# Developing Applications to Automatically Grade Introductory Visual Basic Courses

*Full Paper*

**John Gerdes**
University of South Carolina
jgerdes@sc.edu

## Abstract

There are many unique challenges associated with introductory programming courses. For novice programmers, the challenges of their first programming class can lead to a great deal of stress and frustration. Regular programming assignments is often key to developing an understanding of best practices and the coding process. Students need practice with these new concepts to reinforce the underlying principles. Providing timely and consistent feedback on these assignments can be a challenge for instructors, particularly in large classes. Plagiarism is also a concern. Unfortunately traditional tools are not well suited to introductory courses.

This paper describes how AppGrader, a static code assessment tool can be used to address the challenges of an introductory programming class. The tool assesses student's understanding and application of programming fundaments as defined in the current ACM/IEEE Information Technology Curriculum Guidelines. Results from a bench test and directions for future research are provided.

**Keywords**

Assessment, Visual Basic, Plagiarism, Design Science, Automatic Grading System.

## Introduction

Teaching introductory programming classes presents unique challenges, particularly in larger classes. Student have to learn how to work in a new development environment; become familiar with the programming language's vocabulary and syntactical rules; develop the skill to decompose problems into simpler processes, and develop the logic to address those processes. Students new to programming often struggle with these concepts, and consequently require increased personalize attention from the instructor. Regular programming assignments allow students to gain proficiency with these skills [Brito and de Sá-Soares 2014]. An important component in the learning process is the assessment of these assignments and the timely, consistent feedback provided to the student. Evaluating student work is complicated by the fact that it is often possible to use different approaches to accomplish the same result. The displayed result may be less important than the approach used to reach that result. To assess student work, the instructor must analyze and understand the logic used by each student and then provide individualized feedback relevant to the approach used.

The internet is a useful resource to find help with coding issues, however code snippets found on the Internet can often add to student confusion because they may utilize a different, incompatible approach to the problem than the one being presented by the instructor. Without a core foundation to understand the subtleties of each approach, students can become more confused which leads to added frustration.

If students are not adequately motivated to spend the time to learn the material, they may be tempted to plagiarize the work of others. The issue is complicated by the fact that students may be encouraged to work together on assignments but submit their own work. The Internet also is also a significant source of plagiarism. An Internet search will often find multiple solutions to typical assignments given in introductory programming classes. If a student just copies the code without understanding the logic, the learning objectives are not being met.

There is extensive research on code assessment automation and the identification of code plagiarism. However, most of the automated assessment literature focuses on assessing moderate to complex applications, and doesn't concentrate on fundamental programming concepts addressed in introductory courses. Similarly, the plagiarism literature focuses on methods of identifying code similarity, which is not effective in introductory classes. Assignments in these classes have very low program complexity, and the degree of code similarity is quite high. In addition, the very little research focuses on plagiarism detection in Visual Basic applications (most of it deals with programming in C, C++ and Java).

Contributions of this work are three fold. First, it presents a tool designed to automate the static assessment of Visual Basic programs, analyzing the code for required programming elements, and provides a score based on the level of compliance with the assignment's requirements. The coverage of Visual Basic source code is significant because there is little published work that focuses on this language. Second, it provides two different plagiarism checks not previously discussed in the literature. These two approaches are effective even when there is a high degree of similarity of submitted work as typically seen in introductory programming classes. The third is the open source availability of the AppGrader tool described (available through GitHub - *https://github.com/ProfGerdes/AppGrader*).

The paper continues with a review of prior research. We then discuss the design concepts involved with this tool. This is followed by results of bench trials. The paper concludes with a summary of the work and directions for future research.

# Prior Work

Automatic Assessment of software has received a great deal of attention over the last 30 years. Benefits of adopting automated assessment tools include more timely feedback, and the ability to assess more items (Malmi et al. 2002), improved grading consistency, objectivity, accuracy, and efficiency (Jackson and Usher 1997; Gupta and Dubey 2012; Fuyun and Wenjuan 2010; Poženel, et al. 2015), and allowing instructors to be more efficient by focusing on tasks that cannot be graded automatically (Blumenstein, et al., 2004; Poženel, et al., 2015). The interested reader is directed to the various literature surveys that have been published on the subject (Douce, et al. 2005; Ala-Mutka 2005; Liang, et al. 2009; Ihantola, et al. 2010).

There are two complimentary approaches used to automate software assessment, namely dynamic and static testing. Dynamic testing analyzes the performance of the running application using a set of test cases. The test application results are then captured and compared against the target answers for each test case. Static testing on the other hand uses code understanding and semantic analysis to assess the submitted source code files. Software plagiarism is also an important part of static analysis, and has long been an issue with software development classes. There is extensive literature on different approaches to assess the degree of similarity between software applications. Each of these topics is discussed below.

