An Experiment in Refactoring an Object Oriented CASE Tool

Nacer Boudjlida
UHP Nancy 1, LORIA UMR 7503, BP 239
54506 - Vandœuvre Lès Nancy Cedex (France)
E-mail: nacer@loria.fr; Tel: +33 3 83 59 30 76; Fax: +33 3 83 41 30 79

Taegyun Kim*
Pusan University of Foreign Studies, Department of Computer Engineering
55-1, Uam-Dong, Nam-Gu, 608-738, Pusan, Korea
E-mail : ktg@taejo.pufs.ac.kr; Tel: 051-640-3178; Fax: 051-640-3038

Abstract:

This paper describes experience gained and lessons learned from restructuring OODesigner, a Computer Aided Software Engineering (CASE) tool that supports Object Modelling Technique (OMT). This tool supports a wide range of features such as constructing the three models of 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. A version 1.x of OODesigner has been developed for 3 years since 1994. Although this version was developed using OMT (i.e. the tool has been designed using OMT) and C++, we recognized the potential maintenance problems that originated from the ill-designed class architecture. Thus that version was totally restructured, resulting in a new version that is easier to maintain than the old one. In this paper, we briefly describe the tool's functionality, its development process and its refactoring process, emphasizing the fact that the refactoring of the tool is conducted using the tool itself. Then we discuss lesson learned from these processes and we exhibit some comparative measurements of the developed versions.

Key-Words: Object-Oriented Design and Programming, Computer-Aided design environment, Refactoring, Process, Measurement

1 Introduction

Object Oriented (OO) research and development projects [14, 7, 4, 10, 15, 11, 20] 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 [2]. It is more and more recognized that OO design and OO programming has a long learning curve and as such, iteration on the design as well as on the produced code is usual to make the implemented software more efficient and more flexible. In this iterative process, refactoring [18, 17] is a very useful technique. Indeed, instead of completely re-designing the software system, refactoring aims at changing the software structure to make it more understandable, easier to maintain and easier to

*Acknowledgements: This work was partly supported by a grant No KOSEF 981-0923-123-2 from the Korea Science and Engineering Foundation.

be enhanced with new functionalities. This paper describes and analyzes a refactoring experiment that concerned an OO design tool: OODesigner. The initial development of OODesigner started in 1994 with the objective to provide an environment to support Rumbaugh's OMT [21]. At the early stage of the project, two types of goals were fixed: product goals and process goals. The product goals were the functional requirements of OODesigner and the initial requirements included the following:
- Support the design of the three models of OMT: class, functional and dynamic models,
- Document the class resources,
- Maintain a repository for the designed object models,
- Generate code for C++,
- Reverse engineer class diagrams from C++ code,
- Store/retrieve class definitions for reuse,
- Collect metrics data for C++ program.

Some qualitative process goals were also fixed. Among these goals, we especially willed to *(i)* improve our capability to conduct OO design and implementation, as for [15], *(ii)* to practice seamless and iterative characteristics of the OO development process, *(iii)* tose OODesigner as a CASE tool, to develop and to refactor OODesigner itself and *(iv)* to ensure maintainability for further enhancements and for platform migration.

In 1996, the version 1.x of OODesigner was released and it satisfied almost all the product goals. This version was implemented as 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 version worked correctly for the given requirements, but it had an inappropriate class structure that made the enhancement of its functionality difficult to achieve. Thus we started refactoring OODesigner since mid-1996. We first identified the main drawbacks of the initial version, then we adopted some refactoring goals and principles before actually refactoring the tool. *One of the originality of the refactoring process we followed is that we supported the production of the new version of the tool by the first version.*

This paper describes and tries to measure this experiment. We first give a brief overview of the facilities that are offered by the current version of OODesigner (section 2). Section 3 is the core of the paper: we present the deficiencies of the initial version, the refactoring decisions (sections 3.1 and 3.2), the refactoring process (section 3.3) and the results of that process (section 3.4). Section 3.4 also gives some comparative measurements of both the versions. We conclude, in section 4, with the description of the on-going and the further development of the system.

2 Current State of OODesigner

In this section, we provide general information about OODesigner including a rapid presentation of the provided facilities. 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 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 2, depicts the life-cycle of OODesigner as a spiral model [1]. This figure shows the history and the current state of the OODesigner development process.
OODesigner currently supports the following functions:

- *General graphical editor* to draw primitive diagrams (line, triangle, rectangles, etc).

