# An Investigation of Java Abstraction Usage for Program Modifications

Pamela O'Shea and Chris Exton
Computer Science and Information Systems Department,
University of Limerick
Castletroy, Limerick, Ireland
pamela.oshea@ul.ie

## Abstract

*This paper reports upon the results of an investigation concerning the use and type of Java abstractions employed during software maintenance. The source of data consists of eighty-eight program summaries extracted from online developer mailing lists. Specifically, the summaries describing modifications, thirty-six in total, were examined from the perspective of five task types, including adaptive, corrective, emergency, perfective and preventive. Corrective and perfective task types were the two most commonly found. Abstractions are examined per task type and are also presented in three sequential stages as beginning, middle and end of the summaries. The results show that middle (within program level) abstractions dominate each task type, with the higher (system and architecture level) and lower (code and java virtual machine level) abstractions following respectively. The results detail the type of abstractions used in each task type and summarise the abstractions found for modifications in general with potential applications to support the design of Java software visualisation tools.*

## 1. Introduction

*"Program maintainability and program understanding are parallel concepts: the more difficult a program is to understand, the more difficult it is to maintain"* cited by [17] (p. 849). It is this *understanding* which provided the motivation for the study described here. That is, by investigating which abstractions are more commonly used during software maintenance, a set of requirements may be generated to aid the design of support tools.

The research question under investigation may be stated as follows, *"Which abstractions are most important to the experienced software engineer during software maintenance ?"*. The terms *"software maintenance"*, *"experienced software engineer"* and *"abstractions"* will now be defined and discussed in turn.

Software maintenance has been defined many times by researchers, including Lientz and Swanson in 1978, where it was stated that *''maintenance and enhancement are generally defined as activities which keep systems operational and meet user needs"* [12]. A more recent definition may be taken from the software engineering institute's glossary at carnegie-mellon which defines maintenance as *"the cost associated with modifying a software system or component after delivery to correct faults, improve performance or other attributes, or adapt to a changed environment"* [2].

These and other definitions show that maintenance always consists of *"activities"* which involve the modification of the existing source code in some way. This paper presents an investigation into the abstractions used and described by programmers when performing such modifications to well established open source, object oriented, Java systems.

For the purposes of this study, the term *"experienced software engineer"* may be placed in the context of online open source projects where accepted peer developers contribute source code to the repositories. Open source projects were chosen for investigation [1] for a number of reasons, firstly as Capiluppi et al stated, *"open source software provides a good opportunity for observing software products in various phases of their evolution, growth and maturity"* [4].

Secondly, a large amount of publicly accessible resources are associated with well established projects, such as CVS (Concurrent Versions System), issue-tracking, communication changes (e.g. mailing lists and newsgroups) and online documentation [7].

Thirdly, experienced programmers may be observed through the archives of developer mailing lists. For well established projects, at least two separate mailing lists exist, one for end-users and another for project developers. It is the latter, developer mailing lists which have been examined in this study. The developer lists were chosen from

[1] Approved by the University of Limerick Ethics Committee (Application Number: 03/52).

Proceedings of the 13th International Workshop on Program Comprehension (IWPC'05)
1092-8138/05 $ 20.00 **IEEE**

the Apache foundation's Jakarta site for open source Java projects [1].

The final reason for choosing open source data is that accounts closely representing program summaries may be found within the online mailing lists, where developers discuss problems and design issues, modifications made, etc. Program summaries are accounts written by a programmer to summarise their understanding of the program under investigation. Such summary accounts have previously been gathered within a laboratory setting by Pennington [15] and Good [9] for analysis, where programmers were asked to provide a written summary of a given procedural program. Building upon Pennington's work, Good applied a developed schema to categorise the documented summaries. For example, categories within the schema included, data flow, control, function, action, operation, state-high, state-low, meta, elaborate, incomplete and unclear (due to space constraints, the reader is referred to [9] for full definitions and examples, however their definitions do not affect the understanding of the results presented here). One possible analysis could be to classify the summaries as either control flow or data flow, etc. by examining the frequency of occurrence of each category within the program summaries.

For the research question in this paper, a schema was developed during a pilot study, based upon Good's work, to describe the abstractions found within the Java program summaries. An overview of this schema will be presented in Section 2.1. The schema was then documented and a coding manual written to allow other researchers to use the schema. The schema was then applied to the data gathered from the online mailing lists.

It is this developed schema which leads us to the definition of *"abstractions"*, as it was developed to encompass both the *views* of the software system (those categories developed by Good with a number of additions for the application to object oriented summaries) and the *abstractions*. It was found that the programmer's accounts could be classified into *what* the programmer was describing and *how* it was being described. The *views* were used to describe *how* the programmer was describing the item in question, for example, if the programmer was describing a variable, was it the control flow or data flow, etc., of the variable being described ? In contrast, the *abstraction* was *what* the programmer was describing, e.g. was the programmer speaking of a package, class, object, variable, etc ?

