An Initial Approach to Assessing Program Comprehensibility using Spatial Complexity, Number of Concepts and Typographical Style

Andrew Mohan, Nicolas Gold, Paul Layzell
Information Systems Group
Department of Computation
UMIST, UK
{a.mohan@postgrad.umist.ac.uk, n.e.gold@co.umist.ac.uk, p.layzell@umist.ac.uk}

Abstract
Software evolution can result in making a program harder to maintain, as it becomes more difficult to comprehend. This difficulty is related to the way the source code is formatted, the complexity of the code, and the amount of information contained within it. This paper presents an initial approach that uses measures of typographical style, spatial complexity and concept assignment to measure these factors, and to model the comprehensibility of an evolving program. The ultimate aim of which is to identify when a program becomes more difficult to comprehend, triggering a corrective action to be taken to prevent this.

We present initial findings from applying this approach. These findings show that this approach, through measuring these three factors, can model the change in comprehensibility of an evolving program. Our findings support the well-known claim that programs become more complex as they evolve, explaining this increase in complexity in terms of layout changes, conceptual coherence, spatial relationships between source code elements, and the relationship between these factors. This in turn can then be used to understand how maintenance affects program comprehensibility and to ultimately reduce its burden on software maintenance.

Keywords: software evolution, software maintenance, program comprehension, software quality

1. Introduction
Accounting for 50-90% of the lifetime cost of a software system [16], software maintenance is indeed costly. The largest contributor to this cost is the process of comprehending the source code in order to apply maintenance to it [4, 17, 24].

Given the high cost and importance of this activity much research effort has been devoted to helping software engineers comprehend source code (e.g. program slicing [14] and concept assignment [11]). In addition to supporting the comprehension process by generating alternative views, techniques such as coding standards [11] and style guidelines [23] have been proposed. Here the aim is to ensure the quality and comprehensibility of code during development [10] and to ensure its quality is maintained [1]. Coupled with this are complexity measures that are used as a gauge of the ‘degree of difficulty’ for the comprehending of code [8].

The common goal of all this research is to reduce the comprehension burden. This paper presents an initial approach to assessing program comprehensibility that shares this goal. In particular, the approach is designed to continually monitor the evolution of a program to ensure that where necessary, the comprehensibility of a program is at least maintained.

This paper is organised as follows. Section 2 outlines the problem of program comprehension and factors that can be used to assess it. The next 2 sections present these factors and methods of capturing them, and the COMP metric, which is an approach to model comprehensibility. Section 5 describes an investigation into how COMP and these factors change in evolving source code. Section 6 presents initial results from this investigation, with a case study of several commercial COBOL II programs. Section 7 discusses these results and Section 8 presents the conclusions of this paper and directions for further work.
2. The Problem

2.1 Program Comprehension

To maintain existing code, the maintainer firstly requires a sufficient level of comprehension of that code [4, 17, 24]. The key artefact in this process is the source code itself [9]. The ease with which a program can be comprehended, its comprehensibility, is dependent upon the ease to which the maintainer can gain access to the information contained within the source code itself.

Due to the increasing size of programs, maintainers usually do not comprehend a program in full in order to apply maintenance to it [25]. They identify where the maintenance is to be applied, using tools such as a program slicer [14], or methods, such as software reconnaissance or grep searches [29], to assist them. Once identified, the maintainer then seeks to obtain the required level of comprehension of the program to enable them to complete this task [13].

This required level of comprehension determines the strategy applied by the maintainer. However to attain the desired level, information must be acquired from the program. From this point a “bottom-up” approach can be used, i.e. working from the syntax of program statements in order to devise semantic information [24, 25]. Alternatively a “top-down” approach can be used, i.e. working from an initial hypothesis about the functionality based, for example, on its name [4, 28]. This initial hypothesis is then tested by examining the code. The maintainer then iteratively refines both the initial hypothesis and their mental model of the actual program until they match [17].