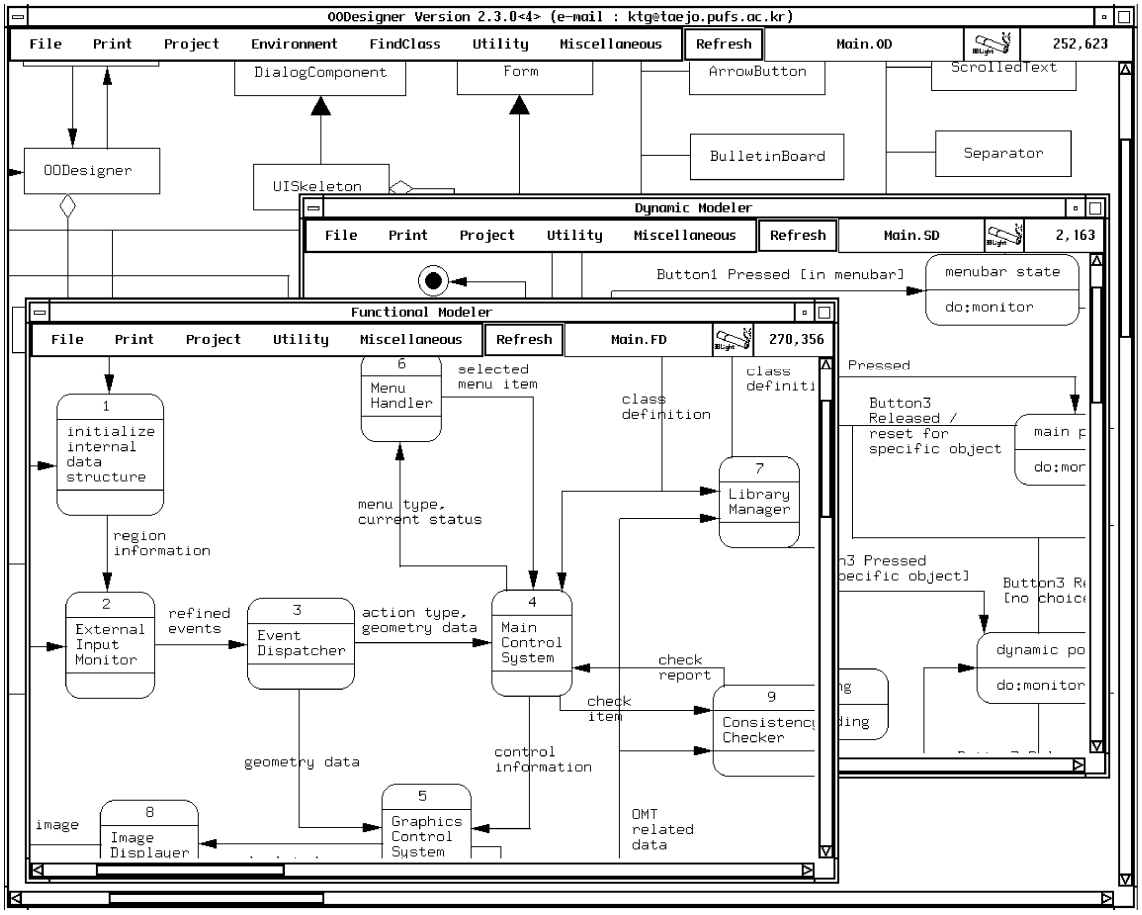

Figure 1: Typical screen session of OODesigner

- *Three modellers for OMT*: class, dataflow and state transition diagrams (Figure 1).
- *C++ and Java code generation, editing and documentation*.
- *Reverse engineering facilities for C++ and Java code*.
- *Class search facilities* to retrieve existing classes in the repository.
- *Metrics data collection*.

3 Refactoring OODesigner

This section is devoted to the rationale of the OODesigner refactoring (sections 3.1 and 3.2), to the refactoring process (section 3.3) and to a discussion of the benefits of the refactoring (section 3.4).

3.1 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 rendered 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 modeller and the dynamic modeller. At that time, one year effort was expected to develop the remaining modellers. Thus there were two choices: **(i)** simply continue to develop the remaining modellers with bad architectural design, **(ii)** totally restructure the tool and develop it again with a better architectural design. After serious considerations, the latter approach was adopted. By mid-1996, the restructuring began, even

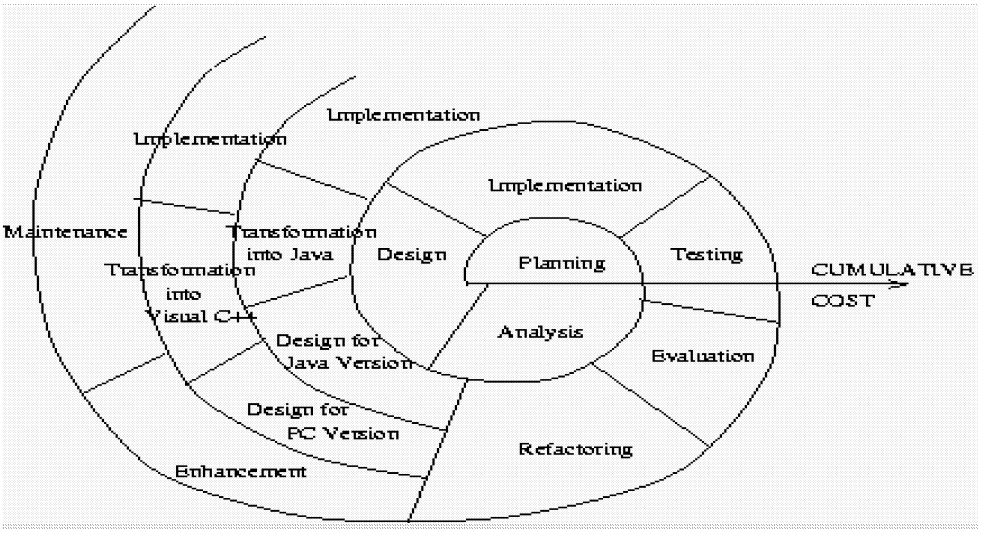

Figure 2: The OODesigner's Lifecycle

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.

3.2 The Refactored Items

"Trial-and-error" development process 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 refactoring. In the following sections, we present the important items which were patched for refactoring.

