Object-Oriented Application Development in CAD: A Graduate Course

Ulrich Flemming, Halil Erhan, Ipek Ozkaya

School of Architecture,
Carnegie Mellon University

Abstract

The programming languages typically offered by CAD systems for third-party application developers were procedural or functional. A major shift is currently occurring in that new versions of commercial CAD software will support object-oriented programming languages for application development. Developers who wish to take advantage of this new kind of environment must undergo a considerable cognitive "retooling" and adopt new software engineering strategies. We describe a graduate course that aims at introducing students to effective object-oriented development strategies, especially use case-driven development and the tools provided by the Unified Modeling Language (UML). Students gained experience with these tools by forming, together with the instructors, a single development team writing an application on top of MicroStation/J using JMDL as programming language. The paper describes the instructors' experience with this approach.

1 Background and Motivation

In the second half of the last decade, researchers from the School of Architecture and the Human-Computer Interaction Institute (HCII) at Carnegie Mellon University (CMU) investigated the interaction between MicroStation (MS) and its users in a real-world setting (Bhavnani et al. 1996, 1999). For the purposes of the present paper, the most important result of the study has been the discovery of *strategic knowledge* and its crucial role in making the use of a tool like a CAD system efficient. To use a metaphor used by one of our sponsors: Suppose a cabinetmaker has a well-organized woodshop, with each tool well understood, sharpened, and easy to reach. But if she is asked to create a table, knowing each individual tool well is not enough: She also has to know how to structure the overall task effectively, which tools to select for any step, and how to use them for each step. Note that this strategic knowledge is task-dependent. It represents a *task-specific layer of knowledge above the tool level*. The study showed that MS users often do not employ the most efficient strategies in this sense.

In response to this, members of the study team developed a course called *Strategic Use of CAD* at CMU. The goal was to develop and test methods that teach students strategic knowledge hand-in-hand with command knowledge. The course was taught three times between 1998 and 2000. Course assessments based on in-class data collection strongly suggest that it succeeded. The course described in the present paper aims at extending this strategic approach to application programming on top of a CAD system.

The programming languages typically offered by CAD systems for third-party application developers have been procedural or functional. A major shift is currently occurring in that new versions of commercial CAD software support object-oriented programming languages for application development. Developers who wish to take advantage of this new kind of environment must undergo a considerable cognitive "retooling" and adopt new software engineering strategies:

> "Using object-oriented design, the designers will stay away, as long as possible, from the

(ultimately inescapable) need to describe and implement the topmost function of a system. Instead, they will analyze the classes of objects of the system. System design will be based on successive improvements of their understanding of these object classes... For many programmers, this change in viewpoint is as much of a shock as may have been for some people, in another time, the idea of the earth orbiting around the sun rather than the reverse." (Meyer 1988, 50)

Novices may, in fact, have an easier time:

> "While professionals are reluctant to abandon their beloved data, function, and process models and always try to fit the new ideas somewhere into their set thinking patterns, computer rookies can light-heartedly get acquainted with object-orientation as an easily accessible approach." (Oesterreich 1999, 18)

Unlike other programming paradigms, the object-oriented paradigm requires an initial modeling step that precedes any coding, and modeling aspects remain significant throughout object-oriented software development. It is therefore not enough to know the syntax and understand the semantics of the programming language used. Software developers, be they rookies or seasoned professionals, need to employ efficient *modeling strategies* to ensure the envisioned functionality of the application and to take maximum advantage of object-oriented programming.

In response to this need, the course described here has the following goals:

- to introduce graduate and advanced undergraduate students to effective object-oriented modeling strategies; and to use these strategies to develop an interesting application as a software prototype.

The course is called *Object-Oriented Application Development in CAD*. It is a one-semester (16 weeks) course and part of the graduate curriculum in Computational Design at the School of Architecture, whose focus traditionally has been the education of competent application programmers in building design. The present course fits right into this tradition.

It was taught as a graduate elective in the fall of 2000 and as a required/elective course in the fall of 2001 with participation by students enrolled in the School of Architecture and in the Department of Civil and Environmental Engineering at CMU. The first author was the instructor of record (IOR); the other authors were co-instructors responsible for specific technical portions of the course. The CAD system used was MicroStation/J (MS/J) and JMDL, an extension of Java, was the programming language.

Space limitations prevent us from providing here a full description of the course. Rather, we introduce the overall approach we took and then highlight certain experiences and insights that may be of interest to educators who plan to develop similar courses. A fuller account can be found in (Flemming et al. 2001).

2 Application

We developed different applications in the two installments of the course. The first time around, our application supported building renovation or remodeling projects. We assumed that an existing building was described in a set of CAD drawing files. The application allowed designers to open (a copy of) such a file and interactively identify parts to be removed or relocated and elements to be added. The application automatically (re)drew these elements in the appropriate style. In the background, it build a collection of persistent objects representing these elements so that we could add features to the application like the automated generation of part schedules.

