Transparent Persistence Appears Problematic for Software Maintenance
A Randomized, Controlled Experiment

Pietu Pohjalainen
Department of Computer Science, University of Helsinki, Helsinki, Finland

Keywords: Transparent Persistency, Self-configuring Software Architecture, Randomized, Controlled Experiment.

Abstract: Information retrieval from a database is the backbone of many applications. For building object-oriented systems with a relational persistence engine, a common approach is to use an object-to-relational mapping library for handling the mismatch between object-oriented and relational data models. These components should make the application programmer oblivious to the choice of relational database.
To study the effects of transparent persistency, we conducted a randomized, controlled study for 16 students, who were given a number of maintenance tasks on a sample software. For half of attendees, the sample software was written using the transparent persistency approach. For the second half the sample software used a self-configuring component for automatically generating database queries.
We found out that the group using transparent persistency were performing worse than the group using self-configurative queries. Attendees in both groups were using the same amount of time for performing the given maintenance tasks, but the transparent persistency group was outperformed by a factor of three in the number of correct submissions. The use of transparent persistency turned out to be a major error source. This gives us a reason to doubt the usefulness of transparent persistency in the long run.

1 INTRODUCTION

Relational databases are used for storing data in virtually all kinds of information systems. Telephone switching systems, e-commerce transaction systems, human resources information systems and e-mail indexing systems, for example, have implemented data persistency by relying on relational database engines. Although these engines often include some support for building application logic into them, it is popular to build applications using a general purpose language, such as Java or C#.

In these cases the way in which the relational database is used is a major architectural decision. Using plain database interfaces, such as Java's database connectivity interface (Fisher et al., 2003) easily leads to tedious work of manually implementing routines for storing and retrieving data to and from the database. Some researchers estimate that the low-level code for accessing a database can be up to 30% of the total line count of a normal business application written in direct access style (Atkinson et al., 1983; Bauer and King, 2004). Manual labour is known to be error-prone and have low productivity figures. For this reason, many architects choose to use object-to-relational mapping components, such as those defined by the Java persistency interface (DeMichiel and Keith, 2007) to handle the mismatch between object-oriented program logic and relational database models.

Object-to-relational mapping (ORM) components aim to reduce the manual labour needed in building object-oriented database applications. Instead of manually defining database queries for retrieving application data, an external component handles the generation of the needed queries. This is done by defining mappings between application objects and database tables. These mappings define how object-oriented structures, such as inheritance, object collections, and other object links should be made persistent in the database. The goal of this component is to liberate the object-oriented programmer from thinking at a relational database abstraction level. For this, the we use the term *transparent persistency*.

Transparent persistency refers to the idea of data persistency should not affect the way programmers build the application logic. Instead, persistency should be a modular aspect, as envisioned in aspect-oriented programming (Kiczales et al., 1997). At first glance, this seems to be a clever idea: in many cases, the routines needed for storing and retrieving data from the database makes algorithms look unnecessar-

Pohjalainen P..
Transparent Persistence Appears Problematic for Software Maintenance - A Randomized, Controlled Experiment.
DOI: 10.5220/0004417900250035
In *Proceedings of the 8th International Conference on Evaluation of Novel Approaches to Software Engineering* (ENASE-2013), pages 25-35
ISBN: 978-989-8565-62-4
Copyright © 2013 SCITEPRESS (Science and Technology Publications, Lda.)

ily complex and cluttered.

However, the drawback of the approach is that the exact workings of the software system are blurred, since the database accessing code is faded away from the working view of the programmer. Since transparent persistency removes the link between algorithms using the data from the database and the accessing code, programmers who are not intimately familiar with the system may conceive an incorrect understanding of the system.

Self-configuring components are a recently introduced novel approach to building links between different software layers. The idea is to recognize automatically resolvable dependencies within the codebase, and to implement corresponding resolver components. This helps in maintenance, since instead of the programmer manually following all the dependencies in the event of a functionality change, the automated component does the task. Examples have been implemented in areas of building consistent user interfaces (Pohjalainen, 2010) and database queries (Pohjalainen and Taina, 2008).

Currently, it is not very well known how these architectural decision affect programmer performance. To gain understanding on what works in practice, empirical validation is needed. To better understand the effect of transparent persistency's consequence for maintenance, we have formulated the following research questions regarding development speed and quality:

An initial hypothesis for introducing configuring component that is based on meta-programming on byte-code level would suggest that at least initially it would be harder and slower to produce correctly working code. Probably after a run-in period, the programmers involved would learn to understand the workings and responsibilities of the meta-programming component. This leads us to formulate a null hypothesis:

RQ_1: Do self-configuring components make it faster to perform maintenance tasks?

RQ_2: Do self-configuring components produce better quality in maintenance tasks?

RQ_3: Do self-configuring components make programmers more productive?