3.2.1 Reduction of Duplicated Data Members and Code

We tried to reduce *duplication of data members* in class clusters and to reduce duplication of code in member functions of the same name by dividing member functions properly. This work causes separately designed classes to be given a common superclass. It also makes the trend of migrating functionality up into the superclass [22]. For example, duplicated data members existed

in classes *GenTion, AggTion, ColTion and AssTion.* that were defined to support the notations of generalization, aggregation, collaboration and association respectively (see figure 4). To reduce that duplication, we introduced a new class named *AnyTion* and made the duplicated data members migrate "up" into the *AnyTion* class (see figure 5).

We also were faced to duplicated source code. The duplication of source code does not mean that the bodies of the corresponding methods are exactly the same. Some methods with the same name were distributed into corresponding classes at the same inheritance level, and they usually had some partial duplication. In other words, the methods had not only same parts but also different parts. For example, duplication of source code typically occurs in the following style, where X and Y are pieces of code duplicated into A and B.

```
void A::foo() {                          void B::foo()
    X;   // same as X in B                 X;  // same as X in A
   Y1;                                     Y2;
    Z;   // same as Z in B                 Z;  // same as Z in A
}                                        }
```

We reorganized these member functions as follows assuming that the class C is the superclass of the classes A and B.

```
class C : public D {
        ...
        virtual void foo_prolog(); // This function deals with X
        virtual void foo_epilog(); // This function deals with Z
        ...
};

   void A::foo() {                         void B::foo() {
           Y1;                                     Y2;
   }                                       }
```

We carefully choose the names of member functions newly introduced like *foo_prolog()* and *foo_epilog()*. In our opinion, the reduction of duplicated code was the most important work in refactoring, because it resulted in facilitating changes of any part of the source code after the elimination of the duplicated code. More importantly, *easy to change means easy to maintain.*

3.2.2 Encapsulation of Graphical User Interface Components

We tried to encapsulate Motif widgets with the corresponding callback functions. All graphical user interface (UI) components have been implemented as classes (Figure 3 shows the designed class diagram for version 2.x.).

Due to this style of encapsulation applied to every Motif widget, several classes were created in version 2.x. There were no corresponding class diagram in version 1.x to this class diagram. In Figure 3, most of the class names correspond to the user interface components that constitute the application. For example, classes like *Button, Label, Separator* and *ArrowButton* are classes for primitive widgets of Motif and classes like *ObjectModeler, MainControl* and *OMMenuBar* are composite classes which are constructed by primitive UI components. It was very interesting to find out that the classes that were obtained had very similar interfaces to the corresponding classes in Microsoft Foundation library and in classes in Java Development Toolkit library. Thus, the new version of the environment is very portable to other platforms.