### Dynamic Testing

Dynamic testing does not depend on the source code, and thus does not know how the application operates. It deduces the correctness of an application by running it against a set of test cases and comparing the resulting output against expected results. If the target application supports scripting, test cases can be run using this script mechanism. Another common approach is to develop an application which uses API calls to interact with the target application and record and compared results against the target solution [Fei, et al., 2012]. A limitation of this approach is that not all user interface elements are accessible through standard API calls. A third approach addresses this limitation by using accessibility features found in the underlying software development environment. These features are meant to provide access to assistive technology such as screen readers used by the blind or visually impaired. These assistive technologies typically provide access to all the user interface objects.

Dynamic testing does have its drawbacks [Ala-Mutka, 2005]. If code does not run due to software bugs, the application cannot be assessed. Even when the code does run, it can only assess if the result is correct without any measure of how it reached that result. Also, running the code exposes the test machine to the possibility of malicious code. Programs not intentionally malicious can still cause severe damage to the host system (i.e., misdirected delete operations could cause permanent loss of data).

### Static Assessment

Static assessment focuses on reviewing and evaluating the code without actually executing it [Ala-Mutka, 2005]. While static analysis does not capture the dynamic or timing related aspects of the code, it can capture and assess elements related to coding style, programming errors, software metrics, and user interface design. It can also perform various plagiarism checks. Static analysis can collect various Software Metrics which capture characteristics of the source code such as module length, number of code lines, and number of variables to assess the complexity of the code.

Alternative approaches have also been proposed to do Static testing. Most modern development environments incorporate code checkers that automatically catch design time syntax errors as the code is being written. For example Microsoft's Visual Studio will catch declaration errors, and design times syntax errors. There are various external code checkers that can evaluate source code [Louridas, 2006]. Graph based approaches have also been used in static assessment, including the use of program dependence graphs [Korel, 1987], and UML collaboration diagrams [Abdurazik and Offutt, 2000].

### Source Code Plagiarism

Source code plagiarism can be defined as trying to pass off all or significant parts of source code written by someone else as one's own without proper attribution [Hage, et al., 2010, Bin-Habtoor and Zaher, 2012, Cosma and Joy, 2008]. It is relatively easy to make modifications to a plagiarized application so that it has a different visual appearance [Bin-Habtoor and Zaher, 2012]. This is true of both the user interface as well as the source code.

The standard approach to testing for plagiarism is to do pairwise comparisons and look for similarities [Ganguly, 2014]. Alternative approaches have been used to screen for software plagiarism, including graph based analysis based on source code feature extraction [Chae, et al. 2013]; comparing software birthmarks (characteristics extracted from the software that uniquely identify the program) [Tian, et al. 2015, Tian, et al. 2016]; binary code comparison [Luo, et al. 2017]; and an information retrieval approach which parses source code to compare the application's structural nature [Ganguly, 2014]. Additional approaches are detailed in published plagiarism surveys. [Kowaltowski, 2010, Martins, et al. 2014]

Unfortunately, these traditional plagiarism techniques are not well suited for introductory programming courses. Their effectiveness is based on the ability to identify core differences in the code, however the simple nature and inherent similarity of application developed in an introductory course would likely find a high level of similarity across all student applications. Things like object and variable naming conventions may be specified by the book or instructor. Coding assignments are likely designed to provide practice with specific programming concepts, which severely constrains differences in program logic. These factors result in traditional plagiarism tests reporting very high percentages of false positives.

