Similarity in the Design and the Implementation of a Multi-Platform CASE Tool[1]

Taegyun Kim [1] , Gyusang Shin [2] , Nacer Boudjlida [3]

[1] *Pusan University of Foreign Studies, Department of Computer Engineering*
55-1, Uam Dong, Nam Gu, 608-738, Pusan, Korea
Tel: +82 51 640 3178, Fax: +82 51 640 3038, Email: ktg@taejo.pufs.ac.kr
[2] *Real-Time Computing Department, ETRI-CSTL*
161, Kajong Dong, Yusong Gu, Taejon, Korea
Tel: +82 42 860 6566, Email: gsshin@etri.re.kr
[3] *UHP Nancy 1, LORIA UMR 7503, BP 239*
54506 – Vandoeuvre Les Nancy Cedex, France
Tel: +33 3 83 59 30 76, Fax: +33 3 83 41 30 79, Email: nacer@loria.fr

Abstract

This paper presents similarity in the design and the implementation of a Computer Aided Software Engineering (CASE) tool on three platforms. OODesigner is a tool that was initially developed to support Object Modeling Technique (OMT). An initial Unix version has been developed since 1994. In 1997, after the completion of the Unix version, we began developing a Java version and a Windows version to support Unified Modeling Language (UML). The development of a CASE tool is a typical application of the Model-View-Controller (MVC) paradigm. Thus, we obtained a common design pattern among the versions in the MVC point of views. This design similarity can be used to develop several kinds of CASE tools with the corresponding design notations. In this paper, we present the process we followed to develop the three versions and we discuss the similarity found among them. We also outline a kind of generic architecture for the design and the implementation of CASE tools.

Keywords: Design Pattern, Object Modeling Technique, Unified Modeling Language, Computer Aided Software Engineering Tool, Model-View-Controller Paradigm

1. Introduction

Although there may be as many different views of what the object-oriented (OO) paradigm should be as there are computer scientists and programmers, it is more and more commonly admitted that the paradigm substantially contributes to overcoming the software crisis. Research and development projects, in the past decade, has shown that OO techniques are a more natural way of problem solving than structured techniques[7,11]. Software engineers can take full advantage of principles like abstraction, encapsulation, modularity, hierarchy, typing, concurrency, and persistency in a synergistic way[2]. Moreover, the long term controversial problem of software reuse can be solved by means of OO concepts and mechanisms. Since its emergence, around two decades ago, the OO technique has gained tremendous popularity and great momentum.

From a software engineering point of view, major research areas in OO technology concerned the following issues:

- OO programming languages such as Smalltalk, Objective-C, Eiffel, C++ and Java,
- OO analysis and design methodology like Booch's OOD, Wasserman's OOSD, Coad's OOA, Beck and Cunningham's CRC, Wirfs-Brock's RDD, Rumbaugh's OMT and Rational Rose's UML[3,17,5,1,18,14,8],
- Component-based distributed systems with CORBA and OLE/DCOM technology[13,4,16],
- Automated software engineering tool, variously known as Computer Aided Software Engineering (CASE), Integrated Project Support Environment (IPSE), Software Engineering Environment (SEE) and meta-CASE tools[6].

As CASE tools may importantly contribute in productivity improvement, many vendors nowadays supply OO CASE tools to support the above-mentioned

[1] In proceedings of *Proceedings of COnstructing Software Engineering Tools, COSET'99, ICSE'99 Workshop.* Pages 137-146. Los Angeles, CA, May 1999.

methods. Since 1994, we also have been developing an OO CASE tool named OODesigner on a Unix platform. The initial goals of OODesigner included drawing facilities for the three models in OMT, documenting class resources, automatically generating C++ code, reverse engineering C++ code, storing, searching and reusing classes in a repository and collecting metrics data.

In 1996, the development of the Unix OODesigner version was achieved. But we had to totally restructure it due to encountered maintenance problems. After one man/year successful restructuring effort, we got a version that is easier to enhance and to port to other platforms[9,10]. As a case study for porting to other platforms, we tried to port the Unix version into a Java application. We also tried to port the Unix version to a Windows version using Visual C++. While performing the platform migration experiments, we identified a kind of general or common OO CASE tool design and implementation pattern. In other words, OO CASE tools may not only have a common design architecture, as a typical application of Model-View-Controller (MVC) paradigm[15], but they also have implementation similarity as they are written in an OO programming language. In this paper, we present the similarity across the three versions of OODesigner. We also suggest the general architecture of OO CASE tools.