Section 2 will present the decisions made during the design of the study, including the sample size, reliability, schema design and the themes and task types used to categorise the summaries. Section 3 will present the results where the summaries were considered as part of their assigned task types and presented both by triad and in general. The summaries were divided into thirds, named triad 1, 2 and 3. Each triad will be reported upon before providing the general results of the summary. Finally, the summaries were all grouped together under the *modification* umbrella and analysed by triad and in general. Section 4 concludes with a short summary of the results and discusses the implications for future work.

## 2. Study and Procedure

As stated in the introduction, the source of the data was chosen to include program summaries extracted from online, open source, mailing lists. As no single study with observed participants is without flaws, the following disadvantages were considered when comparing this methodology to a laboratory study: 1. Cannot easily monitor any of the user's interactions with the debugger, 2. Cannot verify the correctness of the participants' statements as it is too time consuming to read the open source project source code in order to verify the participants' statements, 3. Contact would be required to be made with the participants off-list in order to verify their exact level of expertise, 4. Cannot control the target audience of the summary i.e. whether the participant was writing a summary for a peer programmer in the same project/group or whether the summary was intended for an unfamiliar programmer or novice etc.

The following advantages were deemed important enough to choose this data gathering method over the controlled laboratory method or even in-situ/action-research studies where the presence of an experimenter is required: 1. Increased possibility of achieving strong ecological validity i.e. the participants are unaware of being monitored and are forming the summaries within their own working environment using familiar tools, 2. Larger sample sizes are possible, i.e. 88 program summaries (see Section 2.3) were gathered in total, 36 of which provided the *modification* results presented here (Table 1), 3. The programs being described are both large and real and not manufactured by the experimenter in any way, 4. The tasks performed are real tasks, 5. Regular contributors to the developer mailing list are most likely to be experienced as they are accepted by their peers.

### 2.1. Schema Development

To encompass all the abstractions found within a typical Java program summary, the following *abstraction* schema structure was created with three levels, high, middle and low. The higher abstractions are used when the programmer describes abstractions above the program level in their descriptions, i.e. at the software architecture/design level. The middle abstractions are used when the programmer was describing the program/modification within the program level. The lower abstractions are used when the programmer is

describing the program/modification near the source code level and Java virtual machine (jvm) level.

The higher abstractions (system/architecture level) include the following abstraction descriptions: package (pac), program (prog), thread (thr), component (com), interface (int), abstract class (abs), class (cla) and object (obj). For example, if a programmer refers to a class as part of the program summary, then the class category may be applied to that description.

The middle abstractions (within program level) include the following abstraction descriptions, all of which can be contained within the higher abstractions: external (ext), feature (fea), algorithm (alg), program unit (pru), constructor (con), method (met), variable (var), and process (pro). The less verbose terms will now be discussed. The external abstraction here refers to any output of the program, for example, the program output itself or a file. A feature classification was used when the programmer was describing a particular feature of the program and makes references to it as such. The reason feature becomes a middle abstraction was due to the fact that it refers to items within the program/class itself. The program unit was used to refer to a small unit of executable code i.e. an action performed by the program.

The lower abstractions (source code and jvm) include the following abstraction descriptions: block (blo), code excerpt (cex), line of code (loc), Java virtual machine (jvm). The block abstraction is equivalent to *"chunking"*, where the programmer refers to a block of code as an item, for example, a *try-catch-finally* block. A code excerpt (cex) is when the source code greater than a single line of code was directly included within the summary. A line of code (loc) was a used to classify a single line of code only. The Java virtual machine (jvm) refers to any output of errors (exceptions, stack traces), garbage collection or any other jvm activity. The program unit (pru) referred to in the middle abstraction does not belong here as it describes executable code, the code here must be shown and described statically.

## 2.2. Procedure

The program summaries were gathered from the developer mailing lists of the following projects: bcel, cactus, commons, ecs, hivemind, james, jetspeed, log4j and oro. Each project was listed on the jakarata site [1] and program summaries were collected in a backwards crongology order until the target sample size was reached.

The described schema was then applied to each program summary. This process employs the content analysis method where a number of predefined categories are applied to a text. The presence of such categories and their relationships can then be examined allowing inferences to be made, which in our case are inferences regarding the experienced programmer's abstraction usage.

The first step involves, splitting the summaries into segments. A segment consists of a subject and a predicate (either of which may be implied [9]). Each segment was then examined and categorised into one of the 22 abstraction categories. Each categorisation is mutually exclusive, e.g. a segment classified as describing an *object* cannot also be classified as describing a *class*. The following is an example sentence from a summary, with the numbered segments denoted by a forward slash:

*"(1) In org.apache.jetspeed.services.security.turbine. TurbineUserManagement / (2) in Method encryptPassword() / (3) the OutputStream is not flushed."*

The assignment of abstractions to segments is described in a constructed coding manual which includes all 22 definitions with examples. In this case, segment 1 was classified as a class (cla) due to the fact that *"TurbineUserManagement"* was a class. Segment 2 was coded as a method (met) and segment 3 was coded as an external (ext) since it describes a stream which is defined in the coding manual as an external reference.