The maintainer’s task when comprehending a program is to identify hypotheses in the program. Then to verify these hypotheses from the information, or beacons, within the program, until an adequate level of comprehension has been reached [4]. These beacons play a large role in initial, high-level comprehension of programs [28]. This follows the idea that maintainers recognise the function of parts of a program, pulling together these chunks until the entire program is comprehended [25], or at least until the desired level has been attained.

The cost of program comprehension in software maintenance is dependent upon the programmer building their mental model, by searching for beacons in the code [28]. The ease of this search, or the comprehensibility of the program, is in turn strongly influenced by:

- The format the information is presented in
- The degree of difficulty in acquiring the information
- The amount of information to be acquired.

2.2 Assessing Comprehensibility

A program used in a real-world environment necessarily must change as the system’s environment changes, new requirements are specified, etc. or become progressively less useful in that environment [15]. In effect the program must maintain its fitness in satisfying the purpose it was designed to fulfil.

During the initial stages of identifying where maintenance is to be applied, and therefore the initial stages of comprehension, the maintainer can enter the source code at any point. They then start to explore the program, expanding their comprehension to the desired level. However as it is impossible to tell at this moment what point in the program the next maintenance activity will affect, a more general view of comprehensibility of the program must be taken. Where this view must represent the ease at which a maintainer can search for information in the program to build their mental model.

The typographical style used when initially writing and then applying maintenance to a program determines the format the source code is presented in. Consequently it has a direct impact upon the comprehensibility of it [2, 23]. Furthermore, it impacts upon the ability of the maintainer to extract the information from it in order to comprehend the program [7, 23]. This is in effect stating that for a program to be fit for purpose, it must also meet the requirements that the maintainer can extract the necessary information from it, i.e. the typographical style exhibited by the program matches that preferred by the maintainer.

Currently we do not know how much of a program the maintainer will need to comprehend. This program can comprise many chunks of information, identifiable by a maintainer by beacons or concepts [11]. The number of such concepts a program contains is representative of the amount of information that the maintainer must acquire in order to comprehend a program. Here there is a conflict between the size of a concept that can be processed by the maintainer’s short term memory in one instance [25] and the total number in a program that a maintainer might have to comprehend.

If the concepts within a program that the maintainer must comprehend are positioned throughout it, then this tests the spatial ability of the maintainer. Spatial complexity measures the psychological complexity of a program, i.e. the degree of difficulty faced by a maintainer in acquiring information from a program [8]. Therefore the more spatially complex a program is, the greater the possible difficulty faced by the maintainer in identifying and comprehending the necessary information required for them to maintain the program.
3. Proposed Approach: The COMP Metric

The comprehensibility of a program is strongly influenced by its typographical style, the number of concepts it contains and its spatial complexity. These factors relate to how evolvable a program is through its level of comprehensibility.

The approach we propose is to capture these three factors in a combined comprehensibility metric COMP, giving them equal weighting. COMP will also include the number of non-blank lines of code as a reflection of the actual size of the program to be comprehended.

COMP is calculated by taking the values of the constituent metrics of the original, or base, version of a program and determining the percentage increase from that base for each subsequent version. With an increase in the value of COMP indicating that the subsequent version has a lower level of comprehensibility. The aim here is to identify the coding standard originally employed in the writing of the program and the amount and format of information contained within the program (as we assume that the program conformed to company standards at this point). This can then be used to measure the degradation of code quality in respect to comprehensibility, by means of deviation from this original standard or baseline. This also provides the flexibility for incorporating the fact that the standard could evolve, thereby resetting the baseline.

4. Constituent Metrics

The intention of this section is to outline the constituent metrics of COMP and their capture.

4.1 Typographical Style

The format the information within a program is presented in is determined by the style the programmer and subsequent maintainers have used whilst writing the source code. The aim of which should be to ensure that source code is written in a manner that improves its readability and maintainability [22].

