Refactoring for Software Migration

Dennis Mancl, Lucent Technologies

ABSTRACT

Refactoring can be an important ingredient in the strategy for development and evolution of complex telecommunications software systems. Refactoring is one way to reuse and extend a successful software system. This article will present some design tactics that will assist a development team when they choose evolution from an existing software system over building a completely new system from the ground up. Strictly speaking, the refactoring process changes only the internal design of the software. Refactoring does not add any new functionality. However, the goal of refactoring work is to pave the way for the software to be modified and extended more easily. The simplest example of redesign is the creation of simple "wrapper classes" that contain groups of functions extracted from the legacy code. More complex design patterns are also useful when attempting to improve the design. Developers usually divide the redesign work into stages so that each stage can be implemented and tested separately. This article describes a real-world example of this approach that shows how refactoring improved the design of a wireless base station controller product.

WHY REFACTOR?

Refactoring saves development time and effort by reusing much of the existing design and code. The refactoring process also ensures that the new software will fit in with the same interfaces as the old software.

A new replacement software system (e.g., a new-generation product that supersedes an old product line, or a higher-performance system within an existing product line) may need to reimplement a large share of the basic functionality of the system it is replacing. It is tempting to do the development of such a system completely from scratch. Old software sometimes deserves to be discarded — the accumulation of changes from fixing bugs and adding features may have made the software so brittle that any attempt to make additions might be more time-consuming than a complete rewrite. But it can be difficult and expensive to completely redesign and recode data structures, message formats, file formats, and algorithms for a sizable system. The refactoring approach is a compromise between doing a complete system rewrite and piling on more small system changes.

For many complex problems, compatibility with a previous product is important. A system that is built by refactoring an existing system and adding new features to the refactored system is much more likely to be compatible with the existing system than a system built from scratch based on the existing system's requirements. In a big system, a complete rewrite might misimplement some of the features of the old system. This is a common problem for a new development team working with incomplete documentation and with no previous experience in the problem domain.

CASE STUDY: A WIRELESS BASE STATION CONTROLLER SYSTEM

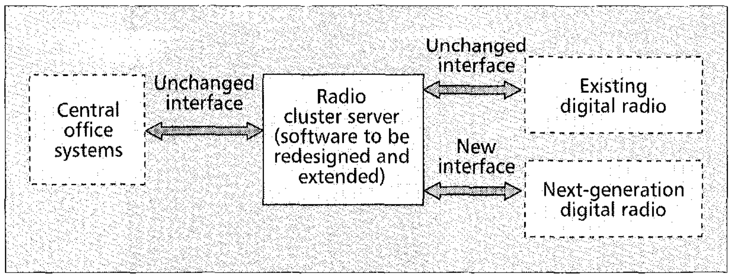
Lucent Technologies builds software systems that control different versions of wireless base station hardware. This case study presents the experiences in the recent evolution of one of these wireless base station controller systems: a radio cluster server. This software system contains two major subsystems: call processing, and operations, administration, and maintenance (OA&M) functions. The base station software has become increasingly complex over time as wireless networks have been extended to implement more features. This case study describes one recent development effort in which the development team chose to build on the existing software, but also decided to do some refactoring of the design to prepare the way for adding some new features (Fig. 1).

The software for the radio cluster server system was originally well-designed for the system's architecture. It initially had a small memory footprint and a simple, modular structure. Some of the most important characteristics of the legacy system are:

- **Purpose:** The purpose of the system was to handle the lower-level details of the main call scenarios. Each scenario was triggered by either a message sent down from a supervisory central office system or a message sent up from the cell's digital radio hardware.
- **Communications to external systems:** The system communicated with other parts of the network by passing messages to a supervisory central office system and to the cell's radio hardware and software.
- **Basic architectural style:** The radio cluster server software's call processing subsystem was divided into four main tasks. Each task was dedicated to processing a limited set of messages. Each task had its own message queue, but all of the internal data structures were shared among all of the tasks. The application environment ran only a single execution thread at a time, so the system gave each task an opportunity to perform processing on the top message in its input queue. Most of the end-to-end scenarios were implemented as a series of relatively short segments of code in a huge switch statement.

