Object-Oriented Application Development in CAD

An Interdisciplinary Graduate Course

Research Team: Ulrich Flemming (PI); Halil Erhan; Ipek Ozkaya
School of Architecture and Institute for Complex Engineered Systems (ICES)
Carnegie Mellon University
Pittsburgh, PA 15213

June 6, 2001

The work reported here has been supported by grants from Bentley Systems and the Pennsylvania Infrastructure Technology Alliance (PITA).
Abstract. This report describes a graduate interdisciplinary course offered to students in the graduate program of the School of Architecture at Carnegie Mellon and related departments in fall 2000. The motivation was the realization that when commercial CAD (Computer-Aided Design) systems recently switched from procedural application programming languages to object-oriented ones, third-party application must undergo a significant cognitive “retooling”; i.e. they must know more than the syntax and semantics of the new programming language to be used and must be able to employ appropriate software development strategies that are appropriate for the new paradigm, especially with respect to the importance of modeling, a distinguishing characteristic of object-oriented programming. The goal of the course was (a) to introduce and test strategies of object-oriented application development in general and in the context of MicroStation, a state-of-the-art commercial CAD package; (b) to develop—as a course team project—an interesting application that gives students practice with these strategies and team work; and (c) to document our approach and findings so that others can learn from them. The strategies introduced were the use-case approach of Jacobson et al. and the complementary object-modeling tools of Rumbaugh that were recently integrated into the Unified Modeling Language UML. The software platform supporting the course comprised MicroStation, JMDL (a superset of Java) and ProjectBank on the CAD side and RationalRose on the modeling side. The application developed by students in the course supports the generation of drawings for remodeling projects from a set of dgn files describing the existing state of the building to be remodeled.

The course was supported by a grant and in-kind contributions from Bentley with matching funds from the Pennsylvania Infrastructure Technology Alliance (PITA).
## Table of Contents

1. Motivation 1  

2. Approach 4  
   Use Case-Driven Application Development 4  
   Application 5  

3. Fundamental Concepts 7  
   Basic Concepts of Object-Oriented Programming 7  
   Use Case-Driven Software Development 11  

4. Course Outline 13  

5. Initial Data Collection 14  
   Application Context 14  
   Interviews 14  

6. Use Cases 19  
   Use Case Modeling 19  
   Key Terms and Concepts for Our Application 22  
   Group 1: Session-Related Use Cases 23  
   Group 2: Component-Related Use Cases 24  
   Group 3: Computation-Based Use Cases 31  

7. Object Model 33  
   Static Object Model I 33  
   Dynamic Object Model 37  
   Static Object Model II 38  

8. Implementation 41  
   The GUI. 41  
   ProjectBank and Remodeler Interaction 42  
   Output 44  

9. Conclusions 45  

10. References 46  

Appendix A. Sequence Diagrams 47
1. Motivation

During the last decade, researchers from the School of Architecture and the Human-Computer Interaction Institute (HCII) at Carnegie Mellon University (CMU) have been investigating the interaction between MicroStation users and the system in a real-world setting. The results were published in a series of articles, notably in the proceedings of the CHI conferences from 1996 on. For the purposes of the work described in this report, the most important result is the discovery of what we call strategic knowledge and its crucial role in making the use of a tool like a CAD system efficient. This knowledge is crucial even for tools whose user interfaces are well-designed. To use a metaphor used by Keith Bentley during one of our visits with Bentley Systems: Suppose a cabinet maker has a well-organized woodshop, with each tool sharpened, easy to reach on a pegboard, and well-understood. 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.

We discovered that specific strategies which make the execution of a task more efficient respond to the nature of the tool in one of two ways: they either benefit from the power of the tool by delegating certain actions to the tool, or circumvent some of its limitations. If the tool at hand is a CAD system, a typical delegation strategy is to use the copy and manipulation commands to create repetitive elements so that they do not have to be drawn individually. An example of a circumvention strategy in CAD is the use of multiple views at varying resolutions to overcome the generally small size of the screen, which makes it impossible to work out a detail and see it in the overall context in one view. We discovered several important strategies for CAD, many of which can be generalized to other applications, even if they are seemingly very different from CAD.

Our empirical data show that CAD users often do not employ the most efficient strategies. One reason is that this knowledge is not covered in a typical, command-focussed tutorial or in textbooks. In response to this, we have developed a course called Strategic Use of CAD, in which we develop and test methods to convey strategic knowledge hand-in-hand with command knowledge. The course has been taught four times. Course assessments based on in-class data collection suggest strongly that we are succeeding. The CAD system used has been MicroStation versions 5 and 95 and MicroStation/SE. The tasks covered were traditional CAD tasks like the production of 2D drawings or 3D models.

The present work aims at extending the strategic use of CAD to include object-oriented application programming as it becomes possible with MicroStation/J (MS/J). The programming languages typically offered by CAD systems for third-party application developers have been procedural. A major shift is currently occurring in that new versions of commercial CAD software will support object-oriented programming languages for application development. A case in point is JMDL, an extension of Java offered with MS/J. 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

1. see Bhavnani et al. (1996, 99)
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.3

The object-oriented paradigm requires an initial modeling step that precedes any coding, and modeling aspects remain significant throughout 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 the object-oriented paradigm. As was the case with CAD users, software developers need to have strategic knowledge if they are to use the possibilities offered by the object-oriented programming paradigm.

In response to this need, the course described in this report introduces students to an object-oriented modeling method that has been successfully employed for more than eight years in the Software Environment to Support Early Building Design (SEED) project, an inter-disciplinary research effort at CMU aiming to construct the prototype of a design support system for the early, conceptual stages of building design.4

As was the case with the cabinet maker, a software development team confronted with the task of creating an object-oriented application needs to know—above the tool level—strategies

- to specify the desired functionality of the application clearly in order to assure, on the one hand, that the expected benefits will justify the development costs and, on the other hand, that the application indeed fulfills its purpose once it has been developed;
- to define the objects of interest needed to capture the domain information that the application must be able to handle, given its functionality; this task is facilitated if the application can extend a given object framework, but remains non-trivial even in this situation
- to define the object interactions and interfaces needed to deliver the desired functionality
- to implement a first software prototype
- to field-test the prototype
- to make improvements given feedback from the tests.

3. Oesterreich (1999) p. 18
The course, then, has the following goals:

- to introduce graduate and advanced undergraduate students to strategies able to achieve the above objectives
- to use the strategies to develop an interesting application as a software prototype within ProjectBank, using JMDL as programming language\textsuperscript{5}
- to document the process so that other teams can learn from it.

As a side effect, students will become familiar with ProjectBank itself, which may prove useful when they seek employment. (We are not stressing this as main objective because the students in the computing concentration of the graduate program in the School of Architecture at CMU are very knowledgeable about computing and able to learn new tools from documentation alone; i.e. they would not require an entire course focused on ProjectBank as a main objective).

The course is called Object-Oriented Application Development in CAD. It has been taught as a graduate elective once in the fall of 2000 with participation from a small number of students: 4 students who were taking the course for credit (three of them enrolled in the School of Architecture and one in the Department of Civil and Environmental Engineering); 2 auditing students; and two Ph.D. students who were co-instructors with the PI.

The small number of students was actually a benefit because it made it much easier for the instructors to absorb the glitches that necessarily occur when a highly experimental course is offered for the first time and the support technologies are at the outset as unfamiliar to the instructors as they are to the students.

The present report describes the work done in the fall 2000 course. It is addressed to students who may take the course when it is taught again in fall 2001; to instructors considering teaching a similar course; to software developers, especially in the CAD area, who consider shifting to an object-oriented approach; and, of course, to our sponsors.

\textsuperscript{5} This goal is very much in line with the perspective outlined in [Aish 1999].
2. Approach

2.1 Use Case-Driven Application Development

The course concentrates on the strategies that have been used successfully for more than eight years in the SEED project. 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. Our selection of this approach was based on a careful comparison of the object-oriented methods available when we started work on SEED in 1992.

The use case approach had—and still has in our experience—several advantages over competing object-oriented software development methods. First of all, it does not take the objects of interest for granted at the outset as some other object-oriented approaches do. Rather, it relies on a principled strategy to discover appropriate objects based on a higher-level, user-oriented description of the desired functionality, formulated in terms of the above-mentioned use cases. Furthermore, the use cases structure and tie together all subsequent development phases so that the intended functionality is always the focus of attention. For example, object interactions can be diagrammed use case-by-use case in the form of “sequence diagrams”, which we found especially useful for novice object-oriented programmers, who are immediately forced to think in terms of object interactions. The code itself is then naturally structured around use cases (down to the name of methods). Testing also proceeds use case-by-use case, and the user documentation is structured likewise (even if the term “use case” itself does not appear).

Some SEED development teams have used—in addition—OMTool to generate graphical documentation of the object model underlying the SEED component they are working on and source files. We note with satisfaction that others who reviewed object-oriented methods at the time have made the same selections. We take this—together with our altogether positive experience with these methods and tools—as a strong endorsement of our choices.

These methods have recently been combined by their respective inventors (with the addition of G. Booch) to form the Unified Modeling Language (UML). Along with this turn of events, new development tools have become available that allow for seamless transitions between methods originating from different sources.

The course under discussion serves 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 impressions that the “three amigos”, 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.”

6. Jacobson et al. (1992)
7. Coyne et al. (1993) give a detailed account.
We also found the UML books published by the inventors themselves unwieldy and not appropriate for use as texts in class. It is not surprising, then, that shorter texts have started to appear with greater promise as introductory texts. We reviewed those by Rosenberg and Oesterreich and have learned from both. Although we agree largely with Rosenberg’s selection of methods and his strong practical orientation (and appreciate his sense of humor), we disagree with some of his tactical suggestions based on our experience. Oesterreich’s text is much more to our liking and we would have used it as text for the course had we come across it in time. It will be the course text in the fall of 2001.

Independently of the text we select, we have to realize that the terminology used in the UML-related literature is not stable and sometimes outright contradictory. For example, Jacobson et al. call their collaboration diagrams sometimes analysis diagrams. Rosenberg calls the same diagrams robustness diagrams; he uses the term collaboration diagram for a different type of diagram. As a result, we had to make choices with respect to the terminology in the course even if we wanted to avoid joining the terminology wars as a matter of principle. We will get back to this issue in later sections when the occasion arises.

