A comparative study on the re-documentation of existing software: Code annotations vs. drawing editors

Marco Torchiano
Politecnico di Torino
Italy
marco.torchiano@polito.it

Filippo Ricca and Paolo Tonella
ITC-irst,Trento,
Italy
{ricca, tonella}@itc.it

Abstract

During software evolution, programmers spend a lot of time and effort in the comprehension of the internal code structure. Such an activity is often required because the available documentation is not aligned with the implementation, if not missing at all. In order to avoid wasting the time devoted to this activity, programmers can record the knowledge they have gained in the form of multiple, structural views that address the specific aspects of the system that they have considered.

Re-documentation of existing software through design views can be achieved either using a drawing editor or annotating the source code. In the first case, diagrams are produced interactively, starting from the reverse engineered information. In the second case, diagrams are produced by an annotation processing tool.

Most of current reverse engineering tools fall into the first case but they have serious limitations in the information they can recover automatically and they eventually require human intervention.

The aim of the empirical work reported in this paper is the comparison of these two approaches, in order to understand which is easier to use and which the current limitations of both of them are.

Keywords: Reverse Engineering, Usability, UML, Code Annotations.

1. Introduction

Program understanding has been reported as one of the most difficult and time-consuming activities during software evolution. Often the programmers who evolve a given program are different from those who developed it, so that it is difficult for them to get a picture of the internal organization and to locate the code portions affected by the requested change. Similar difficulties have been observed even when the original developers are involved in the evolution task after some time has passed since the initial development.

When programmers face program understanding, they demand for information about the internal structure of the code. This kind of information helps them locate the change and evaluate its impact. However, it often happens that no documentation is available to support their activity or that the available documentation is not aligned with the implementation. Another typical scenario is one where the available documentation does not address the comprehension needs involved in the maintenance task at hand. Alternative views and diagrams, with a different focus and with different information shown would be necessary.

In the scenarios described above, programmers usually acquire knowledge about the given system by means of code reading and by executing the program with appropriate input sequences. The output of such a process is an increased knowledge about the system. In order to avoid that such a knowledge gets lost over time, thus maximizing the return on investment, it must be stored persistently in some format, such as a set of design diagrams. To this aim, programmers have (at least) two main options:

1. defining a set of diagrams that capture the acquired knowledge, through a drawing editor;
2. annotating the source code, so that diagrams can be eventually extracted from the code.

When going for the first option, programmers use some design tool to draw the diagrams. Properties and relationships that are relevant for the current evolution task are displayed on such diagrams. This is obtained by interacting with the design tool through a graphical user interface. Currently most drawing tools are also able to extract the basic information needed to produce the diagrams from the code; however, there are serious limitations to what can be extracted automatically, so eventually the human intervention is mandatory.

A similar result can be obtained declaratively, by specifying the relevant properties and relationships directly in the source code, in the form of annotations. Annotation processing tools can then generate the diagrams from the annotated code.

Clearly, the drawing editor approach has the advantage

that the interaction with the tool is straightforward and that there is immediate graphical feedback. On the other hand, the code annotation approach is focused on the declaration of *what* information should be displayed, eliminating the problem of deciding *how* to actually display it. An analogy can be drawn with the WYSIWYG document editors (e.g. Microsoft Word) and the document composition languages (e.g. LaTeX).

In this paper, we report the results obtained in the execution of an empirical study in which a re-documentation task was executed twice: once with a drawing editor tool and another time using code annotations. We collected feedback from the user in the form of a questionnaire, including both closed questions (ranging on an ordinal scale) and open questions with the users' comments on specific aspects of the task. The drawing editor approach resulted to be the most preferred and usable one, with no penalty on the quality of the resulting diagrams. However, some threats to validity that we have identified hint for the execution of further similar studies in a different setting. For example, the age and experience of the users (undergraduate students) might have affected the results.

The paper is organized as follows: after describing the two approaches being compared (Section 2), we give details on the experimental design (Section 3). Then we report and discuss the results (Section 4). Conclusions are drawn at the end of the paper.

2. Recovery of design diagrams

Design Recovery is in general the process of analyzing a software system to recreate meaningful design abstractions. In the case of an Object-Oriented software system, different useful views can be recovered by means of Reverse Engineering tools. Among them, the class diagram is the most important and most widely used. The **UML class diagram [7]** shows the classes that compose a system, the inter-class relationships and the properties of each class. Available reverse engineering tools are able to recover the class diagram from the target code but they present some heavy limitations. To have a more accurate and useful result the reverse engineering process cannot be completely automatic: the human intervention is necessary.

