Inspecting and Improving Code Duplication with Rascal and Minecraft

Simon Baars
University of Amsterdam
simon.mailadres@gmail.com

Sander Meester
University of Amsterdam
sander.meester@student.uva.nl

***Abstract*—Diminishing the amount of duplicate code in a codebase improves the maintainability and prevents bug propagation. To do this, a developer needs a tool to firstly detect clones in a program, and an effective method to refactor the found clones. We propose a tool that use the Rascal MPL to calculate duplication in a codebase and the 3D sandbox game Minecraft to let the developer interact with the found code clones. This approach allows developers and maintainers to gain a quick overview of the amount of code duplication in the code, and improve it in an effective and engaging way. Allowing the developer to refactor the clones in Minecraft allows for many different possibilities, for example making the gamification part of fixing clones more interesting and allowing for different visual representations of the duplication metrics.**

I. Introduction

A significant part of the code in a large codebase consists of duplicate parts of code, either perfect copies or very close resemblance of another part of the code. On average, up to ten percent of the code consists of those clones, created when developers copy and paste parts of code to reuse it somewhere else [2]. Even though it is known to be bad practise, it is still common in codebases since one can easily create new code close to the behaviour of the original clone while being sure to keep the behaviour of the original code the same. To improve on maintenance costs it is beneficial to be able to detect these clones so they can be re-factored or deleted, and to become aware of the problems that can be caused by duplication. [2]

This paper contains an overview of our code duplication calculation and visualisation project done in *Rascal* and the 3D sandbox game *Minecraft*. This program is able to calculate multiple metrics on code duplication in a large codebase, and make a visualisation of duplication metrics found in the codebase in *Minecraft*. It also allows developers to immediately fix found duplicates.

II. Code Duplication

In our approach to code duplication, there were three main points we wanted to achieve.

1) Give the developer/maintainer an overview of duplication metrics of the codebase to show how big the problem is.
2) Help the developer/maintainer in tackling the amount of clone duplication in the code.
3) Confront the developer in an interactive way with the duplication problem of their code and motivate them to solve it.

We did not want to simply give an overview of the duplication metrics and leave the developers to go through his or her own code to fix duplication, separate from the visualisation. We already know where the clones are, so why not help the developer to immediately fix them. We want to prevent developers having to navigate through their entire codebase to find the clones we indicated, but simply allow them to fix them while going through the visualisation. This fixing has to be engaging and rewarding in some way to motivate the developer to put the time needed into it to decrease the code duplication in their code.

Section three elaborates further on the use for a maintainer or developer.

A. The algorithm

Our algorithm to detect code duplication is based on one of the algorithms stated by Baxter et al [2], but adapted heavily. The algorithm as described by Baxter turns the code into an AST representation and then makes buckets based on every subnode in the tree. Every item in a bucket is then compared to each other to check for duplicate lines.

In our algorithm, we use the same AST based approach to compare programs. For every node in an AST however, we return a simplified node representation (normalisation) based on the type of the node, and then create a list of all these "simplified nodes" that occur on a certain line in a file. For example:

```
case \methodCall(bool isSuper, str name,
    list[Expression] arguments) :
                            return [40, isSuper] + (t
                                == 1 ? [name] : []);
```

We see here that when we encounter a methodCall node in an AST, we return the number 40, along with the bool value of isSuper and if we want to check for type 1 clones, also the name.

When having created the simplified representation of an AST per line, we go over all files, and create a map of a hash

ICT.OPEN 2019 - Conference for Dutch ICT Research
19-20 March 2019, Gooiland, Hilversum, Netherlands
https://ict-research.nl/ict-open/

of those representations with the locations of all lines that also have that hash of their line. From this datastructure we can compare all lines to each other and save all sequences of at least 6 lines that are a duplicate of another block. These sequences are all stored together in their own clone class, represented as a list of lists. To differentiate between the clone types, we return different things from the AST representation function we made. The following pseudo snippets show an over-simplified representation of the algorithm. The actual algorithm is way more complex. This complexity is mainly due to optimisations. With infinite computing power the solution to the problem could've been computer with way more concise code.

