Playing the Software Archeologist: Exploring and Conquering an Unknown Legacy System

Software Archeology Workshop, OOPSLA 2001

Position Paper

Walter A. Risi

LIFIA, Facultad de Informática, Universidad Nacional de La Plata.
C.C.11, Correo Central, 1900, La Plata, Buenos Aires, República Argentina.
walter@lifia.info.unlp.edu.ar
URL: http://www-lifia.info.unlp.edu.ar/

Abstract

Understanding and modifying unknown legacy systems is a task that has challenged software engineers for years. While conventional approaches like reverse engineering and reengineering are well known and proven effective, there are cases where these techniques are too heavyweight or time-consuming.

This position paper shows an example of a real situation in which we had to add functionality to an existing legacy system while dealing with a tight schedule. We share our experience and a few techniques that we found useful while digging in the depths of legacy code. We follow the *software archeology* metaphor and present our techniques as ways to *explore* and *conquer* unknown systems.

1 Introduction

Understanding and modifying unknown legacy systems is a task that has challenged software engineers for years. In spite of the high maturity reached in the software reengineering field [1], there are cases when time constraints are too tight to apply such a heavyweight approach. This paper shows an example of a real situation in which we had to add functionality to an existing legacy system while dealing with a tight schedule. We share our experience and a few techniques that we found useful while digging in the depths of legacy code. We follow the *software archeology* metaphor [2] and present our techniques as ways to *explore* and *conquer* unknown systems.

The paper is structured as follows. In section 2, we present an experience report based on a real case we were through. Section 3 shows a number of techniques we isolated from our experience, and shows examples of their usage related to the case previously commented. Finally, in section 4 we draw some conclusions.

2 A Software Archeology Tale

We work in a group developing complex, multi-regional software for large financial institutions. While our group designs and develops its own applications as required by clients, we also work with applications that clients already had – providing support, enhancements and ultimately reengineering. For the latter group, we generally have to deal with obscure code and poor or nonexistent documentation about application design and architecture.

In this section we explain the challenges we faced when enhancing an existing Risk Management System. This application allows clients to know the actual value of their financial positions (consisting in several types of

products) and how would their positions be affected if market conditions change in a number of predefined ways. The system relies on a *valuation engine* which uses *curves* – data structures representing the conditions of a particular financial market – to calculate interest rates and discount factors. The latter are used to calculate the value of the mentioned financial positions.
Although the methodology used by the valuation engine was correct, the system was somewhat restrictive in the definition of those curves. Some clients were already asking for a more powerful curve definition facility. This would enable them to use our application for more complex financial products.

2.1 Our Mission

Our mission consisted in enhancing the system to support the definition and usage of more powerful curves. We first had to identify those parts of the application where curves were defined and used. Then, we had to modify those parts as required. Additionally, we had about a month to have a working version ready.
Before starting to work on the system, our knowledge of its internal structure was very limited. We knew that the system used a – fairly well-documented – financial library (FLIB [1] from now on) for most of its financial calculations. We knew that the system used four particular functions from that library to construct the curves and calculate rates from them. Finally, we knew that the system implemented its business logic in a *C Middle Tier*, while the presentation issues were handled by a *Java Front End* and the data persistence layer was implemented using a *Sybase Relational Database*. However, we had no knowledge at all about the design of any of those layers.

2.2 Initial Exploration

Since we knew almost nothing of the design of the system, our first task was to achieve the necessary knowledge to be able to perform our work. Again, we had not enough time to perform a complete analysis of the system and thus required a faster, lighter strategy.
Fortunately, we knew that the system was using the FLIB library for its internal calculations. We also knew that a necessary step for valuation was the construction of the yield curve upon which valuations were performed. Since we had some documentation about the FLIB library, we knew the names of the functions used for yield curve construction.
Additionally, we knew that the business logic was implemented in the C Middle Tier, so we had to search for FLIB function invocations in that part of the application. Following our assumptions, we searched for those places where these functions were used. That approach left us with a two modules where the yield curve was built.
Since we were dealing with a C program, the next step was knowing which were the data types involved in the calculation. This part was easy, since we only had to see where the function was taking its parameters from and which was the type of the result. Once we knew which were the involved types, we could easily find where instances of these types were retrieved from the database and which were the associated database tables.
Up to this point, we concluded that this initial exploration phase reached its end. We still had little knowledge about the system, but we knew were the function invocations were made and thus where to add the additional cases. Since we had also discovered the data type where parameters were taken from, we also knew that we had to extend or modify that data type. Finally, we knew which were the database tables we had to extend to support the persistence of the modified type.