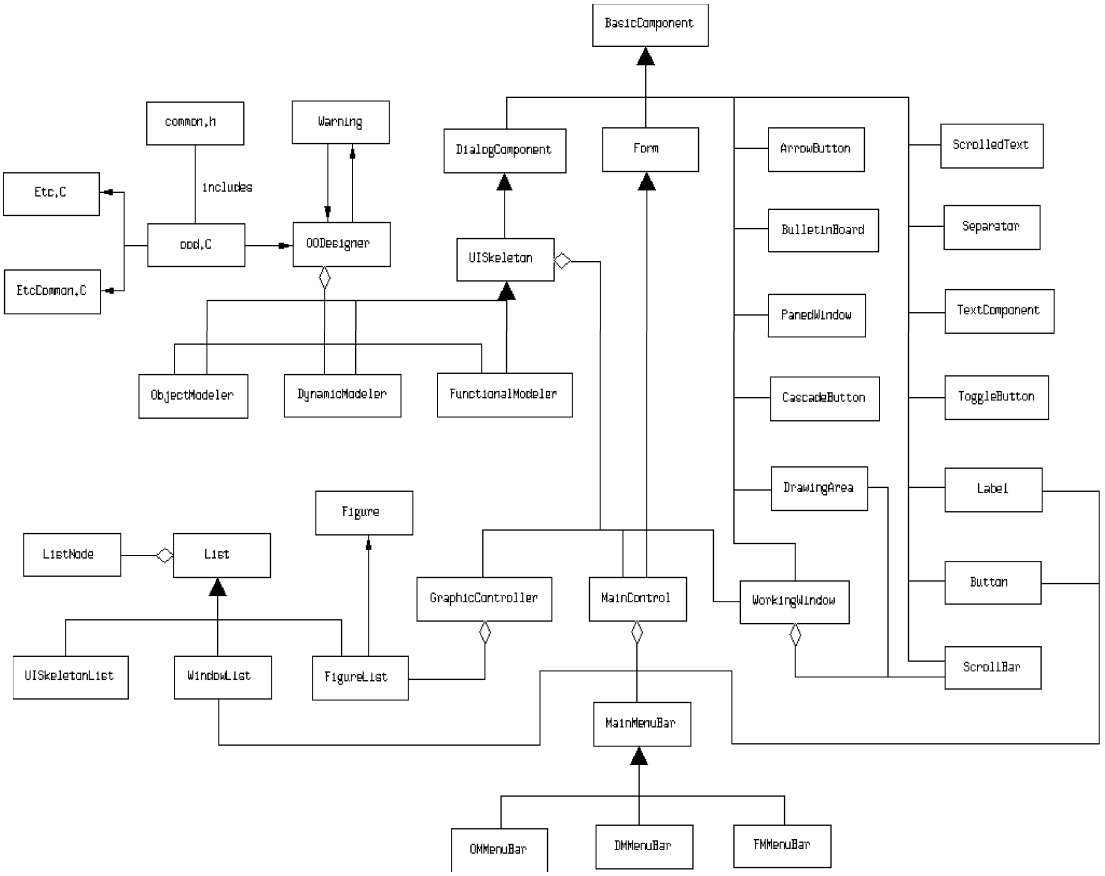

Figure 3: Partial Class Diagram of the UI Components in OOD Version 2.x

3.2.3 Intensive Use of Dynamic Binding

We tried to let all data members be allocated dynamically and use dynamic binding for method invocations as much as possible. C++ offers static binding scheme or dynamic binding scheme alternatively. It is usual for OO novice programmers to use static binding scheme for method invocation. In the early stage of the project, being OO apprentices, we usually used the two binding schemes in a mixed fashion. For example, the source code of version 1.x frequently contained the following mixed style:

```
...
List aList;
List *bList = new List();
aList.insert(someThing);
bList->insert(otherThing);
...
delete bList;
...
```

As we progressively gained experience in dynamic binding, we converted the above mixture of coding style to the only dynamic binding style. The uniformity of coding style gives a uniform way of object instantiation and message passing. More importantly, the intensive use of dynamic binding considerably reduced the control complexity of the software, and notably increased

its flexibility. Intensive use of dynamic binding also introduced several polymorphic container classes. When building the version 1.x, only one class (List) was used for containing objects. But using only one class as a container caused troubles with regard to information hiding. Namely, we had to violate the information hiding principle because the implementation of the List class had to be frequently accessed.

3.2.4 Encapsulation of Global Members

We tried to encapsulate global members or library functions in the corresponding class. Thus most of them were introduced into the corresponding class as static members. For example, the class named *OODesigner* in figure 3 was introduced to store the most important global data members of the system.

This work provided a uniform way of thinking for every component in the software. That means that most of the members are included in the corresponding classes as data members or member functions. This effort also confined platform-dependent library functions to reside in well-identified local parts of the source code.

3.2.5 Reliability Considerations

We tried to make *destructors operations* more complete. Special effort was dedicated to make destructors safer in order to deal with dangling references. This effort contributed to make the tool more reliable and easier to debug. In order to make destructors safer, we always made the complementary destructors for the corresponding *constructors*. That means that whenever any object instantiation for data components exists in the constructors of a class, we associate deallocation statements for freeing the members in the corresponding destructor.

3.2.6 Adoption of Coding Conventions

We used coding conventions of our own. C++ offering many syntactic alternatives, we restricted ourselves to a small subset of C++ syntax. For example, we decided:

- Not to use reference type for argument passing or object reference but to use pointer type,
- Not to use template but to use inheritance,
- Not to use nested classes but to use aggregate components,
- Not to use public data members but to use access functions as interfaces for data members,
- Not to use function pointers at all.

Very fortunately, these choices revealed reasonable and met the Java style, as we noticed after the emergence of this language. Figure 4 and figure 5 show the typical changes that have been made during refactoring. These figures represent the class clusters for modeling notations of OMT. Figures 4 and 5 are the design modules for the versions 1.x and 2.x, respectively.

3.3 The Refactoring Process

OODesigner was restructured for 12 months. The version 1.x was already fitted with the major functions such as C++ reverse engineering and C++ code generation, and it has been used during the refactoring activity. So, the refactoring process was the following (Figure 6 describes the refactoring process in a SADT-like notation [16].).

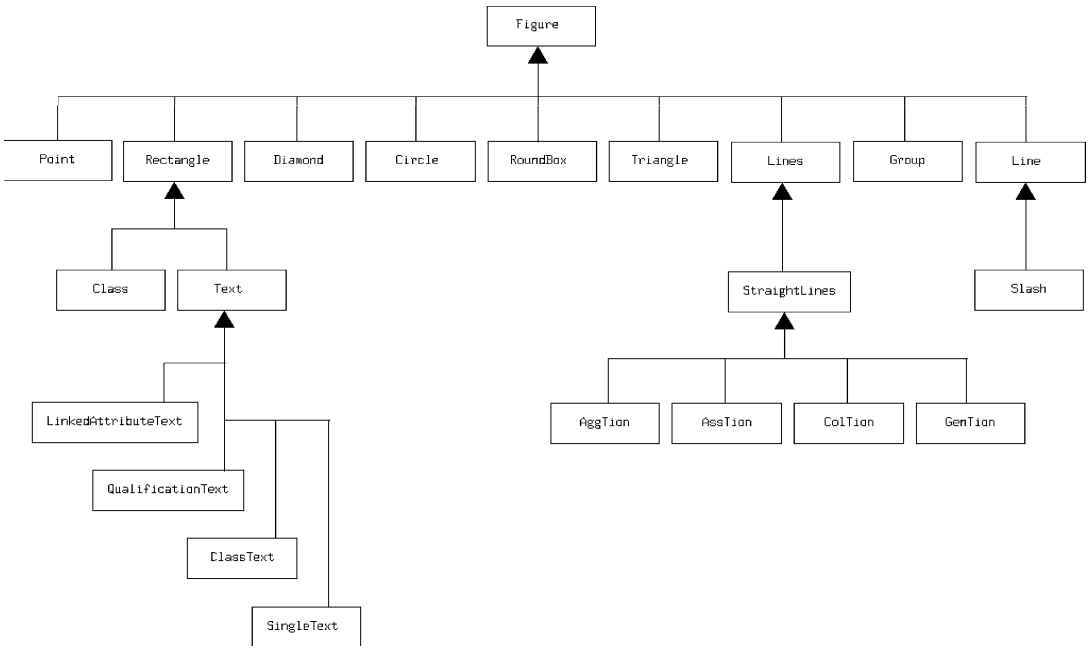

Figure 4: Object Model for OMT Notations in Version 1.x

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. (The design of the new class diagram starting from the reversed one has been done manually, as opposed to some automatic approaches like those depicted in [17]).

