DOES GOD CLASS DECOMPOSITION AFFECT COMPREHENSIBILITY?

Bart Du Bois[1], Serge Demeyer[1], Jan Verelst[2], Tom Mens[*] and Marijn Temmerman[†]
[1]Lab On ReEngineering
[2]Dept. of Management Information Systems
Universiteit Antwerpen
Antwerpen, Belgium
email: {bart.dubois, serge.demeyer, jan.verelst}@ua.ac.be

ABSTRACT
Continuous alterations and extensions of a software system introduce so called god classes, accumulating ever more responsibilities. As god classes make essential steps in program comprehension harder, it is expected that effective and efficient techniques to resolve them will facilitate future maintenance tasks.

This work reports on a laboratory experiment with 63 computer science students, in which we verified whether the decomposition of a god class using well-known refactorings can affect comprehensibility of the relevant code part. Five alternative god class decompositions were derived through application of refactorings, by which the responsibilities of a natural god class were increasingly split into a number of collaborating classes.

Our results indicate that the derived class decompositions differed significantly with regard to the ability of students to map attributes in the class hierarchy to descriptions of the problem domain. Moreover, this effect has been found to interact with the institution in which the participants were enrolled, confirming that comprehensibility is a subjective notion for which we have to take into account people's skills and expectations.

This work indicates that improving comprehensibility is within the grasp of a single maintainer preparing for future change requests by redistributing the responsibilities of a god class using well-known refactorings.

KEY WORDS
Comprehension, refactoring, decomposition

1 Introduction

Class decomposition is one of the key activities in object-oriented software development. It is applied in the refinement from object-oriented analysis to design, by which entities and operations recognized in the problem domain are split up or merged into classes and relationships between them [1]. Furthermore, the incorporation of ever changing – and often unanticipated – requirements into the initial design requires frequent reconsideration of the class decomposition, lest the software system loses its economical value in a highly competitive environment.

Many books have been published on the creation and evaluation of class decompositions. The *creation* of a class decomposition is supported by modeling techniques (e.g. nouns/verbs decomposition [2, 3], and the use of Class-Responsibility-Collaborator cards [4]), modeling notations (mostly UML [5]) and patterns (such as Design Patterns [6, 7] and Analysis Patterns [8]). Its *evaluation* on the other hand, is supported by quality models (e.g. ISO 9126-3 [9]), quality guidelines [10, 1, 11], design heuristics [12] and anti-patterns [13].

Despite the manifold works regarding class decomposition, its relationship with external quality characteristics has hardly been studied. To the best of our knowledge, there is only one empirical study regarding the relationship of class decomposition with external quality attributes. In their work, [14] demonstrate maintainability differences between two alternative Java designs illustrating either a centralized or delegated control style. Their results suggest that novice developers had more problems understanding a delegated control style, indicating that comprehensibility is a subjective notion which should take into account the methods of organization preferred by the people maintaining the system. In other words, working towards the objective notion of optimal comprehensibility will result in class decompositions which are suboptimal for the individual maintainers.

We exploit the subjective nature of comprehensibility by addressing it on the scale of a single maintainer preparing for future maintenance tasks. Accordingly, we ask ourselves whether refactoring a single, yet vital part of a software system using a small number of refactorings can result in improved comprehensibility.

To answer this question, we redistributed the responsibilities of a single natural god class to a number of collaborating classes using small and well-known refactorings. This restructuring was performed in a number of steps, each resulting in a different class decomposition. Subsequently, the comprehensibility of these decompositions was assessed by observing master-level computer science students perform a controlled enhancive maintenance task. This work reports on the verification of comprehensibility differences among the alternative class decompositions.

The paper is organized as follows. We demarcate

[*]University of Mons-Hainaut, tom.mens@umh.ac.be
[†]Karel de Grote-Hogeschool, marijn.temmerman@kdg.be

comprehensibility within the context of this work in section 2. The concept of class decomposition is introduced in section 3. Section 4 illustrates the experimental set-up, of which the collected data is analyzed in section 5 and discussed in section 6. The threats to validity are analyzed in section 7. Finally, we provide references to related work in section 8 and conclude in section 9.

2 Comprehensibility

Program comprehension can be defined as the recognition of the overall program function, an understanding of intermediate-level processes including program organization, and comprehension of the purpose of each statement [15]. This definition accentuates its multidimensional character, making it virtually impossible to construct a complete test for comprehensibility of a software system. Therefore, any reference to comprehensibility should be accompanied with a clear specification of the scope of comprehension.

