Modeling Guidelines for Legacy Applications

by **Christophe Tournier**
Technical Representative
Rational Software, Switzerland

This article provides keys to getting started with modeling legacy applications using the Unified Modeling Language (UML). Legacy applications are often complex and difficult to evolve. Usually, they are written in implementation languages that are not Object-Oriented (OO), the source code is not reverse engineered, and the documentation is not synchronized with prior evolutions. Often, legacy applications have been documented using various methodologies and tools, but there is no synchronization among the tools. If developers need detailed and unambiguous information, they must look to the source code, which is time consuming, requires a good knowledge of the language, and does not easily provide a high-level overview of an application.

The UML-based modeling approach presented here can provide a high-level overview of a legacy application that needs to be rebuilt, replaced, or extended. Our primary goal is to create a simple graphical overview that represents a complementary view of the code. *This overview:*

- *Supplies a different view of an existing system as well as keys to evolving it.*
- *Helps in gathering requirements and discovering reuse potential when preparing a "cosmetic makeover" of the system or initiating a redevelopment.*
- *Helps manage complexity in extended applications that mix OO and legacy; for instance, Java client applications that require business services through a legacy application written in COBOL.*
- *Facilitates communication between company teams developing new technologies and teams working with the old technologies needed to*

maintain the legacy systems.

- *Provides UML and OO training to people working on legacy applications.*

In providing examples of possible analysis and design models for legacy applications, we will focus on analogies between the process of creating and maintaining legacy applications written in non-OO languages and the Analysis and Design discipline for OO applications, described in the Rational Unified Process® or RUP® product. Our aim is to narrow the gap between the legacy world and the OO world and facilitate a smooth integration of legacy resources into the OO world.

As stated in the OMG UML specification, the "UML is not intended to be a visual programming language" that replaces other programming languages; specific concepts are better expressed through textual programming languages. Instead, this article provides starting points for improving the balance between code documentation and graphical documentation with the UML.

Describing a Non-OO Application

To describe an application not written in an OO language(s), the designer can make the most of the RUP process by modeling the context of the application, which includes the following steps:

- Package business objects as organizational units through Business Object Models.
- Express the functional requirements as use cases through Use-Case Models.
- Define the upper-layer package architecture.
- Analyze the application through Use-Case Analysis Models.
- Develop a design model to discover the static and dynamic structure of the system.

Package Business Objects as Organizational Units

Facing the complexity of many existing applications, a designer can build a map of the environment showing dependencies between applications, which are visualized as packages. Packages own application elements: for instance, programs, tables, or artifacts. Packages can reference other packages using a dependency relationship, which indicates that elements within a client package may legally reference elements within a supplier package. This high-level view is at the *organizational level* and is provided by the RUP business modeling discipline through the Business Object Model.

Figure 1 shows a sample retail customer management system that depends mainly on information provided by an *Inventories* system and a *Flows* system. The *Inventories* system needs the two *Flows* systems to work.

Financial Flows and *Furniture Flows* don't know anything about *Inventories* and *Retail Customer*.

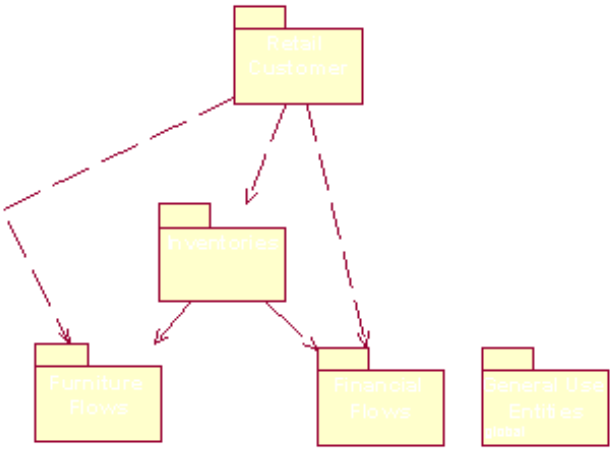

Figure 1: Retail Customer Management System

Express the Functional Requirements as Use Cases

Use-case models emphasize the primary objectives of the "Actors" (users and/or other systems) who use a system (see Figure 2), and define the boundaries of the system. Use cases describe how actors interact with the system; they are a clear way to express the functional requirements of the system. *The advantage of the use-case-driven approach is that it is* ***platform independent***.

Figure 2: Use-Case Representation (Platform Independent)

The properties of a use case are the same for both OO and legacy systems, because *requirements are design independent*. In the case of legacy system extension or redevelopment, don't hesitate to write use-case flows of events and to create mock-ups and prototypes, as described in the RUP.

Legacy applications frequently use batch processing, which involves submitting a unit of work (job) to a machine, which places it in a queue to be processed at a later time. These asynchronous programs require computational resources, and are useful for calculating, reporting, and updating data within the information system. Modeling batch processes with a use case requires a Scheduler Actor to launch the use case. For complex reports, a Printer Actor might be necessary (see Figure 3). Dialog and exchange of information between the Scheduler Actor and the use case are poor. Batch flows can be described briefly in a use-case specification from an external point of view. Textual specifications or UML diagrams describe the internal flow of events during the batch execution of a

program.

Figure 3: Sample Use-Case Representation of a Batch Process Sent by a Scheduler Actor

Define the Upper-Layer Package Architecture

We can split applications into three different levels (layers) of abstraction.

- **UserInteraction**: These packages contain programs with user interfaces. With these programs, users view and maintain application data.
- **Batch**: Batch programs are launched by user-interface programs or by a scheduler. They generate reports and make intensive updates to persistence data.
- **Persistence**: Persistence data contains the object model of the physical view of the database tables. In the legacy world, data is not encapsulated into a specific layer. Applications have direct access to data with SQL or a file operating system. A persistence package represents the data model. No behavior is embedded in this layer except with triggers or stored procedures.