2.2 Application

In order to gain practical experience with these methods and tools, the entire class formed a single development team producing—through the course assignments—an application on top of MS/J. The application we chose is a tool to support the remodeling of existing facilities, like the revitalization of army barracks, whose overplaying are available as.dgn files. This application can take full advantage of ProjectBank’s legacy features.

The focus of the work was the creation and implementation of a schema

- to model those parts of a facility that are to be demolished and their identification in the existing drawings
- to model those parts that are to be added and their addition to a drawing.

Given such a schema, the application may facilitate the generation of demolition and construction drawings, of cost estimates and, possibly, some preliminary checking for conformance with important programmatic requirements. Figure 1 shows an example of a typical remodeling drawing.

We assumed that users of the application will determine the geometry of all objects of interest by working in a MS/J view using MS/J drawing tools: selection tools to identify objects to be demolished or removed, and placement tools to define objects to be added.

---

12. For example, Booch et al. (1999) and Jacobson et al. (1999)
13. Rosenberg (1999); Oesterreich (1999)
Figure 1: Part of a typical remodeling drawing. The plan is shown here courtesy of Tai+Lee Architects, Pittsburgh. All rights reserved.
3. Fundamental Concepts

We introduce in this chapter general concepts of object-oriented programming and the use-case driven approach. Readers who are familiar with these concepts may skip this chapter.

3.1 Basic Concepts of Object-Oriented Programming

This section introduces objects and classes, the key concepts around which object-oriented programming evolves.

3.1.1 Object

Objects are, of course, 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 a wall in a floor plan that has to be removed. An example of an object of interest to the application developers only is an object we call the SessionManager, which, as the name suggests, manages all events that occur in a specific session of our application; the users of our application do not have to be aware of the existence of this object at all. The latter example also illustrates that objects in the object-oriented context can be active at run-time; that is, they are able to do things, or—in object-oriented parlance—have behavior. This distinguishes them from the more passive notion we have of objects we encounter in daily life.

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. 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.

The most important attribute types an object can have in our application are the following:

- **embedded** attributes, which disappear when the respective object is deleted. An example is the length attribute of a wall, which is meaningless when the wall itself does not exist.
- **relational** attributes, which are links with other objects that allow objects to have direct references to other objects
- **method** attributes, which represent functions an object can be asked to execute at run-time. Objects have behavior through these types of attributes.

Attributes are typically identified in a program by their names. The attributes that are accessible by other objects are called public; they constitute the public interface of an object. The other attributes are called private.\(^\text{14}\) The public interface is the only feature of an object that is “visible” to other objects and, possibly, other programmers.

---

\(^\text{14}\) We neglect here the distinction between strictly private and “protected” attributes made in some programming languages.
How the attributes are implemented inside an object is hidden. This feature of object-oriented programming is often called *information hiding* or *encapsulation*; it is a major factor contributing to the flexibility of object-oriented programming because a programmer is able to change the implementation of an object’s attributes without having to worry about other objects. From the perspective of programming languages, an object is an instance of an *abstract data type*, where the public interface defines the operations that can be performed on it.

**Object Attributes**

Embedded attributes can be of two types:

- *primitive* as provided by the programming language. An example is the height attribute of an object representing a wall expressed as a real primitive.

- *another object*. An example is the height of a wall expressed by an object with two attributes: *unit* - a string primitive indicating the system of measurement used (like "metric" or "Imperial") and *value* - a real primitive.

To make the most appropriate decisions as to the implementation of attributes is a crucial aspect of the modelling phase in object-oriented programming.

Relational attributes also fall under two categories:

- *part-of, has-a or aggregation* relations that capture objects with constituent parts. The part objects must be deleted when the containing object is deleted. An example is a structural system consisting of a gravity load- and a lateral load-resisting system, which in turn are linked to parts, all of which are represented by objects. It makes no sense to retain the parts of the system if the system overall disappears, unless, of course, the parts are reused in another system; in the latter case, the relation between the system and its components is not a strict aggregation relation in the object-oriented sense.

- *association* relations linking objects with a “life” independent of each other. An example is the link between an object representing an owner with an object representing a building. Clearly, the owner may exist even if the building disappears, or—conversely—the building may exist even if the owner disappears: it may have simply passed to another owner.

These distinctions reflect the semantics of a particular application and must be managed by the application programmer because a typical object-oriented language does not manage these relations, that is, does not know the distinction between an aggregation and association. In other words, a programmer cannot declare that a certain relation is an aggregation or association and then rely on the run-time system to delete or retain objects accordingly.

Methods are attributes that define the behavior of an object, especially how it is able to interact with other objects. An example of a method attribute is the *area* method of an object representing a closed geometric figure (rectangle, general polygon, circle etc.) where *area* returns the size of the area enclosed by the figure, measured in some dimensional units. We will return to methods when we talk about polymorphism.
Object configurations

A collection of linked objects is an object configuration. An example is a collection of linked objects representing a (semi-)hierarchically decomposed structural system (see Figure 2). Note that all links are part-of relations.15

![Diagram of object configuration]

Figure 2: An object configuration made up of aggregation (part-of or has-a) relations

3.1.2 Classes

Unlike objects, which are run-time constructs, a class is a program construct. It defines the attributes a specific type of object can have, both public and private. Classes are also identified by their names. They are the modules of an object-oriented program.

Objects are created at run-time as instances of a class and have precisely the attributes defined by that class:

An object is a class instance.

For example, our application has a Wall class, which has—among others—a material and height attribute. Any Wall instance created at run-time, then, has these same attributes, but may differ in terms of attribute values; thus, we may have one Wall instance that is 8 feet high and made of studs covered with drywall at both sides and another Wall instance made of brick and 4’ high.

Note that in many explanations found in the literature, no distinction is made between a class and its instances, or between classes and objects. That is, a Wall instance or object

15. A more efficient representation could take advantage of the repetitiveness of structural designs, a detail we don’t consider here.
is also called simply a Wall. This is fine if the distinction is well-understood and it is clear from the context what the author means. We also do this later on in order to avoid clumsy formulations. But when the distinction really matters, we will use more explicit language.

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 or each class, a schema defines the universe of discourse for a particular application.

**Class Inheritance**

A class can inherit the attributes of another class. This establishes super/subclass relations between classes or an inheritance hierarchy (sometimes called is-a relations), which is managed by the programming language. Figure 3 shows an example:

![Inheritance Hierarchy Diagram]

Figure 3: Example of an inheritance hierarchy

It is paramount for novices to understand the difference between inheritance and part-of hierarchies, both as modelling and as programming constructs.¹⁶

**Polymorphism**

The public interface of a class is shared by all of its subclasses. An example is the class Graphical Element that defines a public method, area, which is inherited by all subclasses. But each subclass can re-implement the method to suit its particular shape. This makes the area method polymorphic. As a result, an object can query any object for its area by calling its area method as long as it knows that this object is a Graphical Element; it does not have to know what type of Graphical Element it is.

The decision which method to use is made 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.¹⁷

A class that can never be instantiated is called abstract. Why would a programmer create an abstract class? Precisely to define a public interface shared by all subclasses. Again, this is a crucial aspect of object-oriented programming that we cannot elaborate here, but will pay close attention to during model development in the course.

¹⁶. Hint: In our inheritance example, a rectangle is-a polygon, but in the structural system example, a beam is not a frame, but is part-of a frame. The other huge distinction is that inheritance is managed by the programming language, while aggregation must be managed by the programmer.

¹⁷. We cannot elaborate this point here. Interested readers are referred to Meyer (1988).

---

16. Hint: In our inheritance example, a rectangle is-a polygon, but in the structural system example, a beam is not a frame, but is part-of a frame. The other huge distinction is that inheritance is managed by the programming language, while aggregation must be managed by the programmer.

17. We cannot elaborate this point here. Interested readers are referred to Meyer (1988).
3.2 Use Case-Driven Software Development

A use case is a “sequence of actions an actor performs using a system to achieve a particular goal”,\(^\text{18}\) or, in the words of another author, a use case “describes a sequence of actions a system performs that yields an observable result of value for a particular actor.”\(^\text{19}\) We will return in a later chapter to discuss what the terms “actor,” “goal” or “result of value” concretely mean in this context. For the purposes of this introduction, we simply illustrate these concepts by an example, a use case that may be of interest in the context of our application for an actor “designer”:

**Use case: Remove a wall**

The designer indicates to the system that a wall should be removed and selects sequentially the graphical elements that represent the wall in a MS view. After the designer accepts the selection, the system displays the wall in a different style. Internally, it adds the wall to the list of removed elements and assigns it to the removal layer.

Use cases are the primary artefacts of a use case-driven software development process. 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.

The inventors of the method further claim that this process is (i) **architecture-centric** because the system architecture is the primary vehicle to conceptualize, depict, construct, manage and evolve the system under development; (ii) **iterative** because it requires iterations within and between phases (not a waterfall model); and (iii) **incremental** because it is characterized by releases of increasing detail at the end of each phase. Each phase produces a specific form of documentation, which is a major vehicle for the *traceability of the process*. Figure 4 depicts the phases of the process and the products each phase creates.

Note also that the phases depicted in the figure represent a selection of UML methods that we find particularly useful for our course; our selections do not exhaust the options offered by the language.

- **Initial specification.** Based on the recommendation of Rosenberg, 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 tie these specifications to).
- **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*.
- **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.
- **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. The interaction with the previous phase is so tight that the two phases constitute a single phase, in our opinion (which is not shared by some authors): the sequence diagrams, taken together, specify the public interface for each object and, by implication, class. But in order to develop them, we must have an idea which classes are available in the first place, i.e. must know the class diagrams. Conversely, the class diagrams show not only all classes, but also all attributes, including the public ones, which we find through the sequence diagrams. Thus, the two phases combine to spell out the schema of interest for the application.

- **Code.** In this phase, the code for all classes in the class diagrams is produced.

- **Validation.** This phase test the code, use case-by-use case.