In this work, we focus on comprehensibility of the program organization. More specifically, we are interested in the *structural organization of data and procedures* referred to in the problem domain description. In an object-oriented context, data is organized into attributes, and procedures are organized into methods consisting of algorithmic steps. As stated by [16], *the task of understanding a program for a programmer is one of constructing, or reconstructing, enough information about the modeling domains that bridge between problem and executing program to perform the requested tasks.* Therefore, we demarcate comprehensibility as the ability to map attributes and steps in the execution of a method to descriptions of the problem domain.

3 God Class Decomposition

The object-oriented paradigm introduced the encapsulation of related procedures and data into conceptual groupings called object classes. One of the undoubted benefits of encapsulation is that it allows to control the extent to which changes to attributes and behavior of object classes propagate through the system.

However, the resulting decomposition of the problem domain into a set of collaborating classes leads to a more subtle distinction between procedures and data. At the system level, one can identify *controllers* which exhibit more methods and less data, and *entities*, exhibiting more data and less methods [2]. This distinction is promoted in design patterns such as the Entity-Boundary-Controller pattern [2], the Model-View-Controller pattern [17], the Presentation-Abstraction-Controller pattern [18] and the Use-Case controller pattern [3]. While the concept of a controller originally envisioned the handling of input interactions, today's controllers resemble workflow engines more closely by governing the flow of control associated with a complete user task.

As one of the most agreed upon bad practices, the so called *god class* anti-pattern [12] tends to tangle aspects of both controllers and entities. Its occurrence within numerous software systems can be explained by the way that god classes are formed. God classes are not designed, they are grown. Ever changing requirements cause continuous alteration and extension of a software system. The high rate of source code adaptations tends to introduce design flaws such as misplaced responsibilities. Ultimately, some classes take the form of black holes, accumulating and attracting ever more responsibilities. These god classes are considered to be the main targets for reorganization, and the preferred approach to resolve them is to extract both behavioral and data classes, clarifying the distinction between procedures and data [19].

To exemplify the refactoring of a god class, we illustrate a classical reengineering scenario.

3.1 Illustrative Refactoring Scenario

Consider the problem domain in which the following procedure for filtering incoming mails into appropriate mailboxes applies:

1. Collect the mailheaders of all mails awaiting filtering.
2. Collect the filters to be applied.
3. Apply each filter on all mails awaiting filtering.
   (a) Evaluate the headerfield-values of the current mail.
   (b) Move the mail to the target mailbox in case of a match.
4. Move unfiltered mails to the inbox.

A common – yet bad practice – class decomposition for fulfilling the filtering procedure is:

Decomposition A – The software design consists of a single class. This god class governs the flow of control and manages the internal representation of domain entities.

The responsibilities of the god class can be incrementally redistributed either to collaborating classes – if present – or to new classes that are pulled out of the god class. To ensure behavioral equivalence of the resulting class decompositions, behavior preserving transformations called *refactorings* are applied [20].

Decomposition B – Two filter classes are extracted from the god class: one for filtering according to the equality of a header field with a given string, and one according to the mere presence of a string in a header field. This extraction of two controller classes is done using the refactoring *Replace Method with Method Object* on the two filtering methods from the original god class (step 3).

Decomposition C – The vector in which the mail header field values are stored (step 1) is extracted and encapsulated in a mail header entity class. Using different terminology, this class would be categorized as an entity, model, or abstraction class. The transformation used is similar to the *Introduce Parameter Object* refactoring.

A benefit of factoring out the controllers is that they deal with fewer issues. A benefit of factoring out the entities is that it provides a clear and encapsulated representation of a domain entity.

The refactoring scenario from decomposition A to C would be considered an adequate disentangling of the initial god class. Nonetheless, to anticipate future needs of reusability and flexibility, the extraction of controller and entity classes might continue.

Decomposition D – The action which is to be applied by a filter (step 3b) is extracted in a filter action class. This delegation to a new controller class required the application of the *Extract Method* and *Replace Method with Method Object* refactorings.

Decomposition E – The representation of a mail header field as consisting of a key and value pair (step 3a) is extracted and encapsulated in a header field class. Once again, the extraction of this entity class is done using a refactoring similar to *Introduce Parameter Object*.

As the newly introduced controller and entity classes in respectively decomposition D and E bear little responsibilities, their weight in the overall class hierarchy is very small. These steps might be considered over-engineering. Yet, as numerous cases of over-engineering can be found in practice, these refactoring steps are quite realistic.

Because the god class decompositions were derived by applying behavior-preserving refactorings, they only differ w.r.t. their internal organization. Qualitative differences between them – e.g.. their comprehensibility – are consequently solely due to their organizational differences.