In the case of cosmetic makeovers, UserInteraction programs are rewritten first. Batch programs can be wrapped and called from a new program.

The elements of the UserInteraction layer depend on the Batch and Persistence layers. A package dependencies diagram like the one in Figure 4 provides an overview of the upper layers.

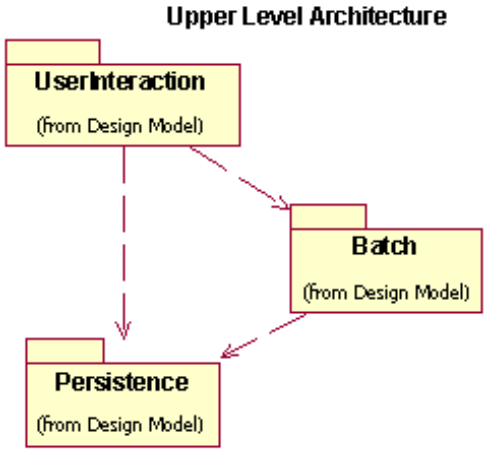

Figure 4: Package Dependencies Diagram

1.1 Analyze the Application Through Use-Case Analysis

Models

Creating an analysis model is a powerful technique for refining requirements and discovering relationships between elements. By identifying high-level messages sent by objects, the analysis model describes responsibilities that analysis objects intend to delegate to other objects. In an analysis model, classes are stereotyped -- as *boundary*, *control*, and *entity*, for example.

Declare one *boundary* class per use case and Actor, and one control class per use case. An analysis model in the legacy world looks like an OO analysis model because the analysis model takes place in an ideal environment. *The analysis model is a kind of "Esperanto" that is not dependent on implementation*.

Our sample application displays a list of customers for a region, and submits a batch program that prints a sorted customer report by region. The batch program that prints the report is considered a boundary class. The application also displays some customer information details.

When modeling an application, the OO designer builds abstractions and uses a vocabulary that reflects the actual world, thereby making the software computing terms understandable and relatable to the actual context in which the system will operate. In the legacy world, using real nouns about data and programs is not a common practice, and developers must make a huge effort to avoid computer gibberish when they communicate with users.

For instance, the AS/400 IBM legacy platform limits program names to eight characters. In our case, we use standard prefixes such "CCLI" (denotes a Customer), "P" on the seventh letter (denotes a program), and numbering to classify programs: CCLI01P1 means the first and main program managing customers within the application. By contrast, our OO terms, FollowCustPgm and ReportCust (see Figure 5), are more understandable.

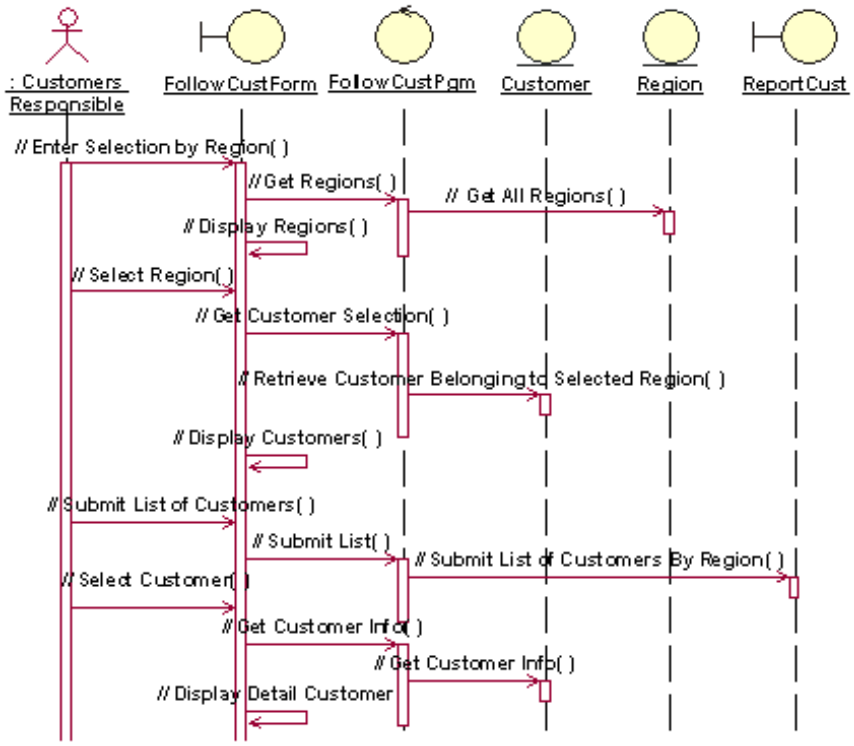

Figure 5: Use-Case Analysis Model for Legacy Application

A significant added value of the analysis model is that it provides natural-language documentation through the responsibilities description: for example, GetCustomerSelection, DisplayCustomers, and SelectRegion, as shown in Figure 5. Using natural language is the most common way to document an application in a form that is understandable to all project participants. Graphical language like the UML helps identify ambiguities and synthesize documentation. Using both natural and graphical languages is the best strategy for managing an application's complexity.

During the OO analysis and design process, the analysis model is useful for validating requirements and discovering the structural and behavioral parts of the application without going into implementation details. Later, the design model can naturally replace the analysis model and reflect what is actually in the code. In a non-OO analysis process, requirements are difficult to trace to the description of the organizational structure and associated behavior. For example, the designers have to establish strong naming conventions and maintain consistency with requirements through comments. The analysis model view carries a lot of extra information that is not directly provided by the code. *Thus, the analysis model can be helpful for maintaining consistency from requirements to implementation. Making good use of natural language by naming responsibilities instead of using program names provides a complementary view of the code.*

Develop a Design Model

Using a design model[1] is a good way to manage the complexity and dependencies of an existing application, or to insulate pieces of an application that developers intend to replace. The designer refines the analysis classes to create a design model that reflects the working code

structure. In fact, *getting closer to the code is the purpose of the design model.* For instance, analysis classes become one or several design classes. The designer replaces analysis classes with programs, and a responsibility is performed by several operations dispatched into different classes or programs.

