which could be interesting to study. However, as a starting step we decided to choose VBA due to its popularity, and close associations with popular end-user tool Excel.

6.3 Confirmability

Threat to confirmability of our results can stem from investigator bias, as all the interviews were conducted by the first author for consistency. This threat was mitigated as much as possible during the interviews by letting the participants speak as freely as possible without prompting, and repeated insistence to speak their mind freely. The same threat arises related to the analysis of the interviews, since the process of coding the interviews, and the categorization of perceptions is subject to our own interpretations, and the analysis was also carried out by the first author. Nevertheless, we tried to attain as much commonality as possible through repeated discussion and brainstorming between the authors.

7 RELATED WORK

Tómasdóttir *et al.* [27], conducted a similar study on JavaScript linter ESLint by conducting 15 interviews. They investigated how developers use ESLint. Their results describe the benefits that developers obtain when using ESLint, the different ways to configure the tool and prioritize its rules, and the existing challenges in applying linters in the real world.

Johnson *et al.* [16] investigated why some developers do not use ASATs to find bugs despite of their benefits proven in research. They also interviewed 20 developers to find that false positives and the manner in which warnings are presented, are the main barriers to usage. Since apart from a wrongly generated warning about an issue that does not exist, 'false positive' is also the case when a true warning is not perceived by a developer as an issue [2], Johnson's results inspire us further to investigate VBA developers' perceptions about the warnings generated by the inspection rules.

Christakis and Bird [4] also investigated developers' perceptions of ASATs and specifically which barriers they face in the adoption of these tools. By surveying 375 developers at Microsoft they found that the largest obstacle in adopting ASATs was the fact that some unwanted rules are turned on by default in the tools' configurations. They proposed having only a subset of rules enabled by default instead. We hypothesize similar situation can arise related to adoption of ASATs and automatic inspection in the VBA context, and therefore it makes our results further important in terms of

prioritizing the rules that should be enabled by default in the VBA context.

Related also is study by Vassallo *et al.* [28] who surveyed 42 developers and 11 industry experts to examine the role of development context in developers' perception of ASATs. They found different development context does affect how developers pay attention to different warning categories, and majority of their participants rely on specific factors such as, team policies and composition, when prioritizing which warnings to fix.

No related work has treated the context of end-user development and VBA. Particularly due to the findings above, related to developer perceptions, our insights into perceptions of VBA developers about automatic codes inspection, offers the possibility of reducing false positives, and developing VBA-specific prioritization strategies for inspection rules.

8 CONCLUDING REMARKS

In this paper, we have studied the relevance of automatic code inspection, a feature of automatic static analysis tools, in the context of end-user development in VBA. Specifically we have examined the relevance of existing automatic inspection rules according to how VBA developers perceive them based on the relative importance of issues they detect. Our results show that, automatic code inspection is relevant in an end-user development context such as VBA. 18 of 20 inspection rules studied, are capable of revealing code quality issues that VBA programmers consider as problems, and only two detect issues that they consider irrelevant. Among the 18 relevant rules, 11 have been considered as detecting serious problems that needs fixing, and the rest 7 have been considered detecting issues that bad practices, that do not mandate fixing. As such VBA programmers can adopt automatic code inspection tools to better maintain their VBA code. Moreover, VBA programmers who are inexperienced, or uninitiated to the concept of code quality, can now be warned and instructed to fix serious issues detected by the most relevant 11 rules, irrespective of their understanding of the issues or the potential dangers. Researchers and developers of code inspection tools can focus their efforts in refining the most relevant 11 inspection rules. Our results could also help them formulate prioritizing strategies for the rules.

In addition, our study also demonstrates the need for new inspection rules that are specific to usage of VBA within the Excel context, which relate to references to Excel objects such as cells and worksheets.

As future work, there are possibilities to expand upon our results by analyzing larger populations for corroborating our findings using quantitative methods like surveys. Also it will be interesting to analyze how effective automatic code inspection is in terms of preventing actual faults or bugs. Through pursuing such research directions, our aim is to help end-users like VBA programmers, in using static analysis tools and automatic code inspection to ensure better code quality and maintainability of their applications.

REFERENCES

[1] 1990. IEEE Standard Glossary of Software Engineering Terminology. *IEEE Std 610.12-1990* (Dec 1990), 1–84. https://doi.org/10.1109/IEEESTD.1990.101064

[2] Nathaniel Ayewah, William Pugh, J David Morgenthaler, John Penix, and YuQian Zhou. 2007. Evaluating static analysis defect warnings on production software. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*. ACM, 1–8.

[3] Moritz Beller, Radjino Bholanath, Shane McIntosh, and Andy Zaidman. 2016. Analyzing the state of static analysis: A large-scale evaluation in open source software. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*. IEEE, 470–481.

[4] Maria Christakis and Christian Bird. 2016. What developers want and need from program analysis: an empirical study. In *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*. IEEE, 332–343.

[5] John W Creswell. 2013. *Research design: Qualitative, quantitative, and mixed methods approaches*. Sage publications.

[6] Ian F Darwin. 1988. *Checking C Programs with lint.* " O'Reilly Media, Inc.".