The three amigos insist that the object model or 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 could not disagree more with this view, given our experience as designers. Specifically for pedagogical reasons, we must insist 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 artefact is being considered. We find the analogous situation in the software development phases shown above (with the exception of the first one and last one): each produces a design at decreasing levels of abstraction. Thus,

**schema development is design, not analysis.**
4. Course Outline

If we adapt the use case-driven approach to the present course, we arrive at the following course outline:

1. **Initial data collection**: We planned initially to interview practitioners in local architectural firms to gain a better understanding of remodeling projects and to derive realistic scenarios for our application. During the interviews, we realized that the process is so open-ended and fluid that it would be hard to capture in a few representative scenarios. We therefore concentrated more on the drawing conventions observed and on specific methods used for cost estimation and scheduling. We documented these observations in a series of reports that served as a general reference throughout the later phases.

2. **Use case development**: We specified the functionality of the planned application of use cases through several iterations.

3. **Object model or schema design**: Given the use cases, we focused on the objects able to capture the information needed by the use cases in combination. The object model was specified through two types of diagrams, class and sequence diagrams, capturing, respectively, the static and dynamic aspects of the schema. Especially the sequence diagrams have a direct relation to the use cases.

4. **Coding or implementation**: The objects and their interactions were implemented use case-by-use case to produce a first software prototype of the desired application.

5. **Field tests**: We planned to test the prototype in the context of realistic remodeling projects. But for technical reasons explained below, we could not complete the first prototype and were therefore unable to put the entire application to the test.

6. **Documentation**: We documented our process and the results obtained in the present report.

The following chapters describe the work done in phases 1-4 in greater detail.
5. Initial Data Collection

5.1 Application Context

Rosenberg observes—correctly, as we believe—that application development often starts or should start with an initial specification of the overall functionality of a system and context-dependent constraints and requirements that can and should be documented before the use cases themselves get developed. In our case, however, these types of specifications were obvious because the general context of our application and the requirements this context generates were sufficiently clear:

- The application had to be written in JMDL on the platform provided by MicroStation/J and ProjectBank.
- Existing dgn files had to be used to obtain a description of a building to be remodelled, and the application had to extract all geometric data of the building components to be demolished or relocated from these dgn files. Users had to be able to identify these elements through an interactive graphical user interface (GUI) that interacted seamlessly with MS views.
- The geometry of added components had to be specified similarly in MS views.
- If possible, ProjectBank had to be used to store descriptions of the components of interest for a specific project persistently and in parallel to the geometric information saved in the dgn files.

A general scenario describing how the intended application could be used in an architect’s office may look as follows:

**Scenario.** A designated person in an office, whom we call the remodeling project manager (RPM), collects all dgn files that describe a building for which the office has been commissioned to develop a remodeling scheme. We make no assumptions about how these dgn files are created. They may already exist because the same office designed the original building or obtained the files from another firm; conversely, the drawings may be the result of on-site measurements conducted explicitly for this project. The RPM combines all of these files to form a ProjectBank project.

Individual designers involved in the project (who may or may not be identical with the RPM) are allowed to check any one of these dgn files out and modify them directly in MS/J or through our application. The application provides various capabilities useful for these designers, like the automatic generation of remodeling drawings, schedules or cost estimates, which should be attractive enough to them so that they will, in fact, use our application whenever possible.

As this scenario makes clear, the term “project” has two meanings for our application:

- it is a collection files and data that are managed by ProjectBank and constitute a “project” in the ProjectBank sense.
- this ProjectBank project is distinct from the remodeling project, of which it is a part.

5.2 Interviews

Even if the general context of our application was known when we started—to the degree described in the preceding section—many details were not clear. They concerned the terms used by architects in the context of remodeling projects, the drawing conventions used, specific services provided during such a project, methods for accomplishing crucial
subtasks (like how to generate an initial cost estimate) etc. In order to obtain a better understanding of these issues, we decided to conduct a series of interviews with architects that had experience with remodeling projects.

**Process**

Since our time was limited, we decided to conduct altogether four interviews. The first one was conducted by the instructors before the start of class in order to see how such an interview could be handled, how long it would take, which questions to ask etc. During one of the first class sessions, we formed three interview teams and scheduled an interview with a different architect in a different firm for each.

The interviews were meant to be informal, but not completely unstructured, because we knew which topics should be addressed. Each team was therefore given a few guidelines to observe during the interviews. In addition, at least one instructor was present during each interview to monitor its general flow and adherence to the guidelines and to participate actively when this seemed to be needed.\textsuperscript{20}

The guidelines stated the following goals for an interview:

- to contribute to our growing understanding of the chosen application domain. We want to find out specifically
  - a. which objects typically appear in a remodeling drawing and how they are graphically displayed. This includes not only building components, but also annotations, side bars, "bubbles" etc.
  - b. what schedules are generated in addition, how they are generated, and how they look
  - c. the same for specifications
  - d. the same for cost estimates
  - e. any process-related issues that seem relevant, but are not directly covered by the above.
- to give students experience with focused interviews as a data collection method.

After completing its interview, each team had to submit the following items:

- (mandatory) an interview report as an html file by e-mail to the instructor who participated in the interview;
- (highly desired, when obtainable) copies of remodeling drawings and schedules, pages from specifications and cost estimates used for a real remodeling project;
- (optional) an interview transcript, again by e-mail (.html or plain ASCII text file).

**Results**

Each of the interview reports consisted of the following sections:

\textsuperscript{20} Our students were generally shy, no doubt in part because most of them were foreigners, and reluctant to take on a leadership role during the interviews. As a result, the instructors ended up participating more actively than we had anticipated.
1. an introduction describing the interviewee, the firm he or she was working in, and the remodeling projects considered during the interview;
2. one or more tables listing the different types of building components discussed and their graphical representation;
3. a description of the method(s) used for cost estimation and a real sample;
4. samples of schedules and specifications;
5. a concluding section highlighting specific findings and insights.

Figure 5 shows as an example for 2 a portion of a component table from an interview report.21

The following are the most pertinent general conclusions we drew from the interviews:

- The graphical conventions for depicting elements in a drawing for a remodeling project are not standardized across firms. However, the variations we found are within a relatively narrow band because contractors and other parties have to understand the drawings regardless of whether they have worked with the architect before or not. A typical set of drawings has on the first page a legend explaining the specific conventions used.

- Objects to be added, like sanitary fixtures, are depicted in remodeling drawings like in drawings for new construction with one exception: load-bearing walls and partitions that are to be added are clearly distinguished in floorplans from existing elements by shading or poché.

- Except for projects in which demolition is a major task, components to be removed and added on a floor or in an elevation are shown in the same drawing.

- Annotations are a crucial component and are typically placed next to the elements they refer to. But they can crowd the graphical elements, in which case they are placed in side bars, with brief references or keys placed next to the depiction of the elements they refer to. Again, if special conventions are used across several drawings, they are explained in a separate legend on a sheet preceding the drawings.

- Changes to the initial design are more frequent in renovation and remodeling projects because of unforeseen conditions that emerge once demolition has started. If such changes occur after a set of drawings has been accepted, gone out to bid etc., the changes have to be explicitly identified. It is usual to do this using “bubbles” or “clouds.” Each such change must be listed in the title box. CAD-based drawings pose a special challenge for the drawing of such clouds: the draftsperson very often misjudges the scale at which the final drawing will be printed because the screen resolution tends to be larger than in the hardcopy, and the bubbles will appear to small in it. As a result, changes are often added in pencil and by hand. This, of course, makes the information inaccessible to a computer program. The management of these changes and their depiction by a CAD system are an interesting special issue that we could not pursue further in the course.

- Cost estimation for elements to be added follows more or less the practice used to estimate cost for new construction. No general method appears to

21. The entire report can be found on the course web site at http://www.andrew.cmu.edu/course/48-756/
exist for estimating the cost of elements to be removed or demolished: contractors base their estimates often on rough guesses about the man-hours involved (based on a site visit), rather than on accurate quantity take-offs. This type of estimation is very difficult to do up-front in an office.

- Clients may nevertheless be interested in an estimate before bidding starts in order to determine if the project is worth pursuing in the first place. Architects therefore have to deal with the uncertainties of cost estimation and rely on a variety of tactics to come up with reliable figures. Experience with similar

---

### Figure 5: Component table from an interview report

<table>
<thead>
<tr>
<th>OBJECTS</th>
<th>GRAPHICAL REPR.</th>
<th>EXAMPLE ANNOTATION IN DRAWING</th>
<th>EXAMPLE ANNOTATION IN SIDE BAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>Walls - existing</td>
<td></td>
<td>(refer to Table 3 for more e.g.)</td>
<td>(refer to Table 3 for more e.g.)</td>
</tr>
<tr>
<td>Existing Walls</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Walls - to be demolished</td>
<td></td>
<td>remove exg. wall</td>
<td>1.9 REMOVE WALL (where 1.9 refers to a member attached to an annotation in text)</td>
</tr>
<tr>
<td>Walls to be removed</td>
<td></td>
<td>1.9</td>
<td></td>
</tr>
<tr>
<td>Walls - to be added</td>
<td></td>
<td>6.11 FILL IN W/WOOD STUDS TO MATCH EXIST.</td>
<td>5.3 5/8&quot; METAL STUD FURRING @ 24&quot; O.C.</td>
</tr>
<tr>
<td>Door opening to be filled in</td>
<td></td>
<td>fill exg. windows</td>
<td></td>
</tr>
<tr>
<td>Window opening to be filled in</td>
<td></td>
<td></td>
<td>9.2 5/8&quot; MOISTURE PROOF G/F</td>
</tr>
<tr>
<td>Wall finish to be added</td>
<td></td>
<td>5.3</td>
<td></td>
</tr>
<tr>
<td>New stud wall with finishing</td>
<td></td>
<td>9.2</td>
<td></td>
</tr>
</tbody>
</table>
projects helps as well as good relations with contractors that can be called and asked.

- We got mixed reactions with respect to the automated creation of schedules. One interviewee saw a clear benefit, at least for larger projects, where a door schedule, for example, may contain many doors on many floors. A mistake in a schedule that results in a door missing at the time it is needed is really harmful, and automating the creation of such schedules may eliminate such errors. Another interviewee, however, commented that current CAD systems already provide this capability; but in order to take advantage of it, every element to be included in a schedule has to be properly tagged and its attributes set, an effort that nobody in his firm is willing to undergo because the general impression is that it exceeds the effort required for generating a schedule by hand.