This paper is organized as follows. In section 2, we briefly review the history of the three versions of OODesigner. The history includes the presentation of the goals of each version, the restructuring and the porting processes. In section 3, we introduce case examples to show the similarity among the three versions. The case examples are separately presented with respect to MVC components, and with respect to the design and the implementation perspectives. In section 4, we present some suggestions for developing OO CASE tools in general. These suggestions are presented from an implementation perspective and from a design perspective. Finally we conclude in section 5 by outlining our work and further research topics.

2. OODesigner Historical Process

In this section, we provide historical information about the three versions of OODesigner including its goals and its development process. While the product goals of OODesigner development were almost the same for the three versions, the process goals were slightly different. In the following, process goals and product goals are presented for every version.

2.1 The Unix Version History

The initial development of OODesigner started in 1994 with the objective to provide an environment to support Rumbaugh's OMT. At the early stage of the project, two types of goals were fixed: *process goals and product goals*. The *product goals* were functional requirements that included the following:

- Support the design of the three models of OMT: object, functional and dynamic models,
- Provide facilities to document the class resources,
- Maintain a repository for the designed object models,
- Generate C++ code,
- Reverse engineer class diagrams from C++ code,
- Store and retrieve class definitions for reuse,
- Collect metrics data for C++ program.

Being OO novices, at the beginning of the project, we also fixed for ourselves some *process goals* like:

- Improve our ability to conduct OO design and implementation activities,
- Practice an iterative development process,
- Apply OODesigner as a CASE tool for developing OODesigner itself,
- Ensure maintainability for further product enhancements and platform migration.

In 1996, the Unix version 1.x was released into the public domain for free use. Since its first release, *Asset Source for Software Engineering Technology (ASSET)* and *ftp://203.230.73.24/pub/OOD* sites are used as major distribution routes. The first version was built on a Sun workstation running OS-4.1.x, X11-R5, Motif-1.2 and C++-2.0. This version was implemented with 60 thousands of C++ source statements. Figure 1 shows a typical screen session of the Unix version. We found that the first version successfully satisfied all the product goals, but it did not meet the process goals, especially with respect to maintenance issues. In other words, the first version worked correctly for the given requirements, but its class structure was so badly designed that it was quite difficult to enhance the tool with additional functionalities. The failure in the process goal satisfaction is mainly due, in our opinion, to the lack of OO technique mastery.

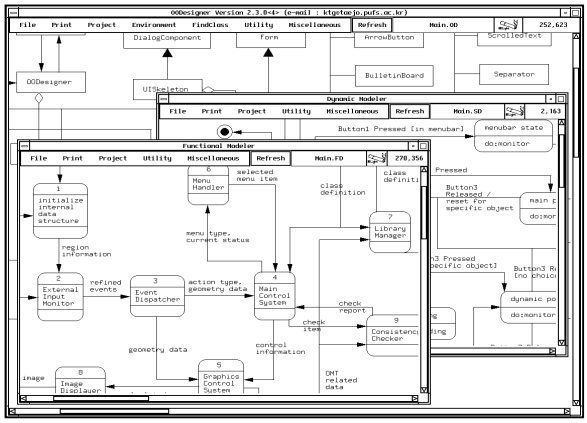


Figure 1. Typical screen session of the Unix version

As the project had no deadline constraint and as it had educational purposes, we decided to totally restructure the produced tool (See [9,10] for details about the

restructuring process). After restructuring, the tool became:

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

Thanks to these gained benefits, we were able to conduct a case study of porting issue.

2.2 The Java Version History

The restructuring activity ended in 1997, and at that time Java and UML were gaining tremendous popularity. Thus we decided to port the Unix version to a Java platform as a case study. We established two types of goals for the Java version. As a *product goal*, we decided to support UML rather than OMT. We also established the following *process goals:*

- Learn about Java,
- Apply OODesigner as a CASE tool for platform migration,
- Try to discover commonalities between C++ and Java programming.

The Unix version was already fitted with the major functionalities like C++ reverse engineering and Java code generation: these functionalities have been used to port the tool from C++ to Java. So, the porting process was the following (Figure 2 describes the porting process in an SADT-like notation[12]).

