An Experience Report Related to Restructuring OODesigner: A CASE Tool for OMT[1]

Taegyun Kim
Department of Computer Engineering, Pusan University of Foreign Studies
Uam-Dong 55-1, Nam-Gu, 608-738, Pusan, Korea
Email: ktg@taejo.pufs.ac.kr; Tel: +82 51 640 3178; Fax: +82 51 640 3038

Nacer Boudjlida
UHP Nancy 1, LORIA UMR 7503, BP 239
54506-Vandoeuvre Les Nancy CEDEX
Email: nacer@loria.fr; Tel: +33 3 83 59 30 76; Fax: +33 3 83 41 30 79

Abstract

This paper describes experience gained and lessons learned from restructuring OODesigner, a Computer Aided Software Engineering (CASE) tool for Object Modeling Technique (OMT). This tool supports a wide range of features such as constructing the three models in OMT, managing information repository, documenting class resources, automatically generating C++ and Java code, reverse engineering C++ and Java code, searching and reusing classes in the corresponding repository and collecting metrics data. We had developed the version 1.x of OODesigner during 3 years since 1994. Although we had developed this version using OMT and C++, we recognized the potential maintenance problem that originated from the ill-designed class architecture. Thus we totally restructured that version during 12 months, and we got a new version that is much easier to maintain than the old one. In this paper, we present what we did to restructure it and what we gained after the restructuring, and we also present a brief overview of the major features of OODesigner.

Keywords: Object Modeling Technique, Restructuring, Refactoring, Computer Aided Software Engineering Tool, Object-Oriented Metrics

---

[1] In *Proceedings of the Asian and Pacific Software Engineering Conference, APSEC'98, IEEE Computer Society Press.* Pages 220-227. Taipee, Taiwan, December 1998.

1. Introduction

Object-oriented (OO) paradigm can provide software engineers with practical and productive ways to develop software in many application areas[7]. The research projects[4,10] in the past decade have shown that OO techniques are a more natural way of problem solving than structured techniques. Software engineers can take full advantage of such principles as abstraction, encapsulation, modularity, hierarchy, typing, concurrency, and persistency in a synergistic way[1]. However, it is more and more recognized that OO techniques have a long learning curve and that OO software is harder to change than it might at first appear to be. OO software often needs to be restructured with several design iterations before it can be reused. In this iterative process, refactoring[12,13] is a very useful technique. While refactoring does not change the behavior of a software, it can support software design and evolution by restructuring a software in the way that allows other changes to be made more easily. This paper describes and analyzes a refactoring experiment that concerned an OO CASE tool: OODesigner[8,9].

Since 1994, we developed a CASE tool, named OODesigner, to support Rumbaugh's OMT[14]. At the early stage of our project, we established two types of goals: *product goals and process goals.* The *product goals* were the functional requirements of OODesigner and the initial requirements included the following:

- Support for the three models of OMT: class, functional and dynamic models,
- Documentation of class resources,
- Maintaining information repository for the designed object models,
- Code generation for C++,
- Reverse engineering for C++ code,
- Storing/retrieving class definitions for reuse,
- Collecting metrics data for C++ program.

We also established some *qualitative process goals.* We started this project not only for making an OO CASE tool, but also for practicing the goodness of OO paradigm. So, it was very important to achieve the following process goals:

- Improve the ability to conduct OO design and implementation activity,
- Practice seamless and iterative characteristics of the OO development process,
- Apply OODesigner as a CASE tool for developing OODesigner itself,
- Ensure maintainability for further enhancement and platform migration.

In 1996, the version 1.x was released and it satisfied almost all the product goals. This version was implemented with 60 thousands of C++ source statements. But the version did not satisfy the process goals, especially with respect to maintenance issues. In other words, the old version worked correctly for the given requirements, but it had a bad class structure that made the enhancement of its functionality difficult to achieve. Thus we started restructuring OODesigner since mid 1996.

This paper presents the experience in restructuring OODesigner. In section 2, we explain the works done during restructuring and the results after restructuring. Some metrics compares version 1.x to version 2.x. In section 3, we briefly explain the features of the tool. Finally we conclude in section 4 by outlining our work and further issues.

2. Restructuring OODesigner

This section is devoted to the rationale of OODesigner restructuring, to the restructuring process and to a discussion of the benefits of the restructuring.