To gain initial insight in these questions, we conducted a randomized, controlled experiment on using Java persistency interface for database queries. In this experiment, the attending students were randomly divided into two groups, the transparent persistency group and the control group. Attendees were given slightly modified versions of the same software and were asked to perform given maintenance tasks upon the software. We tracked the time for performing the asked modifications. Each submission was afterwards graded for correctness. Productivity is defined as a ratio of the number of correct submissions per hour used on the tasks.

The rest of the paper is organized as follows. Section 2 gives a short introduction to self-configuring software components, object-to-relational mappings and implementation techniques used in the presented experiment. Section 3 defines the used research methodology. Section 4 shows results obtained in the experiment. Section 5 discusses these findings. Section 6 concludes the paper and outlines future work.

2 PRELIMINARIES

In this section we first review the object-to-relational approach to using databases in object-oriented programs. We then introduce self-configuring queries in the context of the example used in the experiment. Object-to-relational mapping components are commonly used to allow developers concentrate to code within one paradigm. The idea is that special mappings are used to translate an object's persistent fields into corresponding relational database concepts.

These mappings provide a placeholder for defining handling rules for issues rising from the mismatch between object and relational paradigms. Examples of these problems are e.g. the problem of handling inheritance, navigability of one-way object links versus bi-directionality of relational links, and representation of object collections, to name a few. The mapping also provides a place for defining the fetching rules for certain object classes. This is often a major source of performance problems in applications using the object-to-relational approach for persistency.

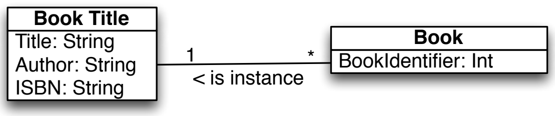

Figure 1: One-to-many mapping in the experiment application.

The fundamental problem in using default rules in database fetches is that most often the set of data objects being fetched from the database is context sensitive. For example, let us consider a fragment of our experiment's class diagram that is presented in Figure 1. The model shows a one-to-many connection between a book title and a number of books. The idea is to show that for each book title, its author name and ISBN number are stored only once; and for each book copy, there's a distinct object in the system.

The use case of listing all book titles in a given database is implemented as Algorithm 1. This algorithm fetches only objects of the *BookTitle* class.

Algorithm 1: List all book titles.

```
procedure PRINTTITLES(List allTitles)
    for all title in allTitles do
        print title.Author
        print ":  "
        println title.Title
    end for
end procedure
```

Another use case for this system is to count all copies of a certain title. This is implemented as Algorithm 2. This version needs to fetch not only objects of the BookTitle class, but also to consult the Book class as well.

Algorithm 2: List all book copies.

```
procedure PRINTBOOKS(List allTitles)
    for all title in allTitles do
        print title.Author
        print ":  "
        println title.Title
        $cnt \leftarrow 0$
        for all book in title.Books do
            println book.BookIdentifier
            $cnt \leftarrow cnt + 1$
        end for
        println cnt + " copies."
    end for
end procedure
```

Given these two use cases, using a default fetching rule for the BookTitle class is always somehow wrong: if the default rule is to eagerly fetch Books with its BookTitle, then the execution of Algorithm 1 is unnecessarily slow, since the algorithm does not need all that information. However, if the default rule is not to fetch associated books with their titles, then Algorithm 2 cannot be executed unless the default rule is overridden or some other internal magic is performed.

A common approach in this case is to use the Proxy design pattern (Gamma et al., 1995) to guard against cases where the algorithm is requiring some object data that is not fetched from the database. The proxy represents the object's external interface, and upon usage of the object data, it generates the necessary database queries needed to lazily retrieve that data from the database.

This is a way to implement transparent persistency: all algorithms can be written in an object-oriented language, and it is the responsibility of the object-to-relational mapping component to provide the implementation for relational database queries.

The downside of this approach is that it is easy to implement inefficient algorithms, as the proxying approach queries the database only at when a certain object is first accessed. This creates the *n+1 queries problem*, which happens when traversing collections of objects. The first query to the database fetches the root object. For each object link in the root object that is being accessed by the traversal algorithm, a new query is generated. Due to $n+1$ round-trip queries to the database, performance usually severely degrades.

For this reason, the object-to-relational mapping frameworks provide means to explicitly specify a query for fetching an initial data set. However, this approach is arguably an inelegant solution, since it breaks the promise of transparent persistency and generates dependencies between the explicitly specified query sites and actual data usage sites.

One solution to provide more precise control over the fetched data set is to use self-configuring database queries (Pohjalainen and Taina, 2008). In this approach, the software contains a query-resolved component, which analyzes the code that accesses the database and estimates the right database query to use. It is claimed that the component improves software maintainability, since the database usage is made more explicit, yet the dependencies between the data accessing algorithms and data fetching code are automatically resolved.