Listing 1. Algorithm in pseudo-code

```
create map[int hash, list[loc] locations]
    locsAtHash
// has all lines with the same hash of
    their simplified AST.
create map[str fileName, list[int]
    lineNumbers] sortedDomains
// this is a sorted list of all lines that
    contain code in a sorted order.

for file in directory
    for line in file
      add line to hash in locsAtHash
    add line number to sortedDomains //
        connecting the line to the file

create list[tuple[int dupLength, list[loc]
    locations]] duplicateList
//every index in the list becomes a
    duplicate class

for every line in sortedDomains
    for identical_line in locsAtHash[line]
        check for chain and register if
            chain is found// A chain means
            that multiple lines in a row
            are a duplicate with multiple
            other lines in a row.
    for every chain of at least 6 lines //
        The "6" in is a parameter which can
        be tweaked as required.
        add number of lines and locations
            to the tempDupList
    for item in tempDupList (starting with
        longest duplicate)
        If no subsumption
            add to duplicateList
```

1) Optimisations: As the algorithm has to do complex computations over big sets of data. Because of this, the main work in the algorithm has been to implement various optimisations. The following optimisations are currently in place:

1) Normalise the AST of the project to compare on basis of a simplified representation of this AST.
2) Compute a hash over all tokens on each line (based on Java's String hash, which is very reliable) to speed up the process of finding duplicates of a certain line.
3) Calculate whether the next 6 (or whatever minimal duplicate size is chosen) lines contain at least x duplicates, where x is the amount of times this line is encountered. With this, **most** lines can be skipped because there's no way they are part of a duplicate group of a minimum of 6 lines. We call this "*checkFutureDuplicates*" as this actually predicts whether future lines have a possibility of being able of a duplicate group.
4) We create a registry of all duplicates found, which have the *uri* and *begin line* of the potential duplicate as a key. Using this, we can find chains (groups of duplicate lines) as an O(2n) algorithm as opposed to an O(n^2) algorithm.
5) Instead of working with the actual line numbers of lines, we use their indices in the sortedDomain. This is useful (and is good for performance), as the algorithm has to ignore all lines of code that do not contain Java source code (but comments, empty lines, or anything else that is set to be ignored). This way a domain is defined in which these ignored lines do not exist anymore to the algorithm, and no computing power has to be used to ignore them.
6) Further metric calculations are done lazily in Java. This way multiple threads can be in use in the calculation of the clones and their metrics.
7) Edits to the clones (because of a user correcting the clones via Minecraft) are done in a separate Rascal shell and thread, so they have no impact on the main calculation of duplicates.
8) ... and more registries and mechanisms are in place to optimise performance.

With these optimisations we try to trade CPU usage for RAM usage. As RAM usage for a code project is typically low (most Java projects are max 3 MB of pure source code) but CPU is scarce, we create many registries to avoid O(n^2) algorithms wherever possible (we have multiple instances where we replaced a O(n^2) algorithm with a O(2n) algorithm).

B. Types

We based our definition of different types on a survey by Roy and Cordy [7]. They state the following definitions for each clone type:

- Type I: Identical code fragments except for variations in whitespace (may be also variations in layout) and comments.
- Type II: Structurally/syntactically identical fragments except for variations in identifiers, literals, types, layout and comments.
- Type III: Copied fragments with further modifications. Statements can be changed, added or removed in addition to variations in identifiers, literals, types, layout and comments.

Our tool detects type one clones by going over all nodes of an AST and returning a list of simplified representations one the AST, wherein each number represents a type of node, possibly with names and bool values. These lists are hashed and every line with that hash is stored together. Since an AST representation already does not include whitespacing and layout (comments and anything that is set to be ignored), these do not influence the duplication found in any of the types.

To change to type two clones, we can simply change a parameter, and for every node in the AST that is not needed for type two clones (e.g. identifier names) our simplify function returns an empty list instead of the name. Using this method, we can select what things to not take into account when searching for type two clones and have full control over it. In line with the definition, we ignored identifiers, literals, and types.

For type three clones, we use the same simplified AST representation as for type two clones. We then use our similiarity calculation function to measure the amount of similarity between two AST's. To get this value, we measure the amount of nodes between to AST's that are different and divide it by the total number of nodes they have. This results in the function below that returns a value between 0 and 100, where 100 is no similarity whatsoever, and 0 meaning the two AST's are identical. This method of comparing for similarity and letting the user set a threshold on the amount of similarty between parts of code to consider it a duplicate is based on the method by Baxter et al [2]. Using our method a developer can decide for themselves how strongly they want to measure their code duplication. When set to 100, it will be the same as our type 2 checker, when set to 0, everything will be duplicate of everything. Using this, one can infer easily which parts of code are perfectly duplicated somewhere else, and which parts of code that differ slightly from each other, but are use almost the same in multiple places. One could for example set the threshold to 75% to find multiple functions that are very much alike, to check if these cannot just be one function with a single different parameter. As this number is a real, the developer can also specify a slight difference (0.5% for instance). Including this method in our program gives the most freedom for the programmer to decide what to look for in code.

```
public real calculateDifference(list[value]
    line1, list[value] line2){
  int differentElements = size(line1 -
      line2) + size(line2 - line1);
  int combinedSize = size(line1) + size(
      line2);
  return toReal(differentElements)/toReal(
      combinedSize) * 100;
}
```

