Using RUP to reverse-engineer a legacy system

Philippe Dugerdil

15 September 2006

from The Rational Edge: Many techniques and tools have been proposed for reverse-engineering legacy information systems. But these techniques are seldom described in the context of a well-defined process. This article describes how the IBM Rational Unified Process can be applied to reverse-engineering and in particular to the recovery of the system’s architecture.

For the last 15 years, software reengineering has become an important subdiscipline within computer science. Larger enterprises increasingly automate critical tasks, which make these businesses highly dependent on their information systems. But in many cases, these systems have been maintained for many years -- i.e., these are "legacy systems" that are important for the enterprise, but they are often difficult to understand and maintain.

For budgetary reasons, the redevelopment of these systems from scratch can be a poor option.

The alternative is reengineering, which can be decomposed into two sub-projects: reverse-engineering (building representations of the actual code) and forward-engineering (restructuring and/or redevelopment of some parts of the code). In particular, one important activity in reverse-engineering is to recover the architecture of the system. This is where RUP is strong. In this article, I describe a situation where the original developers of the legacy information system are not available to provide information on the original structure of the program and no reliable documentation is available. Our process must then rebuild the Unified Modeling Language (UML) models that together provide the architectural view of the system.

One of the key problem in reverse-engineering is the well-known "concept assignment problem" -- i.e., the mapping of domain concepts to elements of the source code. Although the solution to this problem depends on what we mean by "domain concept," we show how rebuilding the UML model of the software through RUP-based disciplines can help solve this problem. At the end of
As in RUP, the construction of the use-case model is central to our reverse-engineering process. First, the use cases are used to recover the business process model the system supports. Second, the use cases are analyzed to build the system analysis model that represents a hypothetical architecture for the software. Third, the use cases are used as the source of scenarios to be run to find the software elements that are involved in the implementation of the business functions. This paper only offers an overview of our process, highlighting its main tasks and work products.

The process

Most of the tasks and work products required to reverse-engineer a system can be found in RUP. In a reverse-engineering project, the key work product will be the representations and models of the system that are recovered from the source code. However, in some cases, the relative order of the tasks must be modified with respect to the original RUP process, since the system is already developed. In particular, the variation of emphasis of each discipline over time will be different from a "green field" software development project. Conceptually, our reverse-engineering process proceeds through the following three steps:

1. Assess the scope of the reengineering project.
2. Build the abstract models.
3. Recover the architecture of the software.

Figure 1 shows where the tasks associated with these steps are located in the traditional 2-dimensional representation of RUP.

![Figure 1: The three key steps of our process mapped to the RUP disciplines and phases](image-url)
In the broader context of a full reengineering, the reverse-engineering project would be followed by a forward-engineering project aimed at restructuring/redesigning/redisveloping the system in part or as a whole. This is where a new paradigm such as service-oriented architecture (SOA) would be introduced. Even though this article focuses on the reverse-engineering part, the potential future platform or paradigm must be dealt with to assess the feasibility of the reverse-engineering project.

**Assess the scope of the reengineering project**

In today's trend for ever more business flexibility and IT cost reduction, any reengineering process should try to optimize the software assets and the IT resources. This is especially true if the reengineering option is in competition with the complete rewrite of the system outsourced to a remote development center.

In this initial step, we set the business scope and desired quality attributes of the reengineered system. Usually, these quality attributes are defined beforehand, and the system is restructured to fulfill them. However, if the quality of the actual code is too bad, it may not be worth the effort of complete reengineering. Management may decide to extract and reengineer some critical component only. In the worst case, when the system structure is so bad that no useful piece of code can be reused, the reengineering work could be limited to the extraction of the knowledge embedded in the old system to help specify a new one. This step is iterative: The more we know about the actual structure of the system, the better we can assess the economic relevance of the restructuring of the system.