Managing Differences Between the Object-Oriented and Legacy Worlds. Objects collaborate, sending and receiving messages in order to achieve a given task or perform responsibilities. From analysis to design, responsibilities become operations, and methods are the implementation of operations.

Figure 6 shows part of an interaction between objects in which the goal is to select and display customers belonging to a region. A double slash "//" precedes the name of a responsibility. (Stereotyping operations with <responsibility>> is another alternative.)

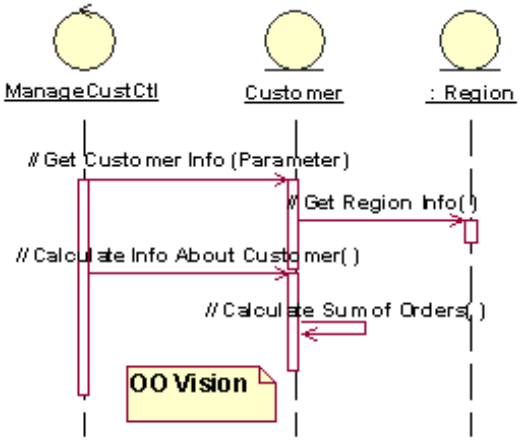

Figure 6: Examples of OO Vision in Which Responsibilities Are Delegated to Analysis Objects

The sequence diagram in Figure 6 matches the class diagram in Figure 7, which represents the static point of view of the collaboration.

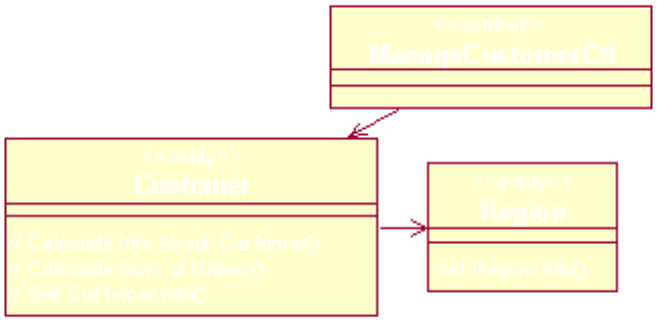

Figure 7: Static View of Corresponding Classes and Responsibilities

In the legacy world, there is no delegation. A program performs most of the responsibilities delegated to entities, controllers, and boundaries; and it centralizes most of the messages. In the legacy world, most of the behavior is embedded in control programs instead of being dispatched through boundaries or entities. In a sense, it's much more difficult to write

and maintain a legacy program because behavior is centralized.

The program stereotyped as "Control" is responsible for retrieving information from a customer. Each time a program needs to get information from a Customer, it has to know the customer data structure in detail and get (or set) all the parameters or an appropriate alias. By default, there is no encapsulation. In contrast to an OO paradigm, in non-OO languages data structures are controlled by programs. *In the non-OO world, data and behavior are separated; even if data exists in a single place within a relational database, behavior is duplicated.*

The following example describes a traditional non-OO interaction written in pseudocode to retrieve a list of customers from the Customer and Region tables.

```
// Data Structure Customer
        KEYID NUMBER(10);
        NAME CHAR(60)
        CDREGION CHAR(2);
        AMOUNT NUMBER(10,2)
        ...
// Data Structure Region
        KEYID NUMBER(10) ALIAS REGID
        CDREGION CHAR(2) ALIAS CDREGIO // Code region
        DESCREG CHAR(25)// region description

        INITIALIZE REGION CURSOR
CDREGIO=parameter
        READ REGION // Search for region description
        SUM=0
CDREGION=parameter
        INITIALIZE CUSTOMER CURSOR sorted by CDREGION;
READ CUSTOMER
        WHILE (CDREGION =parameter)
          // CALCULATE SUM OF ORDERS
          SUM=SUM+AMOUNT
          // SAVE CUSTOMER INTO A LIST BEFORE DISPLAYING
          SAVE CUSTOMER
          READ CUSTOMER
        ENDWHILE

PROCEDURE SAVE CUSTOMER
...
END PROC
```

The sequence diagram in Figure 8 provides a description of the above pseudocode. The program manipulates all the data necessary to retrieve information and is responsible for all the control flow.

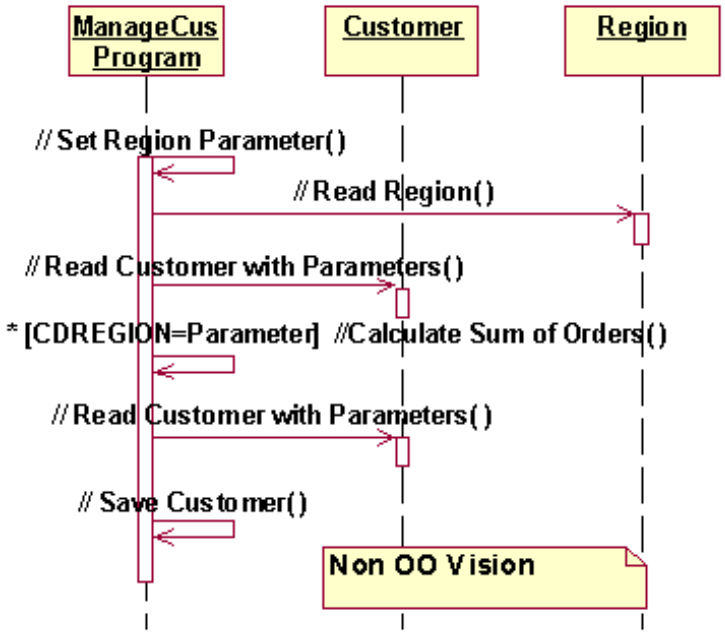

Figure 8: Example of a Non-OO Vision: The Program Carries Most of the Responsibilities

The responsibilities assigned to data entities are the CRUD (Create, Read, Update, Delete) functions; data access functions are implemented at best with SQL queries, stored procedures, or triggers (messages: `// Read Customer with parameters`, `// Read Region`). When using SQL queries, the message `// Read customer with parameters` will correspond to the select query to retrieve information from Customer table. Once the data is loaded into the control program, the calculation is realized by the control program (Message: `// Calculate Sum of orders`). When updating the Customer entity, the program would set data and send the update order to the entity.