The new base station controller software system had some characteristics that merited building on a refactoring of the existing system:

- **Old and new hardware interfaces:** The existing system had to be extended to communicate with newer-generation radio hardware and support new call types. The older radio hardware also had to be supported. In the new system, some OA&M operations needed to be added for the new radio hardware, but the existing administrative operations for the older radio hardware could be preserved by reusing existing code.
- **The existing software was not limited by CPU performance:** The existing system implemented many of the OA&M operations as efficient straight-line C code. Although some efficiency would be lost, similar operations for the newer radio hardware were added by replacing the old straight-line operations with if-then-else logic or function calls. Fortunately, all of the current generation of radio cluster servers had a CPU that was fast enough to allow less efficient but better structured code to continue to meet the performance requirements.
- **Unchanged interfaces to the upstream system:** No changes were being made to the supervisory central office system. The upward-bound messages in the principal call processing code needed to stay stable. For this reason, it made sense to reuse many of the software modules that formatted, transmitted, received, and parsed messages to and from that system.



■ **Figure 1.** *The radio cluster server's interfaces (a simplified view).*

Discovery and Design Work in the Refactoring Process

Refactoring is a software process that follows an unconventional analysis-design-coding-testing life cycle. The process is unconventional because most of the analysis is "discovery," the design is "redesign," the coding is mostly "code copying" or "code transformation," and the testing is "retesting" to ensure that the system still performs the same operations as before.

The work of extending an old legacy system has some real obstacles. Some legacy code can be a problem to work with, because its design documentation may be out of date and completely useless. Refactoring will not be successful in making the system easier to extend unless the refactoring team has a good understanding of the structure of the existing system. The refactoring team will perform some discovery activities followed by some design activities before making any changes to the legacy code.

In the case study, the development team performed some discovery activities at the very beginning of the project:

- The team read and reviewed the code and design documents from the existing base station controller system.
- The team wrote summary-level use cases that describe the interactions between the system and other parts of the network.

In the case study the use cases helped to focus the design activities. Since it is usually impossible to do a complete system redesign, the use cases usually help narrow the redesign effort.

Some of the redesign work was simplified by using some of the standard object-oriented design patterns. Developers use two common object-oriented patterns in the evolution of an existing system: wrapper classes and strategy classes. Complex legacy data structures and hardware-dependent system configuration scenarios are two typical places where legacy system code can be hidden within a wrapper class. Strategy classes can be introduced into the legacy design to permit new variations of an algorithm to be introduced during the evolution process.

The coding part of a refactoring operation is much easier to do in small steps. In the case study, the new wrapper classes and strategy classes were introduced in a way that permitted the complete system to be recompiled and retested at any point.

Call processing use case 1: Call Origination
Primary actor: DCCH radio (DCCH = digital control channel)
Preconditions: DCCH radio is broadcasting; Base station call processing is active.
Success end condition: A Voice channel will be allocated.
Sunny-day scenario:
1. System receives an origination message from DCCH radio.
2. System asks OA&M subsystem for server data.
3. System asks Channel control system for a channel; it returns the id of the channel it allocates.
4. System orders Voice radio to activate itself.
5. System sends DCCH radio a message to inform the Mobile which Voice radio channel is assigned to it.
6. Voice radio sends a message to the system to indicate that the Mobile has successfully accessed the Voice radio.

Failure cases and extensions:
3a. Channel control system fails to allocate a channel:
3a1. Attempt to recover from allocation failure of abandon call.

■ **Figure 2.** *A use case example.*

Discovery Work in the Case Study

In every refactoring effort, the development team must go through a discovery process [1]. In this process the development team learns about the problem domain and the requirements of the original system. The development team may also need to learn details about the structure of the existing software: its design paradigms, programming languages, development environment, and execution environment.