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 proper refactoring, great productivity was reached while refactoring 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 considerably reduces trivial and redundant work, we believe that the use of supporting tools is one of the major factor for conducting successful OO projects. Further, since no "catastrophic" crashes occurred during the refactoring phase, we could also ensure the reliability and the available functionalities of the tool thanks to this experience.

3.4 Refactoring Benefits and Lessons

3.4.1 Benefits from Refactoring

Due to the refactoring efforts, software maintenance revealed easier. The convenience of maintenance results from: the reduction of duplicated code, the proper encapsulation, the simple control thanks to dynamic binding and the robustness of the new version. We feel that the major benefits of the refactoring is that the tool became:

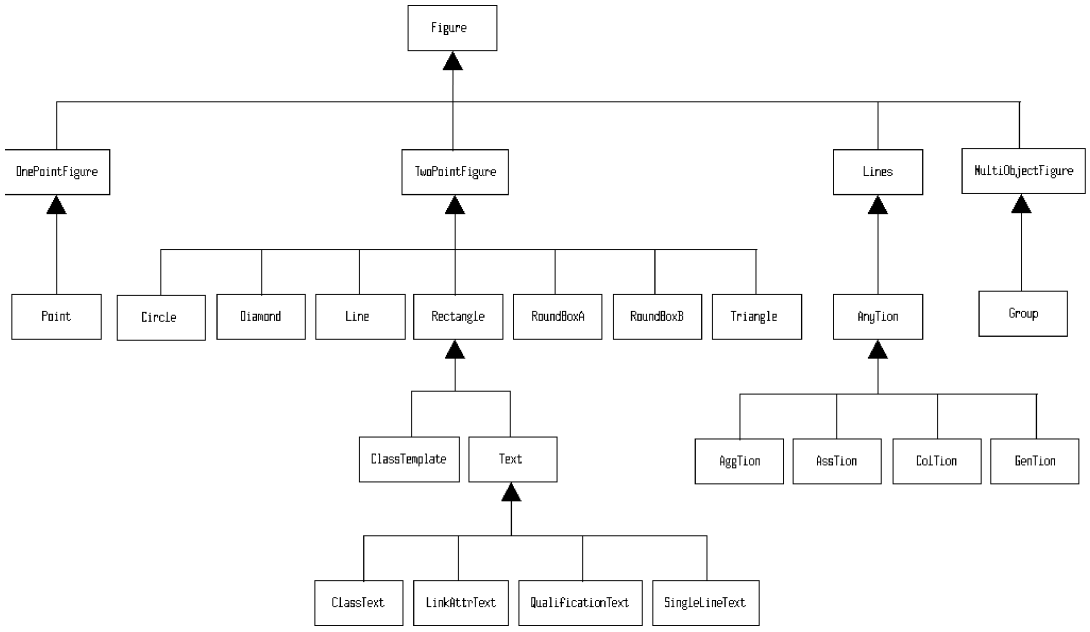

Figure 5: Object Model for OMT Notations in Version 2.x

- *Easier to modify*: We tried to change the data format of diagram several times to increase efficiency and machine independency. We usually spent one day to change data format when we choose another candidate data format.

- *Easier to enhance*: We could enhance it with the functional and the dynamic modellers within one month instead of the 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 facilities for Java.

- *Easier to understand*: We intermittently stopped enhancing the software. That means we sometimes stopped developing OODesigner for some duration. In the worst situation, we were back to its development after six months and we always could quickly re-start the enhancement.

- *Easier to port to other platforms*: After the refactoring process was done successfully, we tried to port the system from a Unix version to a PC version using Visual C++. We could make 53 classes with about 20,000 source lines within one month. We are also tried to produce a Java version concurrently. Again, we also spent one month to make 57 classes with about 20,000 source lines for Java version. That does not mean that we implemented the source code from scratch, but it means that we translated the Unix version into the other versions.

- *More flexible*: 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 thanks to intensive use of dynamic binding.

- *Machine independent*: Because the global functions which are given from libraries are confined to some locality, we feel that it will not be so difficult to adapt the tool to other target