2.3 Planning the Enhancement

Finished the first phase, we still had to plan how we were going to proceed. The obvious approach was to modify types and function invocations using our own style – disregarding the application's own design and coding style. However, that approach had the implied risk of making the application a hybrid of coding styles – with the consequent negative impact on maintainability.

[1] The Applications and Library Names are Changed for Code Ownership Reasons

Therefore, we took some time looking at the application's existing design and coding styles. Some of them were acceptable and some of them were clear examples of bad design (e.g. the use of `GOTO`s instead of structured constructs). Although we were tempted to redesign the whole thing, we knew that we simply didn't have time to do a full reengineering job.
The approach we took was to evaluate each style and decide which to keep and which to redesign, taking into account both pragmatism and design correctness. We knew that once we decided to change a style, we had to change it in the whole system and that was a risk we had to deal with.
We reached the end of this phase once we had an informal criteria of what to keep and what to rewrite. We would guide the implementation of the new features using this criteria, and we would change existing code were adequate so as to keep a coherent style. We also agreed that we would not apply the required redesigns to the whole system, but only to the modules involved in the enhancement.

2.4 Implementing the Enhancement

The implementation phase consisted in the concrete implementation of the enhancement and a moderate redesign aspect, based on the criteria we previously agreed with.
We thus started modifying the environment of the enhancement to adopt to the styles we were going to use. We were particularly careful at ensuring that we were not introducing bugs when redesigning, and thus we worked in incremental builds, where each build was subjected to regression tests.
At this stage, we had only to introduce the enhancement. In the previous phase, we had prepared the invocations to the constructor functions to accept new types of constructors, and thus introducing this particular new one was straightforward. Since constructors have all the same return type (defined in the FLIB library) we knew that the rest of the application would work seamlessly with this enhancement (note that the return type is treated by FLIB as a polymorphic type, although its implementation actually is not). Again, we compiled the modified modules and performed severe tests on them.

2.5 Epilogue

Our implementation was successful and relatively on time. Of course, it was not as clean and cascade-like as explained here. Indeed we performed several iterations on each of the phases. Needless to say, we introduced and corrected defects, made mistakes and found technical difficulties until we reached success.
However, the main flow of activities was mainly as explained here. The difficulties we found and mistakes we made were natural, considering we were dealing with this kind of task for almost the first time. We were involved in similar activities after this particular experience and followed the same process with good results.
Note that we did not mention anything about the process followed to modify the Java Front End. Indeed, we followed the same approach. Since this layer only involves presentation issues, we centered our report in the business layer enhancement – which comprised the most difficult part of our work.
We would like to point out that the whole process was based mainly on three points: capitalizing on what we already knew, understanding the current design and coding styles, and respecting those styles (whether *respect* means either keeping them or changing them wisely).

3 Techniques Learned from Exploration and Conquest

Our work with this legacy system taught us several techniques we found useful for similar jobs. In this section, we comment each of the techniques we learned and show examples of their usage. We follow the *software archeology* metaphor to name each technique.

3.1 Establish your Base on a Safe Place

The first problem a developer finds when confronted with an unknown system is having no idea about where to start. Our approach consists in establishing your start in a place you relatively know.

While you may know almost nothing about the system, you probably know that it uses a particular library, or that you want to modify a particular feature you can easily identify. Searching for that feature over the whole system will restrict the search to a few known places. Establish your base there, and start inspecting the code from those places you understand better.