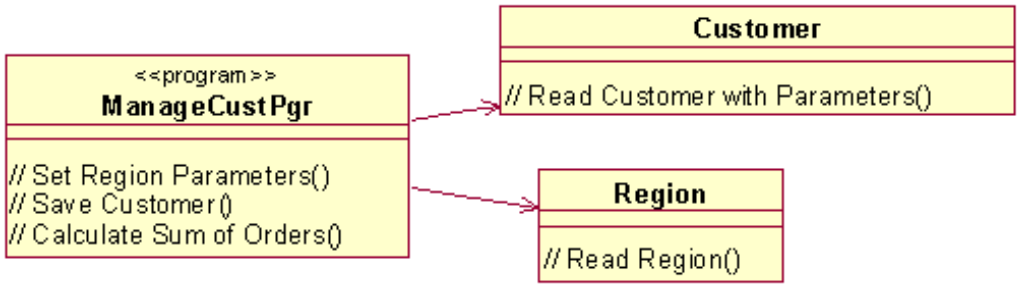

Figure 9: Static View of Corresponding Classes and Responsibilities Delegated to Program and Tables

Designing Boundary Classes. *In the legacy world, user interface classes are embedded into programs. There is no separation between user interface classes and business services.* Getting closer to the code, the designer chooses the level of detail desired to model the application. For instance, if it is necessary to model the navigation between screens, the designer gives details of the boundary classes and their interactions.

Graphical User Interfaces (GUIs) are often divided into three specific kinds of forms:

- ***Enter criteria forms*:** the user enters the search criteria to find data.
- ***List forms:*** the system displays a list of existing data records and the user can select one record.
- ***Details forms*:** the system displays the record selected by the user.

In the legacy world, all forms are either part of the program (with an association relationship) or are embedded into the same program. In our example (see Figure 10), the following forms are considered to be embedded into the CCLI01P1 program:

- CCLI01Pd1 is the *details* form displaying customer information (our naming convention is "d" for details).
- CCLI0lPl1 is a *list* form ("l" for list).
- CCL101Pl2 is a *list* form.
- CCLI01Pm1 is the *enter criteria* form ("m" for enter criteria).

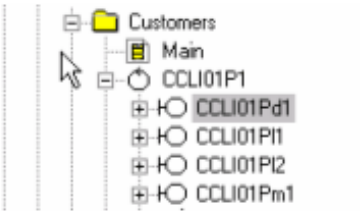

Figure 10: Forms Embedded into the CCLI01P1 Program

Sequence diagrams show the explicit sequence of messages between objects participating in an interaction. The *collaboration diagrams* are similar. If the designer only wants to emphasize the organizational and structural relationships between elements, the collaboration diagram is sufficient and easier to build. For instance, the collaboration diagram (e.g., Figure 11) is a good way to point out the navigation through user interfaces.

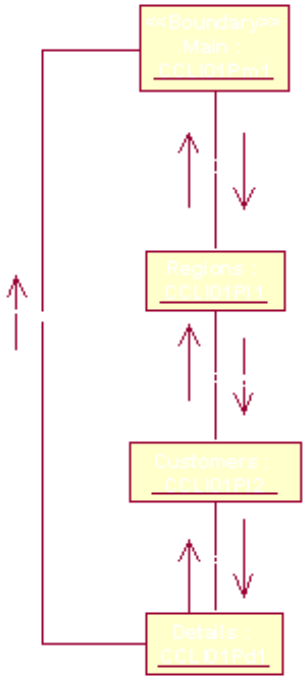

Figure 11: Navigation Between Screens Displayed from a Collaboration Diagram

Designing Entity Classes. Most of the responsibilities delegated to entities are performed by the legacy program. Entities:

- Have *CRUD responsibilities* if the program uses a file operating system.
- Have to *execute queries* if programs use SQL query to access a relational database.
- Have to *execute stored procedures or triggers* when stored procedures or triggers are allowed.

Entity classes in the legacy world map to a table. Foreign keys (relationships between entity classes) and stored procedures are often not implemented. Foreign keys are managed by programs. When a control class is viewing some entities, these entities are candidates to be associated.

Using the OO encapsulation principle -- in which the object provides an interface that manipulates the data without exposing its underlying structure -- tends to normalize data. After many years of an application's life span, the data's physical structure replaces the conceptual vision provided by a model. Within an existing database, when a column manages different types of records into a table (selected by a *where* clause) the table is a candidate to be split into different tables through inheritance relationships if the selection involves a specific behavior.

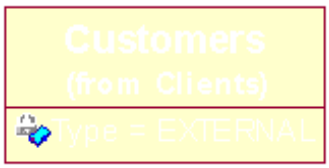

Figure 12: Customers Entity with Different Types of Customers

For example, in Figure 12, consider the Customers entity. The rules of discount for internal and external customers differ. Although those two entities have different behaviors, the attributes are nearly the same. Specifying whether a customer is an internal or external customer is implemented by a single "type" attribute. The two different types of customers can be separated into two entities representing the two types of customers with an inheritance relationship (Figure 13). The specific attributes of internal customers are placed into InternalCustomers entity. Such a transformation, which corresponds to the source of the conceptual data model, refreshes the perspective on the existing data structure.