The most important lesson we learned from the interviews was that our application has to give users some flexibility in selecting drawing conventions and the handling of annotations. The challenge is to design a system that provides this flexibility while automating enough tasks to make it attractive to users.

We also conclude that the automation of quantity take-offs that can be used for various purpose may be useful.
6. Use Cases

6.1 Use Case Modeling

We quoted in section 3.2 two use case definitions that can be paraphrased as follows:

A use case is a meaningful task that an actor can execute using the system.

Several terms in this definition need further explanation. An actor indicates a specific role a user may play in using the system, where a role is a collection of related responsibilities usually based on specific expertise or expectations. Examples of actors in the context of our application are the following:

- draftsperson: creates drawings
- cost estimator: creates cost estimates based on quantity take-offs from drawings; maintains cost data
- specification writer: writes specs; manages standard specification text library.

Note that a single person may play multiple roles.

A meaningful task 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 action 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 the Place Line command in Microstation, which determines a specific task MS users can set for themselves. This is a meaningful task consisting 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 in any way 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.

More generally, what is 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 use case example we provided in section 3.2:

Use case: Remove a wall

The designer indicates to the system that a wall should be removed and selects sequentially the graphical elements that represent the wall in a MS view. After the designer accepts the selection, the system displays the wall in a different style. Internally, it adds the wall to the list of removed elements and assigns it to the removal layer.

Use cases are written in plain English in the actors’ language and from their perspective. They should be written in the present tense and active voice. The formulations should not anticipate any down-stream decisions, that is, they should focus on what the system does, not how it does it.

This is a major disagreement we have with Rosenberg’s recommendations, who suggests having a first cut at an object model before use cases get formulated and changing the use case text based on the evolving object model, even adjusting the GUI to reflect internal attribute names (!). This completely violates the principle that use cases (i) should be written in the actors’, i.e. users’ language, which in all likelihood does not contain computer jargon, and (ii) are the main vehicle to discover the classes of interest. It also...
flies in the face of one of the defining principles of object-oriented programming, information hiding: if classes are not supposed to know details about each others’ internal workings, why should users know them?

A good rule to follow in formulating uses case is to consider the text of all use cases combined the first version of the User Manual for the system. We agree with Rosenberg that terseness is not a virtue: everything should be made as clear and explicit as possible.

Within a use case document, the use case descriptions usually adhere to a common template or boiler plate. However, no generally accepted format exists. Rosenberg recommends just two sections: basic course of action and alternate courses to indicate exceptions. Following this recommendation, our sample use case can be rewritten as follows:

**Use case: Remove a wall**

**Basic course:**
The designer indicates to the system that a wall should be removed and selects sequentially the graphical elements that represent the wall. After the designer accepts the selection, the system displays the wall in a different style. Internally, it adds the wall to the list of removed elements and assigns it to the removal layer.

**Alternate course:**
If the designer does not accept the selection, the system aborts the entire action and changes nothing internally.

Rosenberg discourages stating pre- and postconditions because they add words and often state the obvious. However, some SEED teams have a more positive experience, and following their lead, we allowed pre-conditions as optional specifications when they are not obvious.

We added to this a few additional rules:

- Start a new line every time the active participant (actor vs. system) changes during the execution of a use case. This makes the interaction between actor and system very clear and produces a layout that can be nicely incorporated into sequence diagrams (see Chapter 7.2).

- Precede the detailed explanations with a brief statement that summarizes what the use case accomplishes.

- The use case text may end with an optional comment section.

Later sections of this chapter provide ample illustrations for use cases formulated using our template.

The recommendation to avoid terseness may result in verbose documents with many repetitions when different use cases have subsequences that also occur in other use cases. In such situations, it may be advisable to factor out the commonalities between use cases. To take an example from MS, identifying a datapoint precisely is required for many tools. It would indeed be awkward if the operations involved had to be stated explicitly whenever this action is needed. We can factor out these repeated actions by formulating a separate use case (identify a data point) and refer to it in other use cases.
A use case can have one of two basic relations with another use case that factors out repeated actions:

- **precedes**. Example: two use cases start with the same action, like selecting an option from a pull-down menu, and then diverge depending on the choice.
- **invokes**. Example: sequential selection of elements followed by an accept is needed for many actions.

Other factoring situations are hotly debated.²²

UML allows for the creation of use case diagrams, which can be put together from the elements shown in Figure 7.

Figure 6: Basic relations between use cases

- **precedes**.
- **invokes**.

Figure 7: Elements of use case diagrams

Are UC diagrams useful? In our opinion, only if enough factorization takes place so that a graphical depiction helps readers understand the resulting relations between use cases.

UC descriptions are iteratively refined during use case modeling and through feedback from subsequent phases. In parallel, the developers may produce sketches to capture the

²². see Rosenberg’s very funny appendix.
evolving GUI design and start to create an evolving glossary of terms. The goal is to arrive in the end at an explicit, complete description of the system’s functionality from an user perspective including all alternate courses of action and based on a concrete GUI design.

6.2 Key Terms and Concepts for Our Application

We call our application the Remodeler (RM). In order to distinguish a ProjectBank project (PBP) from a remodeling project, we refer to the latter explicitly as a Remodeling Project (RP). We call the person managing a remodeling project overall in the architect’s office the Remodeling Project Manager (RPM). We call any architect or draftsperson working on a remodeling drawing within an RP an Architectural Designer (AD). The RPM and AD are the main actors we are concerned with. For specific use cases, we need furthermore an actor Cost Estimator and an actor Schedule Preparer.

We abbreviate MicroStation/J as MS-J. To remain consistent with established MicroStation terminology, we call an individual element in a dgn file (line, curve, text, cell etc.) an element. We distinguish the RM-created components from these elements and never use the term “element” for “component” in this sense.

RM’s understanding of a building as a constructed artefact is restricted to three types or classes of components: removed components, added components, and recycled components. Removed components are building parts that are to be demolished, that is, end up in a dumpster. Added components are parts that are added, like in new construction. Recycled components are parts that are removed from their present location, but not demolished; rather, they are put back in a different location (for example, a door). We assume that the RM maintains persistent lists of instances of the three component types across sessions. We call these lists the Removed List, Added List, and Recycled List, respectively.

**Project settings** determine—for the dgn elements depicting the three types of components identified above—their element attributes: level, color, line width and line style (“element symbology” in MicroStationese). Our assumption is that a remodeling project should depict components to be added or removed consistently within and across drawings. The best way to assure this is to determine these settings at the start of the project and to save them independently as project data that can be used by RM as needed. At the same time, this provides the flexibility that seems to be desired by the architects we interviewed.

We include level settings among the project settings to allow elements to be “moved” to different levels depending on their role in the remodeling process; this would make it easy, for example, to generate drawings showing only existing and demolished walls etc.

Schedules, cost estimates and other derived data are generated by RM directly from this information. Drawing elements (lines, annotations, etc.) that do not represent any of these components are not recognized by RM. We may at some later time add a capability for users to create also existing components, that is, interpret an entire drawing in terms of building components.
6.3 Group 1: Session-Related Use Cases

1.1 Start Remodeler

The AD or RPM starts the RM in order to work on a remodeling drawing.

Preconditions:
1. A PBP has been created.
2. The appropriate dgn files have been copied and imported into the PBP.
3. One of these dgn files is checked into the current briefcase and open in a MS-J session.

Basic Course:
1. The AD issues the “Remodeler” command from MS-J key-in window.
2. The RM opens the RM commands box. Internally, it checks if a RP is already defined for this drawing. If it is, RM retrieves the project from the PB data store and makes it available in working memory; otherwise, it creates a new RP.

1.2 Edit Project Settings

The AD or RPM changes the current project settings.

Preconditions:
1. The RM is active.

Basic Course:
1. The AD or RPM issues the “Edit Project Settings” command.
2. The RM opens the “Project Settings” dialog box.
3. The actor inspects the current settings and edits them as desired.
4. The actor commits the changed settings.
5. The RM saves the changed settings.
6. MS-J changes the display of the Remodeler components according to the new settings.
7. The actor closes the dialog box.
1.3 Save Project

Upon the AD’s request, the RM saves the current state of the project in the PB store.

Preconditions:
1. The RM is active.

Basic Course:
1. The AD issues the “Save Project” command.
2. The RM saves the current state of the project in working memory in the PB store.

1.4 Close Remodeler

Upon the AD’s request, the RM exits.

Preconditions:
1. The RM is active.

Basic Course:
1. The AD issues the “Close Remodeler” command.
2. If the AD has made changes after the last save, the RM opens the “save” message box.
3. The AD selects one of three options save, don’t save or cancel.
4. Depending on the AD’s response, the RM does or does not terminate with or without a save. If the RM terminates it also closes the RM commands box.

6.4 Group 2: Component-Related Use Cases

2.1 Remove an Entire Component

The AD designates a component for demolition. “Entire” in the context of this use case must be understood strictly in terms of dgn elements: no element has to be cut or divided in order to represent the component accurately.

Preconditions:
1. The RM is active.

Basic Course:
1. The AD issues the “Remove an Entire Component” command from the RM commands box.
2. The RM opens the “Remove Component” (RECdbox) dialog box.
3. The AD selects the component type, attributes and enters the annotation for the component and commits.
4. The RM prompts the AD to select graphical elements that represent the component.
5. The AD selects the elements in an MS-J window and accepts.
6. The RM creates the corresponding removed component and stores it internally.
7. The MS-J window updates the display of the component.
8. The AD closes the RECdbox.

Alternate course 1 (Remove Finish):
If the AD selects component type “Finish” in step 3, the use case continues as follows:
4. The RM opens the “Remove Finish” dialog box (RFdbox).
5. The AD chooses ‘vertical finish’ or ‘horizontal finish’ in the RFdbox and enters the annotation for the finish. (We assume the current drawing is a plan.)
6. If the AD selected ‘horizontal finish’, the RM prompts the AD to identify all data points to define the area of the removed finish.
Otherwise, the RM prompts the AD for the height of the area and to identify two points to indicate the width of the area.
7. The use case continues with step 6 of the basic course.