III. Duplication Metrics Visualisation

For the visualisation of the code, we chose to use Minecraft, a 2011 sandbox video game where one can build with blocks whatever comes in mind. It lets us approach the codebase in a 3D world where the developer can interact with the world to infer what parts of the code need the most attention considering duplication of code. Our approach focuses on informing the developer in a fun way what parts of the code need attention, and try to increase willingness to improve code quality by gamification of the problem. Gamification of material increases commitment and motivation of students, especially if familiar with the concepts used in the game. Attitude towards the work is better when presented in the form of a game. [5]
There is also the fact that this form of visualising code metrics can make it easier for starting developers to relate to the problems their code has. The field of Software Engineering is a field that is growing rapidly [8]. In most country's there is a shortage of good Software Engineers [8]. Because of this reason, there are organisations that focus on teaching children in primary and secondary school the basics of software development [4]. However, the importance of Software Evolution is not yet taught to this target audience. This is unfortunate, because in industry there is more Software Evolution being conducted than actually developing a new software product [3]. However, the field of evaluating existent software is generally less accessible than software development, because software evolution requires all knowledge of that is required to develop software for a programming language, plus knowledge about best practises and metrics for this language.

To make the field of Software Evolution more accessible, we propose a solution in which the user can inspect and act upon different code metrics in Minecraft. This has also been done in a previous paper, in which G. Baloch and . Beszdes [1] look into creating a city on basis of the source code of a Java application. Where they only focused on displaying various code metrics, we propose a solution where the user can also deal with the code issues in a fun and playful way. This is the second benefit of the gamification in Minecraft.

Current visualisations are mainly focused on showing the developer where the duplication is in the code [6]. Our visualisation does the same, but with less control for the user where to look, giving that up to change the angle of approach to fix code duplication. We hope to make fixing code duplication fun and engaging, resulting in in the end more code duplicates being inspected and improved, because of the method we use. Instead of having to manually go over all files that the visualisation shows has duplicates and fixing them, our tool combines the two. We found the duplicates, so can show exactly where they are in the files, and pop open an editor to allow you to refactor them on the spot.

We use the high expressiveness of Minecraft to let developers really immerse themselves in the problems with their code and make it more concrete where their code needs improvement. User feedback on other similar approaches show that this playlike approach does increase the motivation to improve code [1]. We create a metaphor to the problems caused by code duplication to the enemies in Minecraft, the more duplication the code has, the more and bigger enemies appear one has to overcome. The biggest bonus of our visualisation is the fact that while inspecting their code, the developer gets to refactor their code. It is more than just an overview, and will help the developer improve the amount code duplication instead of only showing the problem.

A. Technical details

Extending the functionality of Minecraft can be done using Minecraft Forge. Minecraft Forge is primarily used as an API to extend Minecraft functionality without having to modify the source code of Minecraft itself. Minecraft Forge has listeners, registries, event handlers and more for various concepts in Minecraft. Using this, Java code can be used in a safe environment to build a so called Minecraft Mod (which is short for Minecraft Modification).

When trying to integrate Rascal with Minecraft Forge (Java/Gradle), we faced some challenges. Rascal does not seem to be optimised for integration in an existing Java project (the other way is fine, integrating Java with Rascal, but due to limitations of Minecraft Forge this is not a possibility). Because of this, we finally decided to include the Rascal Shell jar **within** the resources folder of our Minecraft Forge application (we use the unstable shell due to its reduced size). When the user starts Minecraft, the resources folder is then extracted to disk (see *nl.sandersimon.clonedetection.common.ResourceCommons*) in order to be started as a Java "Process". We start the shell twice, in order to do various Rascal commands while a clone calculation is running.