A programming style is defined as an individual’s interpretation of a set of rules and their application to the writing of source code in order to achieve that aim [19]. An individual in this case can be a programmer or company. The set of rules applied to the writing of source code that only affect the layout of it, but not the execution of a program is classified as Typographical Style [22]. A typographical coding standard would consider, for example, comments, blank lines and their usage.

Typographical styles affect comprehensibility [20, 23, 27], with a consistent application of a particular style aiding comprehension [18]. Moreover, maintainers do impose their own typographical style upon a program they are maintaining, possibly to improve the comprehensibility of that program [19].

The typographical styles measured and incorporated into the COMP metric (each representing one quarter of the contribution made to this metric by typographical style) are as follows:

- Percentage of program lines that are blank
- Percentage of program lines that are comments (noting that an increase in this metric is reflected by a decrease in COMP, and vice-versa)
- The average level of indentation used in the procedure division
- Average number of characters on each non-blank, uncommented line within the procedure division.

These typographical styles have been chosen as a representation of the style exhibited by a program that have been identified as affecting comprehensibility in numerous experiments [2, 18, 20, 23, 26, 27].

4.2 Hypothesis-Based Concept Assignment

Indicators and concepts have been identified as useful in comprehending source code [3, 6, 25]. Indicators are the source code evidence leading to a hypothesis that a concept exists [4]. The definition of concept used by the Hypothesis-Based Concept Assignment (HB-CA) method is: Concepts are defined as descriptive terms at a higher level of abstraction than the source code, nominated by the maintainer to describe some abstractions of interest [11]. Concepts are designated as either actions (e.g. Write) or objects (e.g. File).

The HB-CA method is a three-part, automatic, and non-interactive process (only a summary of this process is provided here, a detailed description of HB-CA can be found in [11]). The three stages of HB-CA are Hypothesis Generation, Segmentation, and Concept Binding. Control and data flow sequentially between them.

Each stage uses a knowledge base that contains two entities: concepts, and indicators. Taking the source code as input the hypothesis generation stage uses the information contained in the knowledge base to analyse the source code for indicators of various concepts. The resulting list of concept hypotheses is ordered by the position of the indicators in the source code. This list is passed to the segmentation stage, which attempts to break them into segments, starting from the COBOL II section boundaries. These initial segments are then analysed to determine whether they have the potential to contain a number of smaller segments. Concept binding analyses each segment’s hypotheses and using the relationship in the knowledge base, determines which concept has the most evidence. The output from HB-CA is therefore an automatically generated list of concepts contained within the analysed program.
In effect the output from HB-CA is reporting on the amount of information (w.r.t. the knowledge base) contained within a program that can be acquired by the maintainer for use whilst comprehending it. The accuracy of this information is determined by the correctness and completeness of the knowledge base, which undoubtedly will change as the program evolves. However for the purpose in hand, we suggest that the more concepts a program contains, the more difficult it is to comprehend, as more information must be acquired by the maintainer to comprehend it. This total number of concepts is included in the COMP metric.

4.3 Spatial Complexity

The process of comprehending software in order to apply maintenance to it substantial involves the spatial ability of the maintainer [8]. This refers to the maintainer’s ability to process the information presented throughout a program to build a program plan. Hence the more widely spaced the information, such as function usage and call are, the more complex the relations between such parts become and the more difficult it is to comprehend [5].

The Douce et al. program spatial complexity metric [8], provides an indication of how much cognitive effort is required to comprehend a program. This is achieved by summing the distances in lines of code between a function call and its declaration, for each function within a program.

To reflect the spatial complexity involved in dealing with nested function calls, Douce et al. proposed a recursive version of the program spatial complexity metric [8] (hereafter referred to as the Douce recursive metric). This metric extends the simple function-oriented complexity metric to account for the additional spatial effort placed on a maintainer in dealing with nested function calls. The metric achieves this by summing the distance from calling functions, where the distance is the sum of the distances that its children call [8].