To the application programmer, the use of self-configuring database queries is shown by the need to parametrize database access with the algorithm that the queried data is being subjected upon. In traditional transparent persistency, the programmer fetches an initial object, e.g. a certain *Title* from the database and then executes the handling code, such as the one shown in Algorithm 2. With self-configuring queries, instead of providing an initial guess for the needed data, the programmer gives the name of the algorithm (*PrintBooks* in this case) to the self-configuring component. The self-configuring component analyzes the behavior of the algorithm and provides an approximation of the required data for running the algorithm with an optimal number of database queries.

In our implementation, the self-configuring query component analyzes the byte code of the method in the containing Java class. The configurator builds a control-flow graph of method execution, and includes database joins to the generated query whenever a class with object-to-relational mapping is encountered in the control flow. The control flow of Algorithm 2 is shown in Figure 2. For this control flow, the self-configurator deduces that for all BookTitles the attributes Author and Title and for all Books the attribute BookIdentifier will be used during algorithm execution.

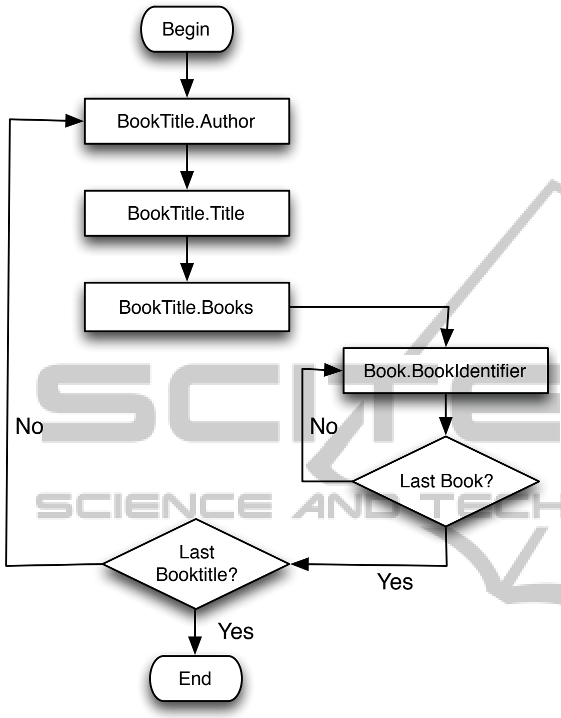

Figure 2: Control flow of persisted entities in Algorithm 2.

From the control flow graph in Figure 2, the self-configurator deduces that a query set of {book titles, booktitle.books} will be a sufficient query.

This approach is believed to give benefits in two ways: improved maintainability and improved modularity. Maintainability is improved, because the database query site does not explicitly define what data to fetch from the database, but instead the self-configuring query determines that. For this reason, the database accessing algorithms can be changed more easily, since dependence on the database query site is automatically updated to reflect the updated algorithm.

Modularity is improved because the one self-configuring component can handle a number of different usage sites. This is an improvement of the traditional n-tiered software architecture, where the database access layer needs to contain functionality specific to business logic. In the context of this experiment, the traditional way would be to implement one database query that fetches all book titles for executing Algorithm 1 and another query that joins all the book titles with the books for running Algorithm 2.

3 METHODOLOGY

To gain understanding on how this approach works in practice, we recruited a sample of 16 students to attend a randomized, controlled experiment. About half of the students were freshmen (*n*=10), and the rest were in their final bachelor year or master's students (*n*=6). Before attending the test, a two-hour lecture on concepts of relational databases, object-oriented programming and object-to-relational mapping tools was given to each participant.

We built two versions of a simple library book-keeping application. The application consists of five classes that are programmed to manage book titles, books, lenders and book loans in a database. Each attendee is randomly assigned to one of the versions, in which he stays during the whole experiment. The baseline application contains a functionally working version of the software with the same set of functionality implemented in both of them. For example, use cases for listing all book titles, as implemented in Algorithm 1 and for listing all books, as implemented in Algorithm 2 are contained in the package given to each test subject.

The first version (group 'transparent persistency, TP') of the sample application was written in the style of transparent persistency: the complexity of handling database queries is hidden behind the persistency framework's internals, which in this case was to use bytecode-instrumented proxying implementation of database queries. In the second version (group 'self-configred, SC'), the application contained method calls to self-configuring instructions to automatically prefetch the correct objects from the database, and to rewrite the corresponding fetching query.

In the variant given to the self-configured group, the database access is not fully obscured away. The database access component is parametrized with the actual algorithm that is going to be executed. The component analyzes the algorithm and produces the database query to be used at the subsequent database access time. This construct helps to remove dependencies between database access code and the actual algorithms that are being executed. A change in the algorithm is automatically reflected by the self-configuring component, thus removing the need to manually update the database query.