2.1. Problems and goals

Drawbacks of the initial design of the architecture of the tool have been recognized during the development of the first version. The ill-designed architecture made difficult the enhancement of the tool with new functions. In the early 1996, almost all the initially established product goals were achieved, except the support for the functional modeler and the dynamic modeler. At that time, one year effort was expected to develop the remaining modelers. Thus there were two choices: (1) simply continue to develop the remaining modelers with bad architectural design, (2) totally restructure the tool and develop them again with a better architectural design. After serious consideration, the latter approach was adopted. By mid-1996, the restructuring began, even though the software worked correctly for the existing functionalities.

Lessons learned from the development of the first version, and especially the "noble" characteristics of OO development, lead to the establishment of the major goals for restructuring:

- Make classes application independent,
- Reorganize the class inheritance tree to reduce the code size,
- Make the control structure loosely coupled,
- Localize the platform dependencies,
- Minimize the duplication of the source code,
- Minimize the global members,
- Increase the robustness of the code.

The most important purpose of these goals is to make the tool easier to maintain. The new version was expected to be more flexible and to be founded on a more reliable architecture for further enhancements.

2.2. The Restructured Items

"Trial-and-error" was a good "teacher" to master the OO paradigm. The experience of miss-using OO technique while developing the first version, enabled to itemize what should be patched for restructuring. More specifically, we tried to:

- <u>Reduce duplication of data members</u> in class clusters and <u>reduce duplication of code</u> in member functions of the same name by dividing member functions properly. In our opinion, the reduction of duplicated code was the most important work in restructuring, because it resulted in a software system to be easily changed. Easy to change means easy to maintain.
- <u>Encapsulate Motif widgets</u> with the corresponding callback functions. All graphical user interface (UI) components have been implemented as classes. It was very interesting to find out that the classes that were newly introduced had very similar interfaces to the corresponding classes in Microsoft Foundation library and in classes in Java Development Toolkit library. Thus, the new version is very portable to other platforms.
- <u>Let all data members be allocated dynamically and use dynamic binding</u> for method invocations as much as possible. C++ offers static binding scheme or dynamic binding

scheme alternatively, it is apt for OO novices to use static binding scheme for method invocation. The intensive use of dynamic binding considerably reduced the control complexity of the software, and notably increased its flexibility.

- Encapsulate global members or library functions in the corresponding class. Thus most of them were introduced into the corresponding class as static members. This work provided a uniform way of thinking for every component in the software. And it also confined platform dependent library functions to reside in local parts of source code.
- Make destructors more complete. Because we suffered from several serious errors resulting from dangling references, we made efforts to make destructors safe. This effort makes the tool more reliable and easier to debug.
- Use some coding conventions of our own. C++ offers many syntactic alternatives: we decided to use a small set of C++ syntax. For example, we decided not to use reference type and template at all. Fortunately, our choice revealed reasonable and met the Java style, as we noticed after the emergence of Java.

Figure 1 and Figure 2 show the typical changes that have been made during restructuring. These figures represent the class clusters for modeling notations of OMT. Figure 1 is the design module for old version 1.x and Figure 2 is the design module for new version 2.x.