4 Experimental Set-Up

As indicators of comprehensibility, we rely on the accuracy and execution time of a task requiring comprehension of both the data and procedural organization.

It is important to note that in comparisons of accuracy and execution time measurements, one must bear in mind that increased accuracy can be compensated by longer execution time and reversely. Accordingly, we clearly distinguish the following research questions:

1. Do some decompositions allow to achieve higher comprehension accuracy than others?
2. Do some decompositions allow to achieve comprehension faster than others?

4.1 Experimental Design

The experiment was designed to use both randomization and a pretest.

Subjects are randomly assigned to one of five groups (A to E). Each group applies two change localization tasks. The first task is identically applied by all groups, and provides pretest observations . The second task , however, is applied for each group on a different class decomposition, and provides *posttest* observations , incorporating manipulation of the experimental factor under study.

Differences between posttest observations are attributable to a) characteristics of the class decompositions (A–E); b) characteristics of individual performance (partially described by the pretest observations); and c) the interaction between the characteristics of a decomposition and individual performance. For example, the effect of class decomposition on task performance might differ between high and low performant participants.

4.2 Software System

The software system used in the experiment is an open source Java mail client (Yamm v0.9.1), which consists of 13KLOC over 66 classes. In order to prepare the assignments, we moved the god class responsible for the filtering procedure to a separate package, thereby forming decomposition A. The total number of non-comment, non-whitespace lines of code of decomposition A was 276, which is a commonly used order in comprehension experiments [21]. By keeping all extracted classes in this package, we had a means to clearly bound the part of source code which was of interest for the assignments.

Incrementally, the god class was refactored in order to extract both controller and entity classes, resulting in decompositions B to E.

This software system was chosen for the following reasons. It has been shown that differences w.r.t. the experience with the application domain has an influence on conceptual modeling effort [22, 23, 24]. By assuring that the application domain was well known by all participants, this effect was neutralized. As a second argument, the system is a typical example of continual extension, which leads naturally to the fostering of god classes. The god class responsible for the filtering procedure is therefore a natural and realistic god class.

4.3 Experimental Assignments

The assignments were chosen to be quite small and simple, which allows us to focus more on comprehension effort than on the degree of accuracy which can be achieved. For practical reasons, the assignments were to be solvable in an average time frame of half an hour.

The pre- and posttest assignments (respectively illustrated in Figures 1 and 2) were chosen to be similar tasks to be applied on different source parts. Therefore, during the pretest, participants could normally not have gained comprehension of the code related to the posttest.

All assignments were provided under the form of a Change Request, which specifies its scope, provides a description of the associated scenario from the user-perspective, illustrates the semantics of domain concepts (e.g. examples of mail headerfields) and dictates the changes to be applied by the participant.

Figure 1 displays the pretest assignment, and requires the localization of two information items which are to be printed to standard output at a specific algorithmic step.

Output the header-fields (name and value) of a mail to standard output at the moment it is being sent. Use the following output format as an example:

```
To: <emailaddress of addressee(s)>
Subject: <subject of the mail>
---
... (next output block)
```

Figure 1. Pretest assignment

The assignments stimulate the following comprehension activities:

- *Map attributes to concepts described in the problem domain.* E.g. localize the attribute representing the To header field.
- *Map steps in the execution of methods to actions described in the problem domain.* As the flow of control passes through the procedural organization, different steps in the execution of methods can match an action as the sending of a mail. E.g. one could print information immediately before or after delegating control. The limited visibility of attributes implies that not all attributes can be printed at the same step in execution.

The posttest assignment is illustrated in Figure 2. It requires the localization of 8 information items, to be printed at the algorithmic step corresponding to 3b in the filtering procedure presented in section 3.

4.4 Experimental Variables

4.4.1 Independent Variables

- **IV1:** *Class Decomposition.* This ordinal variable indicates the number of classes which were extracted from the god class to derive the class decomposition. Therefore, the possible values are {A,B,C,D,E}.

Output the header-fields (name and value) and the properties of the applied filter to standard output at the moment a mail is being filtered. Use the following output format as an example:

```
From=<from header-field value>
To=<to header-field value>
Subject=<subject header-field value>
Reply-To=<reply-to header-field value>
Filtered Header Field Name=<one out of
 {from,to,subject,reply}>
Filtering Method=<one out of
{equals,contains}>
Filtering String=<string of which the
presence in a header-field value triggers
the application of the filter>
Moved Mail To=<indication of the mailbox
to which the mail is to be moved when
filtering, e.g.. boxes/trash>
---
... (next output block)
```

Figure 2. Posttest assignment

- **IV2:** *Institution.* The experiment was replicated with students enrolled in different institutions (see Table 1), providing a nominal variable. Possible values are {MAS1,MAS2,MAS3}. This variable is incorporated as curriculum differences are a potential cause of variation, and can thereby provide an explanation for differences in individual performance.