As in any project, the first task is to develop a vision for the reengineered system whose work product is the Vision document. In particular, such a document should make clear the rationale for the project, the constraints to the reengineered system, as well as the technological choices. For example, the Vision document would state the target architecture to be an SOA. Depending on the size of the project, the information in the Vision may be further elaborated in the Business Case work product that would precisely analyze the economic relevance of the project. This is the output of the Develop Business Case task within the Project management discipline. Alternatively, the target quality attributes of the reengineered system as well as detailed technological constraints may be recorded in the Supplementary Specifications Work Product document that is the output of the Find Actors and Use Cases task of the Requirements discipline. Since, in a reverse-engineering project, these specifications apply to the reengineered system, the actual system must be compared to these specifications. Then a Risk List will record the risks of not reaching this target and the ways to mitigate these risks. The Risk List is produced by the Identify and Assess Risks task of the Project Management discipline (Figure 2).

![Figure 2: Assessing the scope of the project](image-url)
Note that the specification of the target quality attribute for the reengineered system is not specific to our work. For example, it is at the core of the SEI's Horseshoe model. However, the iterative assessment of the risks of the project does not seem to be explicitly dealt with in the few published reengineering processes.

**Build the abstract models**

The trouble with the idea of software architecture is that, in most of the situations, it is not represented explicitly at the code level. Therefore, if the unique source of information on the system is not in the mere source code, how can we rebuild its architecture? In fact, the architectural level of software description is in the head of the designer and, sometimes, in the technical documentation. When confronted with the source code of some unknown legacy system, how can an engineer know what grouping of software elements will produce some meaningful description of the architecture? It may even be the case that no architecture was designed in the first place!

To illustrate this paradox, Kazman and Carrière even speak of a "shared hallucination" to describe the quest for software architecture in legacy systems. Of course, industrial-size software systems are very complex artifacts. Building the architectural model requires understanding the system itself, since its source code does not offer much help. As an aid in system understanding, we can create a hypothetical architecture to potentially be discovered in the code. But this architecture must be sufficiently abstract to fit the large range of designs we are likely to encounter. RUP's system analysis model can fulfill this need, because the analysis class's stereotypes (entity, boundary, and control objects) represent the three responsibilities we will find in almost any information system.

Users often have a good perspective on the legacy system. Although they may have a narrow view of the programs' design, they are usually well aware of the business context and business relevance of the software. Although they generally cannot explain the inner workings of the software, they usually know the purpose of the program they use and the business justification of the computations. They normally know what kind of information must be input to the system and when. By gathering system usage information from all the people involved, we can reconstruct the sequence of processing at the business level. In short, we can build a representation of the business process the system is intended to support as well as a tentative domain model. In RUP terms, we rebuild the Business Use Case and its Business Analysis Model. This is done through the Detail a Business Use Case, Find Business Workers and Entities, and Detail a Business Entity tasks of the Business Modeling discipline of RUP (see Figure 3), but starting from the actual implementation of the business process with business roles and entities.
Figure 3: Documenting the supported business process and the domain model

In order to help find the domain model, one must analyze the actual database tables and record them in a Data Model work product. This is the result of a new Database Analysis task that is not part of the standard RUP, but close to the reverse-engineering part of the Database Design task in the Analysis & Design discipline (see Figure 4).

Figure 4: Analyzing the actual database architecture

At the same time, one identifies the system use cases the users execute while performing their business tasks. This lets us rebuild the System Use-Cases model through the Find Actors and Use Cases task of the Requirements discipline (Figure 5).

Figure 5: Building the use-case model

Figure 6 summarizes the initial models we have rebuilt so far. At the top are the actual users (system actors) together with the use cases we’ve rebuilt. These actors correspond to the business workers in the Business Analysis Model. Together, the tasks these business workers perform represent the steps of the business process the software supports. On the bottom of the figure are the redocumented database tables. Some of them will correspond to the business entities of the Business Analysis Model.
Before proceeding with the detail of the use cases, we must schedule the work according to the Risk List and Business Case. Then the reverse-engineering requirements (use cases) are prioritized in the Prioritize the Use Case task of the Requirements discipline to produce the Reverse-Engineering Requirement Attributes work product, which is similar to the Requirement Attributes repository of RUP (Figure 7). As in traditional RUP, the risk list is updated after we have assessed the current iteration and the difficulties encountered (Iteration Assessment work product outputted by the Assess Iteration task of the Project Management discipline).