2 .1 Limitations of Reverse Engineering Tools

For a medium sized system (in the order of 20k Lines Of Code, LOC) it is quite common to have 50-100 classes. A design diagram reverse engineered from the code that shows them, even without displaying any property, is completely unreadable for human beings, whose cognitive abilities permit grasping information related to about 7 objects at most [6]. There is about 1 order of magnitude that separates automatically recovered information from actually usable diagrams. Two mechanisms can be used to simplify the reverse engineered diagrams: filtering and multiple-views. By filtering, users specify which information is irrelevant and can be skipped. When defining multiple views for a given system, programmers decide which elements (classes, fields, methods, etc.) belong to which view. While a class may be meaningful to understand a given portion of a system, it may be useless to include it inside other views. Thus, each class contributes to one or more views, each of which gives a partial but meaningful representation of the system's organization.

The class properties that are shown in the compartments associated with each class in the class diagram include the class methods and attributes. While some of these properties convey important information about the class state and behavior, others may be completely irrelevant (e.g., setter and getter methods, transient attributes for temporary storage). A filtering mechanism can be used to restrict the displayed information to what gives a relevant contribution to program understanding. Moreover, the information that can be shown for each class property includes the attribute visibility (public, protected, private, etc.) and type, and the method visibility and signature. Not all these data are helpful to program comprehension and suppression of some of them may result in a clearer diagram.

Among the class relationships that are shown in a class diagram, the most important ones are:

- **Inheritance/realization:** a class extends/implements a class/interface.
- **Aggregation/composition:** a class is part of another class.
- **Association:** a class holds a stable reference toward another class.
- **Dependency:** a change in a class might impact another class.

Dependency subsumes all the other relationships, and association subsumes aggregation. It is also possible to consider a special case of aggregation, called *composition*, to which stronger constraints (such as same lifetime) apply.

When recovered from the source code, inheritance and realization are easily identified at the syntactic level. Once implemented, aggregation, association and composition are almost indistinguishable, thus they are usually unified into the association, which subsumes all of them, by available reverse engineering tools. An association (inclusive of aggregation and composition) is recovered from the code when a class attribute references an object of another class, being a pointer (Java reference) or a container, such as a list, a hash table, or an array. A dependency (excluding inheritance and association) can

be detected when a temporary reference (e.g., local variable, method parameter) is used to access another class.

Given the semantic ambiguity in the definition of aggregation (and composition) with respect to association, additional input may be needed to discriminate among them. Moreover, a class diagram with even as few as 5-10 elements becomes quickly unreadable if all relationships of all kinds are shown. Thus, filtering should be applied to relationships as well.

A limitation in the reverse engineering of relationships from the code is multiplicity. Statically, it is in general undecidable the number of objects involved in a given relationship between two classes. Another limitation is the impossibility to recover the relation name and the source role, while the target role can be approximated by the name of the reference variable used to implement the given relationship.

A problem with associations and dependencies is that the target class of such relationships is not always available in the source code. For example, if a variable implementing an association or a dependency is declared of interface type, the actually referenced class cannot be determined from the declaration. Similarly, when weakly typed containers are used, the actual class of the contained objects is not known. Although sophisticated algorithms can be used to approximate the concrete type of the referenced/contained objects [13], a precise solution may be not computable in the presence of incomplete systems.

Tagged values, constraints, properties and comment notes that may enrich a design diagram with important semantic information cannot be recovered automatically from the code and must be provided externally.

As we have seen, to overcome the limitations of reverse engineering tools described above and to refine the default display options into more useful ones the human intervention is essential. To improve the result of the class diagrams produced there are two chances: working at the level of the code and adding *code annotations* that "help" Reverse Engineering tools produce better results or modifying directly the diagrams produced by the tools, by means of a drawing-editor.

2 .2 The code annotation approach

In this approach the user has to add manually annotations to the code. The annotations drive the Reverse engineering tool in the production of more useful and accurate diagrams.

The annotations that we have used in this experiment are an extension of those proposed by Spinellis [11]. They are described with reference to the Java programming language and respect the **Javadoc syntax** of the annotations (i.e. an '@' precedes the name of the annotation).

The reverse engineering tool **UMLGraph [10, 11]** written in Java and based on the Doclets [12], has been used as the starting point to build our tool **XUG** (eXtended Uml Graph). **XUG** allows visualizing the class diagram extracted from an annotated code. The result is a dot [3] file that can be automatically processed to create, for example, Postscript, Gif and Jpeg images.