After we have these shells running, we have to execute commands to them. The output can then be read seperately for every character, or into an array as a longer buffer. We combine the two, using larger buffers when the content size is known and reading per character if the output is unknown. The Rascal application simply uses "println" to communicate with Minecraft. We have defined a contract between Rascal and Java to make the communication run smooth and fast. This results in the desired performance. Because of this, the user does not notice anything of the fact that the Rascal process does not run within the container of the application.

Apart from Minecraft, we use the Java Swing framework to render the code editors. Minecraft does not use Java Swing in any way. Instead Minecraft is based on LWJGL (which is an almost 1:1 mapping of OpenGL to Java). However, due to Minecraft's Java nature it integrates perfectly with Java Swing. When the code editors are created Minecraft is automatically paused. During the paused state the world of Minecraft does not tick (world updates in Minecraft are called "ticks"), so further world command (like spawning spiders or sending chat messages) are delayed (this does not happen automatically, there is a list actively collecting Runnable statements to execute these afterwards).

B. The Visualisation

When in Minecraft, the player can select a codebase from a directory he or she wants the code duplication metrics of, or wants to improve. When selected, the calculation starts running and an arena is spawned around the player, along with a report on the duplicates found which updates while running. In this arena spiders are spawned based on the code duplicates found in the codebase. When struck (or shot with an arrow) these spiders cause a code editor window to popup containing the files that contain the duplicate code, which can then be refactored. For each metric the player improves, points are rewarded.

The report that gets opened in Minecraft automatically shows the % of duplicated lines, number of clones and number of clone classes.
We also chose to highlight the biggest clone in the code in lines of code, the clone that occurred the most in the code (biggest clone class) and the clone class that had the largest total number of lines.
The first one is important is this probably a very large piece of code that has been copy pasted on different places, without significant changes in functionality. Since this part of code consists of many lines it is beneficial to be given attention to clean up the codebase significantly.
The second one is important since if a bug is found in the largest duplicate class, all of those duplicates have to be changed to remove the bug in the entire codebase. The largest duplicate class is therefore class one would start with spending attention to prevent bug propagation.
The last one is the duplication class that has overall the biggest impact on the code when looking at lines. When a developer would want to remove the percentage of code duplication in the code the quickest, this would be the class to start with.

Three screenshots of our visualisation can be found below.

Fig. 1. The player among the clones in the code represented as spiders

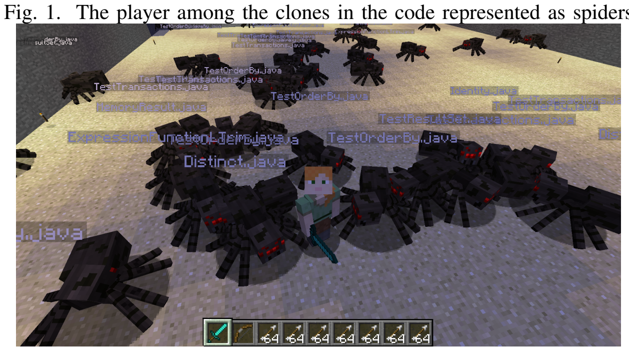
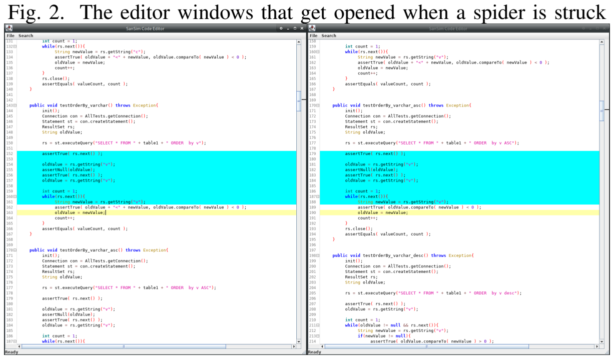