4.4.2 Dependent Variables

In observing the execution of a change request in which attributes and particular steps in program execution have to be localized, we distinguish between:

- **PLA:** *Percentage of Localized Attributes.* Measured as the percentage of attributes that are localized by a participant and match with the requested domain concepts. The total number of attributes to be localized is two for the pretest and eight for the posttest.
- **PLAS:** *Percentage of Localized Algorithmic Steps.* Measured as the percentage of requested information items for which a correct step in execution was found – even if the wrong attribute is printed.
- **ET:** *Execution Time.* Measured as the time frame between the point at which the change request is read and the point at which the task is finished, in seconds.

4.5 Hypotheses

We operationalize our research questions into the following null hypotheses:

$H_{0,PLA}$ The number of localized relevant attributes (PLA) does not differ w.r.t. the class decomposition.

$H_{0,PLA-ET}$ The time required to localize all relevant attributes does not differ w.r.t the class decomposition.

$H_{0,PLAS}$ The number of localized relevant algorithmic steps (PLAS) does not differ w.r.t. the class decomposition.

$H_{0,PLAS-ET}$ The time required to localize all relevant algorithmic steps does not differ w.r.t the class decomposition.

It is clear that the rejection of $H_{0,PLA}$ and/or $H_{0,PLAS}$ would answer research question 1 positively, and similarly for hypotheses $H_{0,PLA-ET}$ and/or $H_{0,PLAS-ET}$ and research question 2.

As an exploratory study, we specify the significance criterion at α=0.10.

4.6 Experimental Subjects

The same set-up was applied in five replications, employing a total of sixty-three master-level computer science students (see Table 1).

As the names of the institutions are of no value within the context of this paper, we have encoded them.

Table 1. Replications of the experiment.

Rep	Institution	Curriculum	#Subjects
1	MAS1	CS	9
2	MAS1	CS	5
3	MAS2	ICT	20
4	MAS3	CS	14
5	MAS3	Cs	15
Total			63

1–2 In October 2004, fourteen third year mathematics and computer science (CS) students (four years master-level) participated in the context of a crash-course on the open source Integrated Development Environment Eclipse. This course is optional in the third year of the four years master level (mathematics and) computer science curricula at the University of X, Belgium. An experimental session was organized for 9 computer science and 5 mathematics and computer science students separately. As third year students, they are moderately experienced in object-oriented software development in Oberon and C++. However, these students have no experience with maintenance.

3 In November 2004, twenty last year students industrial engineering in ICT-electronics (four years master-level) participated in the context of a mandatory course on distributed software in Java at the University of Y, Belgium. These students have limited experience with object-oriented software development in C++ and Java.

4–5 In March 2005, twenty-nine last year computer science students (four years master-level) participated in the context of a mandatory course on software evolution at the University of Z, Belgium. As last year students, they are experienced in object-oriented software development in Scheme and Java. These students previously extended an existing system in a project assignment and therefore have minor experience with the tasks at hand.

The University of Y is a university of professional education (*hogeschool* in Dutch), contrary to the universities of X and Z who provide research-oriented education. Before the Bachelor-Master reformation in Europe, professional education led to the bachelor level while research-oriented education led to the master level.

There is a clear distinction between the curricula of MAS1 and MAS3 students (computer science) and the ICT-electronics curriculum of the MAS2 students. The latter are less focused on software engineering and mostly target hardware and network design as well as datacommunication. While it is not uncommon for people with an ICT-electronics profile to be responsible for development and maintenance in the software engineering industry, we might expect that their different education with regard to software engineering principles had an influence on preferred class decompositions.

4.7 Experimental Procedure

Subjects were randomly assigned to treatments, in which each of the class decompositions represents a different treatment. Due to the practical setup, subjects were aware that they participated in an experiment. However, the purpose of the experiment was not discussed.

To reduce differences in experimental procedure over the experimental replications, subjects were referred to the task assignment sheets as much as possible. As no diagrams were provided, the only source of information to comprehend the data and procedural organization was the source code which could be browsed using the Integrated Development Environment Eclipse (version 2.1.3).

Timestamps after reading and finishing the assignment allowed to calculate the task duration. As the participants were required to extensively use the Concurrent Versioning System (CVS) support, we were able to verify the correctness of the handwritten timestamps, thereby providing the measurement of the dependent variable ET.