Of course this metric does not take into consideration any previous knowledge of the program by the maintainer. However this recursive metric does provide a measure of the difficulty the maintainer will be faced with in acquiring information from a program. Higher values of the metric indicate greater difficulty of understanding. It is incorporated into the COMP metric to reflect the change in a program’s spatial complexity as it evolves.

5. Investigation

We undertook an empirical investigation to determine the COMP metric over the evolution of 9 commercial COBOL II programs to identify how this metric reflected the change in the comprehensibility of these programs.

Further analysis of the differences between versions of each program was undertaken through manually scanning the code and with the unix tool `diff`. The output from this was then used to determine a percentage similarity for each program version (the similarity figure representing the percentage of the previous version that is still present in the current). The “non-similar” percentage indicates the area of the program requiring further comprehension.

6. Results

This section reports on a case study investigating the COMP metric in the 9 programs referred to above. The characteristics of these programs are given in Table 1.

<table>
<thead>
<tr>
<th>Program</th>
<th>Versions Studied</th>
<th>Min Lines</th>
<th>Max Lines</th>
<th>Douce Recursive Metric</th>
<th>Concepts Assigned</th>
<th>COMP Metric</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>First</td>
<td>Final</td>
<td>Max</td>
<td>First</td>
<td>Final</td>
</tr>
<tr>
<td>GB01</td>
<td>14</td>
<td>829</td>
<td>939</td>
<td>31.1 39 39.7</td>
<td>19 20 22</td>
<td>12.4 14.0</td>
</tr>
<tr>
<td>GB08</td>
<td>14</td>
<td>766</td>
<td>1215</td>
<td>75.1 234.8 258.1</td>
<td>18 36 40</td>
<td>85.2 99.8</td>
</tr>
<tr>
<td>GB12</td>
<td>12</td>
<td>461</td>
<td>635</td>
<td>7.9 33.8 56.9</td>
<td>10 16 21</td>
<td>68.2 184.1</td>
</tr>
<tr>
<td>GB92</td>
<td>14</td>
<td>333</td>
<td>484</td>
<td>28 35.5 43.8</td>
<td>15 20 22</td>
<td>19.9 30.3</td>
</tr>
<tr>
<td>GD02</td>
<td>10</td>
<td>623</td>
<td>991</td>
<td>2 75.4 75.4</td>
<td>10 17 18</td>
<td>951.3 951.3</td>
</tr>
<tr>
<td>GD21</td>
<td>12</td>
<td>692</td>
<td>783</td>
<td>78.3 117.5 118</td>
<td>24 25 30</td>
<td>17.0 17.6</td>
</tr>
<tr>
<td>GD28</td>
<td>9</td>
<td>511</td>
<td>784</td>
<td>52.2 131.7 131.7</td>
<td>12 22 22</td>
<td>113.4 113.4</td>
</tr>
<tr>
<td>GD29</td>
<td>9</td>
<td>368</td>
<td>518</td>
<td>28.6 83.2 83.2</td>
<td>9 16 16</td>
<td>114.3 115.5</td>
</tr>
<tr>
<td>GD66</td>
<td>9</td>
<td>558</td>
<td>588</td>
<td>43.2 51.9 51.9</td>
<td>6 10 10</td>
<td>24.9 25.4</td>
</tr>
</tbody>
</table>

Figure 1 and Figure 2 show the various comprehensibility measures for programs GD02 and GD28 respectively and value of COMP. Figure 3, Figure 4 and Figure 5 show the typographical style, spatial complexity and concepts assigned, and the COMP metric respectively, for program GB12 over all versions. Likewise, Figure 6, Figure 7 and Figure 8 show these characteristics for program GB08.
6.1 The COMP Metric

The value of the COMP metric is the result of calculating the percentage deviation of its constituent metrics from the base version of the program, with an increase in COMP indicating a decrease in comprehensibility. The value of this metric for the final version of each program studied, and the maximum value attained are shown in Table 1.