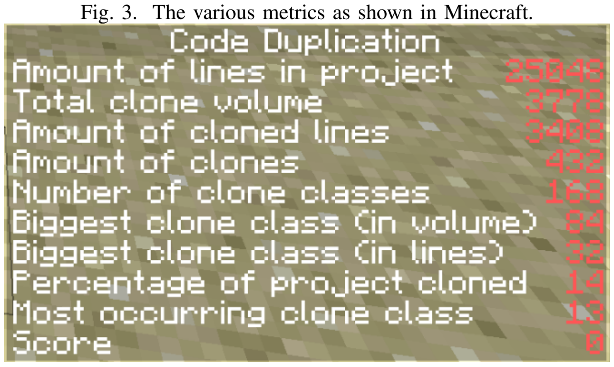
Fig. 2. The editor windows that get opened when a spider is struck



```java
        int count = 1;
        while(rs.next()){
            String newValue = rs.getString("c");
            assertTrue( oldValue + "<" + newValue, oldValue.compareTo( newValue ) < 0 );
            oldValue = newValue;
            count++;
        }
        rs.close();
        assertEquals( valueCount, count );
    }


    public void testOrderBy_varchar() throws Exception{
        init();
        Connection con = AllTests.getConnection();
        Statement st = con.createStatement();
        ResultSet rs;
        String oldValue;

        rs = st.executeQuery("SELECT * FROM " + table1 + " ORDER  by v");

        assertTrue( rs.next() );

        oldValue = rs.getString("v");
        assertNull(oldValue);
        assertTrue( rs.next() );
        oldValue = rs.getString("v");

        int count = 1;
        while(rs.next()){
            String newValue = rs.getString("v");
            assertTrue( oldValue + "<" + newValue, oldValue.compareTo( newValue ) < 0 );
            oldValue = newValue;
            count++;
        }
        assertEquals( valueCount, count );
    }


    public void testOrderBy_varchar_asc() throws Exception{
        init();
        Connection con = AllTests.getConnection();
        Statement st = con.createStatement();
        ResultSet rs;
        String oldValue;

        rs = st.executeQuery("SELECT * FROM " + table1 + " ORDER  by v ASC");

        assertTrue( rs.next() );

        oldValue = rs.getString("v");
        assertNull(oldValue);
        assertTrue( rs.next() );
        oldValue = rs.getString("v");

        int count = 1;
        while(rs.next()){
```

```java

        int count = 1;
        while(rs.next()){
            String newValue = rs.getString("v");
            assertTrue( oldValue + "<" + newValue, oldValue.compareTo( newValue ) < 0 );
            oldValue = newValue;
            count++;
        }
        assertEquals( valueCount, count );
    }


    public void testOrderBy_varchar_asc() throws Exception{
        init();
        Connection con = AllTests.getConnection();
        Statement st = con.createStatement();
        ResultSet rs;
        String oldValue;

        rs = st.executeQuery("SELECT * FROM " + table1 + " ORDER  by v ASC");

        assertTrue( rs.next() );

        oldValue = rs.getString("v");
        assertNull(oldValue);
        assertTrue( rs.next() );
        oldValue = rs.getString("v");

        int count = 1;
        while(rs.next()){
            String newValue = rs.getString("v");
            assertTrue( oldValue.compareTo( newValue ) < 0 );
            oldValue = newValue;
            count++;
        }
        rs.close();
        assertEquals( valueCount, count );
    }


    public void testOrderBy_varchar_desc() throws Exception{
        init();
        Connection con = AllTests.getConnection();
        Statement st = con.createStatement();
        ResultSet rs;
        String oldValue;

        rs = st.executeQuery("SELECT * FROM " + table1 + " ORDER  by v desc");

        assertTrue( rs.next() );
        oldValue = rs.getString("v");

        int count = 1;
        while(oldValue != null && rs.next()){
            String newValue = rs.getString("v");
            if(newValue != null){
                assertTrue( oldValue.compareTo( newValue ) > 0 );
```

Fig. 3. The various metrics as shown in Minecraft.



Code Duplication	
Amount of lines in project	25048
Total clone volume	3778
Amount of cloned lines	3408
Amount of clones	432
Number of clone classes	168
Biggest clone class (in volume)	84
Biggest clone class (in lines)	32
Percentage of project cloned	14
Most occurring clone class	13
Score	0

IV. Discussion

We are aware that our tool is not state-of-the-art in letting developers improve their code as quickly and efficiently as possible, nor does it give the best possible top-down overview of the duplication in the code. We chose to weigh the educative and gamification part of being confronted by duplication in the code over usability, although our tool still improves the code by letting users remove code duplication from their code, just not in the most time efficient way possible. The visualisation we have also lets the user improve code, which is a big plus. It is not just a visualisation tool, but also a tool to improve the code. We reason that making the work done to improve on code duplication more engaging and fun results in people both caring more about code duplication and more willing to put work into improving it. This would in the end result in more code duplicates being investigated and re-factored. A good overview of code duplication is important, but so is a method of investigating and fixing code clones, which our method does in an interactive and engaging way.