The pre- and posttest were performed sequentially, both delimited at one hour. After the experimental session, the CVS repository was harvested to collect the solution of each participant (as well as the path to this solution). This allowed the measurement of the dependent variables PLA and PLAS. As these accuracy ratings were performed by the first author, this can be regarded as an opportunity for experimental bias. However, the evaluation technique was clear and objective, as there is only a single correct outcome for the localization of an attribute or algorithmic step.

5 Data Analysis

In this section, we report the analysis of the data by which we want to answer our two research questions. Firstly, do some decompositions allow to achieve higher comprehension accuracy than others? Secondly, do some decompositions allow to achieve comprehension faster than others?

5.1 Descriptive Statistics

Both pretest and posttest observations of PLA were skewed significantly to the high end of the scale, and exhibit a clear peak. The pretest observations of PLAS demonstrated similar deviations from normality. Its associated posttest observations exhibit a flat distribution. While both pretest and posttest observations of ET were indicated to conform to the normal distribution, this was not the case for PLA and PLAS. Therefore, we must rely on non-parametric tests for comparing accuracy observations.

Table 2. Descriptive statistics for the observations on the posttest (N=63)

	pretest		posttest	
measure	mean	stdDev.	mean	stdDev.
PLA	96.0%	16.3%	86.2%	21.4%
PLAS	93.6%	19.0%	41.2%	43.4%
ET	1284	535	2471	799

The data described in Table 2 suggests that most of the participants had no problems localizing attributes and algorithmic steps for the pretest. During the posttest, most of the participants were able to localize a large part of the attributes.

However, most participants were unable to localize the correct algorithmic steps. Moreover, within the same experimental groups, the individual differences in the ability to localize algorithmic steps in the posttest were large. This finding could only be observed in the possttest observations of PLAS, and persisted even within groups of participants from the same institution comprehending the same god class decomposition. Such variation decreases the sensitivity of tests for differences between different experimental groups. The associated lack of power causes us to remain inconclusive about the effect of god class decomposition on comprehensibility of the procedural organization ($H_{0,PLAS}$ and $H_{0,PLAS-ET}$). We therefore refrain from elaborating on statistical tests concerning the PLAS observations.

Participants performed worse on the posttest, as the information items were less related. E.g. the posttest assignment requires the localization of information about the mailheader, the filter and the action to be applied by the filter.

Table 3 displays the probability (p-value) that individual performance differences between participants were already present at pretest (and posttest) time. In this experiment we reject a null hypothesis on the condition that this probability less then or equal to 10% (α=.10). These values indicate no significant performance differences between the experimental groups working on the different class decompositions (A–E). Performance differences between participants from different institutions are plausible (p=.144), but this effect is under experimental control due to the rigorous procedure of randomization across experimental groups.

Table 3. Significance of the Kruskal-Wallis test statistic w.r.t. the PLA observations

factor	pretest	posttest
decomposition	.445	.089*
institution	.144	.353
interaction	.213	.078*

5.2 Do some decompositions allow to achieve higher comprehension accuracy than others?

Table 3 also evaluates whether there were significant performance differences between experimental groups during the posttest. The significant main effect of class decomposition regarding PLA indicates that the ability of participants to localize attributes differed significantly among the five god class decompositions derived through refactoring (p=.089). Moreover, this effect interacts significantly with the institution, indicating that participants from some institutions had more difficulties localizing attributes in a particular class decomposition than participants from other institutions (p=.078).

Figure 3 presents the mean value of PLA posttest observations per combination of the decomposition and institution factors. The line referred to in the legend as Overall illustrates that the ability of participants to localize attributes differed significantly among the different god class decompositions. Moreover, the results of the institutions differ with regard to the class decomposition at which the ability of participants to localize attributes was highest, as indicated by the different trends across different institutions. The results of the replications with com-