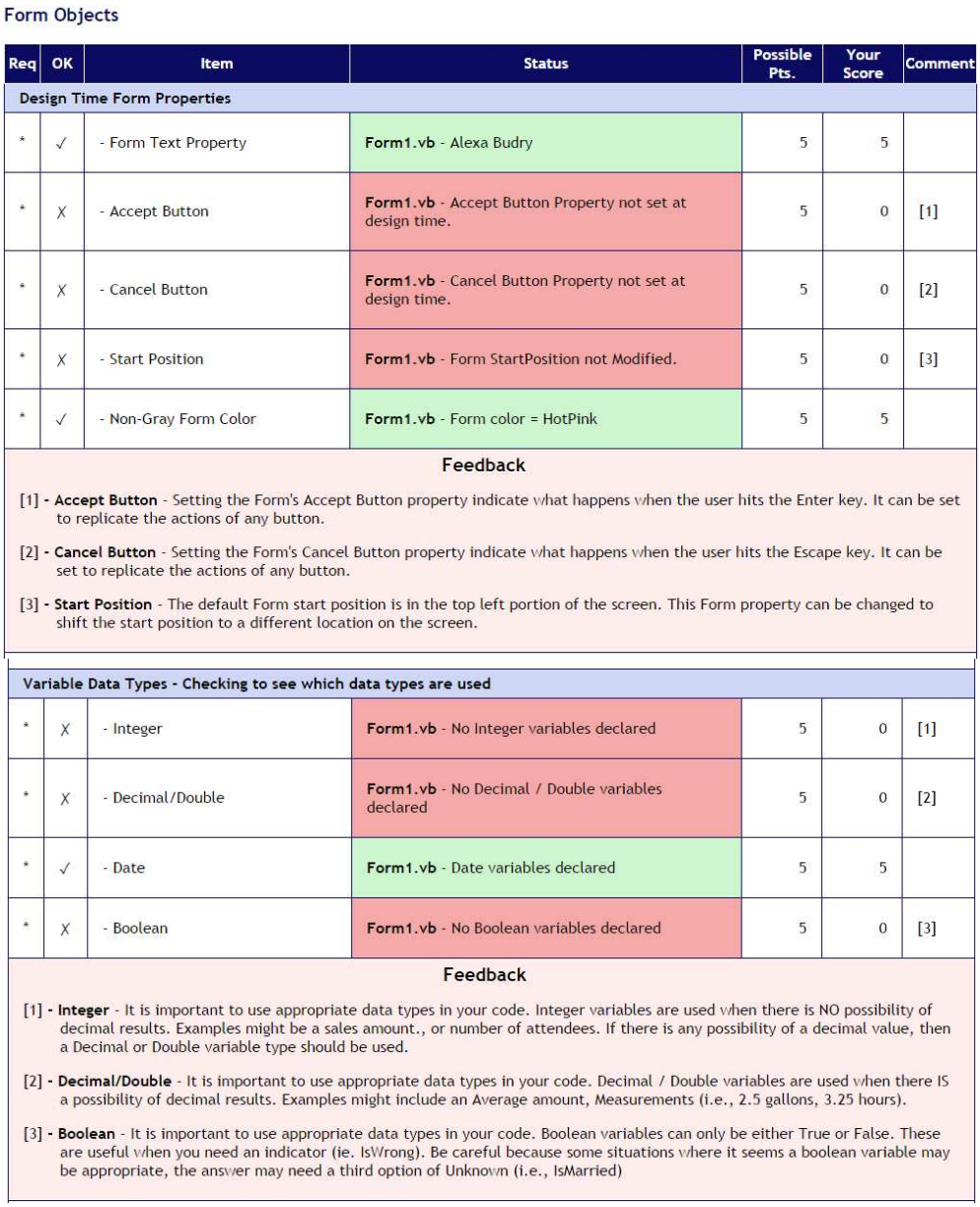
## AppGrader Design Concepts

The AppGrader application is designed to assess Visual Basic applications. It is configurable by the instructor to match the learning objectives of the assignment. Providing too much feedback would likely be confusing to novice students and reduce the usefulness of the tool. For this reason the AppGrader allows the instructor to specify the items to be assessed, and the weight for each item. For a '*Hello World*' assignment, the instructor may only specify that there needs to be a Label, Textbox, and Button objects on the form, and ensure that the whole application folder is submitted properly. As more concepts are covered in the class, the grading template can be adjusted to assess additional items.

The AppGrader uses static code analysis to assess the application. It is designed to objectively and consistently evaluate the structural aspects of the code, and determine if best practices have been followed. It also automates the creation of feedback to the students for identified issues. This allows the instructor to focus on the application's program logic.

The AppGrader analyzes each form designer.vb file to determine which objects have been defined along with their properties. It then processes each class file associated with the application, processing this file line by line against the criteria specified by the instructor for the assignment. Each line of code is indexed so that reference line numbers can report when code issues are identified. It then removes any whitespace

in the code to facilitate the checking of code comment placement. Code is then analyzed to assess the program logic to see if the user incorporated required elements, and if they utilized proper commenting throughout. In each case, the application counts how many times specific VB command word are used, and also how many issues it identified. This information is then processed and reported on a summary report. The sample output in Figure 1 illustrates two areas of analysis – Setting object properties on the form, and declaring different data types in the class file, each with descriptive feedback for identified errors. The instructor specifies which settings to check depending on the assignment requirements.



