the Availability Digest

Migrating Legacy Systems:

Gateways, Interfaces, & the Incremental Approach

March 2007

In their book, Migrating Legacy Systems: *Gateways, Interfaces, & the Incremental Approach*, Michael Brodie and Michael Stonebraker explore the quagmire of migrating legacy systems to modern architectures. They demonstrate through examples and case studies that many legacy migrations fail or are never completed after considerable time and expense have been committed. They argue that this is caused by the urge to do a "big-bang" cutover. Instead, they describe an incremental approach to migration that allows a legacy system to be migrated to a modern architecture in small, controllable steps. They call their approach "Chicken Little" as opposed to the big-bang "Cold Turkey" approach.

What does legacy migration have to do with continuous processing architectures? The answer is another question: "How do I get to there from here?" For instance, how do I migrate my current legacy system to an active/active system?[1]

There are still many legacy applications that provide mission-critical services but are burdened with the inflexibility, high cost, and brittleness that is characteristic of such systems. If we want to move such a system to, say, an active/active architecture, is it as simple as replicating its database to a like system? Generally not. The legacy system must, in general, be migrated to an architecture in which its database is decomposable from its applications. This is not a simple process and is what this book is all about.[2,3]

What is a Legacy System?

Legacy systems are any systems that cannot be modified to adapt to constantly changing business requirements. They often use aged languages and file systems that tie the user interfaces, the application logic, and the databases into an unbreakable monolith. They typically are poorly, if at all, documented and are maintained by an aging staff that knows them intimately. Their maintenance is expensive, and they are quite brittle in that they tend to break easily whenever modified.

Consequently, legacy systems are not very adept at adapting to changing business requirements. Many were built before the concept of layered systems, in which presentation, application, and database layers are separated by clean interfaces.

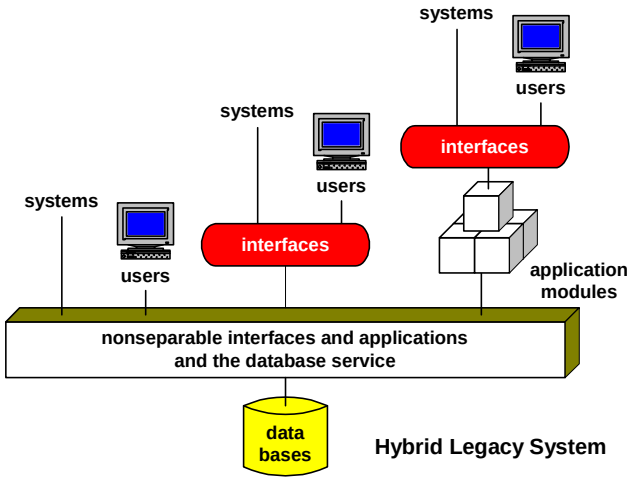
---

[1] See our companion article in this issue, Migrating Your Application to Active/Active, The *Availability Digest*; March, 2007.

[2] Brodie, M. L., Stonebraker, M., *Migrating Legacy Systems: Gateways, Interfaces, & the Incremental Approach*, Morgan Kaufmann Publishers, Inc.; 1995.

[3] Our thanks to Harry Scott of Carr Scott Software for pointing us to this book.

Over the decades, legacy systems often had layered appendages attached in order to provide new functionality. New user devices were attached via interface layers that translated between the legacy interface requirements and those of the new devices. New applications were implemented that accessed the legacy data by imitating legacy user devices. Thus, these legacy systems became a hybrid of legacy and modern systems.



Hybrid Legacy System

In many cases, it has become imperative to phase out these legacy systems and replace them with modern architectures that support the rapidly-changing business conditions of today. However, this migration from old to new has proven to be very difficult; and many, if not most, legacy migrations are never finished.

Chicken Little versus Cold Turkey

The migration of a legacy system could involve hundreds of millions of lines of code, terabytes of data, and millions of customers. It must be done with no downtime as these systems are often fulfilling 24x7 mission-critical functions. Michael Brodie and Michael Stonebraker liken these migrations to trying to overhaul an airplane while it is in flight.

Another serious problem is that the legacy database and the target database to which it is being migrated must maintain transaction integrity even though the legacy database probably cannot participate in global transactions.

In their book, the authors describe a formal methodology for migrating from legacy systems to modern architectures. They couple this with case studies of actual migrations.

Their technique emphasizes incremental migration. They dub this the Chicken Little approach as opposed to the Cold Turkey approach.

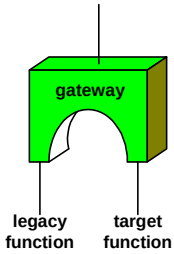
The Cold Turkey approach is the common approach to take. The legacy system is such a black box to the current development team that it seems that the only alternative is to completely rewrite the system and cut over to the new system on one fateful day. However, experience has shown that this approach is very likely to fail after years of development effort and millions of dollars invested.

The intent of the Chicken Little approach is to compartmentalize the migration effort so that only a small amount of functionality is migrated at a time. In the event that a migration increment fails, one can return to the previous system configuration, correct the problem, and try again. In this way, as the business continues to reengineer itself, the migration effort and the partially completed target system can be more easily modified to support newly required business functions as they occur.

Gateways

The basic migration philosophy is to decompose the legacy system as much as possible (if at all), and then to construct gateways that allow the remaining legacy components to cooperate with their modern replacements (the *target* components) as the migration proceeds. At any given point

in the process, any given user request may need the services of either or both of the legacy and the target applications and may need to access data from either or both of the legacy and target databases. This cooperation is provided by gateways, which are at the heart of the Chicken Little technique.