The next task is to Detail a Use Case (from the Requirements discipline) to produce the flow of events of the selected use cases. However, since the use cases are recovered from the user and not from the developers, they are unlikely to be complete. But they nonetheless represent the main part of the functionality of the system. Once some of the use cases are detailed, we build their Analysis Models by the standard Use-Case Analysis task within the Analysis & Design discipline of RUP (Figure 8).
Figure 8: Use-case detail and analysis

In this analysis activity, we use the mapping techniques documented in RUP and especially the heuristics to find the analysis objects from the system use cases. Basically, the business entities become system entities, the interfaces to the actors (for example, the screens of the application) become the boundary objects, and the responsibility for the coordination of a use case is represented as a control object. Figure 9 presents the traceability links between the model elements. This Analysis Model work product represents a hypothetical architecture of the system. It is the best guess we can have about the architecture of the system at this point in time. In the actual source code, we could expect to find software elements that carry out the responsibilities of the boundary objects and the entity objects in one form or another. However, the responsibilities of the control objects will likely be scattered among many software elements. In summary, this Analysis Model is used to record what we can expect to find in the software as far as the responsibilities of the elements are concerned.

Figure 9: The Use-Case Analysis Model and its traceability links to the other model elements. These links are built using the standard RUP heuristics.
Recover the architecture of the software

From the recovered abstract structure of the legacy system (the Analysis Model associated to the System Use Cases), we must now find the components of the information system that implement it. The problem is to create the traceability link between the high-level analysis model elements and the low-level software components. Our approach is "domain-driven" -- i.e., we cluster the software elements according to the supported business tasks and functions. This step is the one that differs most from traditional RUP practices, although where possible, we will draw some parallels with RUP.

This step contains the following tasks:

- Analyze the Implementation Model
- Run the use cases
- Analyze the call graph
- Map the functions to the Implementation Model
- Validate the hypothetical architecture
- Rebuild the high-level architecture

Analyze the Implementation Model

The first task in the reconstruction of the software architecture is to document the Implementation Model of the system. At this stage, one cannot display all the dependency relationships among the elements of the model. One can only represent the containment relationship among the directories, libraries, files, classes, and packages. Figure 10 shows a simple structure of an object-oriented system. In the case of a procedural program, this figure would show files, libraries, and directories.

![Figure 10: The partial Implementation Model](image)

In standard RUP, the Implementation Model work product is the output of the Structure the Implementation Model task of the Implementation discipline (Figure 11). However, in our process, the software architect works backwards from the code, hence the name of the task. This model will be complemented by the tasks that follow in the process.
Figure 11: Structuring the Implementation Model

Run the use cases

In this task, we start identifying the code that implements the use cases. First, we must choose the business task and associated use cases to work out according to the priority list we set up. Since we cannot run the use cases with all possible input values, we must restrict ourselves to the typical values as advised by the users of the system. The latter are recorded in the Execution Case work product from the Define Execution Details task. It specifies the input parameters and execution conditions of the use cases (this work product and task have been named by analogy to the Test Case and Define Test Details of RUP). Then we run the selected use cases and record the execution trace of the software -- i.e., the sequence of functions or methods that is executed. Figure 12 represents the business worker with his corresponding system use case. When this use case is executed according to the Execution case, an execution trace is recorded. This is done either by instrumenting the code, using a debugger, or instrumenting the execution environment. These tasks are performed by a new role: the Use-Case Analyst (Figure 13).

Figure 12: The execution trace of the use case is recorded

Analyze the call graph

As noted earlier, the use case's flows of events retrieved from interaction with the users are unlikely to be complete. We should certainly not expect the alternative flow of events to be exhaustive. Consequently, an execution trace of such a use case will not execute all the functions that actually implement the use case. To complement them, we must transitively find all the
functions that are called by the functions of the trace. Then we perform a static analysis of the code to build a context-insensitive call graph, starting from the function of the trace. Such a call graph is a pair (N,R), where the nodes in the set N are functions and R is the "call" relation; i.e., there is an edge between the functions $f_1$ and $f_2$ if the implementation of $f_1$ includes a call to $f_2$ regardless of the conditions of such a call. The set of functions to retrieve -- the extended set of functions of the use case -- are the functions that could potentially be executed when running all the possible scenarios of the use case. In fact, the Extended Set of Functions work product will likely contain more functions than strictly necessary. This can be sorted out, to a certain extent, by correlating the extended set of functions of all the use cases.