The initial application consists of a functionally consistent set of program code. The test subjects are asked to perform modifications to the sample application; depending on the skill and speed of the attendee, up to 7 functional tasks can be performed during the trial. Each task was time-boxed, meaning that the attendee is asked to return his version even if it is incomplete when the time is up. If an attendee thinks that he/she cannot be performing any more tasks, he is free to leave at any given point. Attendees were awarded with credit units or extra points to a course test regardless of how many tasks they opted to complete during the test. All test subjects were physically present in the classroom during the experiment. The submission mechanism was to send the source code for each application version via e-mail after each completed task.

To moderate the application complexity, the sample application was written as a command-loop tool. In the following code listings we will illustrate the differences between the two groups' variants using the following notation: line prefix "TP" notifies that this line is present only in the transparent persistency variant. "SC" lines are present in the self-configured variant. "C" lines are common to both of the variants.

Using this notation, the command line loop for the use case of listing all book copies in the database is implemented as listed in Program 1.

Program 1: Two variants of the *list books* use case.

```
C: switch(cmd) {
C: [..]
C:   case "list books":
SC:    String alg="BookTitles.PrintBooks";
SC:    Resolver r = new Resolver(alg);
SC:    bookTitles.fetchFromDatabase(r);
TP:    bookTitles.fetchFromDatabase();
C:     bookTitles.printBooks();
C:     break;
C: }
```

As can be seen in the listing, the only difference between the two variants is that the self-configured version contains a stronger hint of the fact that this command is accessing the database. The *Resolver* component is parametrized with the name of the algorithm that is going to be executed subsequently. It is important to note that in the transparent variant, although the name of the method suggests that the method is going to access the database, there is no explicit link between the data-fetching code and the algorithm used to print out the books. Consequences of this omission are further explored later in the results section.

In the resolver version, the component analyzes the internal workings of the method that is given as an argument and produces an estimate of what data objects should be fetched. This gives the maintainer a stronger binding between the data-fetching code and the algorithm used to process the fetched data.

The fetchFromDatabase routine as listed in Program 2 also differs a bit in the two variants. The variant for the transparent persistency group creates a database query by using the object querying language defined in the Java persistency interface. This query fetches an initial set of book titles from the database. In this variant when Algorithm 2 execution arrives to a book copy that has not yet been fetched from the database, the automatically generated proxy instance generates a new query to get the missing information.

Program 2: Two variants of database fetching code.

```
SC: void fetchFromDatabase(Resolver r) {
TP: void fetchFromDatabase() {
TP:   Query q;
TP:   q = createQuery("from BookTitle");
SC:   Criteria q = r.resolve();
C:
C:    bookTitles.clear();
C:    List<BookTitle> list = q.list();
C:    for(BookTitle bt : list)
C:        bookTitles.add(bt);
C: }
```

In the self-configuring variant, Algorithm 2 is given as an argument to the resolver component. The resolver analyzes the code of the algorithm and initiates a database query to fetch the required data. If the resolver happens to underestimate the required set of objects, the normal proxy-based fallback is used as a safeguard. At this point, it is important to note that the self-configured version does not contain the matching between database and business logic, but instead it operates in the business logic domain.

The essential difference between the two variants is the degree of how explicit the database query is. In the transparent persistency variant, the database access code is minimized, with the design goal of reducing the mental load of the programmer, as he should not be worrying about how to access a database. This can also be misleading, since there is no hint of the mechanism of how the persistency framework should be used. In the control group variant, the database accessing is more explicit, since the algorithm calling site generates the database query component before actually executing the query.

Armed with the initial example cases of Algorithms 1 and 2 given to the test subjects, they are asked to implement a number of maintenance tasks to the software. All of the tasks follow the same pattern: fetch some data from the database and then print out information. Task 0, which was to change the source code character set to UTF-8, was not graded. It was used to give the attendees an opportunity to gain understanding of the sample application and to practice the submission procedure. The tasks are summarized below:

Identifier	Task description
Task 0	Fix source code charset (not graded, used to practice the submission procedure)
Task 1	List all book loaners in the system
Task 2	List all book loans in the system
Task 3	Fetch specific book title by its ISBN
Task 4	Modify task 3 result to print out all books associated with the given title
Task 5	Fetch specific lender by his social security number
Task 6	Modify task 5 result to print out all books borrowed by this lender
Task 7	Modify the list of book titles to print out all lent books associated with that title

The tasks list contains maintenance tasks that can be classified as adaptive maintenance in IEEE Std 14764-2006 terms (IEEE14764, 2006). Each of the tasks delivers an added functionality that is requested to make the sample software able to be a better fit for its task of handling a database of library books. Each of the tasks are implementable without inducing new requirements or other needs for architectural changes.

4 RESULTS

We graded each submission based on correctness on a dichotomous scale: a correct submission to a given task is programmed to connect to the database, fetch certain data, and print out corresponding results. Minor stylistic problems, such as output formatting did not affect the grading.