For the triad analysis, each summary was divided into thirds, i.e. triad 1 was the opening third, triad 2 was the middle third and triad 3 was the closing third of the summary. The procedure involved the use of a perl script which divided the number of segments in each summary by three. If the number of segments were not a multiple of three, i.e. could not be divided without a remainder, the remaining summaries were classified as part of the middle triad. The use of triads was chosen due to the varying number of segments within each summary and to also facilitate comparisons between the start, middle and end of a summary. For example, investigating differences between the opening and closing sections of a summary can be supported.

## 2.3. Sample Size

The sample size of 88 was chosen to satisfy the content analysis coding, where Krippendorf's table for the Alpha statistic was chosen to help determine the sample size [10] (p. 122).

The number 88 was derived from the pilot study involving 58 summaries producing 845 segments. Inferences were not to be made regarding the two least frequently occurring categories in the *"views"*: *"Bucket"* (a catch-all category for unclassified segments) and *"Unclear"* (used when the author was describing an unclear entity, e.g. it was not know whether a class or object was being discussed), which left the *"State"* category as the next least frequently occurring category (occurring 36 times in total). At the 0.05 significance level, 44 is the recommended sample size from the Alpha table. This recommended sample size was doubled to 88 for two reasons. Firstly, to improve reliability and sec-

| | Adp. | Cor. | Em. | Perf. | Prev. | Total |
|---|---|---|---|---|---|---|
| Mod. | 1 | 18 | 0 | 14 | 3 | 36 |
| | 2.78% | 50% | 0% | 38.89% | 8.33% | 100% |
| MR. | 3 | 0 | 0 | 9 | 0 | 12 |
| | 25% | 0% | 0% | 75% | 0% | 100% |
| DOP. | 2 | 28 | 1 | 4 | 5 | 40 |
| | 5% | 70% | 2.5% | 10% | 12.5% | 100% |

**Table 1. Distribution of task types within summary themes (88 Summaries)**

ondly to record more summaries for each of the task types (adaptive, corrective, emergency, perfective and preventive) under each of the summary themes (modification, modification request and description of problem).

## 2.4. Reliability

Krippendorf's Alpha [10] and Cohen's Kappa [6] are two common methods of reporting reliability in content analysis studies.

The Kappa statistic was calculated by comparing the results of 8 independently coded summaries from one researcher and measuring the agreement with the results from the coding performed by the first author. The Kappa was found to be 0.8818 using the documented coding assumptions. A further improvement of 0.9449 was found after an agreement was made regarding the *"meta"* category for programmer comments. However, the first Kappa result must be used as protocol. A coding manual was also contstructed to describe definitions and provide numerous coding examples.

## 2.5. Themes and Task Types

Table 1 shows the distribution of the 88 summaries within each task type.

Each summary was found to fit into three distinguishable themes. That is, the theme or purpose of the program summary could be easily classified as either a description of how a *"modification"* was performed, or an account of a *"modification request"* or a *"description of a problem"*. Each theme was further sub-categorised into one of five task types. The themes and task types will now be discussed in the following sections.

### 2.5.1 Themes

- Modifications (Mod): source code for part or all of the solution must be included in the program summary, either embedded, attached or the location referenced.
- Modification Request (MR): is only a request for a feature, no implementation discussion or solution must appear or be attached/referenced within the message.
- Description of Problem (DOP): the problem and/or the possible solution is described, but no solution code must appear or be attached/referenced within the message.

### 2.5.2 Task Types

The five task types are composed of the four types defined in the *IEEE Software Maintenance Standard* [3], *"adaptive"*, *"corrective"*, *"emergency"* and *"perfective"*, while the fifth task type is one that is often referred to in the software maintenance literature as *"preventive"*.

Through the decades, all the literature has agreed on three of the maintenance tasks, those being, *adaptive*, *corrective* and *perfective*. For example, Swanson in the 1970's [19], Martin and Osbourne in the 1980's [13], and the updated IEEE software maintenance standard in the 1990's [3]. It is the *"other"* category which has varied widely, however, *"preventive"* has been consistently appearing most frequently in the literature as the *"other"*, e.g. [17] and [16].

While *adaptive*, *corrective* and *perfective*, may be the most consistently agreed upon task types, the definitions may be seen to vary throughout the literature. This fact is also highlighted recently by Chapin [5].