A. Threats to validity

Since we make use of an AST for the comparison between lines of code, it will not work when code is not valid java code. A missing bracket transforms the entire AST, which could result it just a single bracket causing no more duplicates to be found. We do however advice to first make sure the code is compiling before fixing it for code duplication. It could however still be considered a small flaw of our tool. This same use of the AST also causes some single bracket to not be taken into account when placed on the next line instead of on the line with the method declaration of for loop for example. This is however not a very significant mistake, as technically these two variations also differ a line, so our tool detects what is true.

We cannot 100% validate the correctness of our type two code duplication calculation. Roy et al [7] state that type two allows variations in identifiers, literals, types, layout and comments. We tested extensively to make sure that our type two simplification of the AST does not include any identifiers and names, but we cannot be sure that we do not delete anything that should have been included.

For type three, we do not consider the ordering of the lines, which we do for type 1 and 2. Even though we did think of methods to do this, we chose to not do this since the method as described by Baxter et al [2] also does not taking ordering into account to calculate the similarity between two ASTs. If we would take ordering into account, it will cause duplicates found to look semantically more like each other instead of having the same contents but in random order. We have not validated whether this would result in 'better' clones found, as one can argue for both options being more useful for a programmer. Our method will not care whether lines are swapped or which for loop is done first, while the other method will. Our method will therefor detect clones that may look less like clones than the ones found by the second method. The found code parts will in both cases have very similar functionality and flow however.

B. Improvements possible on the visualisation

Even though our visualisation works as intended, there are some things that could be improved in a future version. Firstly, when a file contains multiple different duplicates, fixing one 'spider' will cause the other spiders that use that file to also display the new file, which may not even contain a duplicate anymore. This could give issues when that spider is struck. To solve this, we added a button that allow the developer to say he already fixed this file, but we cannot prevent that some confusion may occur.
Second, the scores could be used to reward the player instead of just be displayed, making it feel more like achieving points serves a goal in the game itself, not just improving the codebase.
Lastly, the visualisation as is only uses a small part of Minecraft. By using more of Minecraft we could improve the overview of the metrics, showing quickly where in the codebase the most clones are and how files in the project compare to each other.

C. Future work

The next steps in this project are making the visualisation more insightful for developers, and the game more adventurous and fun. Our project should also be tested in practise by developers to proof its functionality and to show that it significantly helps developers in improving on code duplication in their code.

V. Conclusion

We presented an approach to code duplication in Minecraft that helps the developer gain a quick overview of how big the problem is, and immediately reduce the amount duplicates in the code. This approach uses Rascal to find the clones, and Minecraft to let the developer interact with the clones. It can be used by developers to investigate multiple types of clones, and makes fixing them an interactive game.

Acknowledgment

We would like to thank dhr. R.A. van Rozen and dr. A.M. Oprescu from the University of Amsterdam for their feedback, enthusiasm, and unlimited supply of coffee and tea.

References

[1] G. Balogh and A. Beszedes, “Codemetropolis-code visualisation in minecraft,” in *Source Code Analysis and Manipulation (SCAM), 2013 IEEE 13th International Working Conference on*. IEEE, 2013, pp. 136–141.

[2] I. D. Baxter, A. Yahin, L. Moura, M. Sant’Anna, and L. Bier, “Clone detection using abstract syntax trees,” in *Software Maintenance, 1998. Proceedings., International Conference on*. IEEE, 1998, pp. 368–377.

[3] M. Bruntink, “Sig guest lecture,” Master Software Engineering, 2018.

[4] V. Grout and N. Houlden, “Taking computer science and programming into schools: The glyndŵr/bcs turing project,” *Procedia-Social and Behavioural Sciences*, vol. 141, pp. 680–685, 2014.

[5] G. Kiryakova, N. Angelova, and L. Yordanova, “Gamification in education.” Proceedings of 9th International Balkan Education and Science Conference, 2014.

[6] R. Koschke, “Identifying and removing software clones,” in *Software Evolution*. Springer, 2008, pp. 15–36.

[7] C. K. Roy and J. R. Cordy, “A survey on software clone detection research,” *Queens School of Computing TR*, vol. 541, no. 115, pp. 64–68, 2007.

[8] R. T. Yeh, “Educating future software engineers,” *IEEE Transactions on Education*, vol. 45, no. 1, pp. 2–3, Feb 2002.