The legacy code in the wireless base station controller system was reasonably modular, divided into four major tasks that were each responsible for processing different kinds of messages. Each task could access the global data structures that contained information common to all of the tasks, such as the call register (a data structure that contains one record per active phone call).

Each task had been modified gradually over many releases. The addition of new features made the structure of the software more and more complex. The new features were often inserted into the structure of the original application by adding complex conditional logic; many if-then-else constructs added into the code made the basic application logic more difficult to understand.

This modular structure helped simplify the redesign work, as long as the development team limited themselves to working within the structure defined by the original architecture. They could adapt the design and refactor code within a task without significant impact on the other tasks, as long as the outgoing messages sent by the task weren't changed by the redesign.

Using Use Cases in Discovery

The most effective way to start a redesign process is to build a use case model. The use case model is not intended to specify any of the details of the new design. The development team writes use cases in order to have accurate documentation of the high-level interactions between the system and its users. The use cases are an excellent discovery tool: they can be used to describe the functionality of the legacy system that must be preserved in any redesign. The processes for documenting use cases are discussed in detail in many current texts (e.g., [2, 3]).

The use case model describes the sequence of high-level system operations that are triggered by external events. Each use case is a collection of high-level scenarios. The scenarios within the use cases describe the communication between the system and external entities. Each scenario can be shown in text form (as a series of sentences) or in graphical form (as a UML sequence diagram). The use cases are a low-resolution view of the interactions: they don't need to specify the details of "how" the communication takes place; they just need to outline the sequence of interactions and give the message type and purpose for each message passed into and out of the system under design.

The use case process has three main steps:
1. Define the "actors" (the entities outside the system under design that communicate with the system).
2. Document the "sunny-day scenarios" that describe the normal course of events when the system achieves its main goals.
3. Add in "failure scenarios" that describe how to recover from errors.

In the wireless base station controller project, the first two steps (defining actors and sunny-day scenarios) were finished within two weeks. The most important actors were the upstream (central office) and downstream (radio hardware) systems. The set of use cases was not very complex: 10 use cases that described the basic call processing scenarios (e.g., call setup, paging, handoff, and disconnect), plus another 35 simple use cases for the OA&M functions such as the monitoring and control of the low-level radio facilities by the base station software.

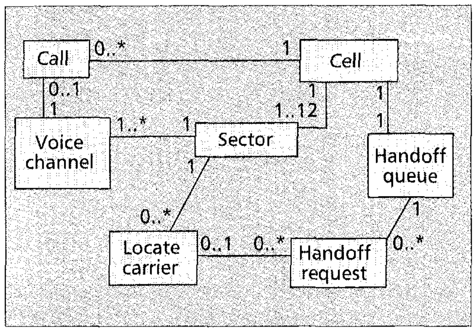
Figure 2 shows an example use case from the early part of the discovery process.

The team quickly determined that most of the use cases were almost identical for the old and new systems. That is, the description of scenarios for the old system could describe the behavior of the new improved system with some minor internal variations. A massive system redesign would not really be necessary. But the development team did discover a couple of use cases where the necessary modifications would not be so simple: those that described the processing of call handoffs. The handoff algorithms were a part of the system where many revisions had been made already, and the existing code for these scenarios was quite complex.

Building a Simple Object-Oriented Design Model

The second model created by a redesign team is a high-level object model. This can be started in several ways: when an existing legacy system is the basis of development, the object model will not always be pure, but the object model will be another good discovery artifact.