Chapin et al make a clear and useful distinction between task type definitions derived from empirical work (activity-based) and theoretical (intention-based) based definitions. For the purposes of this empirical study, the summaries were clearly divisible into the five task types using the following definitions.

- Adaptive: *"accommodation of changes to data inputs and files and to hardware and system software"* [12]. Examples found include, modifications made to support distributed systems were also considered adaptive.
- Corrective: *"emergency fixes, routine debugging"* [12]. Examples of this category were quite frequent.
- Emergency: *"Unscheduled corrective maintenance performed to keep a system operational"* [3]. Examples found include modifications made to correct discovered/reported security bugs.
- Perfective: *"user enhancements, improved documentations, recoding for computational efficiency"* [12]. Examples found include, making modifications to support programmers enhancements/needed features, as well as performance optimisations.
- Preventive: *"work performed on a system in an effort to prevent an error or malfunction from occurring"*,

cited in [16] (originally by [14]). Examples found included the following:

- preventing problems before they occur, e.g. skewed files
- improving design for extensibility
- performing modifications to support practices, such as calling *"super"*
- designing/recoding to avoid situations such as out of memory errors
- performing an operation to be in-line with correct procedures e.g. initialisation of variables
- making modifications for prevention of deadlocks
- correcting access control on variables, e.g. *"transient"*

Referring to Table 1, it is interesting to compare the distribution of task types within the *"modifications"* theme, where the *"corrective"* task types occurred 50% of the time, followed by *"perfective"* at 38.89%. These results are comparable with Vans, von Mayrhauser and Somlo [20] where it was stated that *"Corrective maintenance is a frequent activity during software evolution"*. A more recent study performed by Schach et al [18] showed a 53.4% and 56.7% usage for *corrective* type maintenance at the module and change-log level respectively. The results are also similar to the 50% found in this study. Interestingly, their *perfective* results are similar to the 38.89% usage found here also (36.4% and 39% for the module and change-log level respectively).

In contrast, Pfleeger stated that *perfective* task types take 50% of the total maintenance effort, while the *corrective* tasks are shown to consume as little as 21% [16] (p.420). The results of this study also conflict with Lientz and Swanson [11], where *corrective* maintenance was found to have a frequency of 17.4%, while *perfective* maintenance was found to take 60.3% of the maintenance effort.

# 3. Results

This section presents the results of the summaries classified as *"Modification"* (Mod) in Table 1. The 36 summaries will be examined firstly, by their task type, and secondly, as a collective *modification* group.

Both the *adaptive* and *emergency* task types did not yield sufficient summaries for analysis as individual categories. However, the 18 *corrective*, 14 *perfective* and 3 *preventive* summaries will be presented individually. The table in Section provides a count of the *abstractions* (Abs) assigned to each segment (defined in Section 2.2). The actual number of segments coded (N) is given along with the percentage (%) occurrence.

Significance was measured using the anova test and was followed by a scheffe post hoc if significance was discovered to find which groups differed. Results which were not significant are not reported except in their general form. For example, abstraction usage was not significantly different between the triads of the corrective and preventive tasks, as a result, the abstraction usage is reported for these summaries in general and not their individual triad results.

## 3.1. Frequency of Abstraction Usage

This section presents the frequency of abstraction occurrence. The results will firstly be presented by triad breakdown for each task type, that is, the three triads for each task type will be presented where significant. Secondly, the frequency of abstraction occurrence will be presented for each task type in general (irrespective of triad breakdown).

### 3.1.1 Each Task Type by Triad Breakdown

No significant differences were found for the abstraction usage of the three triads of all the corrective or preventive tasks. Instead, the results of abstraction usage in general will be presented Section 3.1.2 for these tasks. However, the differences found between the triads of the perfective tasks will now be reported.

**Perfective**

A significant difference was noted between the perfective triads with p = .014. The scheffe post hoc test showed that triads 1 and 3 differed significantly from each other, as well as triad 2 and 3 (the tables cannot be shown due to space constraints).

External, class and program unit's were the three most frequently used abstractions in the opening triad. For triad two, the same three categories appeared in a different order as class, external and program unit. In the final triad program unit and external again appeared, this time as the two most commonly used abstractions. A difference was seen in the third most popular abstraction, which was variable.

External and program unit appeared in different orders across all three triads, however they remained in the top three most used abstractions at all times.

Class usage was important in triad 1 and 2 while also appearing in the final triad as the fourth most frequently used abstraction.

Interestingly, algorithm appeared as the forth most frequently used category in the opening triad, it's popularity was reduced to almost half in the second triad while making no appearance in the final triad. This result indicates that perfective modifications were often concerned with algorithm improvements which must be understood as the beginning.

It is interesting to note that program unit's were consistently popular across all three triads for both the corrective

and perfective task types (with the percentages for triad 1 and 2 being extremely close while triad 3 in the perfective tasks doubled it's program unit usage), with external and class appearing consistently high also. Again the program unit usage suggests that code familiarity is a requirement not only for corrective tasks but also for perfective tasks. The external usage highlights the importance of program output for both task types. Class usage again appears more popular than methods for both task types.