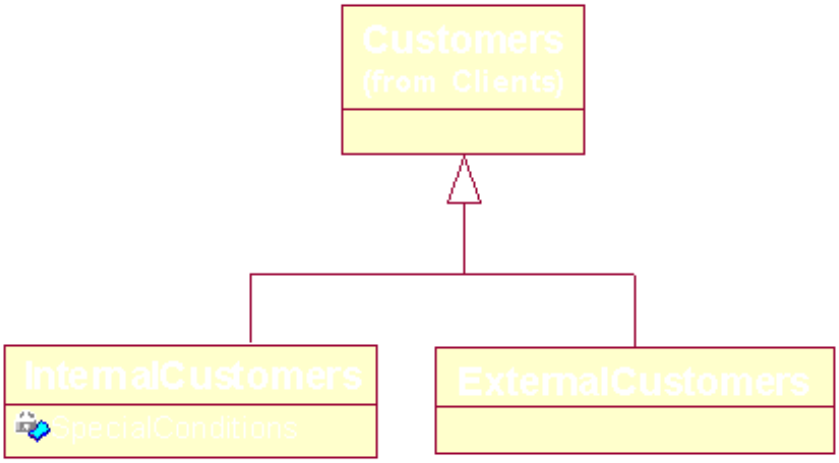

Figure 13: Transformation of the Customers Entity into Two Separate Entities

Designing Control Classes. An analysis control class will be split into one or several programs that realize the use cases. During design, these responsibilities become procedures, functions, or calls to programs. Some of a legacy system's reuse potential comes from these modular programs, which are built to perform some specific behavior.

Designing Subsystems. A subsystem is a grouping mechanism to specify a behavioral unit. Interfaces characterize the behavior of a subsystem. Calls to other application subsystems are specified through interfaces, which expose the services offered by the subsystem. In legacy applications you can use subsystems to call existing applications that provide services. Programs that are invoked by a number of other programs are candidates to be part of a package containing utilities programs, or to be included within a subsystem.

Using a *boundary class* emphasizes the interaction with another part of the application. The interface concept does not exist in legacy applications, despite the fact that applications could be very modular. In our case, the

boundary class is a simple call to another program, and the interface definition with natural language is a way to document it.

A batch process is asynchronous and is separated from the user interface by a "submit" command. Batch processes should at least be grouped into packages. Because batch processes can be considered as a behavioral unit of a system, an alternative is to define the batch part of the system as a subsystem in order to document it with interfaces, and to clearly separate batch programs from the rest of the application.

Detailing the Design Model

The design model is used to describe the "How" of an application and the analysis model is a way to refine the "What." In our case, as we develop the design model, we refine our analysis model and get closer to the code, replacing some analysis classes that were described in natural language to design programs, with real nouns as names of programs.

Designers should feel free to choose the level of detail they want to model, and to create their own set of stereotypes to match the particulars of their applications. For instance, the <> concept, which denotes where specific data on an AS/400 is stored, is an entity class. During the design process, classes with the <> stereotype become a program class. The program can be split into several programs, and a source file, or set of source files, can map to a program. For instance, a program uses script files and report files; if these files do not contain important behavior, you can treat these files as attributes of the class by creating nested classes in the program file. Include files are partitioned into utilities packages to manage shared pieces of code. If these details are required, functions or procedures declared in the source files can be transformed into operations.

If necessary, add dependency relationships by using a <<trace>> stereotype between a class named in natural language and a real program or file name (see Figure 14).

Figure 14: Trace Dependency Relationships Between Classes with Natural Language Names and File Names

To refine class diagrams, distinguish strong structural relationships from light dependency relationships. Strong relationships mean less modularity potential. Light relationships mean easier replacement and maintenance.

Strong relationships are:

- *Associations between programs and tables that can create inconsistencies in the database* -- for example, programs that update tables.

- *Associations between programs and primary tables used by programs for retrieving flow of data* -- for example, most records of a table are read to extract complex data.

Light relationships are:

- *Dependencies when a program reads a table to retrieve simple information without updating files* -- for example, a single read to extract the value of a field into a table; this could be fulfilled by a simple program.
- *Dependencies when a program calls another program* -- for example, invoked programs are candidates for subsystems.

Describing Collaboration. The description of collaborating elements that participate in a use case is presented in an interaction diagram, such as the sequence diagram shown in Figure 15.