We have extended UMLGraph with new annotations and we have introduced a new mechanism to define multiple views. New annotations make XUG more useful and powerful, while the new implementation of the views simplifies their use with respect to the original version.

```
/**
 * @opt attributes
 * @opt types
 * @hidden
 */
class UMLOptions {}
/** @hidden */
class Name {}
/**
 * @has 1..* Member * Student
 * @composed 1..* Has 1..* Department
 */
class School {
        Name name;
        String address;
        /** @show */
        void addStudent() {}
        void removeStudent() {}
        /** @show */
        void addDepartment() {}
        void removeDepartment() {}
}
/**
 * @assoc 1..* - 1..* Course
 */
class Department {
        Name name;
}
class Person {
      Name name;
}
/**
 * @assoc * Attends * Course
 * @note "this is a Person"
 */
class Student extends Person {
        int studentID;
}
class Course {
        Name name;
        int courseID;
}
```

Figure 1: Example of annotated code.

In **XUG**, programmers can decide to change the default show/hide setting for user-defined or library classes/interfaces. By default, all user defined classes/interfaces are shown. On the contrary, library classes/interfaces are hidden. Class attributes and methods are hidden and the shown relationships are by default only inheritance and realization. The user can act on such

default settings in two ways: at the global level or at the local level. It is possible to change the show/hide setting for all classes/interfaces of the project, i.e., at the global level, by adding global annotations. Following the convention described by Spinellis [11], these are specified in a special class named *UMLOptions*, as in Figure 1 (where all fields and field types are shown).

It is also possible to change the setting of an individual class at the local level. In this case the annotation temporarily overrides the global setting for the class being processed.

Annotations can also be added directly to fields and methods. *@hidden* hides a field or method, *@show* shows a field or method, while *@stereotype* and *@tagvalue* respectively add a stereotype and a tagvalue.

Relationships between classes/interfaces are added to the class diagram by putting a relationship annotation before the class that participates in the relation. All relationship annotations, except *@extends*, need four arguments: the source adornments (role, multiplicity and visibility), the relationship name, the target adornments and the target class.

For example, the class diagram extracted from the annotated code in Figure 1 is shown in Figure 2.

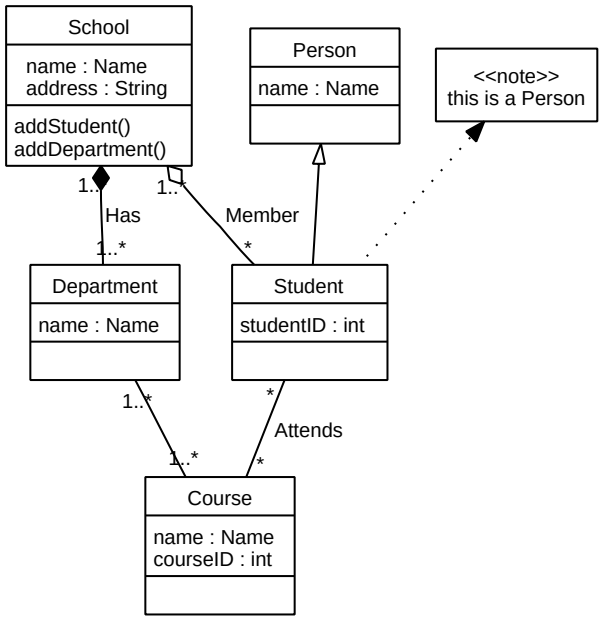

Figure 2: Example of diagram produced by XUG.

Annotations are not used only to add meaningful information to diagrams and to filter them, but also to produce multiple-views, i.e. different views of the same code. To create a new view, it is sufficient to define a new class with a syntax similar to that of *UMLOptions*.

For example, if we want to add a view where all members of the classes are displayed we need to add a new class *view_1* as follows:

```
/**
 * @opt all
 * @hidden
 */
class view_1 {}
```

The setting of a single element (class, field, method, etc.) in a given view can be also changed, by adding an argument to the annotation that specifies the target view. For example, if the annotation *@hidden* with argument *view_1* is added to the class *Student* of Figure 1, the result related to *view_1* will be a class diagram with only four classes: *School*, *Department*, *Course* and *Person*.

2 .3 The drawing editor approach

In the drawing editor approach, the user modifies manually, through a graphical user interface, the diagrams produced by a Reverse Engineering engine. Some tools, such as Visio, provide just drawing aids. Others, such as Rational Rose, Together and Omondo, offer also facilities for round-trip software engineering (i.e., changes in the implementation are propagated to the design and vice versa [2]). However, all drawing editors require the user to place and manipulate figures on a graphical canvas and to change their properties using a GUI.

In our comparative study, we have chosen the tool Omondo [8] for several reasons. Omondo is a visual modeling tool providing forward and reverse engineering facilities, natively integrated with Eclipse, and it supports round-trip design/code development. It is available in a free version that is installed in the labs and is accessible to the students.