Some of the "candidate objects" in the object model of a non-object-oriented legacy system include:



■ **Figure 3.** *A subset of the object model.*

- The main data structures of the existing system
- "Transaction" objects that describe the main work items that are received from the outside, processed, and deleted
- Any entity within the system that has a "state"
- "Interface objects" that encapsulate the application programming interfaces (APIs) or message sets used to communicate to external systems

The base station controller design team used *CRC cards* [4] (CRC: classes, responsibilities, and collaborations) to create a simple high-level object model, but other brainstorming techniques would also work. CRC cards are an informal team object-oriented modeling technique that uses index cards to model the classes, responsibilities, and collaborations in a system. There were only two major kinds of objects in the initial object model of the call processing part of the system:

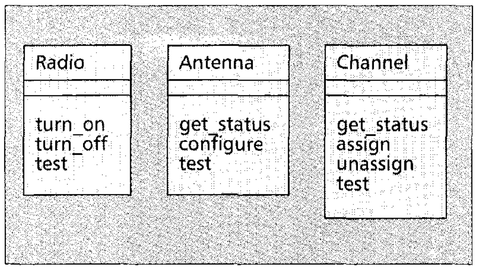
- Long-lived objects that corresponded to data structures maintained in the legacy system
  - Examples: Call, Channel, Carrier, Sector
- "Message objects" that corresponded to messages received and sent by the system
  - Examples: Origination message, Paging message, Request for handoff message, Reply to handoff request message

The object model of the administration part of the system mainly consisted of objects that represented the hardware entities to be controlled. This model was complete within two weeks after the use case modeling, and was converted to a UML class diagram (Fig. 3).

Introducing Refactoring Idioms

In many legacy system reengineering efforts, the design team can't just "work around the edges of the problem" by adding new functions that only work with new messages and new data structures. In real-world redesign, many of the modifications will be intrusive; they will require making changes to the contents of existing data structures and to the control flow within existing functions.

The redesign effort may need to do some selective partitioning of the existing code. Refactoring is an attempt to restructure the code in a way that preserves the original features, but separates the parts of the code that need changes



■ **Figure 4.** *Facade classes.*

into relatively small well-encapsulated modules. Martin Fowler defines refactoring this way: "Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure" [5].

Design Patterns from the object-oriented technology literature [6] are used in many redesign and refactoring efforts. These design idioms are expressed in terms of object-oriented design, but have broad applicability in the reengineering of many kinds of software systems. Two of the design patterns that are useful in redesigns are the *Facade* and *Strategy* patterns.

Facade Classes

A Facade class (also called a *wrapper* or a *wrapper facade*) is an object-oriented view of a non-object-oriented group of functions and data structures. One standard definition of a Facade is "... a set of one or more object-oriented classes that encapsulate existing functions and their associated data. These class(es) export a cohesive abstraction that provides a specific type of functionality" [7].

The redesign process can add one or more Facade classes without changing any functionality in the legacy system. The Facade classes will have each method or member function implemented in terms of the functions that already exist in the legacy application. For example, in the base station controller software system, one class could be defined for each entity (Radio, Antenna, Channel) with operations that provide a simple but complete functional interface (Fig. 4).

Once defined, new functionality that is added to the system should always try to call functions in the Facade classes rather than using the old legacy function calls. The cleanup of the design doesn't have to be done all at once; the development team can slowly replace legacy function calls with calls to Facade class functions. Three positive results from this set of evolutionary changes are:

- The legacy functionality always works the same way it worked before.
- The source code of the system gradually becomes more self-documenting over time.
- The process of introducing extra functionality can be organized better by confining the new functions to the Facade classes.

All three of these results are especially welcome in the development of rapidly evolving telecom software. A new telecom system usually needs to interoperate with other systems that are

The redesign effort may need to do some selective partitioning of the existing code. Refactoring is an attempt to restructure the code in a way that preserves the original features, but separates the parts of the code that need changes into relatively small well-encapsulated modules.

The implementation and testing of the new Strategy-based design was done by a five-person team in conjunction with other additions to the system — in just a few months, even though thousands of lines of code were affected by the changes.

not being replaced, and the new system needs to be at least as reliable as the system it is replacing. In addition, good-quality internal documentation and an orderly migration process are critical for the development team of any software system that has a long lifetime with frequent updates.