Form Objects

| Req | OK | Item | Status | Possible Pts. | Your Score | Comment |
|---|---|---|---|---|---|---|
| **Design Time Form Properties** | | | | | | |
| * | ✓ | - Form Text Property | **Form1.vb** - Alexa Budry | 5 | 5 | |
| * | ✗ | - Accept Button | **Form1.vb** - Accept Button Property not set at design time. | 5 | 0 | [1] |
| * | ✗ | - Cancel Button | **Form1.vb** - Cancel Button Property not set at design time. | 5 | 0 | [2] |
| * | ✗ | - Start Position | **Form1.vb** - Form StartPosition not Modified. | 5 | 0 | [3] |
| * | ✓ | - Non-Gray Form Color | **Form1.vb** - Form color = HotPink | 5 | 5 | |

**Feedback**

[1] - **Accept Button** - Setting the Form's Accept Button property indicate what happens when the user hits the Enter key. It can be set to replicate the actions of any button.

[2] - **Cancel Button** - Setting the Form's Cancel Button property indicate what happens when the user hits the Escape key. It can be set to replicate the actions of any button.

[3] - **Start Position** - The default Form start position is in the top left portion of the screen. This Form property can be changed to shift the start position to a different location on the screen.

| Variable Data Types - Checking to see which data types are used | | | | | | |
|---|---|---|---|---|---|---|
| * | ✗ | - Integer | **Form1.vb** - No Integer variables declared | 5 | 0 | [1] |
| * | ✗ | - Decimal/Double | **Form1.vb** - No Decimal / Double variables declared | 5 | 0 | [2] |
| * | ✓ | - Date | **Form1.vb** - Date variables declared | 5 | 5 | |
| * | ✗ | - Boolean | **Form1.vb** - No Boolean variables declared | 5 | 0 | [3] |

**Feedback**

[1] - **Integer** - It is important to use appropriate data types in your code. Integer variables are used when there is NO possibility of decimal results. Examples might be a sales amount., or number of attendees. If there is any possibility of a decimal value, then a Decimal or Double variable type should be used.

[2] - **Decimal/Double** - It is important to use appropriate data types in your code. Decimal / Double variables are used when there IS a possibility of decimal results. Examples might include an Average amount, Measurements (i.e., 2.5 gallons, 3.25 hours).

[3] - **Boolean** - It is important to use appropriate data types in your code. Boolean variables can only be either True or False. These are useful when you need an indicator (ie. IsWrong). Be careful because some situations where it seems a boolean variable may be appropriate, the answer may need a third option of Unknown (i.e., IsMarried)

**Figure 1: Sample feedback related to Form Object property settings and Variable Usage, with descriptive feedback provided for tasks done incorrectly.**

## *Preprocessing of the Application Files*

An important 'book keeping' feature of the AppGrader is the ability to automate the preprocessing of the student application files and prepare them to be graded. This process is time consuming, and if not done properly can prevent the execution of the student code. For example, students submit zipped files

containing their work to an assignment link in Blackboard. The instructor can batch download a zip file containing all student work. The student's files must be extracted by recursively decompressing all of the individual zip files. In a class of 40 students, each submitting two applications, this amounts to unzipping 81 files (the one assignment file, containing 80 application files). The unzipping process places the files in separate directories. Unfortunately, the resulting directory paths can often exceed the operating system's maximum path length. Blackboard creates a unique filename for the assignment file that can exceed 60 characters. Similarly, each student's submission file length can also exceed 60 characters. Since the maximum path length in Windows is 260 characters, these long directory names can cause the overall maximum path length to be exceeded, which prevents student files from running. To address this problem, each folder path name must be adjusted.

AppGrader has the ability to automate both of these administrative tasks. It can also automate the copying of a grade template document into each of the student directories. When grading 40 student assignments, these 'book keeping' features save the instructor at least 2 hours of repetitive work.

## *Configuring AppGrader*

AppGrader assesses 79 concepts on submitted applications. Table 1 lists eleven broad areas that can be addressed. The **Development Environment** area includes checks for the application's SLN file and vbProj file as well as the version of visual basic used to create the application. The SLN and vbProj files are used to launch the student's application within Visual Studio, and therefore are needed to fully assess the submitted work. Knowing the student's software IDE (integrated development environment) version can be relevant due to version differences in the software.

| | | |
|---|---|---|
| Development Environment | Form Design | Data Types |
| Application Information | Form Objects | Coding Constructs |
| Compile Options | Imports | Subs / Functions |
| Comments | Data Structures | |

**Table 1. Eleven issue categories addressed by AppGrader**

The **Application Information** area checks whether the student's submission includes a splash screen and an About Box in the application. It also checks to see if the application's properties associated with the application have been updated. These properties include the application title, description, product name, company, trademark and copyright information. The **Compile Options** area provides checks to see if option strict and option explicit have been set to on in the code.

The **Comment** area checks to see if Subroutines and Functions, as well as If, For, Do, While, and Select Case statements have been commented. The application looks for comments that are either preceding, on the same line, or immediately following each of these logical constructs. It does not assess the comment content, only if it is present or not.