In the freshmen group, the assignments were generally considered to be hard. This is understandable, since outside the persistency code, the implementation followed object-oriented principles in both versions. However, even in this group, some test subjects succeeded in getting some of the tasks submitted correctly.

The second problem to occur frequently among all test subjects was the failure to fetch the database correctly. Often the modified application seemed to work perfectly, but an underlying problem ("a bug") was introduced by accidentally re-using the applications internal data instead of targeting the query to the database. This problem was most often seen in the transparent persistency group. We believe this to be the consequence of the interface between business logic and database access code. Since the business logic explicitly defines only the initial object of the processed object graph, thus making all other database access translucent, the students often failed to correctly handle the cached versions of the objects.

Other programming errors included various incorrect structures, such as null pointer exceptions, cache-handling errors etc. Each of these were unique to the given submission.

For each test subject, we noted the state of his studies (freshman/advanced), randomization group (transparent persistency / self-configured), time spent on each submission (rounded to next 5 minutes) and the correctness of the submission. Overall, these results are shown in Figure 3. In the figure, the symbol ✓ is used to show a correct submission and the symbol ✗ shows an incorrect submission. Each attempt was also followed by a submission. An empty box means that the test subject did not attempt the task in question.

To study the productivity between the two approaches, we measured the time for producing a submission to all given tasks. However, we are analysing this information in the advanced group only. The test subjects in the freshmen group are omitted due to their submissions having such low success rate; there is only one pair of submissions where two test subjects belonging to different randomization groups have been able to produce a correct submission to the corresponding task (test subjects #7 and #8 for task 3).

In the advanced students' group, there was one case, where the test subject had to leave early due to a medical condition before he was able to send a single submission. He was randomized to the self-configuring. Due to the nature of that situation, his results are not included in the analysis. Outside this case, no test subject left the experiment prematurely.

4.1 RQ1

Given this information, we can return to our research questions. RQ1 proposes that self-configuring queries make it faster to perform maintenance tasks. According to Figure 3, the test subjects in the transparent persistency group returned a total number of 27 submissions, be they correct or incorrect. The test subjects in the self-configured group returned 33 submissions. There are an equal number of subjects in each of the randomization groups, and the maximum allowed time was the same for everybody.

The self-configuring turned in more submissions, but the difference between the two groups is small. From this viewpoint, we cannot support the idea of transparent persistency making the completion of these maintenance tasks faster, but the self-configured group does not get a decisive victory, either.

Subject #	Study state	Randomization	Task 1	Task 2	Task 3	Task 4	Task 5	Task 6	Task 7
#1	freshman	SC	✗	✗					
#2	freshman	TP	✗	✗					
#3	freshman	TP	✗						
#4	freshman	SC	✓	✗					
#5	freshman	SC	✗	✓	✗				
#6	freshman	TP	✗	✗	✗				
#7	freshman	SC	✓	✗	✓				
#8	freshman	TP	✗	✗	✓				
#9	freshman	SC	✗	✗	✗				
#10	freshman	TP	✗	✗	✗				
#11	advanced	SC	✓ 35	✓ 45	✓ 35	✓ 10	✓ 5	✓ 10	✓ 5
#12	advanced	SC	✓ 30	✓ 55	✗ 15	✓ 30	✓ 15	✗ 30	✓ 15
#13	advanced	TP	✓ 45	✗ 50	✓ 20	✓ 10	✓ 20	✗ 15	
#14	advanced	SC	✓ 30	✓ 45	✓ 10	✓ 10	✓ 10	✓ 10	✓ 10
#15	advanced	TP	✗ 55	✗ 35	✗ 20	✗ 15	✗ 20		
#16	advanced	TP	✗ 45	✗ 30	✗ 40				
#17	advanced	(*)							

Figure 3: Summary of results: ✓for a correct submission, ✗for incorrect; number below the result indicates minutes spent on the task.

4.2 RQ2

Figure 4 summarizes the frequency of correct submissions. For example, two attendees in the self-configured group got zero submissions correct and six attendees in the transparent persistency group got zero submission correct.

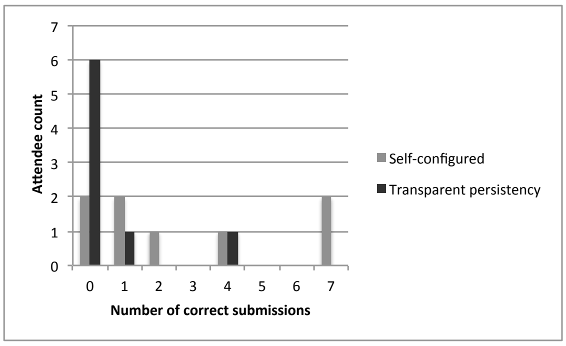

Figure 4: Count of correct submissions for both groups.