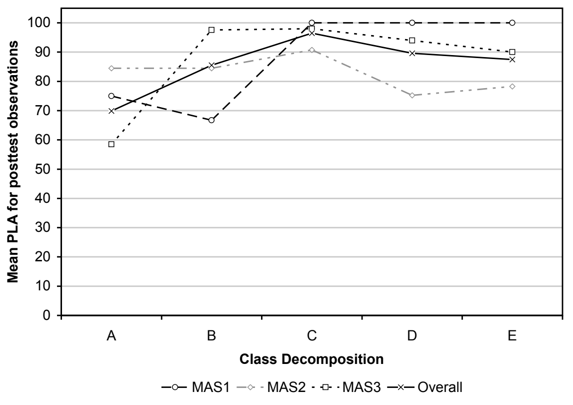

Figure 3. Mean Percentage of Localized Attributes (PLA) posttest observations

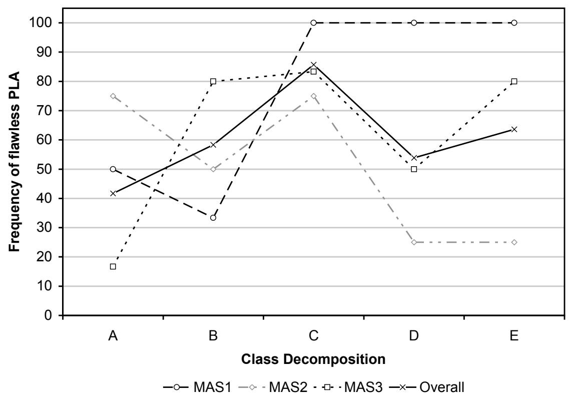

Figure 4. Frequency of observations with PLA=100

puter science students (MAS1 and MAS3) demonstrate absolute differences between class decompositions of more than 30 PLA. This means that for some decompositions, participants were able to localize an additional one third of the attributes in relation to other decompositions. Observed PLA differences between class decompositions of the ICT-electronics students (MAS2) were smaller, differing at most 15 PLA.

Summarizing, these observations allow us to reject $H_{0,PLA}$. Therefore, we can state that some decompositions allow to achieve higher accuracy than others with regard to comprehensibility of the data organization.

5.3 Do some decompositions allow to achieve comprehension faster than others?

To answer this question, we restrict our analysis to only those observations which localized all attributes (PLA=100), of which the frequency over class decompositions is illustrated in Figures 4.

An ANOVA model with factors decomposition, institution and their interaction was composed (α=.10, R^2=.463, Adj. R^2=.136). Table 4 provides the significance of the ANOVA test statistic, demonstrating neither the main nor the interaction effects of factors decomposition and institution factor to be significant.

Table 4. Significance of the ANOVA test statistic w.r.t. the execution time of posttest observations with PLA=100

factor	p-value	power
decomposition	.106	.683
institution	.265	.396
interaction	.893	.260

Figure 5 confirms that the mean ET of different institutions follows the same trend across decompositions. Differences between the time it took participants to localize all attributes reach up to 16 minutes between decompositions D and E, compared to an overall mean ET of 37 minutes. While the probability that differences of this order is low (p=.106), our significance criterion does not allow to classify these as significant.

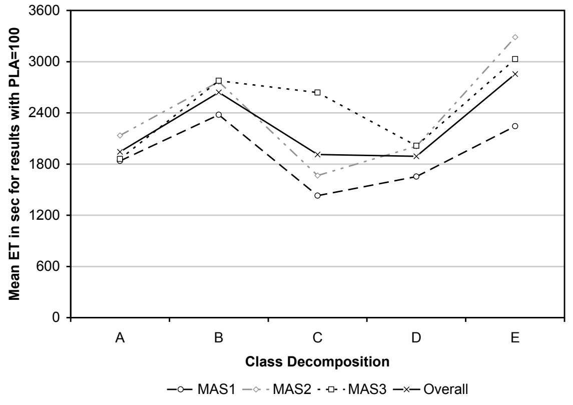

Figure 5. Mean ET of observations with PLA=100

Summarizing, we must accept $H_{0,PLA-ET}$. Accordingly, we cannot state that some decompositions allow to achieve comprehension of the data organization faster than others.

6 Discussion of Results

After analyzing the collected data we are able to reflect upon the two research questions.

Do some decompositions allow to achieve higher comprehension accuracy than others? Yes, our data demonstrates that participants were able to localize significantly more relevant attributes for certain god class decompositions. However, our data does not suggest an objective notion of ”optimal comprehensibility”. On the contrary, while the computer science students had more difficulties comprehending the original god class decomposition A, ICT-electronics students experienced less difficulties. It appears that differences in comprehensibility of the god class decompositions as perceived by the computer science students were larger than as evaluated by the ICT-electronics students. We interpret this finding as a confirmation that advanced object-oriented software engineering training can induce a preference towards particular styles of class decomposition.

However, we were unable to evaluate differences w.r.t. the ability to comprehend the procedural organization . The variation within experimental groups is known as *common cause variation*, and indicates an effect of unknown factors. In other words, participants within the same groups applied different strategies for localizing the relevant steps in the execution. In future work, we plan to perform an extensive post-analysis to identify these different strategies.