The **Form Design** area addresses the properties related to each of the forms in the student's application. This includes checking to see each of the form as well as all objects on the form have been renamed to use standard prefixes for that object. It checks various form properties to see if specific user interface design elements have been implemented. It check to see if the form text has been modified, and if the form background color has been modified from the default gray color. It reports the form start position. In addition it reports if the AcceptButton and CancelButton properties have been set by the student. The **Form Objects** area provides the option to verify if the submitted application includes twelve common object types in the form design. This includes buttons, labels, textboxes, combo boxes, list boxes, radio buttons, check boxes, group boxes, open file dialog, save file dialog, and web browsers. It has the ability to distinguish between static labels which do not change while the application is running, and labels which have their text properties changed programmatically. This is significant because traditional coding practices suggest that only objects that are interacted with programmatically need be renamed with object specific prefixes.

The **Imports** area provides checks for three common name space imports statements, namely Imports System.IO, Imports System.Net, and Imports System.DB. The **Data Structures** area has checks to see if Arrays, Lists and Structures are declared in the application. The **Data Types** area provides mechanisms to determine if different data types are declared in the student's application. It also can check if data type specific prefixes are used when declaring the variables.

The **Coding** area provides check for various common programming constructs. This includes looping constructs (For, Do, While and nested For statements), branching statements (If, Else, ElseIf, and Select Case statements). It checks for the use of MessageBoxes, Try … Catch blocks, Open File Dialog, Close File Dialog, ScreenReader and ScreenWriter, as well as the matching close statements needed to terminate the file IO. It verifies that the student has used string concatenation, commands to convert numbers to strings as well as formatting those strings. It also checks for complex logical checks involving two or more operands (i.e., it looks for AND or OR statements in statements requiring logical checks).

Finally, the **Subs** area checks to see if the student application includes any subroutines or functions that are not form object methods. It checks to see if Optional or ByRef variables are specified in the manually created Subroutines or Functions. It determines if the application includes multiple forms, and if it includes a module. It also checks if a Form Load method has been defined for each of the application forms.

To use the AppGrader, the instructor must specify which concepts are to be assessed and the weight that should be applied to each concept. For each assessment item the instructor can indicate the maximum weight for that item as well as the number of points for each error. So for example, the assignment may require three buttons on the form, each worth 3 points for a maximum of 9 points. Improper commenting of the code might be worth 2 points each, with a maximum deduction of 10 points. If the student must use at least one nested IF statement, the individual and maximum point values could be set to 5 points each. If the point values are set to zero, then errors are flagged, but do not impact the overall grading of the assignment. Alternatively, items that are not specifically required can be set to be without grading. Once the instructor sets up the grading configuration to meet the requirements of the assignment, the settings are saved in a configuration file that will be loaded when the student assignments are graded.

## *Methodology*

To assess student applications, the instructor specifies grading weights and aspects to be assessed in a configuration file. Then AppGrader parses each of the *.vb files in the student submission to extract information related to form objects and student generated code. The extracted information is then mapped against the instructor's specifications which results in the overall assessment of the application. HTML/CSS reports are generated for each student, along with summary files for the instructor.

As defined in the current ACM/IEEE Information Technology Curriculum Guidelines [ACM/IEEE 2008], the Information Technology Body of Knowledge related to Programming Fundamentals includes:

- Fundamentals of Data Structures
- Fundamentals of Programming Constructs
- Object-Oriented Programming
- Algorithms and Problem Solving
- Event-Driven Programming.

AppGrader supports these concepts by automating the automating the assessment of student's work, thereby improving responsiveness, accuracy, and consistency in grading. The application assesses if the student has followed good programming practice with variable and object naming, along with code commenting. It checks for the use of programming concepts, such as IF … THEN, IF … THEN … ELSEIF, SELECT CASE, FOR … NEXT, DO WHILE, and DO UNTIL statements. AppGrader also reports on human computer interface (HCI) issues such as proper naming of form objects, as well as the inclusion of specified object types such as TextBoxes, Labels, Buttons, Listboxes, and ComboBoxes. It can also check for compliance with ADA (American's with Disability Act) Section 508 color contrast compliance of the user interface (testing if the foreground/background color meets the specification).

### *Assessment Process*

AppGrader can be used to assess either a single application, or a group of assignments in batch mode. In batch mode, the application accepts a zip file of compressed student submissions. Prior to assessing these files, all the student files must be extracted so that they can be processed individually. During this process an attempt is made to automatically shorten the folder names to avoid running into the maximum path length constraint. Simply unzipping these files can result in directory paths that exceed the maximum lengths allowed in windows. The program addresses this issue by renaming the zip files to just reflect the student ID.