The grading distribution in Figure 4 leads us to formulate the null hypothesis for RQ2:

> H_0: Transparent persistency makes it easier to produce correct database handling code than self-configuring queries.

When the test data is normally distributed, a parametrized test should be used to determine differences between populations. Otherwise, a non-parametric test should be used (Robson, 2011, p. 450-452). Our data is based on human behavior, and could be believed to be normally distributed. However, the results look like a non-normal distribution. For this reason, we calculated both parametric and non-parametric statistical tests.

Both the one-tailed independent samples t-test and Mann-Whitney U test result a p-value of $\alpha < 0.05$, giving a suggestion that the null hypothesis needs to be rejected. Informally, the self-configured group was performing much better than the transparent persistency group: two attendees who got assigned the self-configuring, query rewriting-based base software got all seven graded tasks implemented correctly. On the other hand, six attendees using the traditional transparent persistency failed to produce a single correct submission.

4.3 RQ3

Finally, as a measure of productivity, we consider the time consumed in programming tasks in the non-freshmen group. The times for producing a *correct* submission (n=23) varied between 5 and 55 minutes.

There was no statistically significant difference in time consumed between the two groups.

As the number of correctly returned submission series is low, statistical analysis of these series would not serve a purpose. As a reference information, when ordering the correct by-task submissions as a speed contest, all test subjects in the self-configured group were producing submissions faster, on average.

We defined productivity as a number of correct submissions per time consumed. Thus, for an individual we calculate his productivity index as

$$\frac{\sum_{i=1}^{7} time_i}{\sum_{i=1}^{7} corr_i}$$

where the variable $time_i$ refers to the time for producing the answer for task i and $corr_i$ is 1 for a correct answer and 0 for an incorrect answer in task i. This formula slightly favors the productivity of incorrect answers, since the time needed to rework an incorrectly piece of software, as would happen in a real-world software development situation, is not included in this productivity index.

Applying this formula to the test subjects in the advanced group yields a productivity index of 105 minutes per correct task submission in the transparent persistency group. For the self-configured group, the index stands at 24 minutes.

For producing correctly working code, the self-configured group outperformed the transparent persistency group by a factor of four.

4.4 Threats to Validity

We have found a statistically significant effect between two alternative ways of querying a database in an object-oriented program. However, a number of threats to the validity of this finding do exist.

In general, a controlled experiment like this differentiates from industrial use in the scope of time that can be spent in understanding the associated technologies. In an industrial setting, programmers usually work with the same technology stack for months or years. In this study, some of the test subjects were exposed to the technologies for the first time. However, some of this concern can be argued to be mitigated by the randomization process; these technologies can be assumed to be equally unknown to both groups. A related problem is the small size of the software used in the experiment. It might be the case that the complexity of transparent persistency pays off only after a certain software size has been achieved.

Another concern is that using students in controlled experiments limits the generalizability of results to a industrial setting. However, previous studies, e.g. (Arisholm and Sjoberg, 2004) have found a high correspondence between (1) undergraduate and junior programmers; and (2) graduate and senior programmers in terms of understanding object-oriented programming, which can be interpreted to mean that at least half of the experiment population were up to industrial standards. This distribution of attendees is probably not fully unrealistic, since companies often use junior programmers to perform maintenance tasks (Gorla, 1991). Another interpretation of our work is that the finding in (Arisholm and Sjoberg, 2004) is probably valid: the advanced group in our study was able to grasp the functionality of the object-oriented system, while the freshmen group clearly had problems understanding it.

Our third concern is the the sample size, which is small. We conducted this study on 17 students, of whom 10 were in their first year, and 7 students were more advanced. Since one student in the advanced group was excluded from the experiment due to a medical condition, the total number of test subjects is 16. Due to the small sample size, individual performance shows a large role in our statistical analysis.

Our fourth concern regards the use of number of submissions as productivity measure. Most of the tasks are equidistant in substance in the sense that for most tasks, the task n is not dependent on whether he succeeds in $n+1$. For tasks $T4$ and $T6$ there is a dependency to the previous task. This dependency plays some role for subjects #12 and #13. Subject #12 failed in task T3, but was able to fix the problem in T4 submission. He also got task T5 correct but failed his task T6. The latter dependency is present for subject #13 as well.

5 DISCUSSION AND RELATED WORK

The use of transparent persistency for storing objects is not a new idea. A number of systems for various languages and programming environments have been described in the literature; (Atkinson et al., 1983; Atkinson and Morrison, 1995; Bauer and King, 2004), to name a few.

Similarly to the self-configured queries, researchers have used the program source as a source model for model transformations in various contexts. For example when building portable interpreters, the opcode definitions can be implemented in the interpreter implementation language. When the definition is compiled, the end result, in byte code or other low-level representation can be used as a building block when interpreting the opcode in question (Elliott et al., 2003; Yermolovich et al., 2008).