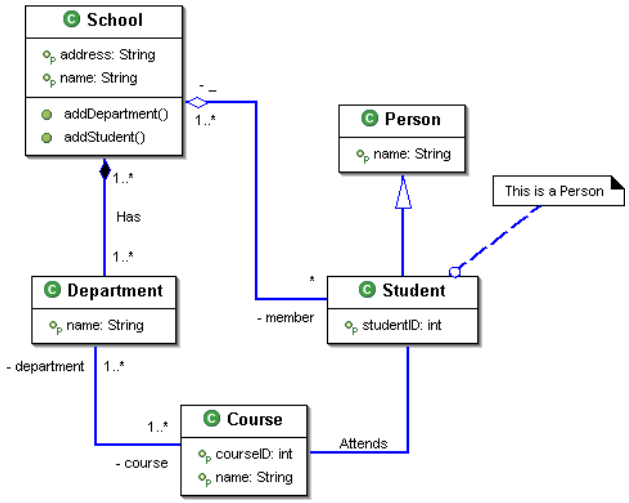

Figure 3: Example of class diagram produced by Omondo.

In the following the most important features provided by the free version of Omondo are listed:

1. **Reverse engineering:** Omondo can generate class and package diagrams from the source code. The result can be manipulated graphically, via context sensitive menus. Filtering and multiple views are supported for the reverse engineered diagrams.
2. **Forward engineering:** code (classes, interfaces, methods, attributes, documentation) is generated when designing/drawing a UML class diagram.
3. **Round-trip engineering:** code and design diagrams are kept aligned automatically. Updates on one of the two are immediately propagated to the other one.
4. **A wide range of UML diagrams are supported** (Activity Diagram, Class Diagram, Collaboration Diagram, Component Diagram, State Diagram, Use Case Diagram, etc).

3. Experimental design

We adopted a simple design with two experimental groups, a single object and two treatments applied in sequence in inverse order for the two groups. The same measurements were taken for both treatments. The subjects performed the same re-documentation task with both techniques in two different orders.

3 .1 Goal of the study

We characterize the goal of our study according to the GQM template [1]: *the goal is to analyze code annotations and drawing editors in order to compare them with respect to ease of use and expressive power from the point of view of the software developer in the context of a master degree course*.

With this goal in mind we formulated three main questions:

Q1: Is it easier to annotate the code or to edit the diagrams?

Q2: How do recovered diagrams differ in their quality?

Q3: In which case do the documentation guidelines provide more help?

For Q1 we identified four main features to consider:

- the difficulty of splitting the model into several diagrams and showing the same class on more than one diagram;
- the difficulty of selecting the attributes and methods to be shown for a class in each diagram;
- the difficulty of specifying the relationships among the classes, in particular those not recovered automatically by the tools;
- the effort spent to generate the diagrams.

As far as Q2 is concerned we focused on two aspects:

- the quality of the default diagrams, extracted automatically by the tools from the code without any user intervention;
- the quality of the final diagrams, resulting from adding annotations or graphical editing made by the users.

For Q3 we decided to observe both the perceived usefulness of the guidelines we provided and the process conformance.

On the basis of the above questions we can outline three high-level null hypotheses that we will try to reject:

Ha_0: code annotations and drawing editors are equally easy to use in software re-documentation.

Hb_0: the quality of the diagrams obtained using code annotations and drawing editors is the same.

Hc_0: the guidelines for the documentation are equally useful.

3 .2 Procedure

First of all let's consider the overall context of the course; the students have three main assignments:

1. They are given a software system in executable-only form and documentation, they have to install the system and write black-box (acceptance) tests.
2. They are provided with the source code, they perform code and design reviews and write white-box (unit and integration) testing, while reporting and correcting failures.
3. Using the source code they have to "re-document" it using both techniques.

The empirical study took place in step 3, which can be divided into three phases:

- Preparation: consisting of a 4 hours lecture. The topics of the lecture were: an introduction to the problems of reverse engineering, description of the XUG annotation language, description of the tool Omondo.
- Work: the students were left free to work on the system and produce the required diagrams.
- Wrap-up: the students delivered the diagrams together with a filled-in questionnaire.

The questionnaire was composed of three parts (see Appendix A for the complete questionnaire). Part I was about XUG, part II was about Omondo, and part III asked the tool preference and a validation question.

For the closed answer questions in the questionnaire we used a five points Likert [4] scale coded, as customary,

with integers ranging from 1 to 5. In addition we had a question about the effort, measured in person-hours, and three open questions designed to collect comments from the subjects.