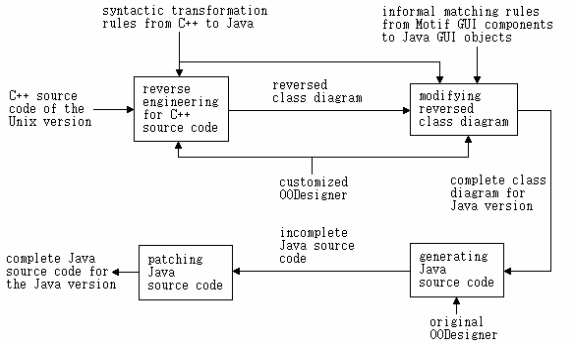


Figure 2. Porting process for the Java version

i) Reverse the source code of the Unix version, thanks to the Unix version itself: this step resulted in the class diagrams for the Unix version. Moreover, since we customized the reverse engineering facility of OODesigner to enable simple syntactic transformation from C++ to Java, we did not have to perform syntactic conversion from C++ to Java.

ii) Modify the class diagrams generated by the reverse procedure using the Unix version itself; i.e. draw new class diagrams for the Java version using the Unix version. Especially, in this step, we converted Motif GUI components to the corresponding Java GUI objects.

iii) Generate the Java code for the modified class diagrams using Java code generation facility of OODesigner.

iv) Manually modify the generated source files whenever a problem is encountered during the implementation phase.

An important feature of the porting process worth noting is that the production of the Java version was supported by the Unix version, so increasing our productivity. For instance, at the early stage of the porting process, we could produce 53 classes with about 20,000 source lines within one man/month. Figure 3 shows a typical screen session of the Java version that is currently under development.

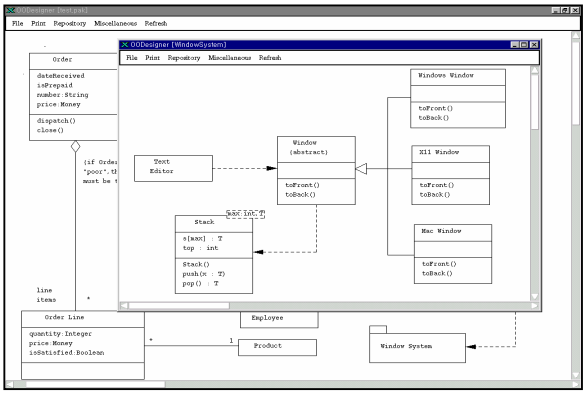


Figure 3. The Java version of OODesigner

2.3 The Windows Version History

While the Unix version was being ported into a Java version, a PC version of OODesigner was concurrently being developed using Visual C++. As Windows 95 gained worldwide popularity, we received many requests from users of OODesigner to provide a Windows version. Thus we started to develop it from late 1997. We established the product goal to make a CASE tool for UML, and we established the process goals as follows:

- Learn how to use Visual C++ development platform. Especially we wanted to find the compatibility between Motif GUI and Windows GUI.
- Apply OODesigner as a CASE tool for platform migration as it was the case in the Java porting process. In this case, we tried to customize OODesigner to transform some identifiers in the Unix version into corresponding names in the Windows version.
- Identify platform-independent classes and reuse classes from the Unix version as much as possible.

While translating the Unix version to the Windows one, we noticed that the object models for the model component in the MVC perspectives could be totally reused in the Windows version. Although the view component and the controller component of OODesigner

are slightly different from those in the Motif and the Visual C++ development platforms, we tried to find common control mechanisms and to translate the Unix version to the Windows one as automatically as possible. Thus we gained productivity while building the Windows version. For instance, we could produce 57 classes with about 20,000 source lines within one man/month. Figure 4 shows the Windows version currently under development.

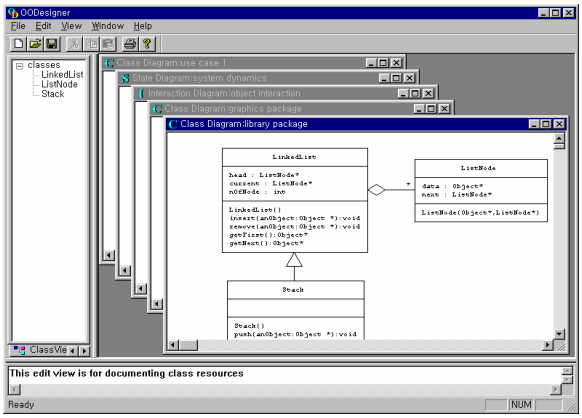


Figure 4. The Windows version of OODesigner

3. Versions Similarity

The MVC paradigm divides an interactive application into three components. The *Model Component* encapsulates the core functionality and data. The *View component* displays a model's data to the user. And the *Controller Component* handles GUI events and communicates them to the model. Since all the CASE tools have view components to show diagrams and to provide control mechanisms to edit diagrams with a model for the corresponding notation, the development of CASE tools can be categorized as a typical application of the MVC paradigm. We recognized some design and implementation similarity among the three versions during the porting processes. In this section, we discuss some case examples of the similarity with respect to the MVC components.