Using program analysis for extracting database queries has been proposed by various researchers. We used the component documented in previous work (Pohjalainen and Taina, 2008), using the same object-to-relational mapping framework, Hibernate (Bauer and King, 2004). In addition to this style, Ibrahim and Cook have proposed a dynamically optimizing query extractor (Ibrahim and Cook, 2006). Wiedermann et. al have implemented a version which combines static and dynamic techniques (Wiedermann et al., 2008). In the simple examples used in this experiment, a simple analyzer was able to deduce the required queries. However, when the business logic involves more complex decisions, the automated self-configuring component needs to be updated to be able to handle the more complex case. In a way, the situation resembles the full-employment theorem of compiler writers (Rice, 1953): since the business logic can be arbitrarily complex, the evolution for implementations of automated resolvers are restricted by laws of economics rather than some self-configurator being optimal to every case.

Although various systems for implementing transparent persistency have been available for decades, it seems that its actual usage patterns have not been studied very well. This implementation was initially employed in an industrial project for building a product-line for provisioning governmental mobile subscriptions (Pohjalainen, 2011), where the need for modular constructs was critical to development success and transparent persistency turned out to lack the support for efficient modularity. However, according to our best knowledge, comparisons of the effect of transparent persistency versus any alternative have not been empirically studied so far. Considering the fact that transparent persistency is a frequently employed technique in industrial software engineering, the lack of empirical studies is a surprise.

The use of orthogonal, transparent persistency can be seen as a specialized case of modularizing orthogonal features. To relate our findings, we can refer to literature on empirical studies of aspect-oriented programming (Kiczales et al., 1997). We can consider transparent persistency to be one of the cross-cutting aspects of the software.

The empirical response to productivity in aspect-oriented programming seems to be mixed. For example, in (Bartsch and Harrison, 2008), the researchers were unable to find any positive impact of using aspect orientation for building maintainable and easier-to-comprehend software.

Kulesza et al. studied the use of aspect-oriented programming for a maintenance task (Kulesza et al., 2006). They implemented two similar systems, the first one in an object-oriented fashion and the second one in an aspect-oriented fashion and compared both systems' internal metrics, such as size, coupling and cohesion, before and after a specified maintenance task. Their conclusion is that the aspect-oriented style was superior in terms of observed software characteristics. They did not measure the time spent on producing the different versions of the software, nor the time spent on implementing the maintenance tasks. Thus, productivity of the maintenance task cannot be assessed. Contrary to our study, the person doing the maintenance task was presumably an expert programmer and familiar with the system being maintained. In our setting we had a sample of junior programmers performing a series of maintenance tasks on a previously unknown system.

Endrikat and Hanenberg measured the time for building a system and then performing a number of adaptive maintenance tasks on a number of test subject (Endrikat and Hanenberg, 2011). For many of the tasks, the object-oriented way in the control group was faster, but for combined build+maintenance time they suggest that aspect orientation can be beneficial.

Another, indirect finding on this study was the poor programming performance in the freshmen group: 70% of test subjects in this group were having serious problems with the experiment. Although they had completed the first programming courses offered by the university and were given a fully functioning system, they failed to perform even simplest modifications to it. For this reason the university has already adjusted our first programming courses to use a new studying method based on extreme apprenticeship (Vihavainen et al., 2011; Vihavainen and Luukkainen, 2013). In future work, it would be interesting to compare how these adjustments to teaching methodologies have affected freshmen programming skills in maintenance tasks.

6 CONCLUSIONS

We performed a randomized, controlled experiment to assess the usefulness of transparent persistency. In the experiment we built two versions of a small sample application and asked a number of test subjects to perform a number of maintenance tasks upon their software variant.

We measured the time used to do these maintenance tasks and related them to successful submission rates. As a result, we concluded that the self-configured group was performing much better in terms of producing correctly behaving code.

In terms of correctly returned submissions, the self-configured group outperformed the transparent persistency group by a factor of three. This result is a statistically significant difference (p value $\alpha < 0.05$) between the two populations.

In productivity, the self-configured group outperformed the transparent persistency group by a factor of four: on average, the self-configured group was able to produce a correct submission in under half an hour. In the transparent persistency, on average it took almost two hours to do the same. However, due to the low sample size, we did not perform any statistical analysis on productivity.

This result casts a shadow of disbelief on the concept of transparent persistency: if the application is going to access a database, it probably is not a good idea to try to disguise its functionality to be something else. This disguise seemed to be the main source of program miscomprehension within the test subjects in the transparent persistency group. However, the small sample application limits the generalizability of the result. Experiments with larger software and larger populations are needed to understand the usefulness of transparent persistency for software development. On the other hand, the result gives an initial empirical validation of the usefulness of using self-configuring software components to reduce the maintenance effort and to improve architectural modularity.