Alternate Course 2:
If the AD selects the graphical elements before issuing the remove component command, the AD skips steps 4 and 5 in the basic course.

Comments:
We assume that the AD knows the component to be removed before-hand.

2.2 Add a Component

The AD defines a component to be newly constructed.

Preconditions:
1. The RM is active.

Basic Course:
1. The AD issues the "Add Component" command in the RM commands box.
2. The RM opens the "Add Component" dialog box (ACdbox).
3. The AD selects the component type, attributes and enters the annotation for the component and commits.
4. The RM prompts the AD to draw or place the component.
5. The AD draws or places the component in the MS-J window and accepts.
6. The RM creates the corresponding added component and stores it internally.
7. The MS-J window updates the display of the component
8. The AD closes the ACdbox.

Alternate course (Add Finish):
If the AD selects "Finish" as component type in step 3, the use case continues as follows:
4. The RM opens "Add Finish" dialog box (AFdbox).
5. The AD chooses 'vertical finish' or 'horizontal finish' in the AFdbox, enters the attributes of the finish and the needed annotation. (We assume that the current drawing is a plan.)
6. If the AD selected 'horizontal finish', the RM prompts the AD to identify all data points needed to calculate the area of the added finish. Otherwise, the RM prompts the AD to enter the height of the finish area and two datapoints to indicate its width.
7. The AD identifies data points and accepts.
8. The use case continues with step 6 of the basic course.

2.3 Remove Parts of a Component

The AD removes parts of a component; i.e. only parts of dgn elements become a component removed by the RM.

Preconditions:
1. The RM is active.

Basic Course:
1. The AD issues the “Remove Parts of a component” command in the RM commands box.
2. The RM opens the “Remove Parts of a Component” dialog box (RPCdbox).
3. The AD selects the component type from the components list displayed in the RPCdbox.
4. The AD selects the desired settings, like materials and graphical representations, for the removed parts of the component and enters the needed annotation for the removed part of the component.
5. The RM prompts the user to select between remove by fence or remove by points.
6. The AD makes a selection, which either way has to stay inside the existing component area.
7. The RM validates the selection entered and prompts to select again if the selection set is not part of a dgn element.
8. If the selection set is valid, the RM adds the removed component parts to the Removed List and changes their symbology.
9. MS-J re-displays the component parts with the new symbology.
10. The AD closes the RPCdbox.

Alternate Course:
1. The AD edits in MS-J the parts of the component to be removed before-hand (e.g. cuts continuous lines) so that the removed part consists of complete dgn elements.
2. The AD issues the “Remove an Entire Component” command. The use case continues with the “Remove an Entire Component” use case.

Comments:
This use case is applicable only when the AD needs to remove parts of dgn elements. If entire components are to be removed, “Remove an Entire Component” should be used. The alternate course indicates an option that we can fall back on if the basic course turns out to be too difficult to implement.

2.4 Put Back Removed Component

The AD puts an already removed RM component back into the drawing as a dgn element.

Preconditions:
1. The RM is active.
2. Parts of at least one component or at least one entire component have been removed through the RM and stored in the Removed List.

Basic Course:
1. The AD issues the “Put Back Removed Component” command in the RM commands box.
2. The RM opens the “Put Back Removed Component” dialog box (PBRCdbox).
3. The PBRCdbox displays a list of components removed through the RM and prompts the user to select component(s).
4. The AD selects the component(s) to be put back and accepts.
5. The RM deletes the selected component from the Removed List and restores its previous symbology.
6. MS-J re-displays the component in its original symbology.

Alternate Course:
1. The AD opens the ProjectBank Explorer and asks for a history display.
2. ProjectBank Explorer displays the history of actions committed.
3. The AD selects the item in the list where the component to be put back has been removed.
4. The ProjectBank Explorer restores the file to the state where the component has not yet been removed.
5. The AD commits.

Comments:
In principle, ProjectBank should be able to work seamlessly with the Remodeler as to recording the activities done through the Remodeler to a dgn file. But note also that this course is sensitive to the sequence in which components were removed; i.e. it is impossible to restrict the alternate course to components not removed in sequence.
2.5 Remove Added Component

The AD modifies the drawing by removing components that have been added earlier during the current or previous RM sessions.

Preconditions:
1. The RM is active.
2. The component to be removed has been added through the RM with use case 2.2.

Basic Course:
1. The AD issues the “Remove Added Component” command.
2. The RM prompts the AD to select an added component.
3. The AD selects the component in the MS-J window and accepts.
4.1 If the component is not an RM component, the RM prompts the user again to select an added component.
4.2 If the component is an RM component, the RM removes the dgn elements representing the added component from the design file and removes the added component from the Added List.
5. MS-J refreshes the view(s).

Comments:
If the added component’s location is a demolished (removed) component location, the removed component should reappear. We should check if we can use MS-J standard delete function instead of writing our own to accomplish this use case.

2.6 Recycle a Component

The AD edits the currently open RP drawing by changing the physical location of a component represented by dgn elements and/or Remodeler components.

Preconditions:
1. The RM is active.

Basic Course:
1. The AD issues the “Recycle Component” command.
2. The RM opens the “Recycle” dialog box (RCdbox), which shows a list of removed components.
3. The AD is prompted to select a component from the list or select a component in the drawing.
4. If the AD chooses by list, the AD selects the component in the list, and the use case continues with 2.6.2, else the use case continues with 2.6.1.

Comments:
The difference between the two options is that selection from the list requires a previous designation of the component as a recycled component without immediate relocation, while selection in a drawing starts from scratch. We made this distinction in order to enable an AD to move a component from one drawing to another one, for example, when a door has to be moved from one floor to another.

2.6.1 Recycle Component in Drawing

The AD designates a component for recycling and, possibly, relocates it.

Preconditions:
1. The AD has issued the “Recycle Component” command.
2. The MS-J elements representing the component in question are not part of any Remodeler component.
3. The AD has selected the “select component in the drawing” option in the RCdbox.
Basic Course:
1. The RCdbox opens the “Annotation” dialog box (ANdbox).
2. The ANdbox displays the component types.
3. The AD chooses the component type and adds an annotation in the ANdbox.
4. The AD selects the dgn elements representing the component in the MS-J window and accepts.
5. The RM creates the component and adds it to the Recycled and Removed Lists.
6. The RM displays the component with new symbology.
7. If the AD chooses to insert the component later, the use case ends here, else it continues with 2.6.2

Comments:
‘Recycle’ implies two actions: remove and add, which may not necessarily be consecutive; i.e. the AD may remove a component in one session and place it somewhere else in a different session.

2.6.2 Recycle Component from List

Preconditions:
1. The AD has issued the “Recycle Component” command.
2. There are already removed components in the list.
3. The AD has selected the “select component from list” option in the RCdbox.

Basic Course:
1. The AD browses the Recycled List, which shows components that have been removed, but not yet been re-inserted elsewhere, and selects a component.
2. The RM asks for the new location of the component.
3. The AD selects a location for the recycled component and accepts.
4. MS-J displays the component at its new location with the appropriate symbology.

Comments:
1. The new location of a recycled component can be in a different design file from the original one.
2. Since walls and finishes can not be recycled, only recycled doors are considered.

2.7 Edit Component Attributes (Display Component Attributes)

The AD changes the attributes for a previously placed component.

Preconditions:
1. The RM is active.

Basic Course:
1. The AD issues the “Edit Component Attributes” command.
2. The RM prompts the AD to select the component in an MS-J window or from a list of existing components in the current RP and accepts (the AD can select only one component).
3. The RM opens the “Component Attributes” dialog box (CAdbox) and displays the attributes of the component.
4. The AD changes the desired attributes and accepts.
5. The RM updates the changed attributes.
6. The AD can select another component and repeat the previous 3 steps.
7. The AD closes the CAdbx.
8. The RM displays the components according to the new attributes.

Alternate course:
The AD selects the components to be edited before issuing the command.
2.8 Highlight Components

Upon the AD’s request, RM highlights selected components.

Preconditions:
1. The RM is active.

Basic course:
1. The AD issues the “Highlight Component” command in the RM commands box.
2. The RM opens the “Highlight Component” dialog box (HCdbox) and displays added, removed and recycled components in the current RP in a selection list.
3. The AD selects the component to be highlighted and accepts. (The AD can select more than one component).
4. The AD closes the HCdbox.
5. The RM changes the display in all MS-J views according to the “highlight” symbology. (The symbology can be edited from the “project settings” dialog box with use case 1.3).

Alternate course:
1. The AD selects—in an MS-J window—the components to be highlighted before issuing the command.
2. The AD issues the “Highlight Component” command in the RM commands box and skips the rest of the steps.

Figure 9: Preliminary user interface sketch for use case 2.6
Comments:
We assume that the AD can issue all other RM commands while the “Highlight Component” command is active; i.e. the dialog box remains on display. When the AD issues the “Highlight Component” command again while the command is active, he/she can highlight additional components by selecting them in the component list.

2.9 Eliminate Highlights

Upon the AD’s request, RM eliminates the current highlights.

Preconditions:
1. The RM is active.
2. A component is highlighted.

Basic Course:
1. If the HCdbox is not open, the AD opens the HCdbox by issuing the “Highlight Component” command.
2. The AD clicks on the ‘Eliminate Highlights’ button in the HCdbox.
3. The RM reverts the display in all MS-J views to the original symbology and resets the active component to null.
4. The AD closes the HCdbox.

Comments:
In case anybody wonders why this is a separate use case from 2.8, we have to remind readers that use cases represent steps that must be executed together. Since we allowed in 2.8 for the HCdbox to remain open, the steps in the present use case are not required to complete 2.8: they can be done at any later time. Therefore, the present use case is a use case in its own right.

2.10 Add Annotation

The AD adds annotations to elements in a drawing.

Preconditions:
1. At least one RM component has been generated.

Basic Course:
1. The AD issues the “Add Annotation” command in the RM commands box.
2. The RM displays the “Add Annotation” dialog box (ANdbox).
3. The AD selects an annotation placement option (in drawing or in side bar).
4. The RM prompts the AD to select the component to add annotation to.
5. The AD selects an RM component and accepts.
6. The AD writes the annotation and accepts in the MS-J window the appropriate placement.
7. The RP creates a new annotation and associates it with the selected component.
8. MS-J displays the new annotation in the drawing.