3 .3 Population and sampling

We observed seven groups of students working in an advanced software engineering course at the Politecnico di Torino. Each group was made of two or three students. The groups are the subjects of our study.

The students are in their final year of a Computer Engineering master degree. In their curriculum they had software engineering, data base, information systems, and object oriented programming courses.

We applied a convenience sampling: all the students attending the course are part of the population; the questionnaire was administered as a part of the course assignments.

The groups were assigned randomly to the two experimental groups: the first group was required to use first XUG and then Omondo, the second group adopted the inverse order.

3 .4 Instrumentation

The object of the experiment is a Heating Control System (HCS). The HCS monitors and controls the temperature in a building composed of several rooms. A house is composed by many subsystems (e.g. rooms or devices as heaters), that are connected with doors and tubes. It is also connected with the outside world through doors and windows. A house is also made of a texture of subsystems that strictly interact with each other. HCS can also check the weather conditions, an alarm, and safety conditions (e.g. critical heating statuses, weather effects).

It is a good example of a software system. It has been originally developed in C++ for use within the context of software engineering courses at the University of Kaiserlslautern. Then it has been ported to Java and adopted in the advanced software engineering course at Politecnico di Torino. The system is made up of 33 named Java classes or interfaces, distributed among a total of 28 source files.

The subjects used XUG version 1.0 (available at http://softeng.polito.it/projects/XUG) and Omondo EclipseUML Free Edition Version: 2.0.0.

3 .5 Variables

There is only one independent variable, the tool used to extract diagram: either XUG or Omondo.

On the basis of the questions defined in section 3 .1 we defined the set of metrics that are shown in Table 1. These metrics were measured from the answers to the questionnaire; as a result, most of them are expressed in a 1 to 5 ordinal scale.

The variables can be grouped according to which high-level hypothesis they address. The hypothesis is indicated in the first column of the table.

Table 1: Dependent variables used in the study.

Hp.	Var.	Unit	Description
Ha	MV	[1..5]	Difficulty in splitting the system into Multiple Views
	AS	[1..5]	Difficulty in Attribute Selection for display in the diagram
	REL	[1..5]	Difficulty in specifying the RELationships
	H	Person-hours	Effort in generating the diagrams
Hb	DD	[1..5]	Level of satisfaction with the Default Diagrams
	RES	[1..5]	Satisfaction with final RESult
Hc	TR	[1..5]	TRaining usefulness
	PC	[1..5]	Process Conformance

3 .6 Statistical analysis

In the hypothesis testing phase we split each high-level hypothesis into several detailed hypotheses: one for each related variable. All the detailed hypotheses share the following form:

HxVar_0: $\tilde{Var}_{XUG} = \tilde{Var}_{Omondo}$

Where x can be a, b, or c; Var is one of the related variables, and $\tilde{Var}_{tool}$ is the median of the variable Var for the given tool. Hypothesis Ha_0 was split into $HaMV_0$, $HaAS_0$, $HaREL_0$, and HaH_0. Hb0 resulted into $HbRES_0$ and $HbDD_0$. Hc_0 was divided into $HcTR_0$ and $HcPC_0$.

Due to the nature of the variables and the limited number of data point we decided to apply non-parametric statistical tests. In particular we selected the Mann-Whitney test [5] that is very robust and sensitive. Since H is a continuous variable, it can be checked for normality and, in case it is so, the t-test can be applied instead.

We decided to adopt the most commonly used value for the alpha-level: we consider statistically significant a test with a p-value lower than 5%. Therefore we will consider acceptable a probability of 0.05 for the type I error, i.e. rejecting the null hypothesis when it is true.