REFERENCES

Arisholm, E. and Sjoberg, D. I. K. (2004). 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.

Atkinson, M. and Morrison, R. (1995). Orthogonally persistent object systems. *The VLDB Journal*, 4(3):319–402.

Atkinson, M. P., Bailey, P. J., Chisholm, K., Cockshott, W. P., and Morrison, R. (1983). An approach to persistent programming. *Comput. J.*, 26(4):360–365.

Bartsch, M. and Harrison, R. (2008). An exploratory study of the effect of aspect-oriented programming on maintainability. *Software Quality Control*, 16(1):23–44.

Bauer, C. and King, G. (2004). *Hibernate in Action (In Action series)*. Manning Publications Co., Greenwich, CT, USA.

DeMichiel, L. and Keith, M. (2007). JSR 220: Enterprise JavaBeans 3.0. Technical report, Sun Microsystems.

Elliott, C., Finne, S., and de Moor, O. (2003). Compiling embedded languages. *Journal of Functional Programming*, 13(2).

Endrikat, S. and Hanenberg, S. (2011). Is aspect-oriented programming a rewarding investment into future code changes? A socio-technical study on development and maintenance time. In *Proceedings of IEEE 19th International Conference on Program Comprehension*, ICPC'11, pages 51 –60.

Fisher, M., Ellis, J., and Bruce, J. C. (2003). *JDBC API Tutorial and Reference*. Pearson Education, 3 edition.

Gamma, E., Helm, R., Johnson, R. E., and Vlissides, J. (1995). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA.

Gorla, N. (1991). Techniques for application software maintenance. *Inf. Softw. Technol.*, 33(1):65–73.

Ibrahim, A. and Cook, W. R. (2006). Automatic prefetching by traversal profiling in object persistence architectures. In *Proceedings of the 20th European conference on Object-Oriented Programming*, ECOOP'06, pages 50–73, Berlin, Heidelberg. Springer-Verlag.

IEEE14764 (2006). International Standard - ISO/IEC 14764 IEEE Std 14764-2006 software engineering #2013; software life cycle processes #2013; maintenance. *ISO/IEC 14764:2006 (E) IEEE Std 14764-2006 Revision of IEEE Std 1219-1998)*, pages 1–46.

Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C. V., Loingtier, J.-M., and Irwin, J. (1997). Aspect-oriented programming. In *Proceedings of the 11th European Conference on Object-Oriented Programming*, ECOOP'97, pages 220–242.

Kulesza, U., Sant'Anna, C., Garcia, A., Coelho, R., von Staa, A., and Lucena, C. (2006). Quantifying the effects of aspect-oriented programming: A maintenance study. In *Proceedings of 22nd IEEE International Conference on Software Maintenance*, ICSM '06, pages 223 –233.

Pohjalainen, P. (2010). Self-configuring user interface components. In *Proceedings of the 1st International Workshop on Semantic Models for Adaptive Interactive Systems*, SEMAIS '10, pages 33–37, New York, NY, USA. ACM.

Pohjalainen, P. (2011). Bottom-up modeling for a software product line: An experience report on agile modeling of governmental mobile networks. In *Proceedings of 15th International Software Product Line Conference*, SPLC'11, pages 323–332.

Pohjalainen, P. and Taina, J. (2008). Self-configuring object-to-relational mapping queries. In *Proceedings of the 6th international symposium on Principles and practice of programming in Java*, PPPJ '08, pages 53–59, New York, NY, USA. ACM.

Rice, H. G. (1953). Classes of recursively enumerable sets and their decision problems. *Trans. Amer. Math. Soc.*, 74:358–366.

Robson, C. (2011). *Real World Research: A Resource for Users of Social Research Methods in Applied Settings*. John Wiley & Sons.

Vihavainen, A. and Luukkainen, M. (2013). Results from a three-year transition to the extreme apprenticeship method. In *To appear in Proceedings of the The 13th*

IEEE International Conference on Advanced Learning Technologies, ICALT '13.

Vihavainen, A., Paksula, M., and Luukkainen, M. (2011). Extreme apprenticeship method in teaching programming for beginners. In *Proceedings of the 42nd ACM technical symposium on Computer science education*, SIGCSE '11, pages 93–98, New York, NY, USA. ACM.

Wiedermann, B., Ibrahim, A., and Cook, W. R. (2008). Interprocedural query extraction for transparent persistence. In *Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, OOPSLA '08, pages 19–36, New York, NY, USA. ACM.

Yermolovich, A., Gal, A., and Franz, M. (2008). Portable execution of legacy binaries on the Java virtual machine. In *Proceedings of the 6th international symposium on Principles and practice of programming in Java*, PPPJ '08, pages 63–72, New York, NY, USA. ACM.