After the preprocessing phase, the instructor either configures the application settings or loads the configuration file, then clicks a button to begin the assessment process. The program recursively processes each student's submission. Depending on the criteria, one of two different assessment modes is used. First, the instructor can specify that a certain element or construct must be used a specific number of times. AppGrader will count how many times this construct is used and grade it accordingly. Examples would be the inclusion of the SLN file, inclusion of 3 buttons on the form, use of an IF statement, and declaring of a 5 Boolean variables.

The second approach allows the instructor to deduct for improper coding practice – that is failure to follow best practice guidelines. Examples include inadequate commenting, and not closing file IO connections. In these cases that instructor can indicate the number of points to be deducted for each infraction, along with a maximum deduction for that criterion. The combination of these two approaches allow the instructor to monitor if the desired programming practices are being utilized in the application, while also monitoring when best practices are not being followed.

The program creates two reports for each student assignment, as well as two additional reports which summarize the assessment process for the instructor. The first student report analyzes and reports assessment information on a file by file basis. It assesses each of the criteria specified by the instructor for the assignment, provides a grade for the element, and also provides feedback based on the issues identified. The second student report provides an integrated assessment across all files in the application. For each criteria, it reports each issue observed in each file contained in the student's submission. It also provides an overall numerical assessment of the student's work.

Both a detail and summary report is generated for the instructor. The detail report is a combination of each of the student summary reports into a single file. A faculty summary report provides just the summary statistics for each of the student submissions. This includes the time and date of the submission, the number of lines of code (not including comments), and also the overall numerical assessment score.

### *Plagiarism Checking*

Two mechanisms are used to check for plagiarism. Because of the inherent similarity of these introductory programming assignments, the effectiveness of traditional plagiarism checking tools is limited. AppGrader uses two different techniques that check for identical signatures across the student submissions rather than just close matches.

The first approach is to determine the MD5 hash for the individual files, and compare these values across all students files. A MD5 hash is derived from a message digest algorithm used to verify data integrity. It creates a 128 bit value based on the whole file. Typically this value is expressed as a 32 digit hexadecimal number. Any differences in two files will result in a different hash value, with the odds of two random files having the same hash value being 1 in $3.4 \times 10^{38}$. Identical MD5 hash values strongly indicates that the indicated students submitted identical files.

Note that the MD5 algorithm has been found to be susceptible to attack, where the attacker can generate two files with identical MD5 checksums [Wang and Yu, 2005, Black and Cochran, 2006]. This vulnerability is not a concern for the AppGrader application for two reasons. First, unlike a traditional data validation application, the expectation is that each file is different, and therefore has different checksum values. There is no incentive to specifically craft a file to have the same checksum as a different file submitted by another student. Doing so would likely not perform properly. Second, the plagiarism checks performed by AppGrader are designed to highlight potential instances of plagiarism. It is

recommended that the specific files be manually checked by the instructor to verify if the files are indeed the same. There are special cases where having identical files does not indicate plagiarism was involved. One such case is where students have access to sample code or a custom library provided by the instructor. Inclusion of this code would not necessarily indicate collusion or plagiarism. Similarly, students can end up with identical class files if they include a special purpose form in the application, such as a Splash Screen or About Page that does not require any modification of the form logic.

The second plagiarism checking approach is to extract the application's GUID (Global Universal Identifier) embedded in the application when it was created in Visual Studio. As its name implies, this identifier is meant to uniquely identify each application no matter where and when it was created. The likelihood of two applications independently having the same GUID is 1 in $5.3 \times 10^{36}$. Unlike the MD5 checksum, the GUID is independent of the code in the file. If an application is copied, the GUID will be identical even if every bit of code is modified. So this approach will catch plagiarism even in cases where the student actively tries to disguise it by changing the appearance of the user interface, modifying the text in comments, renaming variables or reordering program logic.

## Bench Tests

AppGrader's performance was assessed based on two scenarios. The first represented a typical homework submission for an introductory programming class. Students submitted a zipped copy of their complete development folder through Blackboard. This work was subsequently bulk downloaded as a zip file containing the 33 individual zipped submissions. AppGrader extracts each application, analyses the code, generates individualized reports, and checks for plagiarism. The processing results for this case are given in the second column of Table 2. AppGrader generated 33 grade reports (one for each student submission), two summary reports for the instructor, and a plagiarism report. The analysis was run 10 times to test the variability if program execution time.