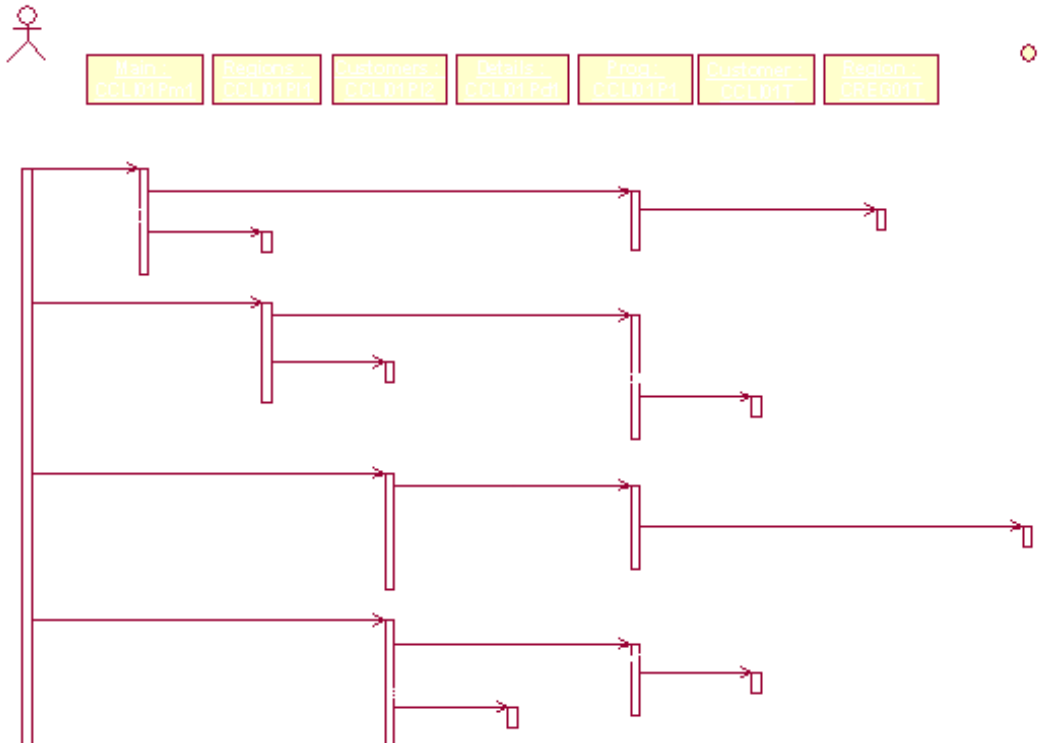

Figure 15: Sequence Diagram Presenting a Scenario within the Use Case "Manage Customer Information"

This basic flow happens in a specific context involving participating elements belonging to classes. At the end of the analysis or design process, a collaboration is created. This UML class diagram shows a set of design elements used by the current use case. CCLI01R1 represents the set of submitted commands characterizing the batch programs dedicated to customers. The sample collaboration concerning objects in Figure 15 is shown in Figure 16. (The association between the program CCLI01P1 and Batch "Interface" CCLI01R1 is considered a light relationship.)

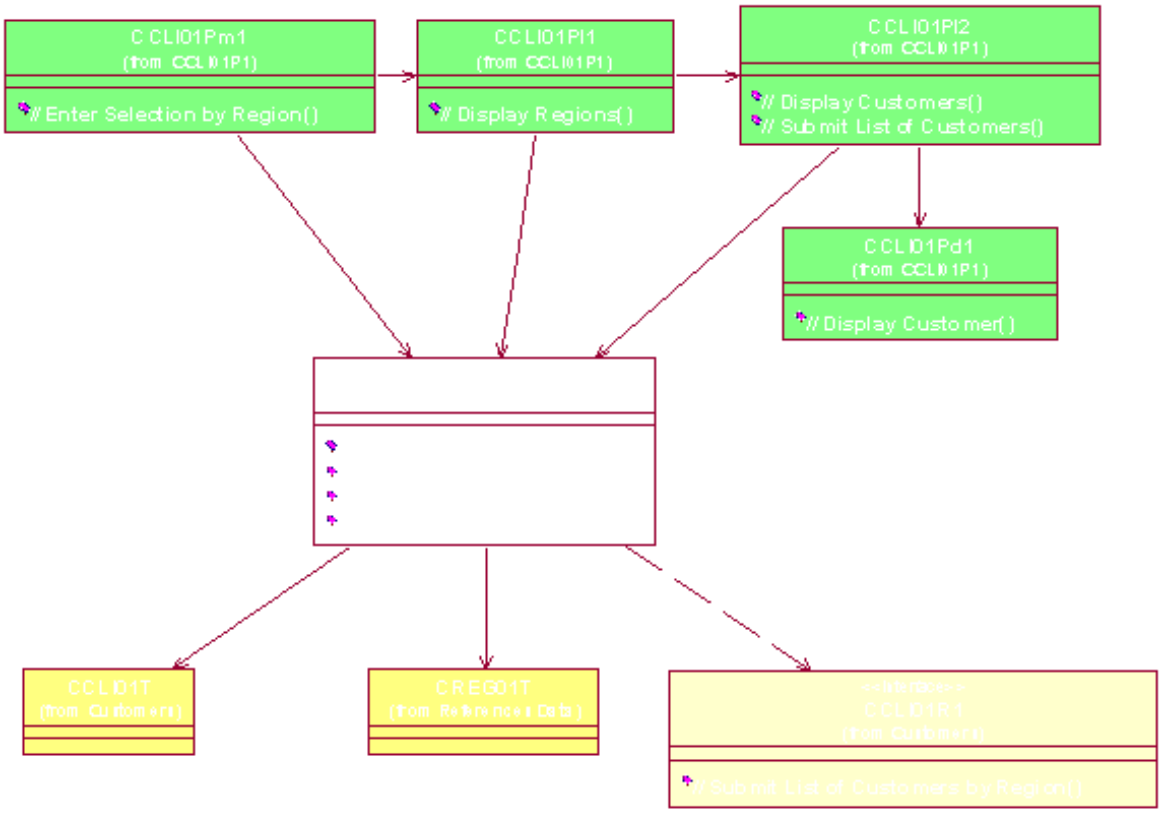

Figure 16: Collaboration Showing Static View of Participating Elements

Describing Batch Workflow with Activity Diagrams. Unlike the other UML diagrams, activity diagrams come not only from OO technology, but also from many other sources. Activity diagrams are useful to describe control flow and object flow in computational and organizational processes, so they are the primary diagrams to consider when modeling batch processes. In addition, by using subactivities, we can create a zoom effect that moves from a global perspective to a detailed perspective.

Starting from major activity, with subactivity diagrams we detail programs with swimlanes, conditions, and eventually loops. The objects in each swimlane show the entity created, read, or updated by the batch program chain.

In this example (through CCLI01R1 interface), the client program from UserInteraction package submits CCL01. The CCL01 program reads CLH1 to retrieve a date and calls CCLI02P2. The program CCLI02P2 uses the table CREG01T and the view CCLI01V1 (selecting customers in state "Open" and sorted by region; "V" is our convention to represent views) based on the table CCLI01T. The program CCLI02P2 prints result statements in a report file named CCLI01O2, which is associated to the CCLI02P2 program.