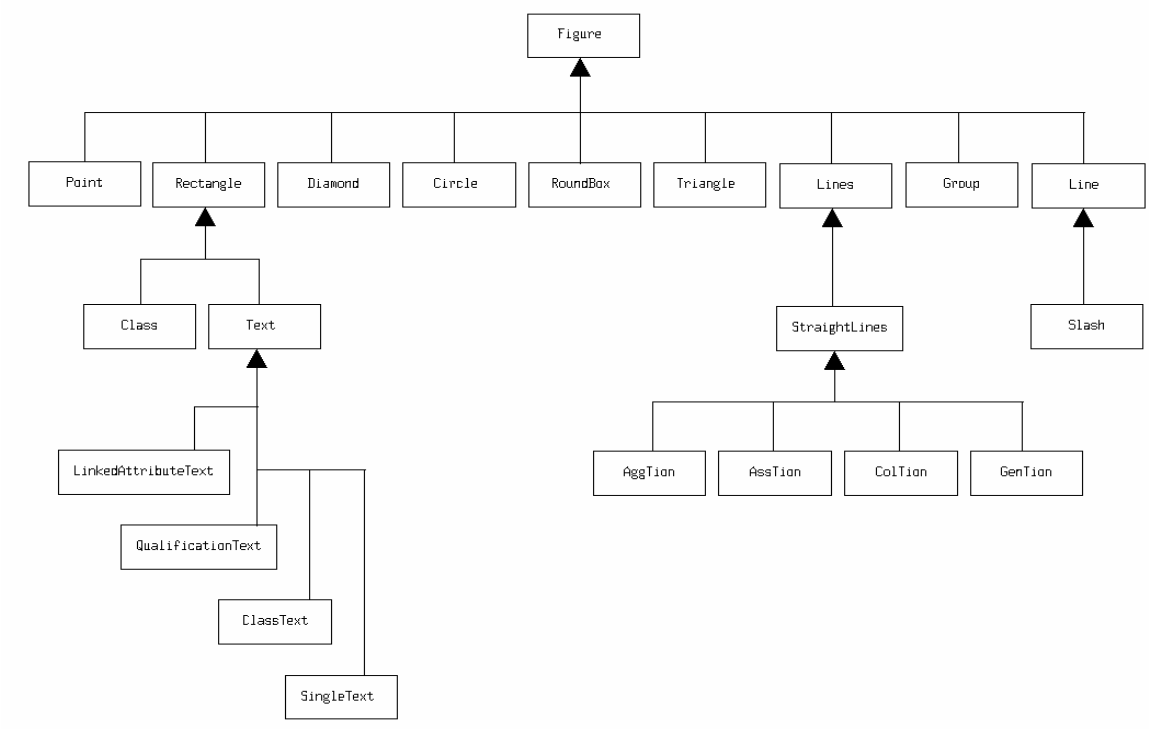

Figure 1: The object model for OMT notations in version 1.x

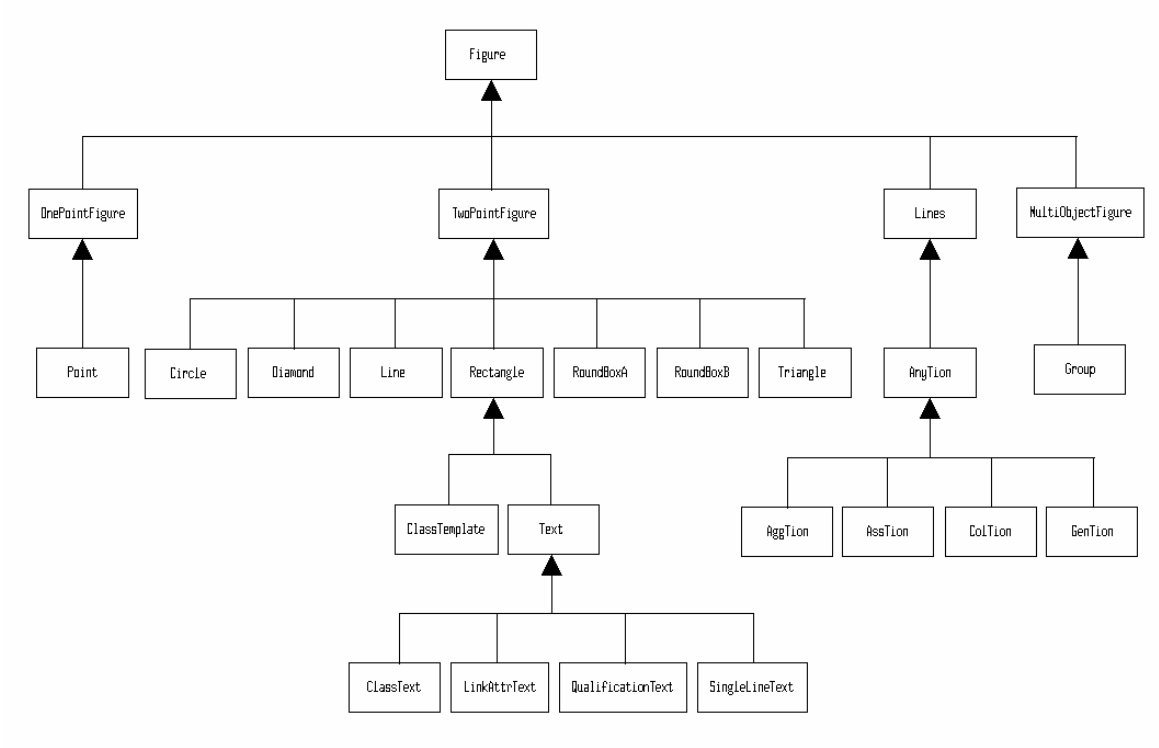

Figure 2: The object model for OMT notations in version 2.x

2.3. The Restructuring Process

OODesigner was restructured for 12 months. The version 1.x already had the major functions such as C++ reverse engineering and C++ code generation, and *it has been used* while restructuring it. So, the restructuring process was the following:

1. Reverse the source code of the version 1.x thanks to the version 1.x itself: this step resulted in the class diagrams of the version 1.x.
2. Modify the class diagrams generated by the reverse procedure using the version 1.x itself, i.e. draw the new class diagrams for the future version using the existing version.
3. Generate C++ code for the version 2.x using the version 1.x itself.
4. Manually modify the generated source files whenever a problem is encountered during the implementation phase.

As OODesigner 1.x supported the steps 1 to 3 of its own restructuring, great productivity was reached while restructuring it. This experience of applying a CASE tool for a project was one of the most important lessons that we gained. Because an OO CASE tool like OODesigner dramatically reduces trivial and redundant works, we believe that use of tools is one of major keys for conducting successful OO projects. Further, since no "catastrophic" crashes occurred during the restructuring phase, we could also ensure the reliability and the available functionalities of the tool thanks to this experience.

2.4. Benefits Gained

We feel that the major benefits of the restructuring is that the tool became:

- Easier to modify,

- Easier to enhance,
- Easier to understand,
- Easier to port to other platforms,
- More flexible,
- Machine independent,
- More stable and reliable,
- And finally easier to maintain.

For example, we could enhance it with the functional and the dynamic modelers within one month instead of one year duration that was expected for the first version. And the other enhancements related to Java were also accomplished very productively: one month was sufficient to develop the Java code generator and another month to provide reverse facility for Java. We sometimes modified not only the implementation but also the interface of some classes. But we feel that the modification of some classes does not affect too far from the place where the modification occurred. This means that the classes in the version 2.x are loosely coupled.

The new version 2.x is currently being ported from Unix version to PC version using Visual C++. A Java version is being also developed concurrently. Furthermore, while we are porting to other platforms, we are trying to enhance the tool functionalities to support Unified Modeling Language (UML) [5]. This work is being done very productively because of the flexibility and machine independence structure. As for restructuring, OODesigner is used as a CASE tool for porting activities. For example, while porting to Java version, we reversed C++ code of OODesigner, and then generated Java source code using OODesigner itself. Nevertheless, we had to customize the tool temporarily for that work.

2.5. Lessons learned after restructuring

We gathered some metrics data after the restructuring was completed. Maintaining a metrics database for process tracking is very useful in setting future project schedules. But if metrics are to cumbersome and time consuming to collect, collecting metrics is apt to be neglected[11]. Thus our approach to collect metrics is deliberately restricted to metrics that can be collected automatically on the basis of simple syntactic analysis. Some of the metrics that are presented below, relate to Chidamber and Kemerer ("C&K" as abbreviation) OO metrics suite[2], even though some authors [3,6] criticized the incompleteness of that suite. Some other metrics were gathered according to subjective viewpoints without relying on any theoretical basis.

The metrics data collected in the table 1 provides brief comparisons between ill-structured OO software and well-structured OO software. We don't believe that the absolute values in this table have great significance but we believe that the relative ratio of the metrics value give us some insights of making good OO software.

Metrics	Version 1.x	Version 2.x	Increment Ratio
1. Total number of lines in the system	58925	59562	+ 1%
2. Total number of semicolons in the system	33245	30535	- 8%
3. Total number of conditional statements	6382	5622	-12%
4. Total number of loops	1261	1091	-13%
5. Total number of classes	97	155	+60%
6. Total number of semicolons in classes	25692	25916	+1%
7. Average number of semicolons per class	265	167	-37%
8. Total number of unique words in classes	2635	3561	+35%
9. Average number of unique words per class	27	23	-15%
10. Total number of data members in classes	795	1030	+30%
11. Average number of data members per class	8.2	6.6	-20%
12. Total number of methods in classes	918	1430	+56%
13. Average number of methods per class	18	15	-17%
14. Average number of semicolons per method	14	11	-21%
15. Maximum inheritance depth	3	4	+33%
16. Total number of inheritance depth	134	233	+74%
17. Average number of inheritance depth	1.38	1.50	+9%
18. Maximum number of children	19	29	+53%
19. Total number of children	76	126	+66%
20. Average number of children	0.78	0.81	+4%
21. Total number of coupling between object classes	526	1002	+90%
22. Average number of coupling between object classes	5.4	6.5	+20%

Table 1: Metrics comparisons between the two versions

In table 1, the big difference of number of classes between the two versions is due to the fact that all graphical UI components of Motif were introduced as classes. The big increment in the number of classes makes some metrics values of no importance. In other words, several cases of total values like the values for the items 6, 8, 10 and 12 give us misleading interpretation for the trend of changing metrics values between the two versions. Thus these values are not to evaluate the empirical data, but use the corresponding average values per class. Some points of interest with the empirical metrics can be listed as follows.