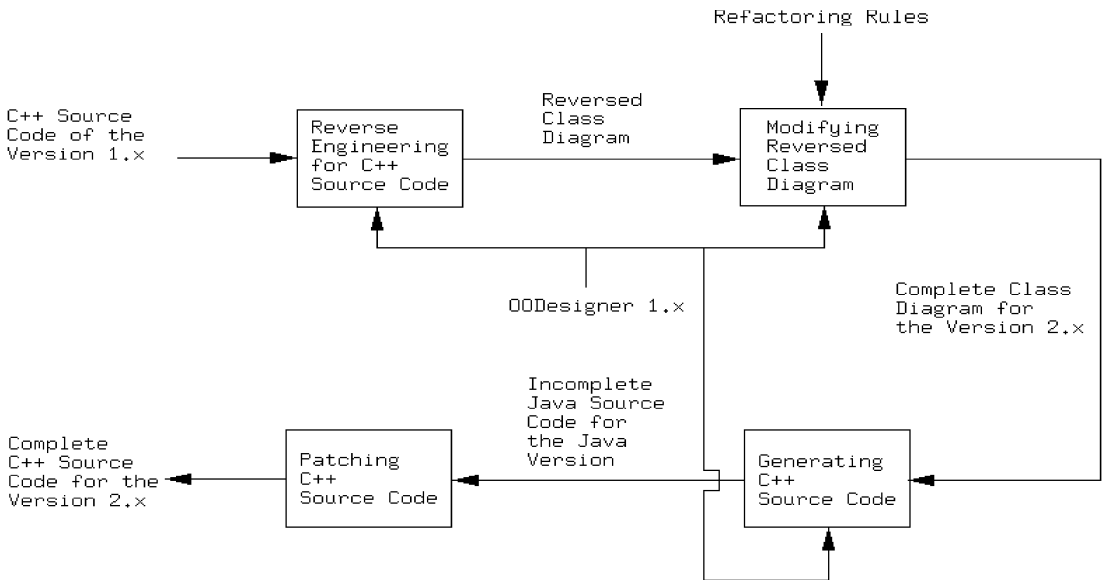

Figure 6: The Refactoring Process

machines.

- *More stable and reliable*: We were used to spend one day to find bugs and to fix them when any failure occurred in the version 1.x. In the worst case, we spent 4 days to find a bug which was due to a dangling reference. With the version 2.x, we usually spend a few hours to fix bugs.

- *And finally easier to maintain*: The above-described experience makes us feel that the maintenance of the resulting product is easier than we expected before refactoring.

3.4.2 Metrics and Learned Lessons

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 [20]. But if metrics are too cumbersome and time consuming to collect, collecting metrics is likely to be neglected [15]. 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's (abbreviated into C&K) OO metrics suite, even though some authors [3, 9] criticized the incompleteness of that suite. Some other metrics were gathered according to *subjective viewpoints* without relying on any theoretical basis [5].

The metrics data collected in table 1 provides brief comparisons between ill-structured OO software and well-structured OO software. We do not believe that the absolute values in this table have great significance but we believe that the relative ratio of the metrics gives some insights for making good OO software. This table compares the metrics between the two versions of OODesigner. From the values provided by that table we can state, as it is more and more admitted, that *"The smaller the classes are, the better the design is"*.

In table 1, the big difference between the number of classes in the two versions is due to the fact that all the graphical UI components of Motif were developed as classes. The big increment in

Metrics Items	Version 1.x	Version 2.x	Increment Ratio
01. Total number of lines in the system	58925	59562	+1%
02. Total number of semicolons in the system	33245	30535	-8%
03. Total number of conditional statements	6382	5622	-12%
04. Total number of loops	1261	1091	-13%
05. Total number of classes	97	155	+60%
06. Total number of semicolons in classes	25692	25916	+1%
07. Average number of semicolons per class	265	167	-37%
08. Total number of unique words in classes	2635	3561	+35%
09. 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%
23. Total number of violation of Demeter's Law	485	891	+83%

Table 1: Metrics comparisons between the two versions

the number of classes makes some metrics values of no importance. In other words, several cases of total values like the values for the metrics 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 used to evaluate the empirical data, but the corresponding average values per class are used instead. Some points of interest with the empirical metrics can be listed as follows.

Items 1 and 2: These values informs that the size of the new version is almost the same as the size of the old version. But that does not mean that the architectures are very similar. The new version is better documented and the increment of the number of lines to employ new classes makes up for the decrement of the number of lines to delete global functions and data structures in the old version.

Items 3 and 4: These values express the fact that the general complexity of the system is reduced. The reduction of the complexity was gained by the elimination of duplicated code and the intensive use of dynamic invocation.

Items 5, 7, 9: These values provide a good guideline for OO beginners. The guideline may be: *"make classes as small as possible"*. We do not think that there exists an absolute value for the optimal size of classes, but making classes smaller yields more reusable classes. 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, because a smaller set of vocabulary in the implementation

provides a simpler way of thinking.

Items 11, 13, 14: These values relate to members of the corresponding class. Specifically, the item 13 tightly relates to C&K's *Weighted Methods per Class (WMC)* with unit complexity. In C&K's viewpoints, classes with large numbers of methods are likely to be more application specific. 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 thence more reusable.

Items 15, 16, 17: These values relate to C&K's *Depth of Inheritance Tree (DIT)*. As C&K already noted in their viewpoints, DIT has contradictory characters. A deeper DIT may result in a more complex system but with greater reusability. In our empirical data, DIT seems to increase in the new version, and that means that the software is more reusable.

Items 18, 19, 20: These values relate to C&K's *Number of Children (NOC)*. NOC has also contradictory features with respect to reusability and miss-use of subclassing. The increased NOC in the new version also means that we tried to make the software more reusable sacrifying the other feature.

Item 21, 22 relate to C&K's *Coupling between Object Classes (COB)*. These values were simply gathered as "a count of the number of other classes to which a class is coupled". The result shows that we have a 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 noticed by Hitz [9]. If the latter guess is right, the way of measurement for coupling should be improved by considering, for example, some additional aspects as those listed in [9].