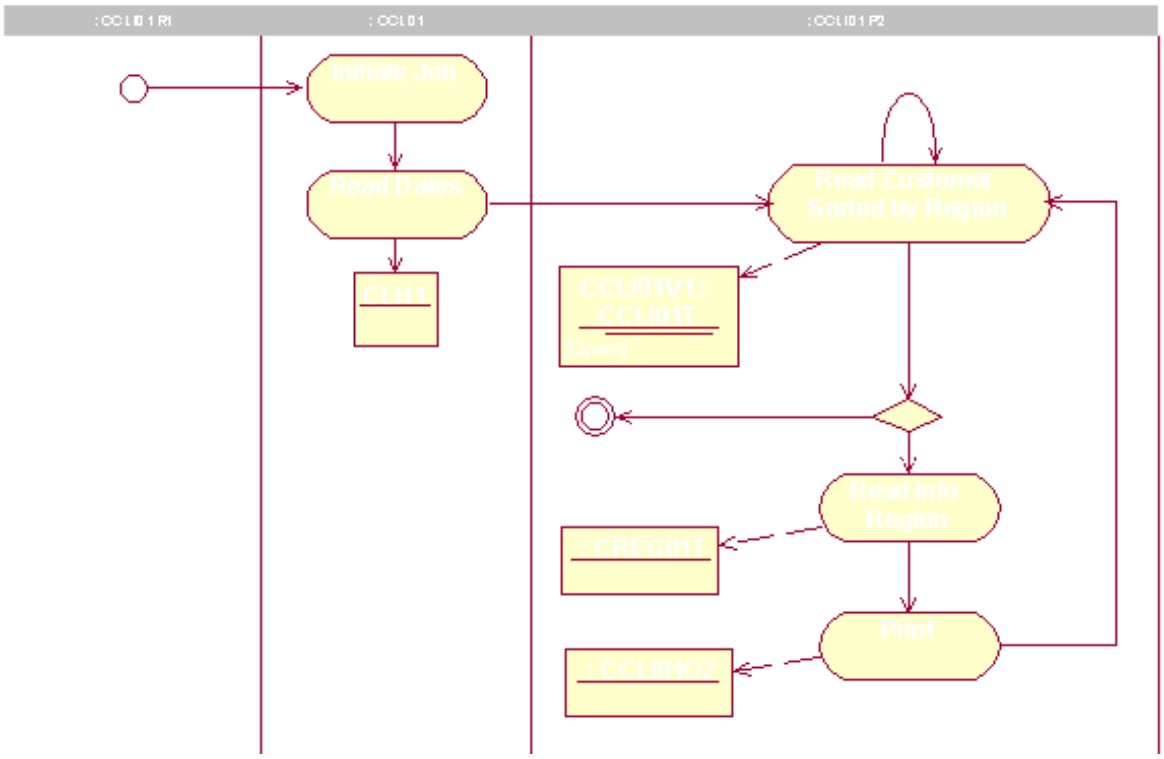

Figure 17: Batch Control Flow-In Details with Participating Programs and Tables

Describing Package Dependencies. After discovering all the major programs and relationships with entities, we refine models in order to assign programs or tables to a specific package. In this way, we can detail the partitioning of the whole application. The detailed package dependencies diagram provides a vertical slice through our application, from UserInteraction packages to Persistence layer packages. If necessary, we display elements contained in packages that are dependent on each other.

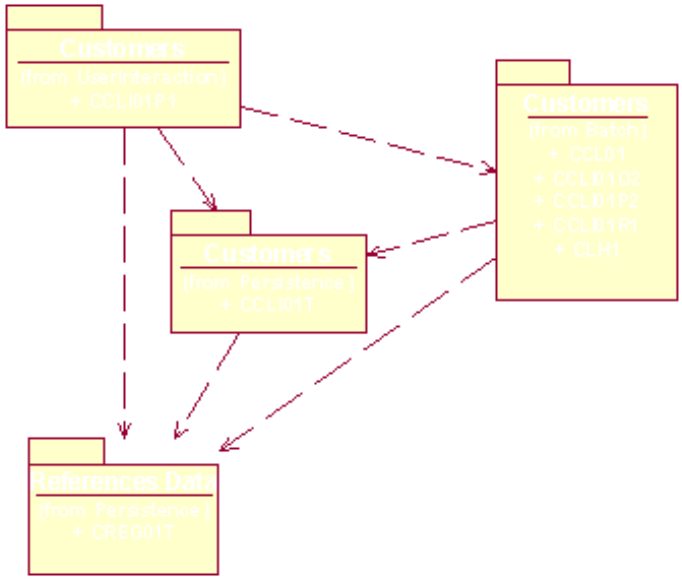

Figure 18: Packages Partitioning Through Layers with Elements Contained in Packages

Capture a "Big Picture" of the Application. For a complex application that contains many programs, it is helpful to capture an overview of the application. The design model gives the navigation map of the application and provides the behavior of the application through the architecture. When many programs are involved in an application and/or when the batch flow of programs is complex, it is useful to create a summary overview of the application from both the static and dynamic points of view.

This "big picture" is founded on the data model of the application. Usually, data is well documented through models or textual description showing the structure of data, but dynamic behavior is poorly documented. The idea is to transform a strong relationship between a program and a table to a major behavior of a table. An interaction diagram (see Figure 19) shows the communication between entities where programs become operations of entities classes.