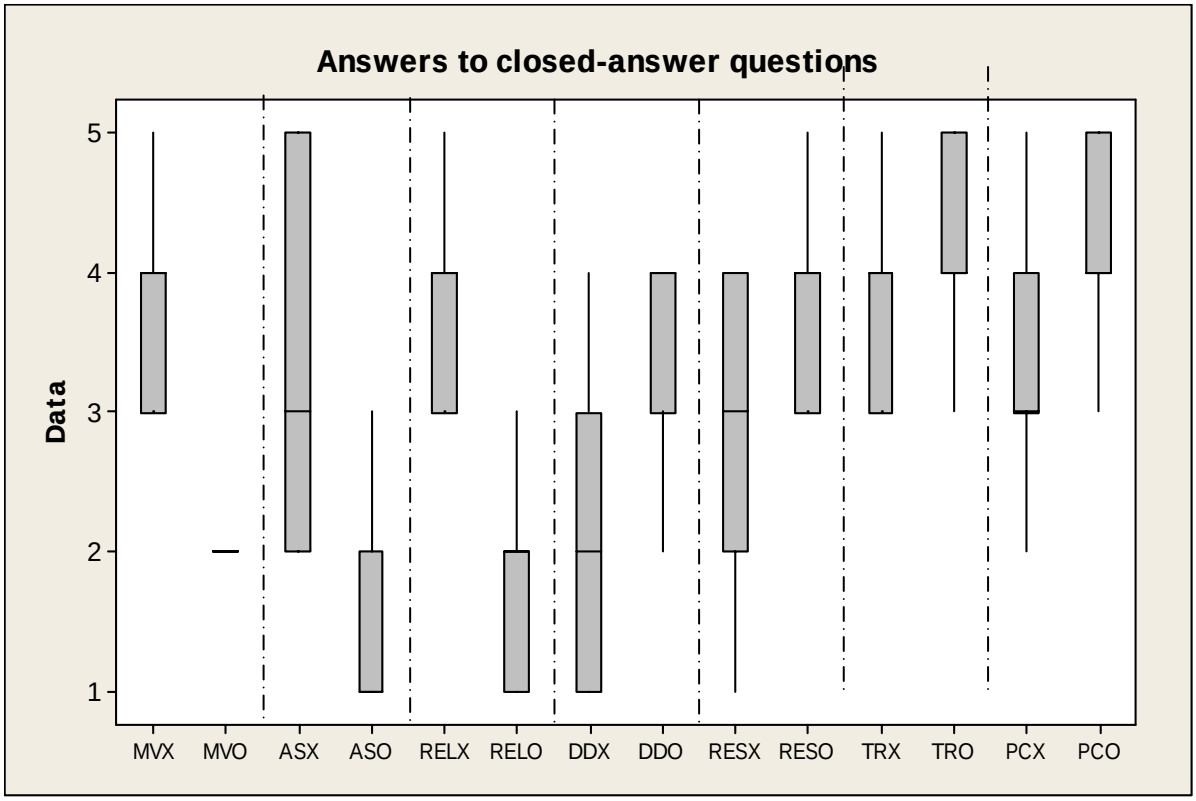

Figure 4: Answer to c.a. questions.

4. Experimental results

4 .1 Data Analysis

We received the questionnaire from all the seven subjects involved in the study.

From a validation question present in the questionnaire we know that three subjects used XUG first, three subjects used Omondo first, and one subject used them in parallel. Therefore we consider the experimental groups balanced.

The answers to the closed-answer questions are shown in Figure 4; we use the name of the variable with an extra X or O to indicate the value for XUG or Omondo respectively.

As far as the effort is concerned, EX has mean 12.71, standard deviation 10.05, and is normally distributed, EO has mean 10, standard deviation 10.44, and is not normally distributed. The condition for applying the t-test does not hold.

The metrics described in Section 3.6 have been collected after the completion of the re-documentation task and have been analyzed through the Mann-Whitney test. The results are shown in Table 2. Moreover, the answers to the open questions in the questionnaire have been carefully read and used to interpret the quantitative results.

Table 2: Results of Mann-Whitney test on the dependent variables.

Hp:	Var.	Median/Mean		p-value
		XUG	Omondo	
Ha	MV	**4**	**2**	**0.28%**
	AS	**3**	**1**	**1.52%**
	REL	**4**	**2**	**0.40%**
	H	10	5	56.53%
Hb	DD	**2**	**4**	**4.76%**
	RES	3	4	20.13%
Hc	TR	4	4	52.29%
	PC	3	4	9.67%

Based on these results, we reject the null hypotheses $HaMV_0$, $HaAS_0$, $HaREL_0$, and $HbDD_0$.

It should be noticed that for the variables MV, AS, REL and H a lower value is better, while for RES, TR, PC and DD a higher value is better. The p-values below the chosen alpha-level are in boldface. They indicate a statistically significant difference between XUG and Omondo.

Since the statistically significant differences involve variables on an ordinal scale we cannot say anything about the effect size.

4 .2 Interpretation

According to the results in Table 2, it is easier to specify multiple views of the same system in Omondo. With Omondo it is also easier to select which class attributes (fields and methods) to display, as well as to specify the inter-class relationships (see metrics MV, AS, REL). The *default* diagrams produced by Omondo are also judged substantially better than those obtained by XUG (see DD).

Among the metrics which do not show a statistically significant difference between Omondo and XUG, RES and H appear particularly interesting. Although there is a tendency in favor of Omondo (higher RES, lower H), the null hypothesis cannot be rejected. This means that we could not find any significant difference between the quality of the resulting diagrams in the two cases and the effort necessary to produce them.