> *In our particular case, we knew financial valuation was strongly based on the FLIB library, and we knew the potential names of the functions involved. We thus searched for those names, and found a few places where they were used. Further inspection showed which of those places were the right places to start.*

3.2 Understand and Respect Ancient Cultures

Since we cannot do a complete reverse engineering and posterior reengineering, we must be particularly careful at understanding the existing design and coding styles. Not only this would help to understand the system better, but it would also help us to plan the enhancement.
When we talk about planning the enhancement, we mean defining the style we will use for the new code. If we use a code style radically different from the existing one, we risk turning the system into an hybrid – which may become very difficult to maintain. If we really want to change the style, we must accept that this will imply changing the style in several places to leave the application in a coherent state. Defining an ambitious criteria for redesign has the implied responsibility of respecting it.

> *In our example, we performed an informal review of the existing style and established a criteria. Once defined, we changed and added code strictly following that informal standard.*

3.3 For Each Conquered Area, Ensure it Remains Conquered

Although we are following a light approach to legacy system enhancement, that does not imply we must disregard good programming practices. Far from reengineering, we still must ensure that when we achieve understanding of a part of the system, we will still have it in the future (e.g. documenting it appropriately). In the same way, when we redesign or rewrite some part of the application, we should take care that the new code is not likely to be changed again in the next enhancement.
We are following a pragmatic approach, not a negligent one. When we *conquer* some part of the application (e.g. adding comments or redesigning code) we must ensure we really conquer it. If the new comments are lousy or the new code is buggy, we haven't really conquered anything.

> *In our particular example, we found several cases of code badly commented and functions badly named. We decided to comment those obscure parts and redesign using the best practices at hand.*

3.4 Be Careful when Handling Ancient Artifacts

A real archeologist is extremely careful when handling ancient relics and artifacts, in order to avoid spoiling historical treasures. The same applies to software archeology, and thus we must be very careful when introducing variations to the existing code. The least we want is to introduce defects in code that was working perfectly before we modified it.
When we modify a portion of the existing application we must be very careful at ensuring that no existing functionality was lost or corrupted. Therefore, we must perform a rigorous regression test on that code. Other quality assurance techniques like code-reviews [3] and functional verification [4] can also offer valuable help. Pair programming – as in *Extreme Programming* [5] – can be used as a low-cost alternative to peer reviews.

> *In our example, we performed regression tests on each feature we decided to redesign. Although we did not use code reviews when we first introduced the enhancement, we did use them on further iterations over the same application. We also did some pair programming on the most difficult parts of our work.*

4 Conclusions and Future Work

Understanding and modifying unknown legacy systems can be a real challenge when time constraints are tight. Traditional methods like reengineering are simply too expensive and time consuming for these cases. This paper showed a real experience at modifying a legacy system and the lessons we learned during our task. The techniques we presented here provide general guidelines for the software engineer confronted with this kind of situation. We learned those techniques from actual cases and proved them useful through further experiences.

While the techniques we mentioned are useful by themselves, they do not prescribe a structured method to follow. We are currently considering organizing these techniques into a methodological framework. This comprises our main line of future work.

References

[1] John Bergey, Dennis Smith, Nelson Weiderman, and Steven Woods. Options Analysis for Reengineering (OAR): Issues and Conceptual Approach. Technical report, Software Engineering Institute, 1996. (CMU/SEI-99-TN-014).

[2] Brian Marick. Wiki Forum on Software Archeology. `http://wiki.cs.uiuc.edu/Visualworkings/`.

[3] Watts S. Humphrey. *A Discipline for Software Engineering*. Addison-Wesley, January 1995.

[4] Stacy J. Prowell, Carmen J. Trammell, Richard C. Linger, and Jesse H. Poore. *Cleanroom Software Engineering: Technology and Process*. Addison-Wesley, April 1999.

[5] Kent Beck. *Extreme Programming Explained: Embrace Change*. Addison-Wesley, 1999.