For the second course installment, we decided to make the application more interesting in the way it was able to handle geometry. We assumed again that it worked on a given set of CAD files, in this case, drawings as they are typically created by HVAC consultants. These drawings often depict the planned system in an abstract version; for example, ducts and pipes are shown as single lines and objects such as outlets as graphical symbols (Figure 1). The application developed in the course allowed architectural designers to interactively identify the HVAC components in such a drawing and supply

some additional parameters so that the application could generate a more accurate representation of the system in the CAD file (Figure 2). The motivation to create such a drawing would not only be the desire to depict the system more accurately, but also to be able to run spatial interference tests or analyses that need a representation of the system in terms of the real objects it depicts. The present paper deals with this second application, which we called *HVACTools.*

We do not claim that these applications are novel or without commercial precedents. They were meant to support the pedagogical aims of the course, for which we needed applications of practical interest that could be implemented quickly without much background research, let alone real innovations.

3 Approach

3.1 Use Case-Driven Application Development

The course introduces the software engineering strategies with which the IOR gained experience during a decade of work in the Software Environment to Support Early Building Design (SEED) project (Flemming and Woodbury 1995).

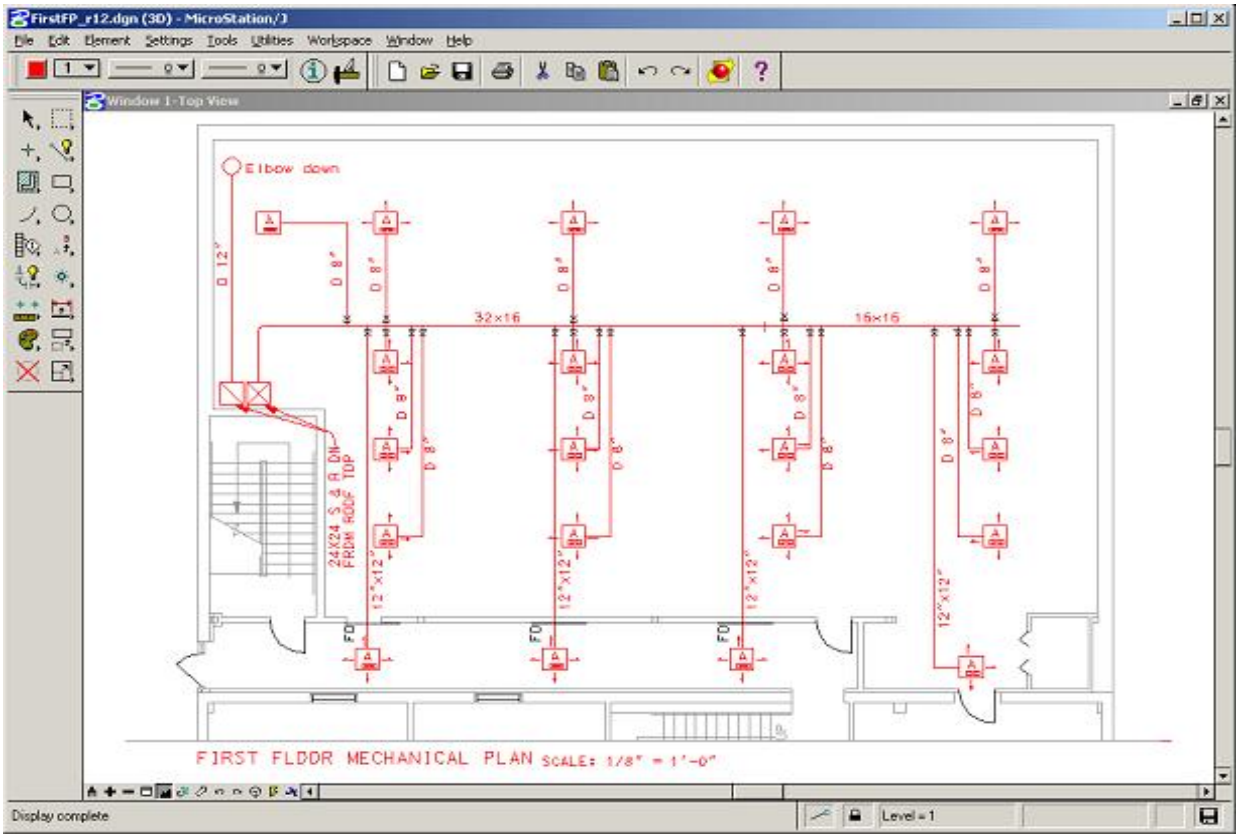


Fig.1 Typical HVAC drawing

The approach relies on *use cases* as the central concept to pin down the desired functionality of the application and to provide a common thread that runs through all development stages (Jacobson et al. 1992). The selection by the SEED team of this approach was based on a careful comparison of the object-oriented methods available at the time (Coyne et al. 1993 give a detailed account). We believe that the criteria used then are still valid today.