One practical Facade design tip: in order to make the transformation from the legacy code to the Facade classes as smooth as possible, it is a good idea to *not* keep any state information within the Facade class. The Facade class may contain pointers into the legacy data structures, but the Facade class functions should always access the legacy data structures to determine the state of the object. This allows the Facade class to work properly even if there are parts of the system that haven't been converted yet, so they still access and modify the legacy data structures directly.

In practice, Facade classes are easy to introduce into legacy code. In the redesign of the wireless base station controller software, the first Facade classes to be added to the system were classes to format and transmit upstream and downstream messages. Other Facade classes were designed to simplify the introduction of new radio hardware by creating an abstraction layer.

STRATEGY CLASSES

A Strategy class is an easy technique to restructure complex if-then-else logic. This technique requires the use of inheritance: an abstract superclass with one or more abstract functions is introduced into the design, plus several concrete subclasses that have different implementations of the abstract functions of the superclass. In the simplest case, a single if-then-else block can be removed by moving the code from one leg of the conditional statement into a function in subclass A and moving the code from the other leg of the conditional statement into the corresponding function in subclass B. The if-then-else block is then replaced by a single polymorphic function call.

This technique is an extremely inefficient way to eliminate single conditional statements, because it creates a design structure that is more elaborate than the simple conditional. The payoff for applying the Strategy idiom is only sufficient for larger applications. Using Strategy pays significant dividends if similar conditional statements are found throughout the system, if nested and complex conditional statements are the rule, and if new variations of the basic Strategy operations are planned in future product releases. These conditions are often true in telecommunications software; new variations are often added to a system to support new hardware and new protocols.

The following illustration is a simplified version of one of the key Strategy classes in the updated wireless system. If the signal strength of any Call falls below a threshold, the central office system (ECP) will be sent a message containing a sorted list of Sectors that are good handoff candidates for the Call.

The actual implementation of this Strategy technique in the wireless base station controller system is more complicated than shown in this example. In fact, the original legacy system had eight different variations in the algorithm to sort the handoff candidates, so the new design was built with eight different concrete subclasses of HandoffStrategy. Some variations have different weightings for uplink and downlink signal strengths, some variations use different calibration and correction algorithms, and some variations are designed with built-in preferences for special sectors.

The code within each concrete Strategy class was not built from scratch; it was extracted from the legacy application by tracing a single execution path for each possible scenario through the entire handoff process. This process ensured that the Strategy class implementation would perform exactly the same operations as the legacy system.

The design details of the new Strategy-based handoff process were difficult, and the Strategy classes somewhat tedious to code. The new design was very easy to test: each variation could be built and tested separately, and it was easy to show that the sequence of messages sent by the new code and the states of the internal data structures were identical in the new implementation and in the legacy code. The development team actually performed some initial tests of the first Strategy class within two weeks of completion of the design. These tests included a performance test, which measured the performance difference between the legacy code and the refactored code. The actual performance of the Strategy class in the refactored system was actually about 2 percent faster than the legacy code, probably due to increased locality of reference within the Strategy classes and the smaller number of branches in the compiled code.

The code structure shown in Fig. 5 is actually a simplified version of the actual redesign. The process of finding the best handoff candidates was divided into five phases, and the HandoffStrategy class therefore contained five abstract functions that were implemented in each concrete subclass. This increased the opportunity to share code between different concrete classes. Two concrete classes, for example, might share the same implementation of their calibration function but have different uplink weighting functions.

The implementation and testing of the new Strategy-based design was done by a five-person team in conjunction with other additions to the system in just a few months, even though thousands of lines of code were affected by the changes. The new design was much cleaner and easier to understand, and it was trivial to add new variations.

SUMMARY

The most important lessons we learned from our legacy system redesign:

- Plan to invest time in discovery work: create high-level use cases and object models.
- Look for opportunities to refactor existing code, using some standard design patterns in the redesign work.