Every program studied during this investigation had an increase in the value of COMP over the versions studied. The smallest such increase occurred in GD21 with 17 percent and was due to the small change in the similarity of the program between versions. The largest increase occurred in GD02 with 951 percent, as this program incurred major changes, in particular between versions 5 and 7 (Figure 1, #1). Most programs did not reach the maximum value of COMP at their final version, the exceptions being programs GD02, GD29 and GD66.

6.2 Douce Recursive Metric

Table 1 shows the value of the Douce recursive metric for the first version of each program, the final version and the maximum value. The values given here have all been divided by 100. However it can be seen in Table 1 that the range in this complexity metric is considerable, both between the first and final versions, for example, program GD02 (Figure 1), and between the programs themselves.

In most programs studied the Douce recursive metric changed between versions and all had an overall increase throughout. However what is more interesting is that several of the programs studied showed very large increases over just one version. The biggest such increases occurred in GD28 between versions 3 and 4 (Figure 2, #2), GD02 between versions 6 and 7 (Figure 1, #1) and GB12 between versions 3 and 4 (Figure 4, #4).

A number of the programs studied, also at varying points had a decrease in the Douce recursive metric. Programs GB12, between versions 6 and 7 (Figure 4, #5) and to a lesser extent, GB08 between versions 13 and 14 (Figure 7, #8) both exhibited such a reduction.

6.3 Number of Concepts

The number of concepts assigned by WeSCA (an implementation of the HB-CA method) for the first and final versions of each program studied, and the maximum number are shown in Table 1.

All programs studied had an increase in the number of concepts. However not every program’s final version had the maximum number of concepts in it. An example of this can be seen in program GD02, where the maximum number of concepts occurred at version 6 (Figure 1, #1).

The change in the number of concepts were seen to increase, as in GD28 between versions 3 and 4 (Figure 2, #2), to decrease or remain constant, as the level of the Douce recursive metric increased. Furthermore, the number of concepts were seen to decrease, as in GB08 between versions 13 and 14 (Figure 7, #8) or remain constant, when the Douce recursive metric decreased.

6.4 Typographical Style

Programs GB08 and GB12 both show that changes in typographical style occur when both the number of concepts and the Douce recursive metric change.

With GB08 it can be seen that all four typographical styles changed between versions 6 and 7 (Figure 6, #6). This in turn is matched by an increase in the number of concepts and the Douce recursive metric (Figure 7, #6). Furthermore, that a slightly less marked change in GB08’s typographical style (Figure 6, #7), is matched to a large increase in the Douce recursive metric and only a small increase in the number of concepts (Figure 7, #7).

In GB12 the changes in typographical style and the various comprehensibility measures are more marked. This is particularly the case between versions 3 and 4 (Figure 3, #4), and between versions 6 and 7 (Figure 3, #5).

7. Discussion

7.1 General

During this investigation we were able to identify that, through the use of COMP, maintainers are not only altering the program stylistically to reflect their preferred style, but also complexity. An example of this can be seen in GB12, between versions 3 and 4 (Figure 3, Figure 4 and Figure 5, #4). Here the maintainer, through the addition of comments, indicates that the program is not very well structured and has an undesirable use of a function (once again this is indicated through his comments). The maintenance applied, with the intention of improving code quality, has in fact caused the opposite effect by increasing COMP to 157 percent (Figure 5, #4). The increase in COMP between versions 3 and 4 was found to be the result of adding sections to the program to deal with this undesirable usage, and hence the increase in concepts (Figure 4, #4). These sections are then called throughout the program, and it is this that resulted in the large increase in complexity. The only confounding factor is that it is not clear how many of the changes were made to improve quality and how many to make the necessary functional changes.

To rectify the above problem, a different maintainer has reduced the complexity introduced through version 4...
of GB12 in version 7 (Figure 3, #5). Additionally it can be seen that there is a marked difference in typographical styles between versions 6 and 7 (Figure 3, #5), as the maintainer has introduced blank lines to separate IF-THEN-ELSE statements. The net result of this was that COMP had decreased to 63 percent.