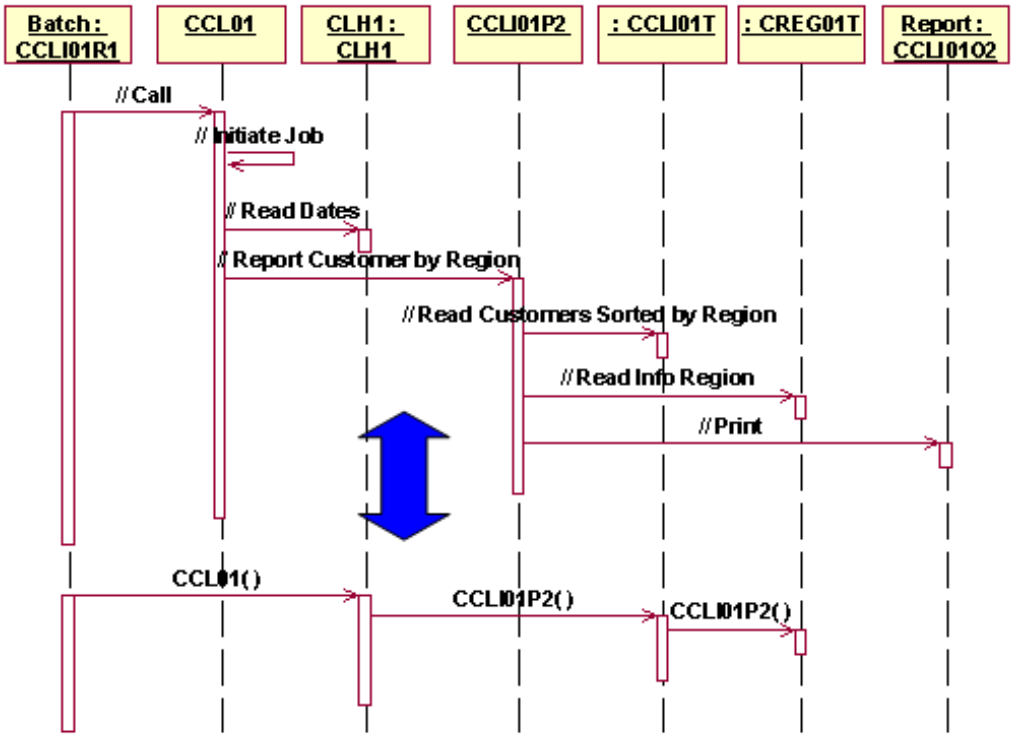

Figure 19: Interaction Diagram Equivalent of the Batch Control Flow of Figure 17

We can use this approach to quickly obtain a global overview of an application. This simple interaction just considered CCL01 and CCLI01P2 as major programs applied to CLH1 and CCLI01T entities. In our example, CCLI01P1, CCLI01P2 uses table CCLI01T as the main entity to read or update. CREG01P1 is a program that maintains CREG01T table. CCLI01P2 is a program involving CREG01T table. CLH1 is an entity in which date parameters for batch programs are registered.

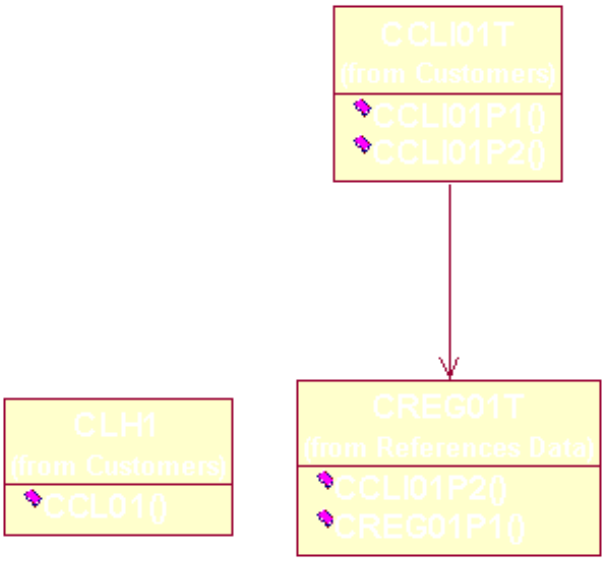

Figure 20: "Big Picture": Static View of Entity Classes and Main Programs

The Power of UML

The UML supports multiple programming languages. When designers don't have tools to synchronize models with code, it is possible to find a "sweet spot" between a description in natural language at the responsibility level and a detailed view at the implementation level that might already be provided by the source code. With UML, designers of legacy applications implemented in non-OO languages can:

- Use analysis models to provide an overview of an application by making good use of natural language.
- Choose the level of detail for the design model to get closer to the code from the analysis model.
- Partition the application through relevant packages and manage dependencies.
- Understand rapidly the global static structure of an application.

For developers used to working with OO applications, modeling and understanding legacy applications with the UML can be an entry point into the legacy world. For legacy development teams in charge of maintaining and creating applications in the non-OO world, using UML tools and sharing standard modeling concepts and tools (through a process inspired by the RUP) can be of great benefit. The UML is a way to unify teams in a company that uses both legacy and new technology applications.

Acknowledgments

I thank Philippe Dugerdil at Pictet & Cie with whom we share a lot of ideas; Jerýme Klotz from Sylog Consulting for his review and comments; and Catherine Southwood at Rational for her editorial help.

References

1. Gary K. Evans, "OO Thinking." http://www.evanetics.com/

2. *Rational Unified Process 2002*. Rational Software, Cupertino, Ca.

3. Philippe Kruchten, "Using the RUP to Evolve a Legacy System." *The Rational Edge*, May 2001.

4. UML 1.4 OMG.

5. Pan-Wei Ng, "Automated Modeling of Legacy Systems Using the UML," *The Rational Edge*, September 2002. This article explains how to reverse engineer a COBOL program with a Rose COBOL Add-In that provides a design model, activity diagrams, and sequence diagrams.

---

[1] If a legacy application needs to be completely redeveloped, the design model we describe in this section will not be useful.

For more information on the products or services discussed in this article, please click here and follow the instructions provided. Thank you!