3.1 The Model Component Point of View

- *From the Design Perspective*

It is clear that the model component of OODesigner had to be platform independent. It consists of object models to represent OMT or UML notations and it is totally reused in the three versions. Figure 5 shows the class diagram to model the notations of the class diagram. The concrete classes in this figure serve as classes for representing the corresponding OMT notations. For example, *GenTion*, *AggTion*, *AssTion* and *ColTion* classes on the right bottom of this figure are defined to represent the notations of generalization, aggregation, association and collaboration respectively. This model can be used for making three versions without modification.

- *From the Implementation Perspective*

We found a very strong implementation similarity of the model component between C++ and Java. Then, as for the design, we could totally reuse C++ source code of the model component to implement the Java version. As already stated in section 2.2, we gained great productivity for implementing the Java version because we could automatically translate C++ syntax into Java using the customized OODesigner. Figure 6 shows a case example of an implementation similarity between the C++ and the Java versions.

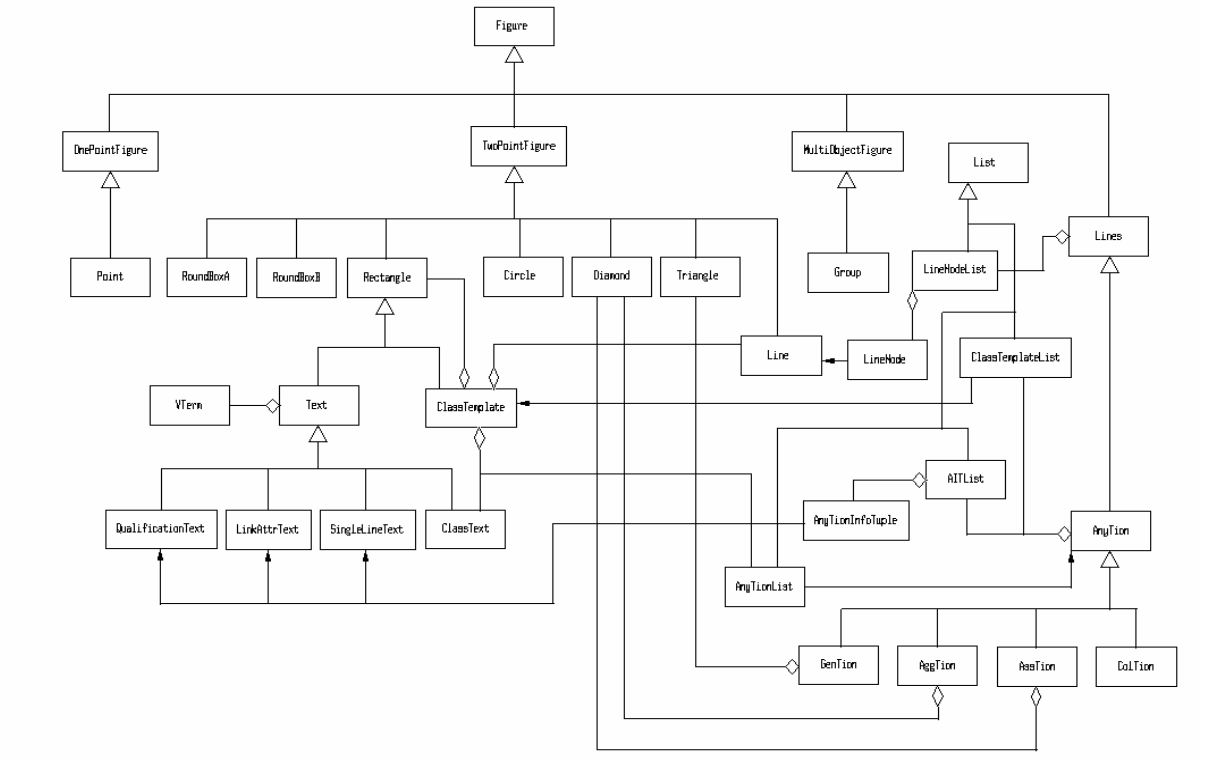


Figure 5. An example of model component: a class diagram for modeling class notations

```
// an example of C++ member function within
// ClassTemplate class
int ClassTemplate::minimized_width(int& maxchar)
{
     _classname->uncentering(TRUE);
     if (_NofCharsinLine>_classname->width()) {
          maxchar = _NofCharsinLine;
     } else {
          maxchar = _classname->width();
     }
     return (_deltaH*maxchar+2*_GapX+_deltaH);
}

// the correponding Java member function
public final class ClassTemplate
extends Box implements ClassLike {
  …
  private int minimized_width(IntVar maxchar) {
     _classname.uncentering(true);
     if (_NofCharsinLine>_classname.width()) {
          maxchar.v = _NofCharsinLine;
     } else {
          maxchar.v = _classname.width();
     }
     return(_deltaH*maxchar.v+2*_GapX+_deltaH);
  }
  …
}
```