Do some decompositions allow to achieve comprehension faster than others? We did observe differences between the god class decompositions concerning the time it took participants to localize all attributes. The statistical significance of these differences approximated the preset significance criterion (p=.106). Curiously, while the class decomposition for which students demonstrated optimal comprehensibility differed between institutions (interaction effect), such an interaction could not be observed w.r.t. the time it took to localize all attributes. When we would regard those students being able to localize all attributes as performant students, this finding could suggest that there are few systematical differences w.r.t. class decomposition preference between the selection of performant students across different institutions.

7 Threats to Validity

There are severe limitations to the extent at which our results can be attributed to the specified cause, and the extent in which these results can be generalized. For clarity, we specify the main threats to validity.

7.1 Internal Validity

Internal validity is the degree to which the experimental setup allows to accurately attribute an observation to a specific cause rather than alternative causes. The threats we have identified are the following.

- Maturation might have affected participant performance due to the application of a pretest. However, randomized assignments of participants to groups should ensure that this effect does not lead to systematic group differences.

- The risk of an instrumentation effect was minimized through the use of a decidedly objective and clear comparison of the participant’s result with the unique correct result.

7.2 External Validity

External validity is the degree to which research results can be generalized outside the experimental setting or to the population under study.

- Characteristics of the problem domain at hand and the associated experimental tasks are that the concepts used were very simple, and familiar to all participants. Therefore, it is unclear whether systematic differences w.r.t. comprehensibility arise when the problem domain is more complex in terms of the number of entities, relationships, and their properties.

- The experiment reported in this work was performed in a laboratory setting with computer science students. Consistency of the observed effect over different institutions suggest that our findings can be generalized to the population of computer science students in general. However, a common characteristic of this population is their perspective on optimal decompositions, as most of the books providing examples are considered standard reading material. Therefore, it is unclear whether the observed effect would reoccur in case professional software maintainers would be used, as their experience with alternative decompositions might have stimulated a comprehension process that is more robust to variability in class decompositions.

8 Related Work

[14] compare differences between two alternative Java designs that had a centralized and delegated control style respectively. It is hard to clearly differentiate between these two styles in our alternative designs, as both controller and entity classes were extracted. The extraction of a controller class introduces delegated control, and extracting an entity class introduces centralized control. Following this argument, our design A clearly represents a centralized control style, designs B and D introduce more delegation, and designs C and E introduce centralization.

Regarding comprehensibility of the data organization, our results confirm the conclusions of [14], which state that novice developers had more problems understanding a delegated control style. Our data indicates a significant interaction effect between the decomposition and the institution factors. On average, our ICT-electronics participants achieved higher comprehension accuracy than the computer science participants for design A, but lower accuracy for designs C, D and E (see Figures 3 and 4). The execution time – for those observations with perfect comprehension accuracy – did not indicate a significant interaction effect, however.

[25] performed research on a so called Psychological Complexity Measure, which uses information on class responsibilities and class collaborations to calculate the psychological development complexity as a measure of effort. By defining a measure called Permitted Collaborations (PC), they suggest a means for estimating the cognitive load imposed on the developers of a software system from a set of Class-Responsibility-Collaborator cards [4] that represent a domain. One of the purposes of this measure is to ”aid the developers in selecting the right class decomposition when there are conflicting choices”. When considering each algorithmic step a separate responsibility, the PC measure confirms the empirical findings in this paper, indicating a decreasing complexity from decomposition A over B to decomposition C and an increasing complexity from decomposition C over D to decomposition E.

9 Conclusion

This experiment has been set up to verify whether god class decomposition can facilitate comprehension. *As one of the indicators of comprehensibility, the ability to map attributes in the class hierarchy with domain concepts was significantly affected by the decomposition and its interaction with the institution from which the participant was enrolled.* Moreover, this increased accuracy was not compensated by longer execution times of the comprehension task.

Our findings lead to two conclusions:

1. Improving comprehensibility is within the grasp of a single programmer or maintainer preparing for future change requests, as demonstrated using our illustrative refactoring scenario. This approach is in contrast with the more traditional perspective on the need for massive reengineering efforts for attaining improvements in external quality characteristics. Small scale refactoring during development can therefore be regarded as a complementary strategy to a separate preventive maintenance phase, differing both in required investments and resulting returns.

2. We are reluctant to accept the concept of *an optimal class decomposition* with respect to comprehensibility, since one of the factors which interacts with the class decomposition is the particular education of the subject performing the comprehension task. This observation has been previously demonstrated by [14], and suggests that guidelines for the reorganization of software systems should take into account the particular skills and expectations of the people maintaining the system. To be more precise, it is important to either adjust the organization of the software system to the methods of organization preferred by the people maintaining it, or to make these people accustomed to these methods of organization, e.g. by training.