A use case describes a “sequence of actions an actor performs using a system to achieve a particular goal” (Rosenberg 1999, 38) or "a sequence of actions a system performs that yields an observable result of value for a particular actor” (Booch et al. 1999, 19). In the context of MS, "Place Line" may

represent a use case, if use cases are formulated at the granularities of individual "tools" (in the MS sense) or commands (our preferred approach; see Section 4). Taken together, the use cases specify the desired functionality of the application completely and from the end users' perspective. In practice, they emerge—through various iterations—in discussions with client and end-user representatives. In our course, we had to do without these as we had no external client.

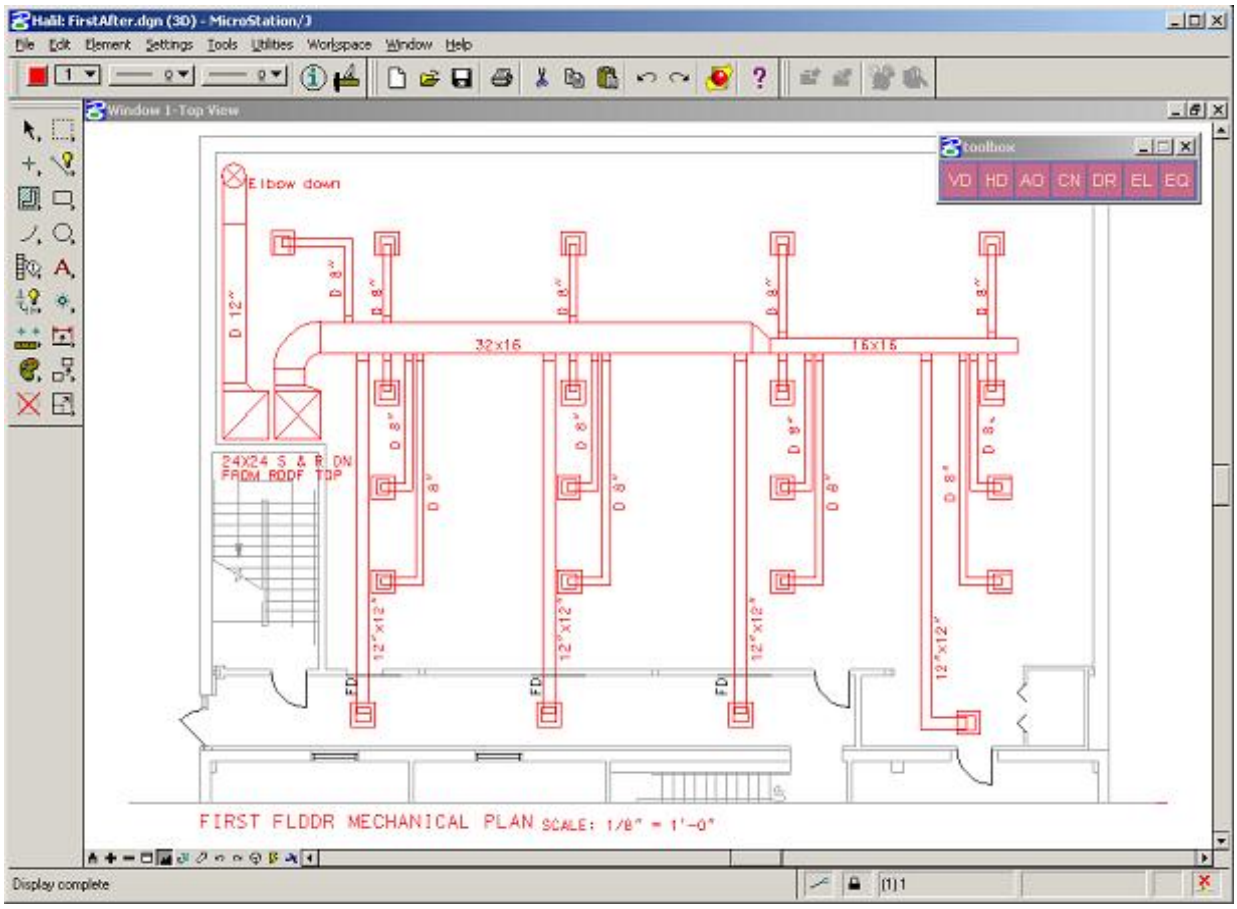


Fig. 2 Drawing generated with the help of HVACTool

Some SEED development teams have used—in addition—OMTool to generate graphical documentation of the object model they were working on as well as source files (Rumbaugh 1991). Others who reviewed object-oriented methods at the time have made the same selections (Rosenberg 1999, 3). These methods have recently been combined by their respective inventors (with the addition of G. Booch) to form the *Unified Modeling Language* (UML) (Jacobson et al. 1999). Along with this turn of events, new development tools have become available that support UML-based software development. An example is RationalRose, which we used in the course to document our decisions [www.rational.com].

The present course served as a vehicle to explore and demonstrate these methods and tools. However, for pedagogical and practical reasons, we decided not to introduce UML in its entire breadth. We are not alone in our impression that the "three amigos" (Booch, Jacobsen, and Rumbaugh), in combining their respective methods and tools, produced a rather unwieldy mix of partially overlapping, even competing concepts and methods. As one observer remarks, "...the fact that UML provides multiple ways to say the same thing is probably not an advantage." (Rosenberg 1999, 111). We therefore introduced UML methods selectively in the course. Among the texts we evaluated, we found (Oesterreich, 1999) much to our liking.