Alternate Course:
The AD selects the component to add annotation to after writing the annotation.

Comments:
1. We assume in general that annotations can be associated only with RM components, not with pure dgn elements. The present use case covers the case where an appropriate annotation was not added when the component in question was created. It is needed because if the AD adds annotation text using MS-J
commands only, the RM will not be aware of it. However, if an annotation already exists, the AD can edit it directly in the MS-J window.

2. This use case can be extended to cover the case where the AD selects a text in MS-J and turns it into an RM annotation. Note that not any text in MS-J is an annotation.

2.11 Switch Annotation Style

The AD changes the annotation style from in drawing to side bar or vice versa.

**Preconditions:**
1. The RM is active.
2. There is at least one annotation in the drawing file.

**Basic course:**
1. The AD issues the “Switch Annotation Style” command.
2. The RM prompts the AD to select the annotation(s) in an MS-J Window.
3. The AD selects the annotation(s) and accepts.
4. The RM switches the annotation display in the drawing.
5. The AD may interactively move the annotation to a more desired location.
6. MS-J updates the view.

**Alternate course 1:**
1. The AD issues the “Switch Annotation Style” command.
2. The RM opens the “Annotation Style” dialog box (ASdbox).
3. The AD selects a component type the annotations of which are to be switched.
4. The RM changes the style of only the annotation attached to all components of the selected type.
5. The AD closes ASdbox.

**Alternate course 2:**
The AD selects the annotation the style of which is going to be switched before issuing the command.

6.5 Group 3: Computation-Based Use Cases

3.1 Prepare Quantity Take-Off

Upon the AD’s request, the RM calculates the quantities of the components known to the RM, sorted by type. (This quantity take-off can be used for cost estimation, project management, etc.)

**Preconditions:**
1. The RM is active.

**Basic Course:**
1. The AD issues the “Prepare Quantity Take-off” command in the RM commands box.
2. The RM opens the “Prepare Quantity Take-off” dialog box (PQdbox). A list of components supported by the RM is displayed.
3. The AD selects the types of components for which the RM should calculate quantities.
4. The RM calculates the quantities of the selected components and displays the results in a special display window.
5. The AD accepts.
6. The RM saves the results in a file from which a hard copy can be produced (in some common format like html or a spreadsheet).
7. The AD closes the PQdbox.
Alternate Course:

The AD selects the components for which quantities should be calculated before issuing the command.

3.2 Prepare Schedule

Upon AD’s request, the RM prepares the component(s) selected in a schedule format either as an html file or to be placed in the drawing file.

Preconditions:
1. The RM is active.

Basic Course:
1. The AD issues “Prepare Schedule” command in the RM commands box.
2. The RM opens the “Prepare Schedule” dialog box (PSdbox).
3. The AD selects the component type(s) for which RM should prepare a schedule from a selection list showing the component types for which there are instances in the current RP.
4. The AD selects one of two options to place the schedule, in an html file or in the drawing.
5. If the AD selects “in drawing” option the RM displays a tentative outline of the schedule in the currently active MS-J display.
6. The AD places the outline of the schedule in a desired position on the drawing by dragging it and accepts.
7. The RM completes the schedule and closes the PSdbox.

Alternate Course:
If the AD selects the option “in HTML” at step 6 an html file is generated and opened for the AD to review. The use case skips to step 9.
7. Object Model

7.1 Static Object Model I

7.1.1 Overview

The use cases lead to a first design of the static object model, which shows the classes of interest known so far and their relations (inheritance and all associations and aggregations). This first model does not show any public class interfaces or internal attributes.

We used RationalRose to document our class diagram. The symbols used go back to Rumbaugh's OMTTool and are summarized in Figure 10.

![Class diagram notation]

Figure 10: Class diagram notation

Our object model underwent several modifications, based both on insights gained from the sequence diagrams and in response to technical issues that emerged only during coding. Figure 11 shows the final version of those parts of the static model that do not represent user interface objects. Time and other resource limitations prevented us from generating a complete class model that would include all user interface objects.

7.1.2 Classes

We describe below the classes shown in the figure and the design principles that led to their creation.

RP (Remodeling Project)

This is the central class in the model. It can be instantiated to represent a specific remodeling project corresponding to a ProjectBank project as outlined in the preceding chapter. All objects generated during the course of the project are linked in some form to this RP instance so that, when the instance is saved in the ProjectBank store, all objects linked to it are also saved in that store. In this way, the ProjectBank store is not only used to save all persistent data, but also in a fashion that makes the management of the store easy: overtly, saving and retrieving the data in a remodeling project can be restricted to saving and retrieving a single object. A prerequisite for this to work in the context of ProjectBank is that all of the persistent objects are JMDL schema objects (that is, instances of JMDL schema classes) and the containers we use are also JMDL schema containers.

Figure 11: Partial class diagram
Grey boxes indicate abstract classes.
A RP collects the data about a specific remodeling project through the following linked objects:

- **three Settings** objects containing, respectively, the project settings for added, removed and recycled components
- **three collections** that contain, respectively, the added, removed and recycled components themselves; they are the Removed, Added and Recycled Lists mentioned above. We decided to separate the components into these three collections (as opposed to one) because it provides a pre-sort that facilitates operations like the creation of a door schedule.

**Settings**

This is a simple data repository for project settings in terms of level and symbology.

**RMComponent**

Instances of this class are the objects manipulated directly by the users of the application, or more specifically, the architectural designers involved in the remodeling project. The class itself is abstract and must be specialized into all practically interesting component types that may occur in a remodeling project. We decided to introduce this specialization because different components may have different attributes. For example, a wall may need only a material attribute, while a door may need several attributes in addition (hinge side, hardware, frame type etc).

We decided to deal explicitly in the course with only a small subset of these concrete components:

- **Walls.** These components raise some of the more intricate problems for geometric reasoning posed by our application and have the most varied depictions in drawings.
- **Doors.** We selected doors because—unlike walls—they are components that appear frequently in schedules or can be recycled.
- **Finishes.** These components are interesting because very often, they are indicated in a drawing only by annotations attached to other components and derive their geometry from these components.
- **Annotations.** They are clearly drawing components with their own geometry and settings. But they have meaning only in association with physical remodeling components.

We believe that other components could be handled by a combination of techniques developed for the selected types so that adding more component classes would not enlarge our students’ learning experience.

The (abstract) RPComponent itself contains all data common to all component subclasses. Specifically, it is linked to

- an **ElementGeometry** object, a collection of references to elements in dgn files that are part of the ProjectBank project to which the remodeling project belongs
- an **RMRole** object.
This design reflects the principles of composition and delegation embedded in the design patterns made popular by Gamma and his collaborators, which by now have become widely accepted in the object-oriented community.\(^\text{24}\) In our schema, these principles are reflected in the separation between a RM component and its geometry, on the one hand, and its “role” in the project, on the other hand. We will explain below the reasons behind each separation because they lead to the heart of our schema.

The separation between a component and its geometry is based on the following observation. Clearly, we need to know the geometry of components to draw them or to compute quantities etc. But we cannot predict at the outset how, for example, walls are shown in an existing dgn file; they may be drawn as two parallel lines, as a polygon (line string), as a multi-line etc. As a result, we cannot create component classes simply by subclassing dgn package classes; we would need an open-ended set of subclasses for each component and, because we cannot have multiple inheritance, incur a major code management problem. This is precisely where the design patterns come into play. They indicate not only how multiply inherited behavior can be implemented in languages allowing only for single inheritance, but that the resulting code is more robust, modular, reusable and extensible, that is, achieves exactly the benefits object-oriented programming is expected to create in the first place.

Our RMComponent illustrates this. By factoring out the geometry of a component and collecting all dgn elements in an otherwise unspecified collection, we allow for multiple geometry representations under single inheritance. Restrictions that may apply to individual components can then be implemented for each component. For example, a wall may make sure that no element enters its geometry that is a circle. Component classes can also offer specific computations on their geometry; for example, a wall or finish may be able to compute its area, which would be meaningless for a sink.

The separation between a component and its role is based on similar considerations. The physical attributes of a component do not vary if it is removed or added, but we nevertheless have to know the distinction in order to draw the component with the proper settings, include it in various lists and schedules etc. We discussed in class three ways in which this distinction can be captured:

- by membership in one of the three collections associated with a RM Component. This is sufficient to maintain the distinction and comes at no extra cost because we decided to create these lists anyway as a presorting device. But this design is also very fragile and works in one direction only; that is, we cannot ask an object directly if it is added, recycled or removed.

- by attaching to each component a role tag that can be used to put the object in the correct list and to query its role directly. This is easy to implement and works, provided no specific attributes or behaviors result from specific roles.

- by creating a new RMRole class, specialized into the three subclasses of interest, and associating an RMRole instance with each RMComponent. This allows not only for direct role-related queries, but also for the specialization of attributes and behavior according to the role.

We selected the last option because there are indeed indications for specialized attributes, if not behavior. For example, we allow a recycled door to be put back into its old position. We consequently have to save not only the new, but also the old geometry for that door or any other recycled component. An added component, on the other hand, has no old

\(^{24}\) Gamma et al. (1995)
geometry. This distinction is easy to capture if the RMRoles are full-fledged objects in themselves.

**SessionManager**

A single SessionManager object is instantiated whenever our application starts. It manages the session in the sense that it creates the RM commands box and mediates the information exchange between the GUI and the RP. The separation between the two is absolute in our model: no interface object is allowed to interact directly with an RP or RPCOMPONENT and vice-versa. The SessionManager is not persistent and written in pure Java. This enables it also to mediate between information collected from dgn elements (via dgn package wrappers) and the RP. The classes used in the GUI are also pure Java classes and written using common Java interface toolkits (AWT, Swing).

The SessionManager is very much a control object as envisaged by the **Model/View/Control (MVC)** distinction associated with object-oriented programming, which our schema design also adheres to. The goal again is greater encapsulation, modularity and reusability of the code.

### 7.2 Dynamic Object Model

The dynamic object model identifies for each use case which class instances and actors are involved and how they interact (which public interface feature they rely on when) **at run-time**, that is, when the use case is executed. It is for this reason that the view of the object model depicted by sequence diagrams is called **dynamic**. A sequence diagram refers to a specific use case, as opposed to class diagrams that cover the entire application across all use cases. Figure 12 shows the symbols used in sequence diagrams.