Code excerpts were not as important in the final triad as was the case with the corrective descriptions. One possible reason for this is that perfective modifications require more explanation from the author and these outweigh the importance of the code excerpts showing the modifications. Variables and algorithms were found to be more important to perfective tasks than corrective.

An examination of the abstractions as high, middle and low for each triad showed a significant difference between triad 2 and 3 ($p = .016$). The middle abstractions were the most frequently used in triad 1 and 3 while coming a close second in triad 2. The higher abstractions occurred most frequently in triad 2 and were the second most frequently used abstraction in triad 1 and 3. The lower abstractions were the least frequently used across all three triads.

The opening triad uses abstractions in the same order as the opening triad for the corrective summaries. Triad two is again similar with an increase in the higher abstractions which were used only slightly more than the middle abstractions. The main difference was seen in the closing triad, where the last perfective triad has more middle abstraction usage compared with lower abstraction usage in the corrective summaries. In contrast, the lower abstractions were the least commonly used in the last perfective triad.

In conclusion, the middle abstractions were common across all triads for both corrective and perfective summaries. The higher abstractions were used more in the perfective summaries than the corrective, while the lower abstractions are used far less in the perfective summaries.

#### 3.1.2 In General for each Task Type

The following results describe the abstraction usage for each task type in general (irrespective of triad breakdown), due to space constraints the tables cannot be shown for the individual tasks.

**Corrective**

The general abstraction usage for the collective eighteen corrective type summaries will now be discussed. It was found that class and external were the two most frequently used abstractions. It is interesting to note the closeness of program unit and external at 10.69% and 9.16% respectively, which shows the importance of code chunking (program unit) and source code (code excerpt) during corrective tasks.

Examining the abstractions as high, middle and low, shows that middle abstractions were most frequent at 43.51%, followed by higher and lower abstractions at 29.77% and 23.66% respectively, while the remaining percent is taken by unclear and other.

**Perfective**

Abstraction usage for the collective fourteen perfective type summaries will now be discussed. Both external and class were again seen to be the most frequently used abstractions with an even higher percentage of occurrence than was seen for the corrective tasks. However, external was used more than class within the perfective tasks. Program unit again is the third most frequently used abstraction, showing that chunking is again an important activity for perfective tasks.

Examining the abstractions as high, middle and low, shows that middle abstractions were most frequent at 55.39%, followed by higher and lower abstractions at 34.53% and 7.91% respectively, while the remaining percent was taken by unclear and other. This pattern is similar to that seen in the corrective results with the main difference being the less frequent use of the lower abstractions.

**Preventive**

The general abstraction usage for the collective three preventive type summaries will now be discussed. External, again was found to be the most popular abstraction. However, a difference was seen in the fact that program unit, class and method were used equally. Despite the fact that only 3 preventive summaries were found, it is interesting to note the importance of external yet again.

Examining the abstractions as high, middle and low, shows that middle abstractions were most frequent again at 66.67%, followed by higher abstractions at 25%, while the remaining percent was taken by unclear. The same trend as seen in the corrective and perfective emerges here also, with the exception of no lower abstractions being recorded.

**Each Task by Triad Breakdown**

With the exception of the adaptive task type (only 1 summary), the middle abstractions were seen to be heavily used across all three triads, followed in order of popularity by the higher and lower abstractions. Triad 2 showed the most variation with perfective tasks displaying slightly more higher abstraction usage than lower.

#### 3.1.3 All Modifications by Triad Breakdown

The results presented here, show the abstraction usage for all 36 summaries under the *"modification"* theme, irrespective of their task type.

Table 2 shows the abstraction usage plotted as high, middle or low, for the entire set of 36 modification tasks.

It can be seen from the table, that the middle abstractions are again consistently used the most frequently across all 3

triads, followed in popularity by the higher and lower abstractions. It is interesting to note that there are no major differences between each of the triads. For example, it may be expected that the opening triad would contain higher levels of abstraction usage than the following triads, while triad 2 and 3 could be expected to contain lower abstractions as the problem solving progresses. However, this was not the case.

| | High | Middle | Low | Total |
|---|---|---|---|---|
| Triad 1 | 34 | 54 | 13 | 101 |
| | 33.66% | 53.46% | 12.87% | 100% |
| Triad 2 | 43 | 55 | 11 | 109 |
| | 39.45% | 50.46% | 10.09% | 100% |
| Triad 3 | 21 | 43 | 18 | 82 |
| | 25.61% | 52.44% | 21.95% | 100% |