At this point it is interesting to note the usefulness of the COMP metric in identifying the problem introduced into GB12 at version 4. Indeed, if the COMP metric had been used as part of the quality review process then the problem could have been rectified by the maintainer at that moment. This would then have saved the cost of having a different maintainer rectifying the problem in version 7.

A further example of the usefulness of COMP comes in GB08, between versions 6 and 7 (Figure 6, Figure 7 and Figure 8, #6), where a major change has taken place. The increase in COMP can be explained by the increase in the number of concepts and complexity. Furthermore, that this has been accompanied by significant changes in typographical style. This implies that the maintainer has used the change to impose their style upon the program (this was confirmed by a manual scan of the code). However the large increase in COMP, if used as part of a quality review process, would indicate the need to reassess the maintenance applied to determine whether it could be done without the increase in concepts and complexity, or whether the program needs restructuring.

7.2 COMP and Comprehensibility

The results presented in this paper indicate that certain changes do have a significant impact upon the comprehensibility of the program, indicated through the large increases seen in the value of COMP.

An example was given in GB08, between versions 6 and 7 (Figure 8, #6). Here the maintenance applied has decreased comprehensibility, as shown by the increase in COMP. This was found to be due to the increase in concepts and complexity (Figure 6, #6) and by the fact that the maintainer has imposed their own style upon the program (Figure 7, #6). However the question remains as to whether this has increased the comprehensibility for the maintainer and for subsequent maintainers.

Every program included as part of this study had an increase in COMP over the versions studied. This was associated with an increase in both complexity and number of concepts. This implies that the comprehensibility of those programs has decreased. In addition it can be seen, for example, in GB12 between versions 6 and 7 (Figure 3 and Figure 4, #5) that where the program is already difficult to comprehend, maintainers use typographical style as a way to improve the source code and to improve its comprehensibility.

Furthermore, that these typographical changes have been reflected by a decrease in COMP (Figure 5, #5).

With the exception of GB12, the largest decrease in COMP occurred in GB08 between versions 13 and 14 (Figure 8, #7). Analysing this program further it was found this change occurred because several statements, including two calling statements, had been removed. Their removal had resulted in a decrease in spatial complexity and concepts (Figure 7, #8). Here it can be speculated that the removal of the calling statements, particularly since they called external functions, has resulted in an increase in comprehensibility as indicated by COMP. This may also hold for the reduction in concepts.

The usefulness of COMP is once again indicated by these examples, in particular, the ability of COMP to identify when maintenance has adversely affected the comprehensibility of the program. This information can then be used by the maintainer to identify why this is the case, allowing them to undertake corrective action.

7.2 COMP and Comprehensibility

The results presented in this paper indicate that certain changes do have a significant impact upon the comprehensibility of the program, indicated through the large increases seen in the value of COMP.

An example was given in GB08, between versions 6 and 7 (Figure 8, #6). Here the maintenance applied has decreased comprehensibility, as shown by the increase in COMP. This was found to be due to the increase in concepts and complexity (Figure 6, #6) and by the fact that the maintainer has imposed their own style upon the program (Figure 7, #6). However the question remains as to whether this has increased the comprehensibility for the maintainer and for subsequent maintainers.

Every program included as part of this study had an increase in COMP over the versions studied. This was associated with an increase in both complexity and number of concepts. This implies that the comprehensibility of those programs has decreased. In addition it can be seen, for example, in GB12 between versions 6 and 7 (Figure 3 and Figure 4, #5) that where the program is already difficult to comprehend, maintainers use typographical style as a way to improve the source code and to improve its comprehensibility.
Figure 3: Typographical Style of GB12 [19].

Figure 4: Comprehensibility Measures for GB12.

Figure 5: COMP Metric for GD12.
Figure 6: Typographical Style of GB08 [19].

Figure 7: Comprehensibility Measures for GB08.