Figure 14 represents the call graph that contains the extended set of functions for the use case. The functions colored red are those of the traces, and the functions colored yellow are those that are transitively called from the former. In the text below, the extended set of functions will be represented by such a graph. This task is performed by a new role, the Implementation Analyst (Figure 15).

![Figure 14: The call graph representing the extended set of functions](image)

![Figure 15: Analyzing the call graph associated to the traced functions](image)

Map the functions to the Implementation Model

Once the extended set of functions for a given use case is recorded, we must map these functions to the recovered Implementation Model. Basically, all the functions must be part of some class or file that has been identified in it. This mapping will let us find the elements of the Implementation
Model that are involved in the implementation of the use cases of a given business task. In Figure 16, all the classes and packages colored yellow participate in the implementation of Use Case 1, i.e., the business task performed by User 1.

![Diagram of Model](image)

**Figure 16: Mapping from the extended set of functions to the Implementation Model**

In non-object-oriented software, the classes in this figure would be replaced by files. The output of this task is recorded in an updated Software Architecture Document. This task is performed by the Implementation Analyst (Figure 17).

![Diagram of Analyst](image)

**Figure 17: Analyzing the mapping of functions to the Implementation Model**

**Validate the hypothetical architecture**

At this point in the process, we know the functions/procedures/classes/files/packages that are involved in the implementation of the use cases. We must now use this knowledge to validate the hypothetical architecture of the code (Analysis Model) that we conjectured in an earlier step. First, the source code of the extended set of functions is searched for any table or file access. This represents the data structures that are actually (or potentially) accessed during the use-case execution. Second, once these tables or files are found, the classes or files that contain the functions that access them are compared to what was expected in the Analysis Model. If necessary, the model must be corrected accordingly. Depending on the legacy system under analysis, the tables or files may be declared both outside the program code (for example, in a
COBOL batch program running under IBM MVS, the files accessed are declared in the job control language) and/or directly within it. The technique to retrieve this information is highly system-specific, but it is generally feasible as each language has an I/O-specific statement to be searched for in the code. Figure 18 represents the validation of the entity objects. On the one hand, the extended set of functions lets us find the accessed database tables. On the other hand, these I/O functions (methods) are contained in implementation classes. Then the latter are matched against the expected entity objects.

![Figure 18: Validating the entity objects](image)

Next, we proceed with the boundary objects (screens and interfaces). In this case, the source code of the extended set of functions is searched for any screen-related functions. Then the containing classes or files are compared to what was expected in the Analysis Model, and, if necessary, the model is corrected. These screen-related functions are highly specific to the programming environment and programming language considered. Normally, they are well-documented in the programming manuals, and their identification should not be too difficult. Figure 19 represents the validation of the boundary objects. On the right, the extended set of functions is searched for the screen-related functions using the programming environment's documentation. Then the containing classes are identified. These classes are then matched against the expected boundary objects (left).
Figure 19: Validating the boundary objects

As a first guess, the remaining classes (from the mapping of the extended set of functions to the Implementation Model) are associated to the control object as shown in the summary map of Figure 20. The dotted lines represent the recovered traceability link from the Analysis Model to the Implementation Model. The output of this task is recorded in an updated Software Architecture Document. This task is performed by the Implementation Analyst (Figure 21).