![Figure 12: Sequence diagrams - notation](image)

We must mention here that our “domain objects” are called “entity objects” in the use case and UML literature. We felt compelled to deviate from this usage because every object is an entity, that is, adding “entity” to “object” does not qualify it at all. The entity objects are really the objects representing the model in the MVC scheme of things. They are also very often the only objects that are persistent across sessions. We use the term “domain object” because the term “model” is used differently and more comprehensively in the UML world.

Figure 13 shows a sequence diagram for use case 2.1, Remove an Entire Component. Observe that the diagram contains for each actor and object involved a vertical, dashed line, sometimes called a “swim lane”. A solid rectangle in that lane indicates a period of time during which the corresponding object is active, that is, has part of its public interface invoked by another participant and reacts to it. We introduced in addition the convention to
repeat the use case text in the left margin so that particular interactions can be associated with particular portions of the use case.

The general point about sequence diagrams is that, taken together, they determine the public interfaces to be provided for all classes, give strong indications as to the internal attributes a class should have, and document precisely how each use case should be implemented through object interactions. Note also that the diagrams include all view or user interface objects; that is to say, we cannot develop full sequence diagrams without a GUI design that shows at least the types of windows and widgets involved, if not their exact layouts.

We found sequence diagrams a very valuable tool for object-oriented design. We believe specifically that they are effective learning tools for developers that are used to other paradigms because the diagrams force them to conceive of the entire application in terms of object interactions before the first line of code is written. Similar benefits occur, of course, for novice programmers, who also have to learn to think in terms of interacting objects.

In complex applications, one has to expect that work on sequence diagrams leads to revisions of the initial object model and vice-versa. In the end, the actions depicted in the diagrams define exactly the public interface that each object must support so that the application can support all use cases.

7.3 Static Object Model II

The sequence diagrams determine the public interfaces for all classes and give strong indications for possible internal, private attributes. We can use this information to refine our class diagrams by adding public and private attributes and methods to the class definitions. If we use RationalRose to do this, we obtain diagrams as the one shown in Figure 14.

At this point, the object model has been specified to a degree that allows for the automated creation of source code files containing all class and attribute definitions and methods or functions with empty bodies. It is the role of the programmer to fill in the details.

Coding then is almost anticlimactic because the interesting decisions as to the overall architecture of the application have been made.
Figure 13: sequence diagram for use case 2.1
Figure 14: Detailed class diagram for class RP
8. Implementation

We concentrate in this chapter on a brief description of the GUI for the initial prototype of our application and sample code to describe the interaction between the ProjectBank and our application. We also indicate briefly what kinds of output the prototype generates.

8.1 The GUI.

The Remodeler can be started within a MS session in which one of the files in the current remodeling project has been checked into the user’s briefcase and is open. The user starts the remodeler through a key-in command. This opens the main RM commands box shown in Figure 15 on the left.

The main panel of the RM commands box contains an array of buttons each of which makes one of the component- or computation-related use cases available to the user. The session-related use cases can be executed from the Project or Settings pull-down menus. Figure 16 shows the Project Settings dialog box that opens from the Settings menu.

The component- and computation-related use cases typically require special settings through which the user indicates, for example, the type of component under consideration, its physical properties etc. This is done through use case-specific dialog boxes; Figure 15

Figure 15: The RM commands box and a settings box opened in a MS view

The Remodeler can be started within a MS session in which one of the files in the current remodeling project has been checked into the user’s briefcase and is open. The user starts the remodeler through a key-in command. This opens the main RM commands box shown in Figure 15 on the left.

The main panel of the RM commands box contains an array of buttons each of which makes one of the component- or computation-related use cases available to the user. The session-related use cases can be executed from the Project or Settings pull-down menus. Figure 16 shows the Project Settings dialog box that opens from the Settings menu.

The component- and computation-related use cases typically require special settings through which the user indicates, for example, the type of component under consideration, its physical properties etc. This is done through use case-specific dialog boxes; Figure 15
42 shows—as an example—on the right the dialog box for use case 2.2, Add Component. Typically, such a box lists the available component types in a selection list on the left. On the right is a tabbed pane that changes with every Component type selection. Each of the individual panels shows a small number of logically related settings with their default settings. The user is able to modify any one of these; this includes the project settings, which the user can override for the component under consideration. Note that these settings are immediately used to (re)draw the component; they are saved as element settings in the dgn file, not in the object store.

8.2 ProjectBank and Remodeler Interaction

The key object that controls the interaction is the SessionManager. It is instantiated when a RM session starts. This happens upon the user’s request when she keys in the Remodeler Command, which instantiates a RemodelerCommand object:

```java
public class RemodelerCommand extends Command {
    ...
    public void start (String oString) {
        SessionManager sm = new SessionManager ();
    ...
    }
}
```

The SessionManager constructor called by the command creates not only the SessionManager (instance) for the current RM session, but performs also important initialization tasks:

1. It checks if a RP is already stored in the PB store. If so, it retrieves the RP with all of its components and creates a copy of all of these in working memory. If the RP does not exist, it creates an RP instance. At this point, components can be added or deleted or otherwise modified in the RP. It is important to note that modifications can only be performed on objects that are not part of the persistently stored RP. This is the reason why copies are created in working memory. When updates have to be made persistent, i.e. stored in the PB store, the existing RP has to be removed from the store and the current copy in working memory put in its place.

2. The SessionManager object creates and displays the RM commands box.

The dialog boxes that open if certain command buttons are hit in the RM commands box often have to display information contained in the current RP. In order to maintain a strict separation between user interface and domain objects, the dialog boxes are not allowed to query RP or RP components directly. All such queries have to go through the SessionManager. Furthermore, the SessionManager also has to mediate the exchange of geometric information between RP components and the current dgn file because dgn
wrappers, which capture the attributes of dgn elements, and JMDL schema objects cannot interact directly—the interaction has to be mediated by a pure Java object like the SessionManager.

These types of exchanges are illustrated by the following function, which is intended to give readers a flavor of programming with JMDL. The function removeEntireComp is a member function of SessionManager and called by the Remove Entire Component dialog box after the user has selected the type of the component to be removed and the dgn elements that describe the component geometry. The selected type is one of the input arguments; the other one is a vector containing the remove symbology.

```java
public void removeEntireComp(String type, int [] compSet){
   /*findGeometry is a SessionManager member function that returns object wrappers linked to the currently selected dgn elements.*/
   Vector geoVec = findGeometry();
   if (! geoVec.isEmpty()) {
      ...  
      ElementGeometry geo = new ElementGeometry ();
      ...
      /*Set component geometry in geo using the information in geoVec.*/
      ...
      /*theRP is a SessionManager attribute that references the current RP in working memory. removedComponent is a RP member function that creates and returns an instance of a removed component of the specified type and adds it to the Removed List.*/
      RMComponent temp = theRP.removedComponent (type);
      /*The SessionManager has to set the geometry of the object because RP, as a JMDL schema object, cannot do this.*/
      temp.setGeometry (geo);
      if (type == "Wall") {
         Wall w = (Wall) temp;
         /*SessionManager assigns the role-dependent symbology of the new component.*/
         w.getRole().setnewSet(new Settings (compSet));
         /*SessionManager sets the wall attributes by querying (inside the function) the Remove Component dialog box.*/
         setWallAttributes (w);
         /*SessionManager informs the user that a component of the desired type has been created.*/
         sendMessageToGUI(w);
      }  
      else if (type == "Door") {  
         ...
      }
      else if (type == "Finish") {
         ...
      }
      else  
         System.out.println ("Unknown type");
   }
   else if (type == "Wall") {
      Wall w = (Wall) temp;
      /*SessionManager assigns the role-dependent symbology of the new component.*/
      w.getRole().setnewSet(new Settings (compSet));
      /*SessionManager sets the wall attributes by querying (inside the function) the Remove Component dialog box.*/
      setWallAttributes (w);
      /*SessionManager informs the user that a component of the desired type has been created.*/
      sendMessageToGUI(w);
   }
   else if (type == "Door") {  
      ...
   }
   else if (type == "Finish") {
      ...
   }
   else  
      System.out.println ("Unknown type");
}
```
else {
    System.out.println("No geometry found");
}

8.3 Output

At the time of this writing, almost all use cases have been implemented in a first version. To illustrate the kind of output an application like ours is able generate, we show in Figure 17 a door schedule created by the Remodeler as an html file.

![Door schedule captured as an html file](image)

Figure 17: Door schedule captured as an html file
9. Conclusions

Our experience in developing and teaching this course has strengthened our faith in the object-oriented development strategies used, specifically, use case-driven software development.

We were also happy with the selection of UML concepts and methods that we used to define our strategy overall—with one exception: following Rosenberg’s recommendation, we interjected between the development of use cases and sequence diagrams a brief phase in which we generated what he calls “robustness” diagrams. The goal of these diagrams is to derive a first picture of the interactions between objects for each use case in order to ascertain that the initial class model is solid. We found that sequence diagrams can be used for the same purpose and that the switch from robustness diagrams that obey Rosenberg’s rules to sequence diagrams is not natural and led—in our class—to considerable confusion on the part of our students.

In addition, we believe that teaching this strategy in the context of a semester-long, realistic application development project can be pedagogically effective. All of this convinced us to offer this course again in the upcoming fall term.

We found ProjectBank as such easy to use and appreciated in particular that it was able to solve the issue of persistent storage of object configurations. More documentation is clearly needed to explain the interactions between ProjectBank and JMDL. The only reason why we were able to get the application to work at all was the fact that we were given sample code by our sponsor. Even then, it took us several days to understand the sample code well enough to be able to adapt it to our application. We assume that this issue will go away as ProjectBank matures and documentation becomes available.