- *Item 3, 4:* These values express the fact that the general complexity of the system is reduced. These values give us an encouragement for doing restructuring.

- *Item 5, 7, 9:* These values provide a good guideline for OO beginners. The guideline may be: "make classes as small as possible". With the value in item 9, we think that the number of unique words in class definition can be a good candidate for complexity measurement of a class.
- *Item 11, 13, 14:* These values relate to members of the corresponding class. Specifically, the item 13 tightly relate to C&K's *Weighted Methods per Class*(WMC) with unit complexity. Because the number of members in classes of the new version is smaller than they are in the old version, we can conclude that the new version is less application specific and more reusable.
- *Item 15, 16, 17:* These values relate to C&K's *Depth of Inheritance Tree*(DIT). With this data, DIT seems to increase in the new version, and that means that the software is more reusable.
- *Item 18, 19, 20:* These values relate to C&K's *Number of Children*(NOC). The increased NOC in the new version also means that we tried to make the software more reusable sacrificing miss-usability of subclassing.
- *Item 21, 22:* These values relate to C&K's *Coupling between Object Classes*(CBO). The result shows that we have worst coupling in the new version. We guess that the bad result comes from the fact that we employed too many classes in the new version or that simple measurement of coupling is not sufficient as noted by Hitz[6].

Two kinds of lessons can be advocated from this project. The first one relates to technical issues and the second one relates to managerial issues. We don't believe that people in OO field could generally accept these lessons, but we believe that these lessons were very important experience for us.

(1) *From the technical perspective*, in OO design and programming, one should:

- keep the class size, the number of class members and the method size as small as possible
- use inheritance "aggressively". We believe that a large inheritance depth and a large number of children encourage good OO design and implementation. Nevertheless, the largest inheritance depth should be limited: from in our informal experience, we suggest that it should not be greater than five or six.
- Use dynamic binding "aggressively" too: C++ novices often hesitate to use virtual functions. Instead, they invoke methods using dotted notations and this practice leads to bad OO programming habits. Dynamic binding should be the "primary way of thinking" in OO programming.

(2) *From the managerial perspective*, we learned the following lessons during this project.

- It is inevitable for beginners of OO paradigm to fail in the first OO project even if they are experts of structured techniques. They might implement operational software, but their system may reveal harder to maintain as time passes. We think that a well designed training programs for OO novices should be conducted before they are deployed into an OO project.
- An OO project could be successfully conducted just in the case of applying OO methodology, OO language and OO CASE tool synergistically. Using OO language alone without methodology to build OO software can not achieve a good productivity.
- If we feel the need of restructuring an OO legacy system, we should not hesitate to restructure it. To defer restructuring will cause a critical maintenance problem that cannot be avoided sometime in the future. Thus we should be aggressive to restructure an OO legacy system.

- In OO design and implementation point of views, choosing proper words of nouns for defining class names or data members and selecting proper words of verbs for defining services are very important to develop successful OO software. That means a good naming convention reduces the complexity of the way of thinking.
- Ill-designed OO software makes maintenance activity terrible, but well-designed OO software makes it enjoyable. We think that the good feeling about well-designed OO software promotes the productivity of project personnel.

3. Current State of OODesigner

In this section, we provide general information about OODesigner including its brief functionality. OODesigner was built on a Sun workstation running OS-4.1.x, X11-R5, Motif-1.2 and C++-2.0. It can be built on most Unix systems with X11-R5, Motif 1.2 and a reasonable C++ compiler. The source code of it contains 190 classes with about 60,000 lines of code. OODesigner was released to public domain for free use since 1995. It can be found at *ftp://203.230.73.24/pub/OOD* and it was also registered at the Asset Source for Software Engineering Technology (ASSET) organization. Figure 3 shows typical screen session.