[7] EuSpRIG. [n. d.]. EuSpRIG Horror Stories. http://www.eusprig.org/horror-stories.htm

[8] Egon G. Guba. 1981. Criteria for assessing the trustworthiness of naturalistic inquiries. *ECTJ* 29, 2 (01 Jun 1981), 75. https://doi.org/10.1007/BF02766777

[9] Felienne Hermans, Bas Jansen, Sohon Roy, Efthimia Aivaloglou, Alaaeddin Swidan, and David Hoepelman. 2016. Spreadsheets are Code: An Overview of Software Engineering Approaches Applied to Spreadsheets. In *2016 IEEE 23rd International Conference on Software Analysis, Evolution, and Reengineering (SANER)*, Vol. 5. 56–65. https://doi.org/10.1109/SANER.2016.86

[10] Felienne Hermans, Martin Pinzger, and Arie Van Deursen. 2011. Supporting professional spreadsheet users by generating leveled dataflow diagrams. In *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 451–460.

[11] Dietmar Jannach, Thomas Schmitz, Birgit Hofer, and Franz Wotawa. 2014. Avoiding, finding and fixing spreadsheet errors–a survey of automated approaches for spreadsheet QA. *Journal of Systems and Software* 94 (2014), 129–150.

[12] Bas Jansen, Felienne Hermans, and Edwin Tazelaar. 2018. *Detecting and Predicting Evolution in Spreadsheets: A Case Study in an Energy Network Company.*

[13] Ciera Jaspan, I Chen, Anoop Sharma, et al. 2007. Understanding the value of program analysis tools. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*. ACM, 963–970.

[14] Jetbrains. [n. d.]. IntelliJ IDEA. https://www.jetbrains.com/idea/

[15] Jetbrains. [n. d.]. ReSharper. https://www.jetbrains.com/resharper/

[16] Brittany Johnson, Yoonki Song, Emerson Murphy-Hill, and Robert Bowdidge. 2013. Why don't software developers use static analysis tools to find bugs?. In *Proceedings of the 2013 International Conference on Software Engineering*. IEEE Press, 672–681.

[17] Henry Lieberman, Fabio Paternò, Markus Klann, and Volker Wulf. 2006. End-user development: An emerging paradigm. In *End user development*. Springer, 1–8.

[18] Microsoft. 2018. Call statement. https://docs.microsoft.com/en-us/office/vba/language/reference/user-interface-help/call-statement

[19] Stack Overflow user ThunderFrame. 2016. Global statement. https://stackoverflow.com/questions/39639488/make-a-global-variable-in-visual-basic-for-applications

[20] Raymond R Panko. 2000. Two corpuses of spreadsheet errors. In *System Sciences, 2000. Proceedings of the 33rd Annual Hawaii International Conference on*. IEEE, 8–pp.

[21] Raymond R Panko and Salvatore Aurigemma. 2010. Revising the Panko–Halverson taxonomy of spreadsheet errors. *Decision Support Systems* 49, 2 (2010), 235–244.

[22] Raymond R Panko and RP Halverson. 1996. Spreadsheets on trial: A survey of research on spreadsheet risks. In *System Sciences, 1996., Proceedings of the Twenty-Ninth Hawaii International Conference on*,, Vol. 2. IEEE, 326–335.

[23] Sohon Roy, Felienne Hermans, and Arie van Deursen. 2017. Spreadsheet testing in practice. In *Software Analysis, Evolution and Reengineering (SANER), 2017 IEEE 24th International Conference on*. IEEE, 338–348.

[24] Caitlin Sadowski, Jeffrey Van Gogh, Ciera Jaspan, Emma Söderberg, and Collin Winter. 2015. Tricorder: Building a program analysis ecosystem. In *Proceedings of the 37th International Conference on Software Engineering-Volume 1*. IEEE Press, 598–608.

[25] Christopher Scaffidi, Mary Shaw, and Brad Myers. 2005. Estimating the numbers of end users and end user programmers. In *Visual Languages and Human-Centric Computing, 2005 IEEE Symposium on*. IEEE, 207–214.

[26] Anselm Strauss and Juliet Corbin. 1998. *Basics of qualitative research: Techniques and procedures for developing grounded theory* . Sage Publications, Inc.

[27] Kristín Fjóla Tómasdóttir, Mauricio Aniche, and Arie Van Deursen. 2018. The Adoption of JavaScript Linters in Practice: A Case Study on ESLint. *IEEE Transactions on Software Engineering* (2018).

[28] Carmine Vassallo, Sebastiano Panichella, Fabio Palomba, Sebastian Proksch, Andy Zaidman, and Harald C Gall. 2018. Context is king: The developer perspective on the usage of static analysis tools. In *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 38–49.

[29] Carmine Vassallo, Fiorella Zampetti, Daniele Romano, Moritz Beller, Annibale Panichella, Massimiliano Di Penta, and Andy Zaidman. 2016. Continuous delivery practices in a large financial organization. In *Software Maintenance and Evolution (ICSME), 2016 IEEE International Conference on*. IEEE, 519–528.

[30] Stefan Wagner, Florian Deissenboeck, Michael Aichner, Johann Wimmer, and Markus Schwalb. 2008. An evaluation of two bug pattern tools for java. In *2008 International Conference on Software Testing, Verification, and Validation*. IEEE, 248–257.

[31] Fadi Wedyan, Dalal Alrmuny, and James M Bieman. 2009. The effectiveness of automated static analysis tools for fault detection and refactoring prediction. In *Software Testing Verification and Validation, 2009. ICST'09. International Conference on*. IEEE, 141–150.