| | **33 Different Apps Average (std dev.)** | **Single App Average (std dev.)** |
|---|---|---|
| Number of applications analyzed | 33 | 1 |
| Submission Lines of Code (excluding comments) | 71.07 (25.37) | 3,137 |
| Number of instructor sample applications | 188 | 188 |
| Number of code files in instructor applications | 1,748 | 1,748 |
| Timing Results | | |
| Initialization | 0.08 (0.02) sec | 0.08 (0.01) sec |
| Load instructor application data | | |
| Application GUID | 0.73 (0.07) sec | 0.79 (0.10) sec |
| MD5 of all code files | 0.99 (0.14) sec | 0.83 (0.01) sec |
| Unzip student submissions | 4.41 (0.75) sec | - |
| Assess student submissions | 2.26 (0.29) sec | 0.84 (0.07) sec |
| Load MD5 & Perform plagiarism checks | 0.07 (0.01) sec | 0.03 (0.00) sec |
| **Overall time** | **8.54 (1.00) sec** | **2.56 (0.16) sec** |

**Table 2: App Grader operational results based on 10 iterations**

The second test represented the assessment of a single application, namely the source code for AppGrader. This test simulates the case where a student uses the tool during program development. This would allow the student to identify issues in the code and correct them prior to submitting the work for a grade. Processing results are given in third column of Table 2.

## Summary and Directions for Future Research

In this paper we discuss the capabilities of a tool specifically designed to aid in the assessment of novice Visual Basic programmers using a static analysis approach. This tool is unique in its focus on Visual Basic as opposed to other programming languages such as C, C# or Java. It is capable of assessing 79 common coding elements important in introductory programming applications.

An innovative feature of the tool is its approach to plagiarism checking. Traditional plagiarism checkers are ineffective on programs written in introductory courses because of the simplicity of the applications. This tool uses a MD5 message digest to look for exact matches of the student source code between student submissions. It also compares application GUIDs (Globally Unique ID) to identify instances of student plagiarism. Using the GUID will identify instances where the students started with a copy of another application, even if the code and user interface are significantly changed. The author is not aware of this approach being discussed in the literature. Experience has shown that both of these mechanisms have been effective in identifying instances of plagiarism in student submitted work.

AppGrader's limitations include the inability to: identify code syntactical errors, evaluate source code correctness, and aid with debugging. Fortunately, Visual Studio has tools which aid the identifying design-time syntactic error, along with features like break points and watch variables that aid with code tracing and debugging program logic. The current version of AppGrader cannot remotely execute the student code. This planned feature would permit unit testing as well as logging of runtime errors.

This work focused on static analysis of student applications. However, to fully assess the proper operation of the applications requires dynamic testing. While there are various tools designed to do dynamic testing of applications, they are typically designed to test a single application on a suite of test cases, and not a large set of similar applications with slight variations in the user interface. This is a critical need for the support of introductory programming classes, for it would the instructor to assign more programming assignments, and provide more timely, detailed feedback to the student.

## REFERENCES


Aaltonen, K., Ihantola, P., and Seppala, O. 2010. "Mutation Analysis vs. Code Coverage in Automated Assessment of Students' Testing Skills", *Proceedings of the ACM international conference companion on Object oriented programming systems languages and applications companion*, pp. 153 – 160.

Abdurazik, A., and Offutt, J. 2000. "Using UML collaboration diagrams for static checking and test generation". In *International Conference on the Unified Modeling Language*, pp. 383-395.

ACM/IEEE 2008. Information Technology 2008 Curriculum Guidelines for Undergraduate Degree Programs in Information Technology. Association of Computing Machinery (ACM), available at http://www.acm.org//education/curricula/IT2008%20Curriculum.pdf.

Ala-Mutka, K. M. 2005. "A survey of automated assessment approaches for programming assignments." *Computer science education* (15:2), pp. 83-102.

Bin-Habtoor, A. S., and Zaher, M. A. 2012. "A Survey on Plagiarism Detection Systems", *International Journal of Computer Theory and Engineering* (4:2), pp. 185-188.

Black, J., Cochran, M., and Highland, T. 2006. "A study of the MD5 attacks: Insights and improvements", In *International Workshop on Fast Software Encryption*, pp. 262-277.

Blumenstein, M., Green, S., Nguyen, A. , Muthukkumarasamy, V. 2004. "GAME: a Generic Automated Marking Environment for Programming Assessment", Conference on Information Technology: Coding and Computing (1), pp. 212-216.

Brito, M. A., and de Sá-Soares, F. 2014. "Assessment frequency in introductory computer programming disciplines". *Computers in Human Behavior (*30), pp. 623-628.

Chae, D., Ha J., Kim, S., Kang, B., and Im, E. G. 2013. "Software plagiarism detection: a graph-based approach". In *22nd ACM International Conference on Information and Knowledge Management*, pp. 1577 - 1580.

Cosma, G., and Joy, M. 2012. "An Approach to Source-Code Plagiarism Detection and Investigation Using Latent Semantic Analysis", *IEEE Transactions on Computers* (61:3), pp. 379-394.