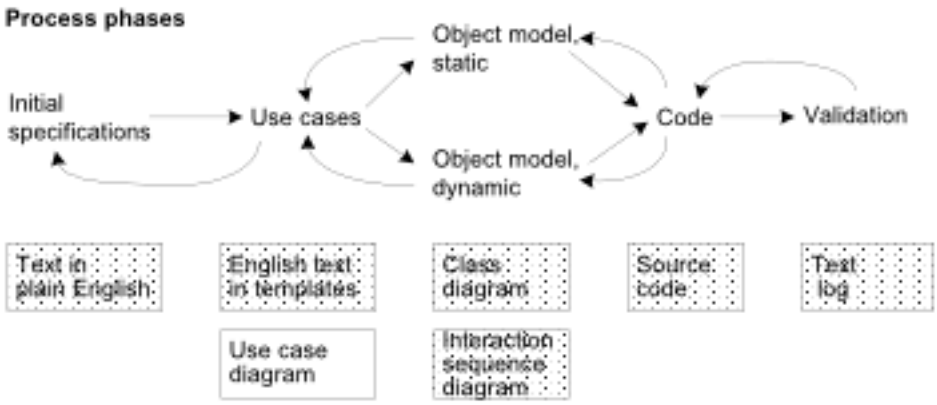


Fig.3 Phases and products of use case-driven software development

3.2 Development Phases

Use cases are the primary artifacts of use case-driven software development. They function specifically

- to *guarantee the desired functionality* through all development phases
- to *integrate all phases* by giving them a shared focus that always keeps this functionality upmost in the developers' mind.

Figure 3 depicts the phases of the process and the products each phase creates. (The phases depicted in the figure represent the selection of UML concepts and methods that we used in our course). The appendix briefly describes how we defined central notions of object-oriented programming.

1) **Initial specification**. Based on the recommendation of Rosenberg (Rosenberg 1999), we precede the use case development phase by an initial specification phase not mentioned by the UML inventors. These initial specifications document basic system and context constraints and identify the overall desired functionality that cannot be naturally captured in terms of use cases (for example, because there is no natural actor to whom these specifications could be tied).
2) **Use case development**. This phase produces, through several refinement iterations, a description of the desired functionality in terms of use cases *in the end-users' language*.
3) **Object model, static**. This phase results in class diagrams that describe the object model or schema for the application. Depending on the tools used, it may also create source files in outline form.
4) **Object model, dynamic**. This phase results in interaction or sequence diagrams that depict which objects are involved in which use case and what part of their public interfaces are being used.
5) **Code**. In this phase, the code for all classes in the class diagrams is produced, use case-by-use case.
6) **Validation**. This phase tests the code, again use case-by-use case.

These phases do not follow each other in a strict "waterfall" model. Rather, the process is meant to be highly iterative with numerous feedback loops between phases.

3.3 Extreme Programming

At both occasions when we taught the course, we had relatively few students and a relatively short time to develop our application. In addition, students had varying degrees of prior programming experience,

with some taking their first programming course (an introductory Java course taught in the School of Computer Science) parallel to ours. In response to this, we limited the portions of the application that were to be implemented and assisted students hands-on, especially during coding.

In doing this, we found ourselves adopting more and more principles of Extreme Programming (XP), a relatively new, lightweight software development methodology for small- to medium-sized teams. The word “extreme” in the name refers to taking commonsense software development principles and practices to extreme levels (Beck 2000). These practices include simplicity, continuous integration, short iterations, refactoring, pair programming and continuous automated testing. Among these principles, pair programming and refactoring proved specifically helpful in our course.

Pair programming is accomplished by two people looking at one machine, with one keyboard and one mouse. The person with the keyboard and the mouse focuses on the best way to implement the portion of the system the pair is dealing with at the time, while the other partner keeps track of the overall system design and the more strategical aspects of the work. A pair may consist of an experienced programmer and a novice, and learning may be a desired side-effect of the work. In our course, a pair typically worked on a specific use case at a time. One member was most often an instructor and the other one a student. Pairs worked in a highly concentrated fashion that produced an intense learning experience for the student.

Refactoring refers to restructuring the system incrementally without changing its behavior to remove duplication, simplify, or introduce other improvements. In our course, refactoring governed the way in which code was shared. For example, we—as instructors—provided general utility methods to the entire team. We typically wrote an initial version that allowed students to get started, and while they were at work, expanded and corrected the methods, taking feedback from students into account. Note that the modularity and encapsulation characterizing object-oriented programming make this especially easy.

The course outline followed the phases depicted in Figure 3, with many iterations and feedback loops between phases 2 to 6. Each phase started initially with one or several lectures that introduced the specific concepts and tools needed for this phase. These lectures were typically followed by individual exercises in which each student applied the tools to a specific use case (once we had an initial use case document), after which the individual responses were coordinated and integrated through class discussions. The coding phase was preceded by two weeks of introductory Java exercises emphasizing features of the language not covered in the parallel Java course, notably the user interface toolkits AWT and Swing, which we needed to develop rapidly the graphical user interface for our application.