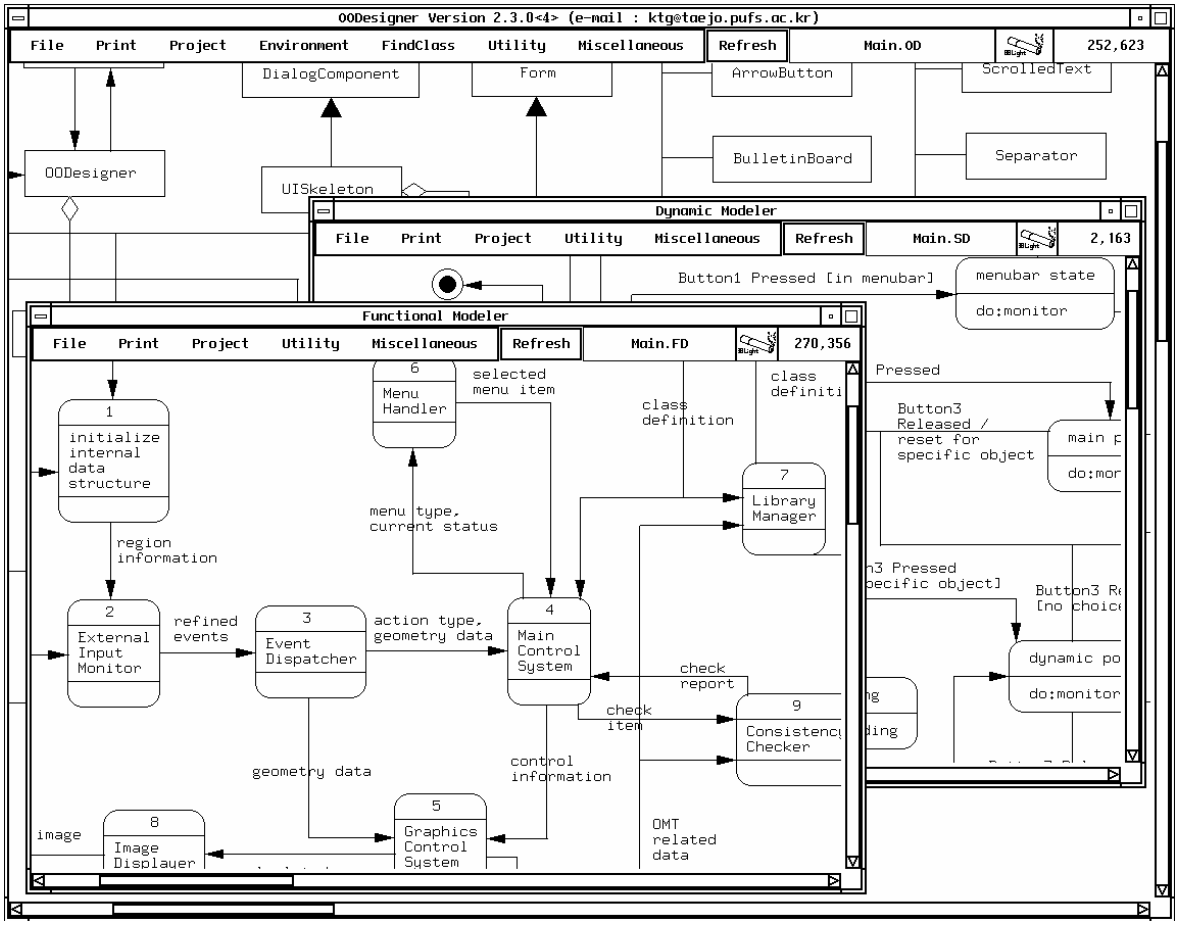

Figure 3: Typical screen session of OODesigner

OODesigner currently supports the following functions:

- General graphics editor: it supports drawing primitive diagrams including points, lines, triangles, rectangles, circles as well as annotations. The reason why we provided these primitive notations was that these are used as basic objects for constructing the notations of OMT models. For example, it was possible for the class notation to be constructed by combining two rectangles, seven lines, and five text objects.
- Three modelers for OMT: it supports drawing OMT notations like class diagrams, dataflow diagrams and state transition diagrams. Figure 3 shows the three modelers of this tool.
- C++ and Java code generation: it supports automatic code generation from class diagrams. The comments and code for individual member functions can be documented or edited within OODesigner using proper dialogs. As OODesigner can contain source instructions of class member functions in class diagrams using the documentation facility, it can generate a perfect header file and source file for the corresponding class definition. In other words, a well-documented class in a class diagram can be transformed into C++ files that need no manual changes for compilation. This documentation facility enables class diagram and C++ files to have a one to one correspondence.
- Reverse engineering facilities for C++ and Java code: it supports making class diagrams from source code. There are two major roles in the reverse engineering facility. One of them is extracting class diagrams from C++ files that are written manually, and the other is ensuring consistency between class diagrams and the corresponding C++ files that are generated initially by OODesigner. The reversed class diagram contains source instructions of class member functions like in the case of code generation. It also can extract some information about relationships found in C++ source files. The information about relationships such as generalizations, aggregations and collaborations will be inserted into the information repository. Figure 4 shows the reversed class diagram for some Java library classes.