We did not gather data metrics for C&K's *Response For a Class (RFC)* and *Lack of Cohesion in Methods (LCOM)*. The reason is due to our unability to measure them but also to the controversy about the way of measuring them.

Item 23: Finally we counted the number of violations with the Law of Demeter [13, 12]. It is "very embarrassing" to find that the new version violates the law more than the old version. Because the law is generally accepted as a good OO programming style, we were disappointed when we found that our empirical results somewhat contradict the law. Writing programs that follow the law decreases the occurrences of nested message sending and decreases the complexity of the methods, but it increases the number of methods. Because we usually have large numbers of methods for classes, we tried to keep from increasing the number of methods. Sometimes we also felt that the readability of code is better when violating the law. For example, the method invocation like **"currentLines->focus()->whoAreYou()"** is more readable than **"currentLines->focus_s_whoAreYou()"**.

The comparisons specified in table 1 may not clearly show the inherent differences between the ill-designed and the well-designed OO software. In order to make the comparison clearer, we provide another set of metrics. Table 2 shows other metrics comparisons between some class clusters of the two versions. The class models in figure 4 and figure 5 contain the same design information to support OMT notations. Thus the classes in figure 4 have exactly the same functionality as the classes in figure 5. So table 2 shows the real differences between the two sets of classes in the two versions.

Metrics	Figure 4	Figure 5	Increment Ratio
01. Total number of conditional statements	3688	2499	-12%
02. Total number of loops	771	517	-33%
03. Total number of classes	22	25	+14%
04. Total number of semicolons in classes	18805	12226	-35%
05. Average number of semicolons per class	855	489	-43%
06. Total number of unique words in classes	1707	1549	-9%
07. Average number of unique words per class	78	62	-21%
08. Total number of data members in classes	346	196	-43%
09. Average number of data members per class	15.7	7.8	-50%
10. Total number of methods in classes	1124	958	-15%
11. Average number of methods per class	51	38	-24%
12. Average number of semicolons per method	17	13	-24%
13. Maximum inheritance depth	3	4	+33%
14. Total number of inheritance depth	41	58	+41%
15. Average number of inheritance depth	1.86	2.32	+25%
16. Maximum number of children	9	7	-22%
17. Total number of children	21	24	+14%
18. Average number of children	0.95	0.96	+1%
19. Total number of coupling between object classes	233	249	+79%
20. Average number of coupling between object classes	10.6	9.96	-6%

Table 2: Metrics comparisons between classes

We finally learned the following lessons during this project.

1. It is almost unavoidable for novices in OO technology 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. Ill-designed OO software is worse than well-designed software by structured techniques.

2. An OO project could be successfully conducted just in the case of applying OO methodology, OO language and OO CASE tool synergistically. Using an OO programming language alone without any methodology to build OO software should bring fake OO software. Moreover, this experience shows the power of the object technology, especially in the refactoring process where the system is refactored using itself.

3. 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.

4. Ill-designed OO software makes maintenance activity "terrible", but well-designed OO software makes it enjoyable. This fact says the importance of OO modelling and design.

4 Concluding remarks

In this paper, we presented the experience acquired while developing and enhancing an OO CASE tool named OODesigner. We discussed: the drawbacks of the initial version of the tool, the goals

of the refactoring process and the refactoring decisions and process. We also presented the benefits of the refactoring, some 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.

As already stated, the version 2.x is currently being ported from a Unix version to a PC version using Visual C++. A Java version is also being developed concurrently. This work is productively being done thanks to the flexibility and the machine-independence structure. As for refactoring, OODesigner is used as a CASE tool to support the porting activities. For example, while porting to a Java version, we reversed C++ code of OODesigner, and then we generated Java source code using OODesigner itself (Figure 7 shows, in an SADT-like notation, the adopted process to produce the Java version.). Nevertheless, we had to temporarily customize the tool for that work.