**Table 2. Modification (36 Summaries): General Abstraction Usage**

| Abs | N | % |
|---|---|---|
| cla | 64 | 21.26 |
| ext | 63 | 20.93 |
| pru | 38 | 12.62 |
| var | 22 | 7.31 |
| met | 22 | 7.31 |
| cex | 20 | 6.64 |
| prog | 14 | 4.65 |
| loc | 11 | 3.65 |
| jvm | 11 | 3.65 |
| obj | 9 | 2.99 |
| com | 7 | 2.32 |
| unc | 6 | 1.99 |
| alg | 6 | 1.99 |
| int | 4 | 1.33 |
| oth | 3 | 0.99 |
| fea | 1 | 0.33 |
| **Total** | **301** | **100** |

**Table 3. Modification: 36 Summaries: General Abstraction Usage**

#### 3.1.4 In General for All Modifications

Table 3 shows the general abstraction usage for the entire set of 36 modification tasks irrespective of triads. The previous results have been summarised in this table, showing the top three most frequently used abstractions to be class, external and program unit, again showing the importance of chunking. Variable and method were both used equally, with code excerpt following closely behind.

Examining the abstractions as high, middle and low, shows that middle abstractions were most frequent at 71.76%, followed by lower and higher abstractions at 13.95% and 11.29% respectively, while the remaining percent is taken by unclear and other.

### 3.2. Patterns of Abstraction Switching

The following sections present the results of abstraction switching and usage by triad breakdown for each task type and the modification group. Patterns occurring 3 or less times are not reported except where only a single type of pattern was found. Abstraction switching here means the programmer used two or more consecutive segments using an abstraction (same or varying abstractions). For example if three segments were coded as class-class-class, then it may be stated that the programmer switched from class-class to class-class two times (ie. class-class means two consecutive segments were concerned with a class). It should be noted that abstraction switching included recording consecutive switches between similar and differing abstractions. However, it emerged that many of the switches were between the same abstractions and appear as such in the results.

#### 3.2.1 Each Task Type by Triad Breakdown

Both preventive and adaptive had 3 and 1 summaries respectively and hence produced no patterns of interest for reporting. For the 18 corrective summaries, no abstraction pattern occurred more than three times.

**Perfective**

Triad 1 of the fourteen perfective summaries had 11 patterns. The most frequently occurring pattern was the external-external pattern which occurred 8 times (72.73%). Triad 2 had 14 pattern occurrences, the most frequently used pattern was the class-class occurring 4 times (28.57%). Interestingly, class was found to be used consecutively 3 times in a row, ie. class-class-class, 3 times (21.43%). Thus, it can be said that the class-class pattern occurred 10 times (58.82%). Triad 3 had no patterns occurring more than 3 times.

In summary, with the exception of the *perfective* task type, no patterns were found for the individual task types when examined over the triads. Within the *perfective* task type, the importance of consecutive uses of the external abstraction in the opening triad was seen, followed by consecutive uses of the class abstraction in the middle triad.

#### 3.2.2 In General for each Task Type

The following results describe the abstraction pattern usage for each task type in general (irrespective of triad breakdown).

**Corrective**
The eighteen corrective summaries only had one pattern occurring more than 3 times, which was the class-class pattern occurring 6 times (23.08%).

**Perfective**
For the fourteen perfective summaries, 4 patterns occurred more than 3 times. These were class-class-class and external-program unit at 5 times each (15.62%), external-external-external and external-external at 4 times each (12.5%). It is interesting to note the consecutive use of the external-external pattern, effectively external-external can be said to have occurred 12 times. The consecutive use of class-class was also noted from both the class-class-class pattern and the less frequently occurring class-class-program unit pattern which occurred twice. Effectively, the class-class pattern can be said to have occurred 12 times also.

**Preventive**
The external-external pattern occurred twice during the 3 preventive summaries.

In summary, examining the patterns over the triads for each task type did not yield many results with the exception of triad 1 and 2 of the perfective task type. The patterns found in these triads were consistent with those found during a general examination of the task types. Resulting in the observation that consecutive use of the external abstraction was common, the same applies to consecutive use of the class pattern.

It is also worth noting that the program unit abstraction was often associated with an external reference during the perfective summaries (program unit-external-external occurred 2 times, and external-program unit occurred 5 times). It should also be noted that the strength of the external followed by an external may be observed as it appears without exception across all three triads for all task types and hence will be shown to be one of the frequent patterns for modifications in general.

The same is true of the class-class pattern, which again appears without exception across all three triads for all task types, appearing more frequently than the external-external pattern.

In summary, the class-class pattern was a frequently used pattern for both the *corrective* and *perfective* task types. The ext-ext pattern was also used to the same extent in the *perfective* tasks. Although, only three *preventive* tasks existed, the external-external pattern emerged again.

#### 3.2.3 All Modifications by Triad Breakdown

The results presented here, show the abstraction pattern usage for all 36 summaries under the *"modification"* theme by triad, irrespective of their task type.