We highlight in the next sections specific aspects of our work in the crucial modeling phases.

4 Use Cases

In our view, use cases describe a *meaningful task* or result of value an actor (i. e. a user of the system in a specific role) may achieve. A meaningful task, in turn, is a sequence of actions or operations that must be *executed together* to achieve some goal. The task is self-contained in the sense that after the last task has been performed, an actor has choices in selecting a next task to execute. Conversely, the sequence of actions cannot be interrupted if the task at hand is to be achieved. An example is again the Place Line command in MS, which determines a specific task MS users can set for themselves. This task consists of a sequence of actions (select tool, adjust settings, identify datapoints) that must be executed if the goal of the task is to be achieved. It is self-contained because it does not determine what went on before or can go on after its execution. On the other hand, adjusting the Place Line settings, in itself, would not be a use case because it is a task that has meaning only within another, larger task.

What constitutes a meaningful task for an application depends very much on the level of *granularity* at which the application is considered and use cases are formulated. Selecting this level is a major design decision. We favor a relatively fine-grained level, as exemplified by the Place Line tool or the use case below (there is no generally accepted template or format for describing use cases; the example illustrates our preferred form, which is an amalgam of various suggestions made in the literature):

Use case: Place Connections

The architectural designer (AD) places connectors between pairs of HVAC components already placed.

Preconditions:

1) HVACTools is active (this implies that MS/J is running and the appropriate dgn file is open).
2) The components to be connected have been placed in the drawing and are visible in the MS window.

Basic Course:

1) The AD issues the “Place Connection" command from the main tool box (see Fig 4a).
2) HVACTools opens the Place Connection dialog box and prompts the AD to select the first component (see Fig.4b). The AD may click the Close button at any time to terminate the use case.
3) The AD overrides the default setting for the type of connection (optional) and selects one of the components to be connected in an MS/J window.
4) HVACTools checks if the selected component is “connectable." If it is, HVACTools highlights the selected component and proceeds to the next step. Otherwise, it displays an error message, and the use case returns to step 2.
5) HVACTools prompts the AD to select the next component.
6) The AD selects the second component to be connected in an MS/J window.
7) HVACTools checks if the selected component is “connectable." If it is, HVACTools highlights the selected component and proceeds to the next step. Otherwise, HVACTools displays an error message, and the use case returns to step 5.
8) HVACTools enables the Place button and prompts the AD to select the next component.
9) The AD either selects the next component to be connected or clicks the Place button.
10) If the AD selected another component, the use case returns to step 7. Otherwise, HVACTools opens a Settings Box that is component-specific and asks the user to determine settings such as the size of the connection.
11) The AD adjusts the component attribute settings and clicks the Commit button.
12) HVACTools draws the connection component(s) between the components selected in the MS/J window. In our first implementation, we restricted ourselves to connecting only two components.
13) The AD inspects the drawing. If it is correct, she clicks the Close button. Otherwise, she clicks the Cancel button.
14) If the Close button was clicked in the preceding step, HVACTools creates (in the background) an object describing the connecting component and adds it to the set of currently defined objects; closes the dialog box; and terminates the use case. If the Cancel button was clicked, it deletes the elements describing the connection in the MS/J window and returns to step 2.

Alternative Courses: (not shown here)

We want to emphasize two points about this example: It was arrived at after several iterations, some of which were the result of difficulties encountered during coding. Secondly, use case development goes hand-in-hand with the design of the graphical user interface (GUI) at the level of granularity we selected; the two are, in fact, inseparable. That is to say, the developers are considering the GUI design from the start—it is never an afterthought! This is a main reason why we prefer use cases at this level of granularity. Initially, GUI ideas were captured in sketches; later on, they were replaced by images of the GUI implementation.

(a) (b)

Fig.4a Tool and Dialog Boxes used in use case Place Connection.
Fig.4b Tool and Dialog Boxes used in use case Place Connection.

Our final application implemented a total of 12 use cases, seven of which dealt directly with component construction (like the example above), while the rest covered session management (open, close, save, etc.) and the handling of general project settings (level assignments by component type, component symbology etc.).

5 Object Model

We show in Fig.5 portions of the schema we developed, again over several iterations, for HVACTools. The diagram shows schema classes and their inheritance and attribute relations in UML notation, which is derived from OMTools and should by now be familiar to readers

We want to highlight two features of the schema. Note first that the schema adheres strictly to the separation between view (not shown in the figure), control and model (or domain) classes that underlies good object-oriented design. In our application, the *domain* classes describe the components of the HVAC system, instances of which are stored persistently (as objects) across MS/J sessions. These objects are never allowed to interact with instances of *view* classes (like GUI widgets). This interaction is always channeled through an instance of the *control* class HVACSessionManager. This strict separation allows the interface classes and domain classes to be independently modified, actually reused in other applications.