Delev, T., and Gjorgjevikj, D. 2012. "E-Lab: Web based system for automatic assessment of programming problems", *Web proceedings ICT-Innovations*.

Douce, C., Livingstone, D. and Orwell, J 2005. "Automatic test-based assessment of programming: A review". *Journal on Educational Resources in Computing (JERIC)* (5:3), pp. 4.

Fei, T. P., Heng, L. Y., and Yun, Z. C. 2012. "Research of VB Programming Automatic Scoring Method Based on the Windows API", Proceedings of the International MultiConference of Engineers and Computer Scientists (1),

Fuyun, L., and Wenjuan, J. 2010. "The implementation of automatic scoring of computer operation of Visual Basic based on Windows Message Mechanism". In *Computer Science and Education (ICCSE),* pp. 1524-1526.

Gupta, S., and Dubey, S. K. 2012. "Automatic assessment of programming assignment". *Computer Science & Engineering* (2:1) 67.

Hage, J., Rademaker, P., and van Vugt, Nike 2010. "A Comparison of Plagiarism Detection Tools", *Technical Report UU-CS-2010-015*, Department of Information and Computing Sciences, Utrecht University, Utrecht, The Netherlands, 28.

Halstead, M.H. 1977. Elements of Software Science. New York: Elsevier North-Holland.

Hoare, C. A. R. 1969. "An axiomatic basis for computer programming". *Communications of the ACM* (12:10), pp. 576–580.

Ihantola, P., Ahoniemi, T., Karavirta, V. and Seppälä, O. 2010. "Review of recent systems for automatic assessment of programming assignments." In *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*, pp. 86-93.

Irving, R. W. 2004. "Plagiarism and collusion detection using the Smith-Waterman algorithm". *University of Glasgow*, 9, available at http://www.dcs.kcl.ac.uk/staff/mac/DOC/Irving-plagiarism.pdf.

Jackson, D., and Usher, M. 1997. "Grading student programs using ASSYST." In *ACM SIGCSE Bulletin* (29:1), pp. 335-339.

Korel, B. 1987. "The program dependence graph in static program testing". Information Processing Letters (24:2), pp. 103-108.

Liang, Y., Liu, Q. Xu, J. and Wang, D. 2009. "The recent development of automated programming assessment." In *Computational intelligence and software engineering,* pp. 1-5.

Louridas, P. 2006." Static code analysis". *IEEE Software* (23:4), pp. 58-61.

Luo, L., Ming, J., Wu, D., Liu, P., and Zhu, S. 2017. "Semantics-Based Obfuscation-Resilient Binary Code Similarity Comparison with Applications to Software and Algorithm Plagiarism Detection". IEEE *Transactions on Software Engineering*.

Malmi, L., Korhonen, A. and Saikkonen, R. 2002. "Experiences in automatic assessment on mass courses and issues for designing virtual courses." *ACM SIGCSE Bulletin* 34, no. 3, pp. 55-59.

Martins, V. T., Fonte, Fonte, D., Heriques, P. R., and da Cruz, D. 2014. "Plagiarism Detection: A Tool Survey and Comparison", *3rd Symposium on Languages, Applications and Technologies*, pp 143-158.

Mengel, S.A., and Ulans, J.V. 1999. "A case study of the analysis of the quality of novice students programs". In *Proceedings of the 12th Conference on Software Engineering Education and Training*, pp. 40 – 49.

Mozgovoy, M. 2006. "Desktop tools for offline plagiarism detection in computer programs", *Informatics in Education-An International Journal* (5:1) pp. 97-112.

Poženel, M., Fürst, L., and Mahnič, V. 2015. "Introduction of the automated assessment of homework assignments in a university-level programming course". In *Information and Communication Technology, Electronics and Microelectronics (MIPRO),* pp. 761-766.

Tao, G., Guowei, D., Hu, Q., and Baojiang, C. 2013. "Improved plagiarism detection algorithm based on abstract syntax tree". In *Emerging Intelligent Data and Web Technologies (EIDWT),* pp. 714-719.

Tian, Z., Liu, T., Zheng, Q., Tong, F., Fan, M., and Yang, Z. 2016. "A new thread-aware birthmark for plagiarism detection of multithreaded programs". In *Proceedings of the 38th International Conference on Software Engineering Companion*, pp. 734-736.

Tian, Z., Zheng, Q., Liu, T., Fan, M., Zhuang, E., and Yang, Z. 2015. "Software plagiarism detection with birthmarks based on dynamic key instruction sequences", *IEEE Transactions on Software Engineering* (41:12), pp. 1217-1235.

Wang, Z., Yu, H., and Wang, X. 2014. "Cryptanalysis of GOST R hash function", *Information Processing Letters* (*114*:12), pp. 655-662.