In triad 1 for all 36 modifications, only two patterns occurred more than 3 times. These were the class-class and external-external-external patterns at 4 times each (20%). It is worth noting that the external-external pattern occurred 2 times, making the external-external pattern effectively occurring 20 times when taken into account with the external-external-external pattern.

Again, only two patterns occurred more than 3 times in triad 2. These were the same abstractions found in triad 1, in this case the class-class occurred 7 times (24.14%) and the external-external pattern occurred 6 times (20.69%). It is also worthing noting that the class-class-class pattern occurred 3 times, effectively making the class-class pattern occur 13 times.

No pattern in triad 3 occurred more than 3 times, however the popularity of the class-class and external-external patterns were evident again through the external-external-external pattern occurring 3 times, effectively making the external-external occur 6 times. While the class-class-class pattern and the class-class pattern occurred 2 times each, effectively making the class-class pattern occur 6 times.

#### 3.2.4 In General for Modifications

The following results describe the abstraction pattern usage for all of the modifications in general (irrespective of triad breakdown).

The patterns found overall for the 36 modifications included 9 patterns, which occurred more than 3 times. The frequency of the external-external pattern can be seen through the external-external pattern occurring 8 times (7.34%), while external-external-external occurred 7 times (6.42%) and program unit-external-external occurring 2 times (1.83%), effectively making the external-external pattern occur 24 times in 36 summaries.

Again, the popularity of the class-class pattern is evident through the class-class-class pattern occurring 7 times (6.42%), class-class pattern occurring 6 times (5.5%), class-class-class-class 3 times (2.75%), class-class-method, method-class-class and class-class-class-class-class 2 times each (1.83% each). Effectively, making the class-class pattern occur 45 times in 36 summaries.

The next three most popular patterns showed the interaction of program unit with varying abstractions including the external and program unit at 5.5% each and the class at 3.67%.

Next in frequency, occurred the variable-variable and external-method patterns, both occurring 4 times each (3.67%).

Of the less frequently occurring patterns, it is interesting to note the consecutive use of the object-object pattern, which occurred 3 times. The variable abstraction was also seen to be used consecutively variable-variable pattern, appearing 4 times. Also the class-variable and class-method occurred 3 times each.

In summary, class-class appears most frequently everywhere, without exception. The same is true, although less frequently, of the external-external pattern.

Program unit's were seen to interact frequently with other abstractions, for example, in triad 1, program unit-program unit and program unit-variable both occurred 10% each. In triad 2, program unit was followed by external (10.34%) and method (6.9%), while preceeded by external (6.9%) and class (6.9%). In triad 3, program unit was preceeded by external (16.67%) and followed by variable-variable (11.11%) and a consecutive program unit (11.11%).

In general for modifications, program unit was seen to be followed by a consecutive program unit, code excerpt, variable, external, program, method, external-external, and variable-variable. Program unit was also seen to be preceeded by external, class, variable and method.

In conclusion, taking all this program unit switching into account, it can be said that abstraction switching for modifications in general, involves switching from programmer assigned chunks/actions (program unit's) to/from the listed abstractions above. Consecutive examinations also take place of the class abstractions as well as the external abstractions.

## 4. Conclusion

Corrective type modifications were seen to be the most frequent maintenance activity. When examined by triad, class usage and program unit usage were consistently popular across all three triads. It is also interesting to note the high level of class usage which outweighs other higher abstractions such as objects and descriptions at the program level. Overall, corrective modifications were shown to be performed using the middle (within program level) abstractions which include heavy usage of external and program unit abstractions. It is informative to compare these results with Vans and von Mayrhauser [20], where it was found that *"programmers work at all levels of abstraction (code, algorithm, application domain) about equally"* during *corrective* maintenance. In contrast, the *corrective* results here show many more occurrences of the middle abstraction usage (within program level). It is also important to note the high class usage which ranked highly in the results for perfective and preventive tasks.

Consistency was observed, when the same abstractions appeared for the perfective tasks, showing that code familiarity is a requirement not only for corrective tasks but also for perfective tasks. The external usage highlights the importance of program output for both task types. Class usage again showed it's importance. The middle abstractions were common across all triads for both corrective and perfective summaries. The higher abstractions were used more in the perfective summaries than the corrective, while the lower abstractions are used far less in the perfective summaries.

The only significant difference in abstraction usage between triads was seen in the perfective tasks where discussions of algorithms were frequent in the opening triad, indicating that algorithm understanding is often the first task for perfective type tasks.

By comparing corrective, perfective and preventive task types, it can seen that the most common order of abstraction usage across all three triads is in the order of middle (within program level), high (program and architecture level), followed if at all, by the lower abstractions, with very few exceptions. A similar trend emerged for the overall *modification* results where the middle abstractions ranked highly, followed by lower and higher abstraction respectively.