Figure 20: Map of the analysis classes to the implementation classes
Rebuild the high-level architecture

In this activity, we must sort out the code that is specific to each use case and rebuild the corresponding high-level architecture of the code. First, we must work at the class and file level. Then we go deeper into the code. Let us have the set UC of use cases that have been reverse-engineered UC= \{UC_1,...,UC_2\} and let us define the following three functions:

- \( \text{classes}(UC_i) \): returns the set of classes associated to the use case \( UC_i \). These are the classes that contain the extended set of functions for the use case.
- \( \text{specific}(UC_i) \): returns the set of classes specific to the use case \( UC_i \) that contain the functions that belong only to the extended set of functions of this use case.
- \( \text{common}({UC_1,...,UC_k}) \): returns the set of classes common to the set of use cases \{UC_1,...,UC_k\}.

Then we have:

\[
\text{specific}(UC_i) = \text{classes}(UC_i) \setminus \bigcup_{UC_j \neq UC_i} \text{classes}(UC_j)
\]

The set of common classes for the set of use cases is simply defined as:

\[
\text{common}(UC) = \bigcap_{UC_i \in UC} \text{classes}(UC_i)
\]

This lets us draw a map of the high-level elements involved in the implementation of a set of use cases. Figure 22 represents both the elements that are unique to a given use case (same color as the use case) and the common elements (green). The packages that contain use-case-specific elements are colored the same as the use case. Those that contain common elements are colored green. Please note that these colors are only relevant to the current state of reverse-engineering. It is likely that the colors will change as more use cases are included in the analysis.
Figure 22: Analyzing specific and common implementation classes among use cases

When all the use cases that belong to the target business task are processed, we must group the classes (files) according to the use case they implement and record the possible mapping to the analysis objects. Once this is done for all the use cases, we analyze the Implementation Model of the code to check if some logical grouping of the classes (files) already exists in the code. For example, these groups would correspond to the packages and/or directories. Among the possible grouping paradigms we have:

- Grouping by analysis object type (example: the classes that implement the boundaries)
- Grouping by use case (example: the classes that implement a given use case are grouped)
- Grouping by information source (example: the classes that access a given database)
- Grouping by actor (example: the classes that implement the interaction with some user role)

Moreover, a given structure could be based on several criteria simultaneously. As a result of this step, we can sketch the high-level structure of the code according to the discovered grouping. For example, Figure 23 represents a grouping by use case, where a common subfunction use case to both UseCase1 and UseCase2 (Subfunction Use Case 1) would have been analyzed. However, the figure also shows that some common code may well not belong to the implementation of the subfunction use case (UC-common 1-2).
Figure 23: Grouping implementation classes by use cases

Figure 24 blends together the two previous architectural representations by showing the correspondence with the analysis objects. This structure may not be found in the code however and is shown only to illustrate the information recovered so far.

Figure 24: Grouping implementation classes by use cases and responsibilities

It is important to note that even if some logical grouping was made when the program was initially developed, it might well be the case that these groupings were not respected during the years of system maintenance. In this case, some apparent grouping may contain classes that do not respect the criterion. The output of this task is recorded in an updated Software Architecture Document. This reverse-architecting task is performed by the Software Architect (Figure 25).

Figure 25: Rebuilding the high-level architecture
Summary of our process

Table 1 summarizes the main tasks and work products of our process. The elements colored green are the same as defined in RUP. Since the third step of our process deals with actual code analysis, there is no equivalent discipline in RUP. Therefore, the new Database Analysis task is associated to the Analysis & Design discipline. On the other hand, the reconstruction of the high-level architecture from the execution of the use cases are associated to the Implementation discipline. This leads us to define two new roles in the process: the Use-Case Analyst and the Implementation Analyst as well as seven new tasks.