Finally, it is interesting to report the following quotes, taken from the answers to the open questions:

XUG:

"A tool to add the annotations automatically to the code would be useful."

"Code annotation is quite time consuming."

"The annotations @show and @hidden should remain valid until overridden by a successive annotation, instead of holding for the following attribute only."

"XUG lacks interactivity/GUI."

"The layout of the diagrams produced by Graphviz is good."

"XUG lacks integration with Eclipse."

Omondo:

"It is not possible to depict two different kinds of relationships between the same classes."

"Omondo is good because it does not require annotating the source code."

"Hiding an entity from a view should be an operation kept more clearly distinct from removing it from diagram *and* code."

"Omondo should not annotate the code with its own special-purpose comments."

"With Omondo, it is easy to find the correspondences between design and code."

The experimental results clearly indicate that the drawing editor is the winning approach, in that it supports the specification of multiple system views in an intuitive way, thanks to the GUI, with no penalty on the quality of the output. As apparent also from the comments, the subjects involved in this study preferred the interactive creation of the views over their declaration through code annotations.

A detailed analysis of the answers to the open questions highlights several improvement areas for both tools. This may suggest that none of the two is currently able to support the user needs during a re-documentation task in a satisfactory way. Actually, Omondo was mainly conceived as a forward engineering tool, so that its use as a re-documentation tool is expected to be sub-optimal. The same holds for XUG, but for other reasons: XUG is a research prototype. This means that it does not reach the level of integration and usability of a commercial tool such as Omondo.

This is the list of the main improvements that can be derived from the open comments:

XUG:

- Annotation insertion is annoying. It should be supported with more automation (control-key sequences, GUI, etc.).
- Application of annotations to a block of consecutive entities should be supported.
- Integration with the overall programming environment is desirable.

Omondo:

- Code editing and view editing are different activities, which are not always well-separated.
- View editing should not result in code modifications, such as the insertion of special-purpose comments.

4 .3 Threats to validity

Although we did our best to minimize the threats to the internal and external validity of this study, some of them might have affected the obtained results. It is possible to identify four type of threats to validity [14]:

- Conclusion: concerns the relationship between the treatment and the outcome.
- Internal: concerns the correct identification of a cause-effect relationship.
- Construct: concerns the link between the theory and the observations.
- External: concerns the capability to generalize to a wider population.

As far as internal validity is concerned we must consider the training time. The training time was somewhat limited by the course schedule. Approximately the same time was devoted to the training of the students with Omondo and with XUG. However, we have reasons to believe that XUG may need more training for an effective use.

The different maturity of the tools can constitute a threat to the construct validity. Omondo is a commercial tool while XUG is a free research prototype. Correspondingly, the latter is trickier to use, it comes with less documentation, and it requires more familiarization. This is also related to the training time issue. It might be the case that a more mature version of XUG would have received a better appreciation from the

students.

The most important threats to external validity are the age and skill of programmers, and the size of the target program.

Since the involved programmers are all young, there might be a bias towards the usage of modern, integrated, graphical tools. This is especially true for Omondo, which is completely integrated into the Eclipse platform. It might be the case that senior programmers, who are used to work in an environment supporting just textual editing and the command line, are more receptive to code annotation tools, such as XUG, than to tools which require the development of the software within a graphical environment.

The use of students in experiments is always subject to debate although there is evidence that similar improvement trends can be observed both for students and professional developers [9].

There could be a mono-operation bias. The size of the target program is an important variable. We had to keep it limited, to comply with the requirements of the course in which the experiment was conducted. It is not clear if on larger programs the interactive facilities offered by Omondo would remain more effective and more usable than the annotation of the source code. Moreover, we could not test the maintainability and evolvability of the diagrams in the medium/long term.

5. Conclusions and future work

We conducted an experiment comparing the use of code annotations to drawing editors for the re-documentation of object oriented code.

The main outcome of the study is that the drawing editors are more usable and produce diagrams with a comparable perceived quality. Notably we could not find any difference in terms of effort required by the two approaches.

As a future work we plan to implement some of the suggested enhancement for XUG, in particular:

- a better integration with the development environment,
- improvement of the annotation language to reduce the number of annotations to be written

As soon as XUG achieve a more mature level we aim at using it within a large project to assess its usability and usefulness in an industrial context with a large code base.

In addition we plan to conduct further empirical studies. The main areas of interest are:

- comparing the maintainability and obsolescence of documentation produced with the two approaches,
- evaluating the usefulness of the documentation produced by means of reverse engineering in terms of improved maintenance effectiveness and efficiency.

6. References

[1] V. Basili, G. Caldiera, and D. Rombach, "Goal question metric paradigm" in *Encyclopedia of Software Engineering*, vol. 1, J. J. Marciniak, Ed.: John Wiley & Sons, 1994.