Secondly, "design patterns" of delegation and compositions are used extensively to make the schema design more modular, flexible and extensible (Gamma et al. 1995). This becomes particularly clear when one observes how the geometry of an HVAC component is handled. It is captured by an associated ComponentGeometry class. This class, in turn, has associations with dgn elements in the dgn file, which depict the component in the drawing, and with a Shape class, which captures the component geometry in a form that makes reasoning with this geometry easy. The basic idea underlying this design is *the separation between the graphical display of a component in an MS/J window and its geometry proper*. This separation allows objects, on the hand one, to reason about their geometry without the need to retrieve it first from the graphical elements in a *dgn* file. Conversely, we can add graphical elements to the display without enlarging the set of graphical elements that have to be stored persistently with the objects. For example, the concentric polylines shown in Figure 2, which indicate the directions of air flow from a diffuser, are displayed in the MS/J window, but not part of the persistent object representation.

One remark on schema design in general: The three amigos insist that a schema is not the result of *design*, but of *analysis*, and call this phase accordingly *analysis* and its product, the class diagram, the *analysis model*. Their argument is that design implies a level of detail not needed for the class diagrams. We disagree strongly with this view, given our experience as *designers*. For pedagogical reasons, we insist in our course that developing a good class diagram or schema in general is a design activity *par excellence*, characterized by the open- endedness, false starts, iterations, sketching by hand, discussions and general hand-wringing as we know them from design, but not from analysis. The detail argument poses no problem for architects and engineers who know from their daily experience that designs progress through phases of increasing detail, characterized by different scales at which the artifact is being considered. Thus, *schema development is design, not analysis*.

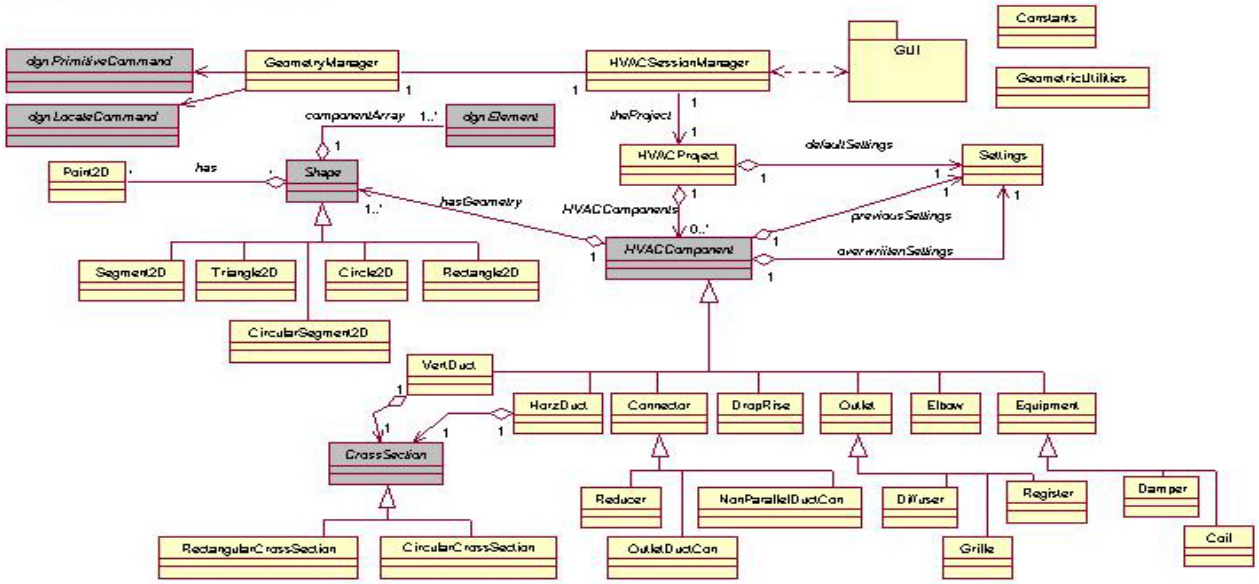


Fig. 5 Static Object Model or Schema (in UML notation as generated by RationalRose)

6 Sequence Diagrams

A sequence diagram needs to be developed for each use case. It identifies the objects (as class instances) that are involved in the execution of the use case and the methods by which they interact. If we collect the methods associated with objects of the same class across all sequence diagrams, we arrive precisely at the public interface we have to implement for this class. The sequence diagrams are called a dynamic object model because they depict object interactions at run time (Fig.6).