Vans and von Mayrhauser stated that programmers *"frequently switch between levels of abstraction"* during corrective maintenance [20]. In contrast, the trend observed here was that middle level (within program level) abstraction switching occurred most often (between consecutive classes and external references).

In summary, the study set out to examine the abstraction usage for modification tasks. The purpose of which was to have implications for the design of supportive software visualisation tools. While this study cannot answer which presentation techniques are appropriate for each abstraction, e.g. call graph, SeeSoft view [8], etc, it can however contribute to the knowledge of which abstractions are appropriate to present on screen for modification queries.

No statistically significant value was obtained when the task types were compared against each other for abstraction usage. This leads us to the summary of modification abstraction usage in general, which showed that the class, external and program unit were the three most frequently used abstractions. Followed by variable, method and code excerpts at similar usage levels.

The implications for software visualisation design indicate that an emphasis on the class level views needs to be made. This is particularly true when considered with the observed consecutive usage of class descriptions from the pattern results, that is, multiple queries at the class level should be supported. The next abstraction was the external, which supports the need for program output, file output, streams, etc. to be accessible during runtime. Program unit's were the next most frequent abstraction, which highlights programmers summarising or chunking the actions within the program. Examples of software visualisation support in this

area could be annotated views, where the programmer can attach post-it notes to unit's of code.

Future work includes providing the coding manual to more independent coders in order to further verify the kappa result. Also, the abstractions will be examined further in order to determine which views were applied, for example, were the class abstractions described from a data structure context or from a functional context etc. In this way, the implications for the design of a supportive software visualisation tool may be detailed further.

## 5. Acknowledgements

The first author has been supported by the Science Foundation Ireland Investigator Programme, B4-STEP (Building a Bi-Directional Bridge Between Software ThEory and Practice).

## References

[1] Apache jakarta, open source java project page. *http://jakarta.apache.org*.

[2] Carnegie mellon software engineering institute glossary. *http://www.sei.cmu.edu/str/indexes/glossary/software-maintenance.html*.

[3] IEEE Standard for Software Maintenance. *IEEE Std 1219-1998*, June 1998.

[4] A. Capiluppi, M. Morisio, and J. F. Ramil. Structural evolution of an open source system: A case study. In *Proceedings of the IEEE International Workshop on Program Comprehension 2004*.

[5] N. Chapin, J. E. Hale, K. M. Kham, J. F. Ramil, and W.-G. Tan. Types of software evolution and software maintenance. *Journal of Software Maintenance*, 13(1):3–30, 2001.

[6] J. Cohen. A coeffi cient of agreement for nominal scales. *Educational and Psychological Measurement*, 20:37–46, 1960.

[7] D. Cubranic and G. Murphy. Hipikat: Recommending pertinent software development artifacts. In *Proceedings of the IEEE International Conference on Software Engineering 2003*.

[8] S. Eick, J. L. Steffen, and E. E. Summer. Seesoft - a tool for visualizing line oriented software statistics. *IEEE Transactions on Software Engineering*, 18(11):957–968, November 1992.

[9] J. Good. Programming paradigms, information types and graphical representations: Empirical investigations of novice comprehension. *Ph.D. Thesis, University of Edinburgh*, 1999.

[10] K. Krippendorf. Content analysis: An introduction to its methodology. *Sage Publications*, Second Edition, ISBN: 0-7619-1545-1, 2004.

[11] B. Lientz and E. B. Swanson. Characteristics of application software maintenance. *Communications of the ACM*, 21(6):446–471, 1978.

[12] B. P. Lientz, E. B. Swanson, and G. E. Tompkins. Characteristics of application software maintenance. *Commun. ACM*, 21(6):466–471, 1978.

[13] R. J. Martin and W. M. Osborne. Guidance on software maintenance. *National Bureau of Standards Special Publication 500*, 1983.

[14] J. Miller. Techniques of program and system maintenance. *Winthrop Publishers*, 1981.

[15] N. Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology*, 19:295–341, 1987.

[16] S. L. Pleeger. Software engineering: the production of quality software. 2nd Edition(ISBN 0-02-395115-X), 1991.

[17] R. S. Pressman. Software engineering: A practitioner's approach. 5th Edition(ISBN 0-07-709677-0), 2000.

[18] S. R. Schach, B. Jin, L. Yu, G. Z. Heller, and J. Offutt. Determing the distribution of maintenance categories: Survey versus measurement. *Empirical Software Engineering*, 8:351–365, 2003.

[19] E. B. Swanson. The dimension of maintenance. In *Proceedings of the Second International Conference on Software Engineering, 1976*.

[20] A. M. Vans, A. von Mayhauser, and G. Somlo. Program understanding behavior during corrective maintenance of large-scale software. *Int. J. Hum.-Comput. Stud.*, 51(1):31–70, 1999.