[2] B. Bruegge and A. H. Dutoit, *Object-Oriented Software Engineering: Conquering complex and changing systems*: Prentice-Hall, 2000.

[3] E. R. Gansner and S. C. North, "An Open Graph Visualization System and its Applications to Software Engineering" *Software Practice and Experience*, 30 (11): 1203-1233, 2000.

[4] R. Likert, "A technique for the measurement of attitudes" *Archives of Psychology*, 140: 5-55, 1932.

[5] H. B. Mann and D. R. Whitney, "On a test of whether one of two random variables is stochastically larger than the other" *Annals of Mathematical Statistics*, 1947.

[6] G. A. Miller, "The Magical Number Seven, Plus Or Minus Two: Some Limits On Our Capacity For Processing Information" *The Psychological Review*, 63: 81-97, March 1956.

[7] OMG, "Unified Modeling Language Specification" version 1.5, June 1999 available at http://www.omg.org/cgi-bin/doc?formal/03-03-01.

[8] Omondo, Omondo home page, available at: http://www.omondo.com (last access May 16, 2005), 2005

[9] P. Runeson, "Using Students as Experiment Subjects – An Analysis on Graduate and Freshmen Student Data" Proc. of 7th International Conference on Empirical Assessment & Evaluation in Software Engineering (EASE'03), April 8-10, 2003

[10] D. Spinellis, "On the declarative specification of models" *IEEE Software*, 20 (2): 94-96, March/April 2003.

[11] D. Spinellis, Drawing UML Diagrams with UMLGraph, available at: http://www.spinellis.gr/sw/umlgraph/ (last access May 16, 2005), 2005

[12] Sun Microsystems Inc., Doclet Overview, available at: http://java.sun.com/j2se/1.5.0/docs/guide/javadoc/doclet/overview.html (last access May 16, 2005), 2004

[13] P. Tonella and A. Potrich, "Reverse Engineering of the UML Class Diagram from C++ Code in Presence of Weakly Typed Containers" Proc. of ICSM 2001, International Conference on Software Maintenance, Florence, Italy, November 7-9, 2001, pp. 376-385.

[14] C. Wohlin, P. Runeson, M. Host, M. C. Ohlsson, B.

Regnell, and A. Wesslen, *Experimentation in Software Engineering: An Introduction*: Kluwer Academic Publishers, 2000.

7. Appendix A – Full questionnaire

We present here the full questionnaire we used.

Part I – XUG

I.1. Showing the same classes on several diagrams has been
[] immediate; [] easy; [] easy enough; [] difficult; [] complex

I.2. Selection which attributes and methods to show in each class has been
[] immediate; … [] complex

I.3. Specifying the associations among the classes has been
[] immediate; … [] complex

I.4. How do you judge the final result?
[] bad; [] not satisfying; [] sufficient; [] good; [] excellent

I.5. The guidelines provided in the classroom have been
[] counterproductive; [] not useful; [] not relevant; [] useful; [] very useful

I.6. The process proposed in the classroom has been followed
[] not at all; [] for a small part; [] partially; [] mostly; [] completely

I.7. How do you judge the diagrams produced automatically without annotations?
[] bad; [] not satisfying; [] sufficient; [] good; [] excellent;

I.8. Which was the main difficulty?
[] annotate the code[] find the correspondences between design and code [] locate the code relative to diagrams [] other, please detail:

I.9. How many person-hours have been used to produce the diagrams?

I.10. How could we improve the annotations in XUG?

I.11. What are your general impressions of XUG?

Part II – Omondo

II.1. Showing the same classes on several diagrams has been
[] immediate; … [] complex

II.2. Selection which attributes and methods to show in each class has been
[] immediate; … [] complex

II.3. Specifying the associations among the classes has been
[] immediate; … [] complex

II.4. How do you judge the final result?
[] bad; … [] excellent

II.5. The guidelines provided in the classroom have been
[] counterproductive; … [] very useful

II.6. The process proposed in the classroom has been followed
[] not at all; … [] completely

II.7. How do you judge the diagrams produced automatically?
[] bad; … [] excellent;

II.8. Which was the main difficulty?
[] adjust graphically the diagrams [] find the correspondences between design and code [] locate the code relative to diagrams [] other, please detail:

II.9. How many person-hours have been used to produce the diagrams?

II.10.How could we improve the Omondo?

II.11.What are your general impressions of Omondo?

Part III – Overall questions

III.1.Which tool did you use first?
[] Omondo [] XUG [] both in parallel

III.2.Which tool do you prefer and why?