Figure 8: COMP metric for GB08.
7.3 Limitations of COMP

An increase in the spatial complexity of a program, with no increase in the number of concepts, suggests that the existing sections are being used more widely in the program. Such changes, for example in GD02, between versions 6 and 7 (Figure 1, #1), indicates the deletion of existing concepts, with existing sections being used more widely.

In this example COMP has risen to reflect the increase in spatial complexity and lines of code and as such indicates a decrease in the comprehensibility of the program. However the limitation here is that the number of concepts does not indicate the information they are providing the maintainer, or if any degradation in this provision has occurred (as a concept specifying what it is doing is more comprehensible than one stating it exists). This would then enable an indication of whether the quality of the existing concepts had improved and as a consequence, increased the comprehensibility of the program.

The addition of new concepts and an increase in spatial complexity indicates an increase in the number of sections that are then used throughout the program. An example of this can be seen in GB08 between versions 11 and 12 (Figure 7, #7). Here the program has undergone Y2K analysis and as a result had additional sections added to ensure compliance. This then has increased the value of COMP (Figure 8, #7), indicating an increased difficulty in comprehending the program. Once again, as with the example above there is the question as to how the quality of the concepts has changed, particularly those concepts that have resulted in the increased spatial complexity. A similar example of this effect can be seen in GD28 between versions 8 and 9 (Figure 2, #3). Here COMP has increased but the main constituent metric that has caused this is the increase in the number of concepts.

In the example given in Section 7.1 above, the change in GB12 between versions 6 and 7 was meant to improve the program. This improvement resulted in a decrease of COMP from 184 percent to 63 percent (Figure 5, #5). However this reduction is the result of decrease in concepts, lines of code and, in particular, the decrease in spatial complexity. The large increase in blank lines (Figure 3, #5) in the program is treated by COMP as a decrease in comprehensibility. This then raises the question of whether blank lines increase or decrease comprehensibility. Moreover, whether COMP should treat this metric the same as comments, i.e. treat an increase in comments in a program as a negative value.

8. Conclusions and Further Work

This paper has presented the initial findings from applying an approach to identifying the comprehensibility of a program through the COMP metric. The approach involves measuring various typographical programming styles, concepts assigned and the Douce recursion metric. The aim was to draw upon these findings to determine if there exists a relationship between these measures and to identify how a program continues to be fit for the purpose of being maintainable, through the COMP metric.

The results this paper has presented indicate that the programs studied have been continually changed to maintain their fitness in satisfying the purpose they were designed to fulfil. These changes have probably made the process of comprehension more difficult, as shown by the increase in COMP.

This investigation has found that the COMP metric can be used to indicate how maintenance has changed the accessibility and amount of information contained within a program. However a number of limitations to this metric have been identified. Consequently for this metric to become more useful the contribution of each factor to COMP needs to be investigated and calibrated to more accurately reflect their influence on comprehensibility.

The scope of COMP needs to be examined. Firstly, to determine whether additional typographical styles should be included and to what level of detail. Additionally, COMP currently uses the number of concepts assigned by WeSCA. This could be expanded to include the quality of the concepts (how much information do they provide) and how their assignments change (using the framework proposed in [12] as a base).

A change in COMP reflects a possible change in the comprehensibility of a program. However during this investigation the change observed in COMP between versions has been both large and small. The usefulness of COMP comes with its ability to detect such changes. To become more useful, these changes need to be identifiable with certain trigger or action points in the evolution of a program, i.e. at what point should maintenance be applied to reduce COMP? The ultimate aim here is to ensure, that in terms of comprehensibility and hence maintainability, that a program can continue to be ‘fit for purpose’.

9. Acknowledgements

We gratefully acknowledge the support of CSC. We would also like to thank Andrew Kirkham for producing analysis software and proof-of-concept results for the spatial complexity metrics. This work was partly funded by the UK Engineering and Physical Sciences Research Council (EPSRC) under grant number GR/R71733.
References