3.2 The View Component Point of View

Figure 6. Example of an implementation similarity between C++ and Java source code

- *From the Design Perspective*

The view component of OODesigner has the responsibility to draw a model's data to a screen. Objects related to the view component consist of the corresponding event handlers, graphics context, and fonts. As these objects are provided by the corresponding platforms as built-in facilities, and as functions of the view component are scattered within the controller component, we did not have to make specific object models for the view component. Because the three platforms provide the same functionality with slightly different names, we only had to find the corresponding names of the built-in objects to port from Unix version to the other ones. Table 1 shows the corresponding objects for the view component across the three platforms.

Although there was a small difference in managing attributes of the graphic objects, we could reuse the classes for the view component of the Unix version by converting the corresponding class names and member function names.

Table 1. Objects cross-platforms correspondence

Functionality	X Toolkit	Java AWT	MFC
graphics	GC	Graphics	CDC
font	Font	Font	CFont
clipping	Region	Shape	CRgn
event handler for repainting screen	user defined event handler	paint() in Canvas class	OnDraw() in CView class

- *From the Implementation Perspective*

As for the case of the model component, we could reuse C++ source code of the Unix version to make the other versions. We had to make minor changes to some names for the other platforms. But the already existing algorithm and the overall architecture for the view component remains unchanged. Figure 7 shows an example of source code of the Windows version and the Java version to manage screen repainting.

```
// the member function in the Windows version to repaint screen
void COODView::OnDraw(CDC* pDC)
{
     COODDoc* pDoc = GetDocument();
     ASSERT_VALID(pDoc);
     CPoint sp = GetScrollPosition();
     if (sp.x != _originX || sp.y != _originY) {
          _originX = sp.x; _originY = sp.y;
          if (_currentFocus != NIL) {
               _currentFocus->make_region();
               _currentFocus->reset_clip_area();
          }
          Figure *all = _figures->get_first();
          while (all != NIL) {
               all->set_in_canvas(all->
                                  check_in_region(_canvasRgn));
               if (all->in_canvas()) {
                    all->make_region();
               } else {
                    all->reset_region();
               }
               all = _figures->get_next();
          }
          recalc_clip_area();
     }
     CPen *oldPen = pDC->SelectObject(_plainPen);
     CBrush *oldBrush = pDC->SelectObject(_plainBrush);
     Figure *ptr = _figures->get_first();
     while(ptr != NIL) {
          ptr->draw(pDC);
          ptr = _figures->get_next();
     }
     if (_currentFocus != NIL) {
          if (_editingTag) {
               _currentFocus->draw(pDC);
          } else {
               if (_doNotRubberband) {
                    highlight(_currentFocus);
               } else {
                    rubberbanding(_currentFocus);
               }
          }
     }
     pDC->SelectObject(oldPen);
     pDC->SelectObject(oldBrush);
     _doNotRubberband = FALSE;
}
// the corresponding Java member function
class GraphicController extends Canvas
implements ActionListener, AdjustmentListener{
  …
  public void paint(Graphics g) {
     int spx = hscrollBar.getValue();
     int spy = vscrollBar.getValue();
     if (spx != _originX || spy != _originY) {
          _originX = spx; _originY = spy;
          if (_currentFocus != null) {
```

```
            _currentFocus.make_region();
            _currentFocus.reset_clip_area();
        }
        Figure all = _figures.get_first();
        while (all != null) {
            all.set_in_canvas(all.check_in_region(_canvasRgn));
            if (all.in_canvas()) {
                all.make_region();
            } else {
                all.reset_region();
            }
            all = _figures.get_next();
        }
        recalc_clip_area();
    }
    Figure ptr = _figures.get_first();
    while(ptr != null) {
        ptr.draw(g);
        ptr = _figures.get_next();
    }
    if (_currentFocus != null) {
        if (_editingTag) {
            _currentFocus.draw(g);
            showCaret();
        } else if (_popupflag == true) {
            draw_dots(_currentFocus);
            _popupflag = false;
        } else {
            rubberbanding(_currentFocus);
        }
    }
  }
  …
}
```