Table 1: Main tasks and work products of our process

<table>
<thead>
<tr>
<th>Discipline</th>
<th>Task</th>
<th>Work Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>Business Modeling</td>
<td>Detail a Business Use Case</td>
<td>Business Use Case</td>
</tr>
<tr>
<td>Business Modeling</td>
<td>Find Business Workers and Entities</td>
<td>Business Analysis model</td>
</tr>
<tr>
<td>Business Modeling</td>
<td>Detail a Business Entity</td>
<td>Business Entity</td>
</tr>
<tr>
<td>Requirements</td>
<td>Develop Vision</td>
<td>Vision</td>
</tr>
<tr>
<td>Requirements</td>
<td>Find Actors and Use Cases</td>
<td>Supplementary Specifications</td>
</tr>
<tr>
<td>Requirements</td>
<td>Find Actors and Use Cases</td>
<td>System Use-Cases model</td>
</tr>
<tr>
<td>Requirement</td>
<td>Prioritize the Use case</td>
<td>Reverse-Engineering</td>
</tr>
<tr>
<td>Requirement</td>
<td>Detail a Use case</td>
<td>Requirement Attributes</td>
</tr>
<tr>
<td>Analysis &amp; Design</td>
<td>Database Analysis</td>
<td>Data Model</td>
</tr>
<tr>
<td>Analysis &amp; Design</td>
<td>Use-Case Analysis</td>
<td>Analysis Model</td>
</tr>
<tr>
<td>Implementation</td>
<td>Analyze the Implementation Model</td>
<td>Implementation Model</td>
</tr>
<tr>
<td>Implementation</td>
<td>Define Execution Details</td>
<td>Execution Case</td>
</tr>
<tr>
<td>Implementation</td>
<td>Run the use cases</td>
<td>Execution Trace</td>
</tr>
<tr>
<td>Implementation</td>
<td>Analyze the call graph</td>
<td>Extended Set of Functions</td>
</tr>
<tr>
<td>Implementation</td>
<td>Map Function to Implementation Model</td>
<td>Software Architecture Document</td>
</tr>
<tr>
<td>Implementation</td>
<td>Validate the Hypothetical Architecture</td>
<td>Software Architecture Document</td>
</tr>
<tr>
<td>Implementation</td>
<td>Rebuild the High Level Architecture</td>
<td>Software Architecture Document</td>
</tr>
<tr>
<td>Project management</td>
<td>Develop Business Case</td>
<td>Business Case</td>
</tr>
<tr>
<td>Project management</td>
<td>Identify and Assess Risks</td>
<td>Risk List</td>
</tr>
<tr>
<td>Project management</td>
<td>Assess Iteration</td>
<td>Iteration Assessment</td>
</tr>
</tbody>
</table>

Milestones in this process

The goal of a reengineering project is to restructure a legacy software system so that some of its quality attributes are improved. This restructuring could imply a change in the technological framework, such as a migration to an SOA framework. Whatever the technology, the reverse-engineering project should assess the feasibility of the restructuring. In fact, even if the target of the reverse-engineering is to remodel a system, it must also prototype the migration of some key component to the new architecture to check that it is possible.

The inception phase is intended to assess the feasibility of the project. In particular, the project manager should check that a sufficient number of users can be accessed and queried, that the current database tables and source code are accessible, and that the proper tools are known and available (for example, for recording the execution trace). Then some critical parts of the system should be reverse-engineered and the migration path to the new platform validated. Finally, the
majors risks must be understood. At the end of the inception phase, the go/no-go decision can be made.

The elaboration phase is intended to reverse-engineer the critical parts of the system and redocument their high-level architecture. In particular, the actual grouping paradigm of the classes (files) of the legacy code, if any, should be known. Moreover, the reverse-engineering environment and tools should be put in place. At the end of the inception phase, the architecture of the critical part to reverse-engineer should be documented.

During the construction phase, all the components in the scope of the project are reverse-engineered. Then, at end of the inception phase, the system is fully redocumented.

During the transition phase, the models and documentation of the legacy system are transferred to the team that will forward-engineer the new system.

**Conclusion**

I have shown that most of the tasks and work products that we need on our reverse-engineering process are found in the RUP toolbox or are close to them. It is worth mentioning the central role played by the use-case model in this work as in any RUP-based software development project. First, the use cases help us to redocument the business process model that is supported by the software. Second, the use cases help us build a hypothetical architecture of the legacy system. Third, the use cases are the source of scenarios to be run to gather the execution trace of the system. In some sense, the use-case model lets us make the link between the high-level business function and the low-level code that implement these functions.

One of the key problems in reverse-engineering a legacy software system is to understand the code and build an architectural representation of it. However, these two tasks are dependent on each other and the problem is always: how to start? With the reconstruction of the UML models through RUP, I have demonstrated one solution.

**Notes**