The application code uses the Strategy classes to trigger a call handoff

```
class Call { ... };
  // each Call object contains the state information
  // for a particular call

class Sector {
  // each Sector object knows its strategy for
  // performing handoff searches
private:
  HandoffStrategy *loc_strategy;
      // plus other internal state information
public:
  void setCurrentPolicy (HandoffStrategy*);
  SectorList * BestHandoffCand (Call *c) {
     return (loc_strategy->sortCandidates (c));
  }
};

main () {
  Call *cptr;
  ... // read in Calls and Sectors – each
    // Sector points to a concrete Strategy
  for (; ;) { // continuously scan all calls
     if (cptr->signal_is_weak() ) {

     // In the old implementation, this code
     // was a complex switch statement.
     // In this design, the BestHandoffCand()
     // function automatically calls the
     // sortCandidates() function from the
     // correct subclass.

       SectorList *slist = cptr->
         Get_Sector()->BestHandoffCand (cptr);
       SendHandoffInfoToECP (cptr,slist);
     }
     cptr = cptr->next_call ();
  }
}
```

The "strategy class" definitions — an abstract class plus two or more concrete classes

```
// In this example, the policy that a Sector
//uses to find the best handoff candidates
//depends on which kind of HandoffStrategy
// object it contains.

//Each subclass of the HandoffStrategy
// class is a concrete Strategy class, with
// a different implementation of the
// sortCandidates operation.


class HandoffStrategy { // abstract class
public:
  virtual SectorList*
     sortCandidate (Call *c) = 0;
};

class MAHO_Handoff :
     public HandoffStrategy {
public:
  SectorList* sortCandidates (Call *c) {
   ...implementation 1...
  }
};

class Hierarchical_Handoff :
     public HandoffStrategy {
public:
  SectorList* sortCandidates (Call *c) {
   ...implementation 2...
  }
};
```

■ **Figure 5.** *A Strategy class example.*

- Modify the code in small stages and retest often during the process of modifying the code.

The time spent discovering the structure of the legacy system is an important investment. Some of the architectural secrets of an existing system are not immediately obvious from browsing the code. Use case modeling and high-level class diagrams can be used to fill in for incomplete legacy system documentation. The use case model and class diagram can be used to discover the best places to focus the refactoring work.

Some standard object-oriented design idioms are useful in the refactoring process. Wrapper classes are the most commonly used design idiom when doing software redesign. In certain places, more complex idioms such as the strategy class technique can be used to make the design of the system easier to understand and extend. It is not at all essential to convert the entire system to fit the redesign. In fact, it is best to make changes a bit at a time, retesting after each group of changes.

References

[1] J. Davison, D. Mancl, and W. Opdyke, "Understanding and Addressing the Essential Costs of Evolving Systems," *Bell Labs Tech. J.*, vol. 5, no. 2, Apr.–June 2000, pp. 44–54.

[2] I. Jacobson, *Object-Oriented Software Engineering*, Reading, MA, Addison-Wesley, 1992.

[3] A. Cockburn, *Writing Effective Use Cases*, Reading, MA: Addison-Wesley, 2000.

[4] N. Wilkinson, *Using CRC Cards*, Reading, MA: Addison-Wesley, 1995.

[5] M. Fowler *et al.*, *Refactoring: Improving the Design of Existing Code*, Reading, MA: Addison-Wesley, 1999, p. 51.

[6] E. Gamma *et al.*, *Design Patterns*, Reading, MA: Addison-Wesley, 1995.

[7] D. Schmidt *et al.*, *Pattern-Oriented Software Architecture*, vol. 2, New York: Wiley, 2000, p. 52.

Biography

DENNIS MANCL [SM] (mancl@lucent.com) is a Distinguished Member of Technical Staff in the Software Technology Center at Bell Laboratories, Murray Hill, New Jersey. He is a member of ACM. He has a B.S. in mathematics from the University of Wisconsin-Madison, and an M.S. and a Ph.D. in computer science from the University of Illinois. He works as a consultant in C++ programming, object-oriented design, and use case modeling for many communications software projects throughout Lucent Technologies.