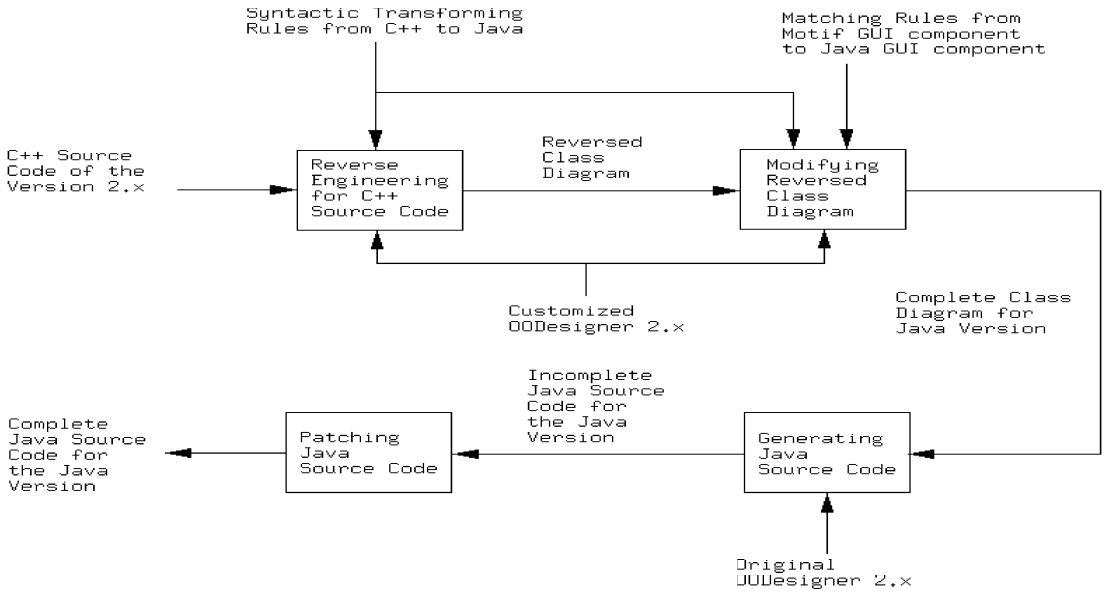

Figure 7: From C++ version to Java version : the Process

Using the new version of OODesigner, we are currently enhancing it with some additional functions and we are porting it to other platforms. Furthermore, as for the first version refactoring, we use OODesigner itself as a CASE tool to enhance it. The ultimate goal of our work is a full implementation of an OO design environment based on graphical tools. We plan to extend this tool so that it can support all the models of the Unified Modelling Language (UML) [8]. We also plan to make it usable in a distributed environment and refactor its architecture into a client/server architecture. A final mid-term objective is to make it support an OO design process [6] for OO Process modelling [19].

References

[1] B.W. Boehm. A Spiral Model of Software Development and Enhancement. *Software Engineering Notes*, 11(4):22–42, August 1986.

[2] G. Booch. *Object Oriented Design with Applications*. The Benjamin/Cummings Publishing Company, Inc., 1991.

[3] N.I. Churcher and M.J. Shepperd. A Metrics for Object-Oriented Design. *IEEE Transactions on Software Engineering*, 21(3):263–265, March 1996.

[4] D. de Champeaux, A. Anderson, and E. Feldson. Case Study of Object-Oriented Software Development. In *Proceedings of OOPSLA'92*, pages 377–391. ACM, 1992.

[5] N. Fenton. Software Measurement: A Necessary Scientific Basis. *IEEE Transactions on Software Engineering*, 20(3):199–206, March 1994.

[6] A. Finkelstein, J. Kramer, and B.A. Nuseibeh, editors. *Software Process Modeling and Technology*. Advanced Software Development Series. Research Studies Press, Taunton, Somerset England, 1994.

[7] R.G. Fishman and C.F. Kemerer. Object-Oriented Conventional Analysis and Design Methodologies. *IEEE Computer*, 25(10):22–40, 1992.

[8] M. Fowler and K. Scott. *UML Distilled: Applying the Standard Object Modeling Language*. Object Technology Series. Addison-Wesley, 1997. (http://www.awl.com/cseng/otseries/).

[9] M. Hitz and B. Montazeri. Chidamber and Kemerer's Metrics Suite: A Measurement Theory Perspective. *IEEE Transactions on Software Engineering*, 22(4):267–270, April 1996.

[10] S. Honiden and al. An Application of Artificial intelligence to Object-Oriented Performance Design for Real-Time Systems. *IEEE Trans. on Soft. Eng.*, 20(11):849–867, 1994.

[11] D. Inga, T. Kaehler, J. Maloney, S. Wallace, and A. Kay. Back to the Future: the Story of Squeak, a Practical Smalltalk Written in Itself. In *Proceedings of OOPSLA'97*, pages 318–326, GA, USA, October 1997. ACM.

[12] K. Lieberherr and I. Holland. Assuring Good Style for Object-Oriented Programs. *IEEE Software*, pages 38–48, September 1989.

[13] K. Lieberherr, I. Holland, and A. Riel. Object-Oriented Programming: An Objective Sense of Style. In *Proceedings of OOPSLA'88*, pages 323–334, 1988.

[14] P.H. Loy. A Comparison of Object-Oriented and Structured Development Methods. *ACM Sigsoft SE Notes*, 15(1):44–48, 1990.

[15] R. Malan, D. Coleman, and R. Letsinger. Lessons from the Experiences of Leading-Edge Object Technology Projects in Hewlett-Packard. In *Proceedings of OOPSLA'95*, pages 33–46, Austin, TX, USA, 1995. ACM.

[16] D. Marca and C. Mc Gowan. *SADT, Structured Analysis and Design Technics*. Mc Graw Hill, 1987.

[17] I. Moore. Automatic Inheritance Hierarchy Restructuring and Method Refactoring. In *Proceedings of OOPSLA'96*, pages 235–250, CA, USA, 1996. ACM.

[18] W.F. Opdyke. *Refactoring Object-Oriented Frameworks*. PhD thesis, University of Illinois at Urbana-Champain, 1992. (ftp://st.cs.uiuc.edu/pub/papers/refactoring/opdyke-thesis.ps.Z).

[19] O. Perrin and N. Boudjlida. Object-Oriented Concepts for Modelling Software Processes: an Initial Proposal. (Deliverable) OOCDPM/CRIN/T3/D/1.0, LORIA (F) and ETRI (KR), UHP Nancy 1 (F), October 1996.

[20] M.W. Price and Sr S.A. Demurjian. Analyzing and Measuring Reusability in Object-Oriented Designs. In *Proceedings of OOPSLA'97*, pages 22–33, GA, USA, October 1997. ACM.

[21] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorenzen. *Object-Oriented Modeling and Design*. Prentice Hall, 1991.

[22] J.G. Wolff. Towards a New Concept of Software. *Software Engineering Journal*, 9(1):27–38, 1994.