This work does not provide guidelines on how to improve comprehensibility using refactorings. However, our conclusions can motivate software engineering researchers, practitioners and tool developers to study and support ways in which comprehensibility improvements can be achieved using controlled refactorings.

10 Acknowledgements

This work has been sponsored by the Belgian National Fund for Scientific Research (FWO) under grants ’Foundations of Software Evolution’ and ’A Formal Foundation for Software Refactoring’. Other sponsoring was provided by the European Science Foundation by means of the project ’Research Links to Explore and Advances Software Evolution (RELEASE)’.

References

[1] P. Coad and E. Yourdon. *Object-Oriented Design*. Yourdon Press, 1991.

[2] I. Jacobson, M.Christerson, P. Jonsson, and G. Overgaard. *Object-Oriented Software Engineering — A Use Case Driven Approach*. Addisson-Wesley, 1992.

[3] Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[4] David Bellin and Susan Suchman Simone. *The CRC Card Book*. Addison-Wesley, 1999.

[5] OMG. OMG Unified Modeling Language specification version 2.0. formal/2005-07-04, July 2005.

[6] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Languages and Systems*. Addisson-Wesley, 1994.

[7] Lutz Prechelt, Barbara Unger, Walter F. Tichy, Peter Brössler, and Lawrence G. Votta. Controlled experiment in maintenance comparing design patterns to simpler solutions. *IEEE Trans. Software Engineering*, 27(12):1134–1144, 2001.

[8] Martin Fowler. *Analysis patterns: reusable objects models*. Addison-Wesley Longman Publishing Co., Inc., 1997.

[9] ISO9126. Software product evaluation - quality characteristics and guidelines for their use. Technical report, ISO/IEC Standard ISO-9126, 1991.

[10] P. Coad and E. Yourdon. *Object-Oriented Analysis*. Yourdon Press, 1991.

[11] Lionel C. Briand, Christian Bunse, and John W. Daly. A controlled experiment for evaluating quality guidelines on the maintainability of object-oriented designs. *IEEE Trans. Software Engineering*, 27(6):513–530, 2001.

[12] Arthur J. Riel. *Object-Oriented Design Heuristics*. Addison-Wesley Publishing Company, April 1996.

[13] William J. Brown, Raphael C. Malveau, Hays W. McCormick, III, and Thomas J. Mowbray. *AntiPatterns: refactoring software, architectures, and projects in crisis*. John Wiley & Sons, Inc., 1998.

[14] Erik Arisholm and Dag I. K. Sjoberg. Evaluating the effect of a delegated versus centralized control style on the maintainability of object-oriented software. *IEEE Trans. Softw. Eng.*, 30(8):521–534, 2004.

[15] B. Sheiderman. Exploratory experiments into programmer behavior. *International Journal of Computer and Information Sciences*, 2(5):123–143, 1976.

[16] Ruven Brooks. Using a behavioral theory of program comprehension in software engineering. In *ICSE '78: Proceedings of the 3rd international conference on Software engineering*, pages 196–201, Piscataway, NJ, USA, 1978. IEEE Press.

[17] G. E. Krasner and S. T. Pope. A cookbook for using the model-view-controller user interface paradigm in smalltalk-80. *J. Object-Oriented Programming*, pages 26–49, August/September 1988.

[18] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-Oriented Software Architecture — A System of Patterns*. John Wiley & Sons, 1996.

[19] Serge Demeyer, Stéphane Ducasse, and Oscar Nierstrasz. *Object-Oriented Reengineering Patterns*. Morgan Kaufmann and DPunkt, 2002.

[20] Martin Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

[21] Dean Hendrix, II James H. Cross, and Saeed Maghsoodloo. The effectiveness of control structure diagrams in source code comprehension activities. *IEEE Trans. Softw. Eng.*, 28(5):463–477, 2002.

[22] D. Batra and J.G. Davis. Conceptual data modelling in database design: similarities and differences between expert and novice designers. *International Journal of Man-Machine Studies*, 37(1):83–101, 1992.

[23] Diana Kao and Norman P. Archer. Abstraction in conceptual model design. *International Journal of Man-Machine Studies*, 46(1):125–150, 1997.

[24] G. Shanks. Conceptual data modelling : an empirical study of expert and novice data modellers. *Australian Journal of Information Systems*, 4(2):63–73, 1997.

[25] John D. McGregor and Sanjay Kamath. A psychological complexity measure at the domain analysis phase for an object-oriented system. Technical report, Clemson University, 1995.