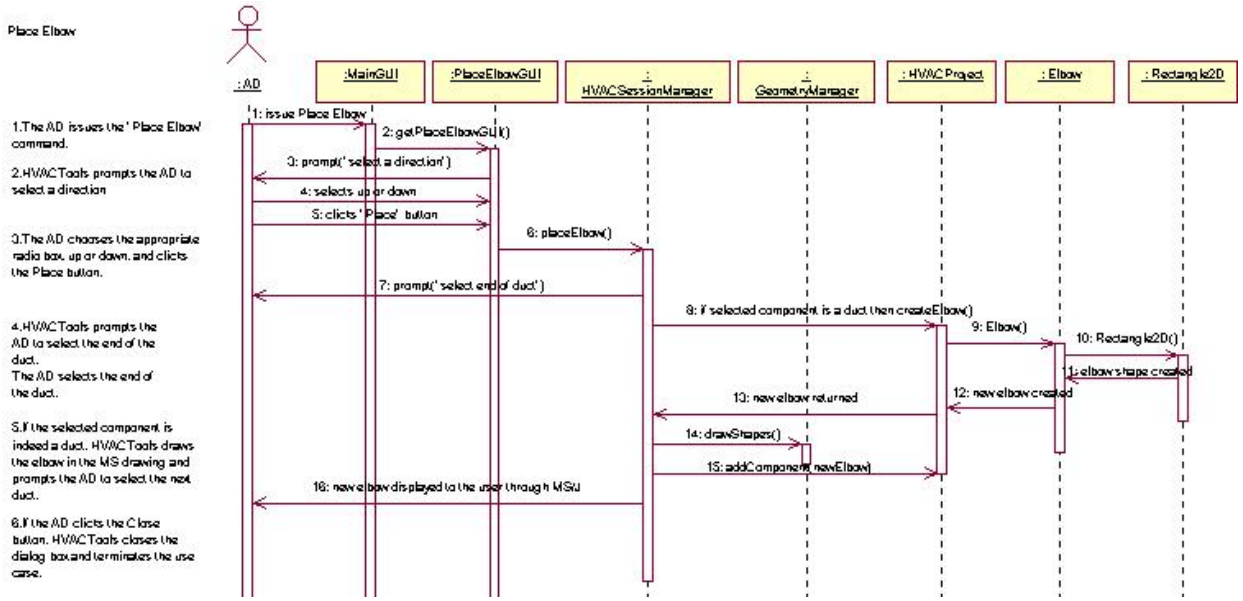


Fig. 6 **Dynamic Object Model as Sequence Diagram**

The sequence diagrams both depend on and affect the static object model. In order to find out which objects may be involved in a use case, we must have class diagrams. Conversely, we may discover a better class composition while working on sequence diagrams. Furthermore if we collect the methods associated with objects of the same class across all sequence diagrams, we arrive precisely at the

public interface of that class. Thus, the development of class and sequence diagrams constitute really a single phase.

In our experience, sequence diagrams are *the* most effective tool to affect the cognitive retooling needed for object-oriented programming. No other device illustrates so clearly how functionality can be delivered by interacting objects and how one can conceive of them.

7 Geometric Reasoning

We conclude our description of the application by indicating how we handled geometric reasoning aspects. We created the class GeometricUtilites, which makes available a suite of static methods (generators and predicates) that perform standard computations in analytical geometry. We show in the table below (Table 1) the signatures of four of these methods and then illustrate how they can be used to generate the geometry of a connecting duct between an air outlet and a supply duct.

Suppose an outlet and a duct are drawn as shown Fig. 7a. We know in addition the width of the desired connection, *w*. Note also that all HVAC Components implement the overloaded method getCenter, which they delegate to the associated Shape. Given this information and these methods, we can compute the geometry of the duct as follows:

1) We request from the outlet its center, point *C*.
2) We compute the foot of *C* on the long sides of the duct and find out which side is closer to *C*. Call the foot on that side *F*.
3) We determine which side of the outlet is closest to the duct and compute the intersection, *X*, between that side and the line through *C* and *F*.
4) We translate the segment *CX* by *w*/2 and -*w*/2.

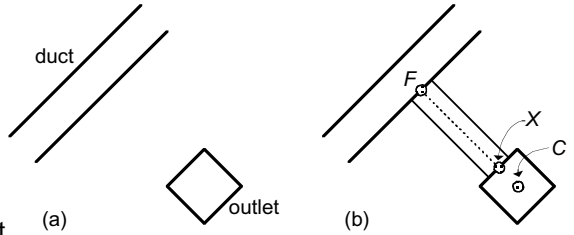


Fig.7 Construction of a connecting duct

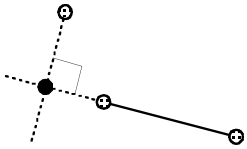
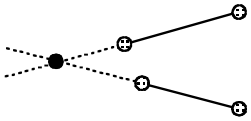
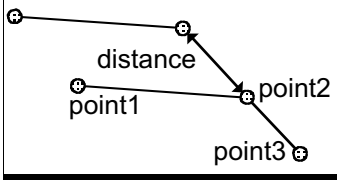
	double distanceBetweenPoints(Point2D, Point2D) returns the Pythagorean distance between the wo points.
	Point2D footOfPointOnLine(Point2D, Segment2D) returns the point at which the normal through the given point intersects the line defined by the given segment. If the given point is on the line, the method returns that point.
	Point2D intersectionOfLines(Segment2D, Segment2D) returns the point at which the lines defined by the two segments intersect. If the segments are (almost) parallel, the method returns NULL.
	Segment2D translateSegmentByDistance (Point2D, Point2D, Point2D, double) returns the segment that translates the segment between the first two points by the given distance in the direction indicated by the second and third point (see diagram on the left).