Figure 7. A sample source code of Visual C++ and Java to repaint screen

3.3 The Controller Component Point of View

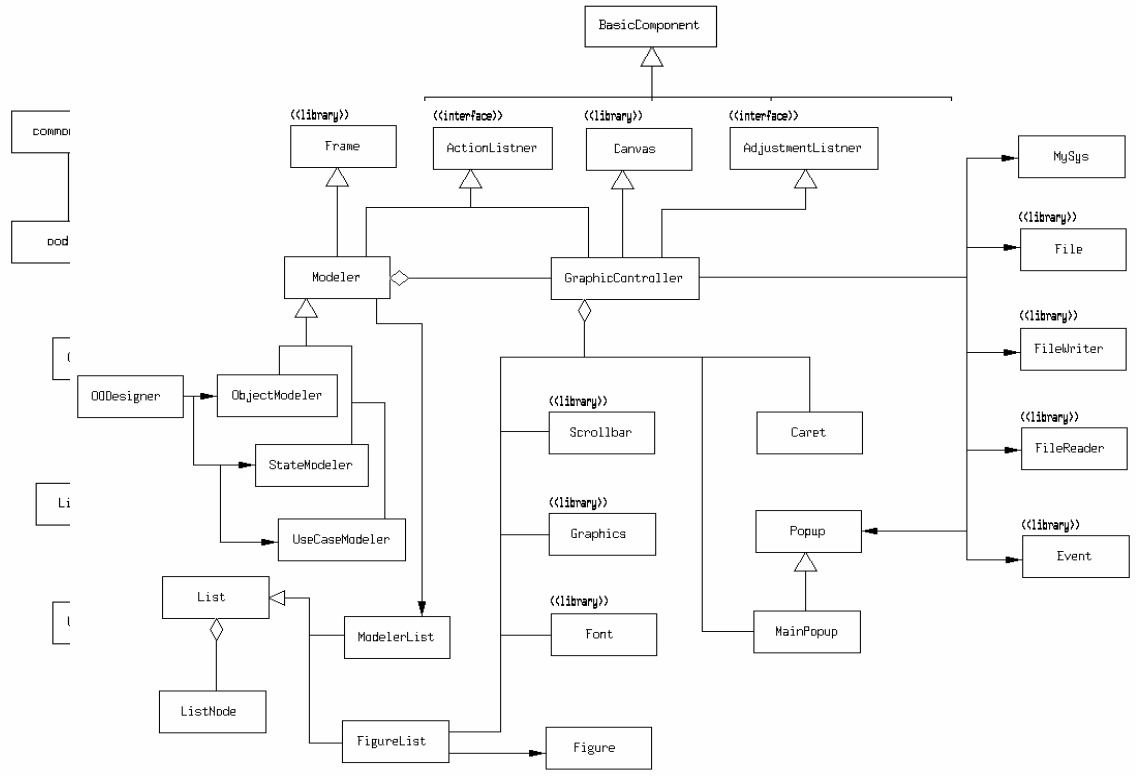
■ *The Design Perspective*

Although the model components and the view components of the three versions are very similar, their respective controller components are slightly different. Therefore the reuse rate for the controller components is relatively small as compared to the other two components. The controller compo
with the physical in
into messages to
differences among th

- *The Unix vers*
function. The
application o
members and
had to enca
corresponding
Figure 8 show
component o
mechanism to
can be effi
unregistering
mechanism t
handlers if n

programming as compared to Java and Visual C++ programming. But the major drawback of Motif programming is that we have to declare the event handlers and callback functions as static members within the corresponding classes. Thus, event handlers always find out the proper object to receive the events that occurs, and they have to delegate their event-handling role to the proper member functions. This mechanism results in increasing the number of member functions.

- *The Java version*: In this version, we directly use Java AWT classes to provide the application frames. The built-in classes reduce the GUI component construction effort. Figure 9 shows the object model for the controller component of the Java version. In this figure, the stereotypes <<library>> and <<interface>> annotations means that the annotated classes are built-in classes and built-in interfaces respectively. Figure 8 and figure 9 show that the design of the classes is similar in the two versions even though their inheritance hierarchies are different. The good point of the event handling mechanism in Java programming is that we do not have to explicitly identify the object that receives the corresponding events. In other words, the built-in member functions that implement the proper interface are not declared as static members, in contrast to Motif programming. Thus, additional member functions are not needed to actually deal with the events. On the contrary, unlike Motif programming the usability to register and to unregister event handlers brings some inconvenience: flag variables have to be used to deal with the dynamic event handlers. The use of these flags makes the system more complicated.
- *The Windows version*: The merits and the demerits of the Windows version are very similar to those of the Java version. Microsoft Foundation Class (MFC) library can be used to construct the