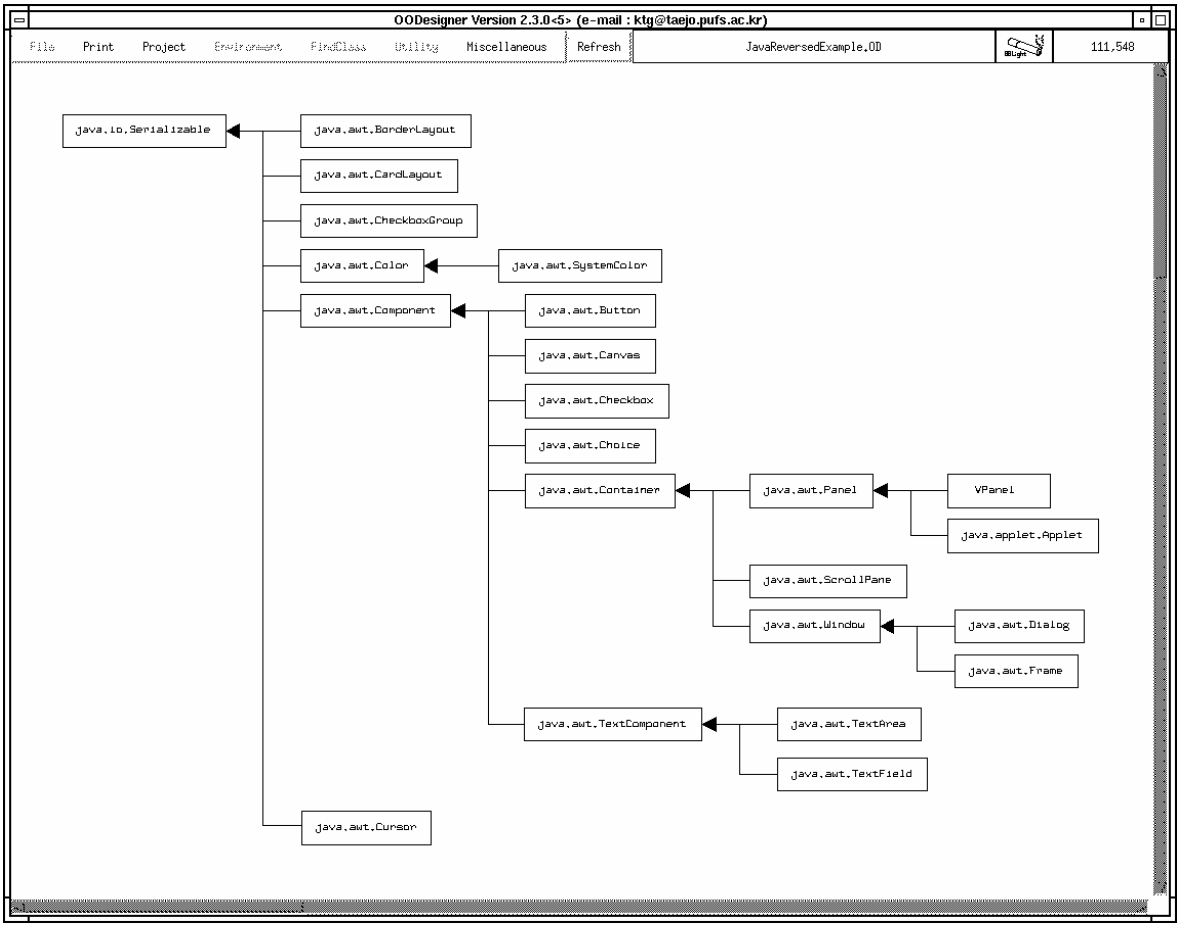

Figure 4: An example of reversed class diagram from Java code

- Class search facilities: it enables us to easily find the existing classes in the repository. The class search facility of OODesigner is implemented for encouraging reuse of classes, and allows three methods for searching classes. The search schemes employed in this tool are the search by exact matching, the search by enumeration and the search by relationships.
- Metrics data collection: it collects metrics data of source code. These metrics data could be used to measure and validate the quality of our project, and it can be used to control the evolution of the project. The collected data has three parts, data for each class, for all member functions and for overall statistics. This data will be used as a basis for serious metrics evaluation. The metrics data in Table 1 and Table 2 was also collected thanks to this facility.

4. Conclusion

In this paper, we presented the experience acquired when we restructured an OO CASE tool named OODesigner. We discussed:

- The problems we had with old version of the tool,
- The goals of the restructuring, the restructuring decisions and process,

We also presented:

- the benefits of the restructuring,
- metrics comparisons between the two versions of the tool,

- and finally the lessons learned from this work.

In summary, the major benefits are that we could get a new version that is easier to modify, to enhance, to understand, to port and to maintain. We also shortly introduced the facilities of the tool in this paper. This tool supports the following functions:

- General graphics editor
- Three modelers for OMT
- C++ and Java code generation
- Reverse engineering facilities for C++ and Java code
- Class search facilities
- Metrics data collection

With the new version of OODesigner, we are currently enhancing some functions and are porting it to other platforms. This work is being done very productively because of the flexibility and the machine independence structure of the tool. Furthermore, as for restructuring, we use OODesigner itself as a CASE tool to enhance it.

The ultimate goal of our work is a full implementation of an OO environment based on graphics tools. We plan to extend this tool so that it can support all the models of UML[5]. We also plan to make it usable in a distributed environment. And, a final mid-term objective is to make it supporting OO design process.

5. Acknowledgements

This work was supported by a grant No KOSEF 961-0906-038-2 from the Korea Science and Engineering Foundation.

6. References

[1] G. Booch, *Object Oriented Design with Application*, Benjamin Cummings, 1991.

[2] S. R. Chidamber and C. F. Kemerer, "A Metrics Suite for Object-Oriented Design," IEEE Tran. On SE., Vol. 20, No. 6, pp. 476-493, 1994.

[3] N. I. Churcher and M. J. Shepperd, Comments on "A Metrics for Object-Oriented Design," IEEE Tran. SE., Vol. 21, No. 3, pp. 263-265, 1995.

[4] R. G. Fishman and C. F. Kemerer, "Object Oriented and Conventional Analysis and Design Methodologies," IEEE Computer, Vol. 25, No. 10, pp. 22-40, 1992.

[5] M. Fowler, K. Scott, *UML Distilled: Applying the Standard Object Modeling Language*, Addison Wesley, 1997.

[6] M. Hitz and B. Montazeri, "Chidamber and Kemerer's Metrics Suite: A Measurement Theory Perspective," IEEE Tran. SE., Vol. 22, No. 4, pp. 267-270, 1996.

[7] S. Honiden, et al., "An Application of Artificial Intelligence to Object-Oriented Performance Design for Real-Time Systems," IEEE Trans. on SE., Vol. 20, No. 11, pp. 849-867, 1994.

[8] T. Kim, *An Object-Oriented Development Environment Based on Extended Object Modeling Technique*, Ph.D. thesis, Seoul National University, 1995.

[9] T. Kim, "Restructuring OODesigner: A CASE Tool for OMT," Proc. ICSE'98, pp. 449-451, 1998.

[10] P. H. Loy, "A Comparison of Object-Oriented and Structured Development Methods," ACM SIGSOFT SE Notes, Vol. 15, No. 1, pp. 44-48, 1990.

[11] R. Malan, D. Coleman and R. Letsinger, "Lessons from the Experience of Leading-Edge Object Technology Projects in Hewlett-Packard," Proc. OOPSLA'95, pp. 33-46, 1995

[12] I. Moore, "Automatic Inheritance Hierarchy Restructuring and Method Refactoring," Proc. OOPSLA'96, pp. 235-250, 1996

[13] W. F. Opdyke, *Refactoring Object-Oriented Frameworks*, Ph.D. thesis, University of Illinois at Urbana-Champain, 1992.

[14] J. Rumbaugh, et al., *Object-Oriented Modeling and Design*, Prentice Hall, 1991.