Table 1: Selected geometric utilities functions

8 Conclusions

Students in the course report that one of the most interesting aspects, indeed an eye-opener, was to observe an application develop from scratch. Moreover, this application had a real GUI running in the context of a real CAD system. This differed markedly from other courses dealing with individual tools in isolation. A course like ours demonstrates how all of this can fit together in practice. It was the combination of use-case driven software development and aspects of Extreme Programming that made this possible.

We find use cases, class and sequence diagrams the most useful UML concepts in such a context. But by themselves, they will not generate good object-oriented design. Their use has to be embedded in established strategies of object-oriented software development such as the separation of model, view and control; design patterns are indispensable. Moreover, these strategies are generally applicable and do not depend on a language like JMDL or even a CAD-based application. That is a course like the present one is of general interest to application programmers. Especially for an audience with a design background, it is important to emphasize the *design* aspects of object-oriented modeling.

Acknowledgments

The course has been funded by grants from Bentley Systems, Inc. and the Pennsylvania Infrastructure Technology Alliance (PITA). Rational, Inc. provided an educational copy of RationalRose free of charge.

9 References

Beck, K. (2000). *Extreme Programming Explained*. Addison Wesley, Boston.

Bhavnani, S.K., U. Flemming, D.E. Forsythe, J.H. Garrett, D.S. Shaw, and A. Tsai. (1996). CAD Usage in an Architectural Office: From Observations to Active Assistance. *Automation in Construction* **5** 243-255.

Bhavnani, S. K., B.E. John, and U. Flemming. (1999). The Strategic Use of CAD: An Empirically Inspired, Theory-Based Course. In *Proceedings of the CHI 99* 183-190.

Booch, G.; J. Rumbaugh, and I. Jacobson. (1999). *The Unified Modeling Language. User Guide*. , New York: Addison Wesley Longman.

Flemming, U. and R. Woodbury. (1995). Software Environment to Support Early Building Design. In *Journal of Architectural Engineering* **1** 147-152.

Flemming, U., H.I. Erhan, I. Ozkaya. (2001). Object-Oriented Application Development in CAD. Technical Report 48-01-01. Pittsburgh, PA: Carnegie Mellon University, Institute of Complex Engineered Systems.

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

Jacobson, I., M. Christerson, P. Jonsson, and G. Overgaard. (1992). *Object-Oriented Software Engineering. A Use Case Driven Approach*. New York, NY: Addison-Wesley.

Meyer, B. (1988). *Object-Oriented Software Construction*. New York, NY: Prentice-Hall.

Oesterreich, B. (1999). *Developing Software with UML. Object-Oriented Analysis and Design in Practice*. Reading, MA: Addison-Wesley.

Rosenberg, D. (1999). *Use Case Driven Object Modeling with UML. A Practical Approach*. Reading, MA: Addison-Wesley

Rumbaugh, J., M. Blaha, W. Premerlani, F. Eddy, W. Lorensen. (1991) *Object-Oriented Modeling and Design*. Englewood Cliffs: Prentice Hall.

10 Appendix

We briefly indicate in this appendix how we defined general concepts of object-oriented programming in the context of the course.

Objects

Objects are the fundamental entities that give object-oriented programming its name. It is useful to distinguish between the *conceptual* and *technical* meaning of objects.

At the conceptual level, objects represent—at run-time—individual chunks of information that are as such meaningful either to the user (because they represent objects of interest in the domain) or to the programmer (because they play a role in organizing the overall program and its execution, but remain invisible for users). An example of an object of interest to the user of our application is an air outlet. An example of an object of interest to the application developers only is the object we call the SessionManager.

From a technical perspective, an object is a run-time construct, namely a distinct region of memory with specific semantics, which include a collection of attributes of various types (value, relational, method). A collection of objects linked through relational attributes is an *object configuration*.

In a typical object-oriented program session, working memory gets populated with objects that interact with each other and—through this interaction—change each other's state (in terms of attribute values); the creation and deletion of objects also happens as side effects of these interactions.

Classes

A class is a *program* construct. It defines the attributes a specific type of object can have, both public and private. The collection of public attributes is the *public interface* of a class. An object is created at run-time as an *instance* of a class and has precisely the attributes defined by that class.

Classes are the *modules of an object-oriented program*. A collection of classes that—taken together—support a particular application is called a *schema* or an *object model*. By specifying the attributes and behavior of each class, *a schema defines the universe of discourse for a particular application*.

A class can inherit the attributes of another class. This establishes super/subclass relations between classes or an *inheritance hierarchy*. The public interface of a class is shared by all of its subclasses. *Each subclass can re-implement a method in the public interface of a superclass*. This results in method overloading or *polymorphism*. The decision which method to use is made not at compile-time, but at run-time (*dynamic binding*). Thus, inheritance and polymorphism go hand-in-hand. Together with encapsulation, they account for the particular power of object-oriented programming [We cannot elaborate this point here. Interested readers are referred to (Meyer 1988)].