■ *From The Implementation Perspective*

Unlike the case of the model component and the view component, we could not totally reuse C++ source code of the Unix version to produce the other versions: to deal with the event handlers, many changes were required in the controller component. We have to introduce more conditional statements in the Java version and the Windows version. Figure 10 shows an example of source code of the Unix and the Windows versions to deal with an event handling.

```
// an example of event handler of the Unix version
void GraphicController::start_draw(
        Widget,XtPointer thisController,XEvent *event,BOOL *)
{
    if (event->xbutton.button != Button1) return;
    if (OODesigner::EditingTag) return;
    GraphicController *controller =
                        (GraphicController*) thisController;
    if (controller->_currentDrawingType == NIL) return;
    controller->_isFixed = FALSE;
    XtRemoveEventHandler(controller->_canvas_widget,
        PointerMotionMask,FALSE,
        (XtEventHandler)&GraphicController::fix_pointer,
        thisController);
    XtRemoveEventHandler(controller->_canvas_widget,
        PointerMotionMask,FALSE,
        (XtEventHandler)&GraphicController::trace_enter,
        thisController);
    controller->local_start_draw(event);
}

// the corresponding Visual C++ member function
void COODView::start_draw(CPoint event)
{
    if (F_start_draw == FALSE) return;
    if (EditingTag) return;
    if (_currentDrawingType == 0) return;
    _isFixed = FALSE;
    F_fix_pointer = FALSE;
    F_trace_enter = FALSE;
    local_start_draw(event);
}
```

Figure 10. A sample source code of C++ and Visual C++ to deal with a button-pressed event

4. Observations

In this section, we provide some recommendations for building OO CASE tools from design and implementation perspectives. The view component in the MVC paradigm is qualified as an independent component of a system. But from our experience, we notice that the view component is tightly coupled with the controller component. And it is almost clear that the view component is only a subset of the controller component. Thus we think that it is better to decompose a system into MC components rather than into MVC components. Figure 11 outlines a package architecture we propose for building CASE tools. At the top level, two sorts of packages serve to build the controller component and the model component. The controller package consists of two types of packages. The *Interactive Controller* package offers the services to deal with the input-devices generated events. As this controller component is the most complicated part in the system, lot of design and implementation effort may be dedicated to its production.

The *Static Controller* package is in charge of processing callbacks by selecting application pulldown menus. As these kinds of services can be processed not interactively but sequentially, we can implement the static controller package more or less concurrently. The functionalities of CASE tools such as loading or saving files, automatically generating or reversing source code are examples of services that are provided by the static controller package.

The model component has also two kinds of sub-components. The *Methodology Specific Object Model* must be constructed to support the notations of the method for which the tool is developed. The object model in figure 5 is a typical example. The *General Purpose Object Model* includes classes like Stack, List, OrderedCollection, and SymbolTable. Some of these may be provided by the specific platforms, otherwise they have to be developed. If these classes are completely developed, they can be very reusable across the different platforms. In our case, we tried to make the general-purpose classes by ourselves as much as possible to easily translate from a platform into the others.

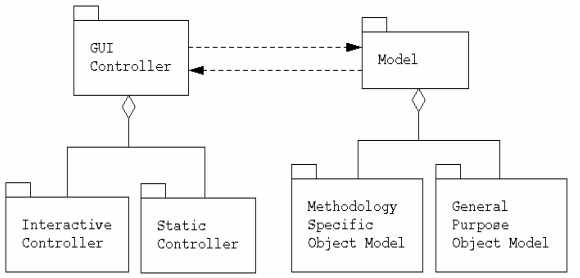


Figure 11. Architecture to build CASE tools

Let us now turn our attention to the implementation perspective. Even though different OO programming languages offer some different syntax structures, their main theme has inherent generality. Thus it is very important to keep the uniformity of the basic features of OO programming while implementing. We believe that the success of this porting study results from the fact that the Unix version became well structured after restructuring process[9,10]. The virtue of the Unix version due to the fact that we tried to:

- Reduce duplication of data members in class clusters and reduce duplication of code in member functions of the same name by dividing member functions properly.
- Let all data members be allocated dynamically and intensively use dynamic binding for method invocation.
- Keep the class size, the number of class members and the method size as small as possible.

- Use inheritance aggressively. But we limited the largest inheritance depth to be less than six.
- Encapsulate Motif widgets with the corresponding callback functions. All graphical user interface components have been implemented as classes.
- Encapsulate global members or library functions in the corresponding class. And we also confined platform dependent library functions to reside in local parts of source code.
- Make destructor more complete to make the software more reliable.
- Use some coding conventions of our own. For example, we decided i) not to use reference type but to use pointer type, ii) not to use template class, iii) not to use nested class.