Our positive impressions are reinforced by the (anonymous) student evaluations collected at the end of the course.
10. References

Aish, R. Migration from an individual to an enterprise computing model and its implications in AEC research. position paper submitted to the Stanford/Berkeley CE&M Workshop (1999)


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


Appendix A. Sequence Diagrams

We show in this appendix all sequence diagrams that we developed in class. They were drawn using RationalRose Enterprise Edition. We wish to express our gratitude to Rational Software Inc., who provided the software free-of-charge for the course.
Group 1: Session-Related Use Cases

1.1 Start Remodeler

Preconditions:
1. A PBP has been created.
2. The appropriate .dgn files have been copied and imported into the PBP.
3. One of these .dgn files is checked into the current briefcase and open in a MS-J session.

Basic Course:
1. The AD issues the “Remodeler” command from MS-J key-in window.
2. The RM opens the RM commands box. Internally, it checks if a RP is already defined for this drawing. If it is, RM retrieves the project from the PB data store and makes it available in working memory; otherwise, it creates a new RP.
1.2 Edit Project Settings

Preconditions:
1. The RM is active.

Basic Course:
1. The AD or RPM issues the “Edit Project Settings” command.
2. The RM opens the “Project Settings” dialog box.
3. The actor inspects the current settings and edits them as desired.
4. The actor commits the changed settings.
5. The RM saves the changed settings.
6. MS-J changes the display of the Remodeler components according to the new settings.
7. The actor closes the dialog box.
1.3 Save Project

Preconditions:
1. The RM is active.

Basic Course:
1. The AD issues the “Save Project” command.
2. The RM saves the current state of the project in working memory in the PB store.
1.4 Close Remodeler

Preconditions:
1. The RM is active.

Basic Course:
1. The AD issues the “Close Remodeler” command.
2. If the AD has made changes after the last save, the RM opens the “save” message box.
3. The AD selects one of three options: save, don’t save or cancel.
4. Depending on the AD’s response, the RM does or does not terminate with or without a save. If the RM terminates, it also closes the RM commands box.
Group 2: Component-Related Use Cases

2.1 Remove an Entire Component

Preconditions:
1. The RM is active.

Basic Course:
1. The AD issues the “Remove an Entire Component” command from the RM commands box.
2. The RM opens the “Remove Component” (RECdbox) dialog box.
3. The AD selects the component type, attributes and enters the annotation for the component and commits.
4. The RM prompts the AD to select graphical elements that represent the component.
5. The AD selects the elements in an MS-J window and accepts.
6. The RM creates the corresponding removed component and stores it internally.
7. The MS-J window updates the display of the component.
8. The AD closes the RECdbox.
2.2 Add a Component

Preconditions:
1. The RM is active.

Basic Course:
1. The AD issues the “Add Component” command in the RM commands box.
2. The RM opens the “Add Component” dialog box (ACdbox).
3. The AD selects the component type, attributes and enters the annotation for the component and commits.
4. The RM prompts the AD to draw or place the component.
5. The AD draws or places the component in the MS-J window and accepts.
6. The RM creates the corresponding added component and stores it internally.
7. The MS-J window updates the display of the component.
8. The AD closes the ACdbox.
2.3 Remove Parts of a Component

**Preconditions:**
1. The RM is active.

**Basic Course:**
1. The AD issues the “Remove Parts of a component” command in the RM commands box.
2. The RM opens the “Remove Parts of a Component” dialog box (RPCdbox).
3. The AD selects the component type from the components list displayed in the RPCdbox.
4. The AD selects the desired settings, like materials and graphical representations, for the removed parts of the component and enters the needed annotation for the removed part of the component.
5. The RM prompts the user to select between remove by fence or remove by points.
6. The AD makes a selection which either way has to stay inside the existing component area.
7. The RM validates the selection entered and prompts to select again if the selection set is not part of a dgn element.
8. If the selection set is valid, the RM adds the removed component parts to the Removed List and changes their symbology.
9. MS-J re-displays the component parts with the new symbology.
10. The AD closes the RPCdbox.
2.4 Put Back Removed Component

Preconditions:
1. The RM is active.
2. Parts of at least one component or at least one entire component have been removed through the RM and stored in the Removed List.

Basic Course:
1. The AD issues the "Put Back Removed Component" command in the RM commands box.
2. The RM opens the "Put Back Removed Component" dialog box (PBRCdbox).
3. The PBRCdbox displays a list of components removed through the RM and prompts the user to select component(s).
4. The AD selects the component(s) to be put back and accepts.
5. The RM deletes the selected component from the Removed List and restores it back to its previous symbology.
6. MS-J re-displays the component in its original symbology.
2.5 Remove Added Component

Preconditions:
1. The RM is active.
2. The component to be removed has been added through the RM with use case 2.2.

Basic Course:
1. The AD issues the “Remove Added Component” command.
2. The RM prompts the AD to select an added component.
3. The AD selects the component in the MS-J window and accepts.
4.1 If the component is not an RM component, the RM prompts the user again to select an added component.
4.2 If the component is an RM component, the RM removes the dgn elements representing the added component from the design file and removes the added component from the Added List.
5. MS-J refreshes the view(s).
2.6 Recycle a Component

Preconditions:
1. The RM is active.

Basic Course:
1. The AD issues the “Recycle Component” command.
2. The RM opens the “Recycle” dialog box (RCdbox), which shows a list of removed components.
3. The AD is prompted to select a component from the list or select a component in the drawing.
4. If the AD chooses by list, the AD selects the component in the list and the use case continues with 2.6.2, else the use case continues with 2.6.1
2.6.1 Recycle Component in Drawing

Preconditions:
1. The AD has issued the "Recycle Component" command.
2. The MS-J elements representing the component in question are not part of any Remodeler component.
3. The AD has selected the "select component in the drawing" option in the RCdbox.

Basic Course:
1. The RCdbox opens the "Annotation" dialog box (ANdbox).
2. The ANdbox displays the component types.
3. The AD chooses the component type and adds annotation in the ANdbox.
4. The AD selects the dgn elements representing the component in the MS-J window and accepts.
5. The RM creates the component and adds it to the Recycled and Removed Lists.
6. The RM displays the component with new symbology.
7. If the AD chooses to insert the component later the use case ends here, else it continues with 2.6.2
2.6.2 Recycle Component from List

Preconditions:
1. The AD has issued the “Recycle Component” command.
2. There are already removed components in the list.
3. The AD has selected the “select component from list” option in the RCdbox.

Basic Course:
1. The AD browses the Recycled List, which shows components that have been removed, but not yet been re-inserted elsewhere, and selects a component.

2. The RM asks for the new location of the component.
3. The AD selects a location for the recycled component and accepts.

4. MS-J displays the component at its new location.
2.7 Edit Component Attributes (Display Component Attributes)

Preconditions:
1. The RM is active.

Basic Course:
1. The AD issues the “Edit Component Attributes” command.
2. The RM prompts the AD to select the component in MS-J window or from a list of existing components in the current RP and accepts (the AD can select only one component).
3. The RM opens the “Component Attributes” dialog box (CAAdbox) and displays the attributes of the component.
4. The AD changes the desired attributes and accepts.
5. The RM updates the changed attributes.
6. The AD can select another component and repeat the previous 3 steps.
7. The AD closes the CAAdbox.
8. The RM displays the components according to the new attributes.
2.8 Highlight Components

Preconditions:
1. The RM is active.

Basic course:

1. The AD issues the "Highlight Component" command in the RM commands box.
2. The RM opens the "Highlight Component" dialog box (HCdbox) and displays added, removed and recycled components in the current RP in a selection list.
3. The AD selects the component to be highlighted and accepts. (The AD can select more than one component).
4. The AD closes the HCdbox.
5. The RM changes the display in all MS-J views according to the "highlight" symbology. (The symbology can be edited from the "project settings" dialog box with use case 1.3).
2.9 Eliminate Highlights

Preconditions:
1. The RM is active.
2. A component is highlighted.

Basic Course:

1. If the HCdbox is not open, the AD opens the HCdbox by issuing the “Highlight Component” command.

2. The AD clicks on the ‘Eliminate Highlights’ button in the HCdbox.

3. The RM reverts the display in all MS-J views to the original symbology and resets the active component to null.

4. The AD closes the HCdbox.
2.10 Add Annotation

Preconditions:
1. At least one RM component has been generated.

Basic Course:
1. The AD issues the “Add Annotation” command in the RM commands box.
2. The RM displays the “Add Annotation” dialog box (ANdbox).
3. The AD selects an annotation placement option (in drawing or in sidebar).
4. The RM prompts the AD to select the component to add annotation to.
5. The AD selects an RM component and accepts.
6. The AD writes the annotation and accepts in the MS-J window the appropriate placement.
7. The RP creates a new annotation and associates it with the selected component.
8. MS-J displays the new annotation in the drawing.
2.11 Switch Annotation Style

Preconditions:
1. The RM is active.
2. There is at least one annotation in the drawing file.

Basic course:
1. The AD issues the “Switch Annotation Style” command.
2. The RM prompts the AD to select the annotation(s) in the MS-J Window.
3. The AD selects the annotation(s) and accepts.
4. The RM switches the annotation display in drawing.
5. The AD may interactively move the annotation to a more desired location.
6. MS-J updates the view.
Group 3: Computation-Based Use Cases

3.1 Prepare Quantity Take-Off

Preconditions:
1. The RM is active.

Basic Course:
1. The AD issues the “Prepare Quantity Take-off” command in the RM commands box.
2. The RM opens the “Prepare Quantity Take-off” dialog box (PQdbox). A list of components supported by the RM is displayed.
3. The AD selects the types of components for which the RM should calculate quantities.
4. The RM calculates the quantities of the selected components and displays the results in a special display window.
5. The AD accepts.
6. The RM saves the results in a file from which a hard copy can be produced (in some common format like html or a spreadsheet).
7. The AD closes the PQdbox.
3.2 Prepare Schedule

Preconditions:
1. The RM is active.

Basic Course:
1. The AD issues “Prepare Schedule” command in the RM commands box.
2. The RM opens the “Prepare Schedule” dialog box (PSdbx).
3. The AD selects the component type(s) for which RM should prepare a schedule from a selection list showing the component types for which there are instances in the current RP.
4. The AD selects one of two options to place the schedule, in an html file or in the drawing.
5. If the AD selects “in drawing” option the RM displays a tentative outline of the schedule in the currently active MS-J display.
6. The AD places the outline of the schedule in a desired position on the drawing by dragging it and accepts.
7. The RM completes the schedule and closes the PSdbx.