In our experience, it was especially important to keep class sizes as small as possible to make the software more portable. On the other hand, we found that there are big risks of misunderstanding and misusing some features of a specific OO language or the functionality of class libraries that may be available on a given platform. The chance of misuse sometimes leads the system into serious hazardous states. Thus if we are supposed to be involved in a new project with new circumstances never experienced, we need a long time and a carefully designed learning course to teach the details of implementation issues.

5. Concluding Remarks

In this paper, we described our observation of recent case studies we conducted to develop three versions of an Object-Oriented CASE tool on different platforms. The experience gained and the lessons learned from first, restructuring the initial version of the tool[9,10] and then from porting it to different platforms enabled us to empirically improve the design and the development process together with the mastering of OO technology. As a matter of fact, the process improvement has been facilitated by the reflexive nature of OODesigner since its restructuring as well as the development of the Java and the Windows versions have been supported by OODesigner itself. Further, these studies helped us in identifying commonalities and a kind of common pattern for the development of CASE tools that support a given OO design method. From this case study, we observed that:

- There is a strong similarity in the design and the implementation of a multi-platform CASE tool in MVC point of views.
- It is better to decompose a system into MC components rather than into MVC components when we design architecture of a CASE tool.
- It is very important to keep the uniformity of the basic features of OO programming while we implement an OO software. The virtue of portability could be achieved when we consistently apply uniformity of OO paradigm to OO software construction.

A further topic we intend to address in the near future concerns comparative metrics between the developed versions, as we did for the Unix versions (the one before restructuring and the one after restructuring). An additional topic will deal with process support[6]: we feel that the same kind of process model can be followed either to develop and to enhance the tool, or to use the tool to support the design and the implementation of OO applications.

Acknowledgements

This work was supported by a grant No KOSEF 981-0923-123-2 from the Korea Science and Engineering Foundation and also by a grant from Electronics and Telecommunications Research Institute.

References

[1] K. Beck, W. Cunningham. A Laboratory for Teaching Object-Oriented Thinking. In *Proceedings of OOPSLA'89*, pages 1-6, New Orleans, USA, October 1989.

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

[3] G. Booch. Object-Oriented Development. *IEEE Transactions on Software Engineering*, 12(2):211-221, February 1986.

[4] D. Chappell. *Understanding ActiveX and OLE*. Microsoft Press, 1997.

[5] P. Coad, E. Yourdon. *Object-Oriented Analysis*. Yourdon Press, 1990.

[6] A. Finkelstein, J. Kramer, and B. Nuseibeh. *Software Process Modelling and Technology*. John Wiley & Sons, 1994.

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

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

[9] T. Kim, N. Boudjlida. An Experience Report Related to Restructuring OODesigner: A CASE Tool for OMT. In *Proceedings of Asia-Pacific Software Engineering Conference*, pages 220-227, Taipei, Taiwan, December 1998.

[10] T. Kim, G. Shin. Restructuring OODesigner: A CASE Tool for OMT. In *Proceedings of 20'th International Conference on Software Engineering*, pages 449-451, Kyoto, Japan, April 1998.

[11] P. H. Loy. A Comparison of Object-Oriented and Structured Development Method. *ACM SIGSOFT SE Notes*, 15(1):44-48, January 1990.

[12] D. A. Marca, C. L. McGowan. *SADT, Structured Analysis and Design Techniques*. McGraw Hill, 1987.

[13] T. J. Mowbray, R. Zahavi. *The Essential CORBA*. John Willey & Sons, 1995.

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

[15] Y. P. Shan. An Event-Driven Model-View-Controller Framework for Smalltalk. In *Proceedings of OOPSLA'89*, pages 347-352, New Orleans, USA, October 1989.

[16] K. Wallnau, E. Morris. P. Feiler, A. Earl, and E. Litvak. Engineering Computer-Based Systems with Distributed Object Technology. In *Proceedings of the International Conference on Worldwide Computing and Its Applications*, Tsukuba, Japan, 1997, *Lecture Notes on Computer Science #1274*, Springer-Verlag.

[17] A. I. Wasserman, P. A. Pircher, R. J. Muller. An Object-Oriented Structured Design Method for Code Generation. *ACM SIGSOFT SE Notes*, 14(1):32-55, January 1989.

[18] R. J. Wirfs-Brock, R. E. Johnson. Surveying Current Research in Object-Oriented Design. *Communications of the ACM*, 33(9):104-124, September 1990.