It may be decided not to migrate certain legacy functions due to cost, complexity, risk, or importance. If this is the case, some gateways may remain in the finished target system.

Some gateways are commercially available, but gateways are so application-dependent that even commercially-available gateways need significant modification. In many cases, gateways must be written in their entirety. The effort of decomposing the legacy system into migratable elements and the effort required to implement and maintain the gateways can be perceived to be daunting efforts. However, the authors point out that without this effort, the migration is likely to fail.

Comparison of Chicken Little and Cold Turkey

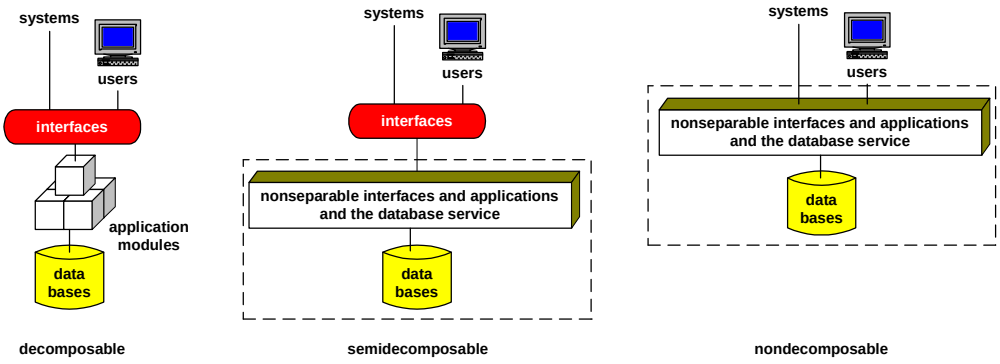
The authors' comparison of the Chicken Little and Cold Turkey approaches is given in the following table:

	Cold Turkey	**Chicken Little**
Risk	Huge	Controllable
Failure	Entire project fails	Only one step fails
Benefits	Immediate, probably short-lived	Incremental over time
Outlook	Unpredictable until deadline	Optimistic

Decomposability

A key consideration in legacy migration is to what extent can the legacy system be decomposed into the presentation, application, and database layers. The complexity of the migration is controlled by the structure of the legacy system.

The decomposability of the system defines the migration approach and greatly affects the complexity of the migration. The more decomposable the system is, the easier will be the migration.



A decomposable system is one in which the presentation, application, and data layers can be easily separated. (The presentation layer is called the "interface" in the author's diagrams.)

A semidecomposable system is one in which the presentation layer can be separated, but the application and data layers are closely entwined.

A nondecomposable system is one in which the system cannot be partitioned in any way.

Gateways

It is the gateway that is the key to the Chicken Little migration process. The function of a gateway is threefold:

- It insulates certain components from changes made to other components.
- It translates requests and responses between the components which it serves.
- It guarantees consistency of data in legacy and target database copies.

Gateways are complex. The implementation of a gateway is a major project in itself as it must understand the nuances of both the legacy interfaces and the target interfaces so that it can translate accurately between the two. Furthermore, as the migration proceeds, the functions required of a gateway change so that it must be continually maintained.

Gateway Direction

There are two directions that a gateway might serve. A *forward gateway* transfers requests from a legacy component to a target component. A *reverse gateway* transfers requests from a target component to a legacy component.

Gateway Types

Likewise, there are several types of gateways.

- A *database gateway* transfers requests from an application to a database as shown above.
- An *application gateway* transfers requests from a presentation interface to an application.
- An *interface gateway* transfers requests from users to a presentation interface.

Typically, a single gateway will present a legacy or target interface to the component which it is servicing, as appropriate. It will route requests to legacy and target components as needed, based on application logic contained in the gateway. It will format responses to meet the expectations of the component which it is servicing.

Thus, a source component (whether it be legacy or target) is unaware of what kind of component (legacy, target, or a combination) is servicing its request. It is insulated from changes in the servicing components as they migrate from legacy to target and sees only the defined target interface.

Migration Gateway Configurations

The use of gateways depends upon the level of decomposability exhibited by the legacy application.

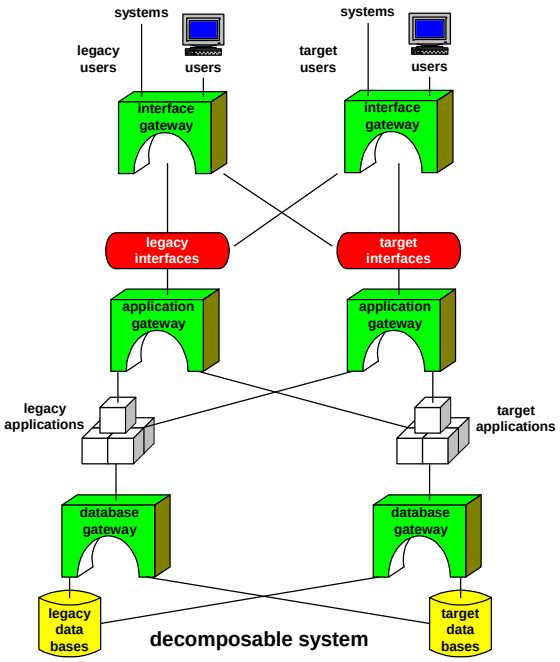
Decomposable

In a decomposable legacy system, each of the layers can be isolated. Therefore, every layer can be migrated individually, independently of the other layers. Gateways provide isolation of the layers and hide the source of servicing, whether it be legacy or target, from the components requesting service.

The users interface with an interface gateway. This gateway provides the interface which the users expect and can route requests to either the legacy interface or the target interface as required.

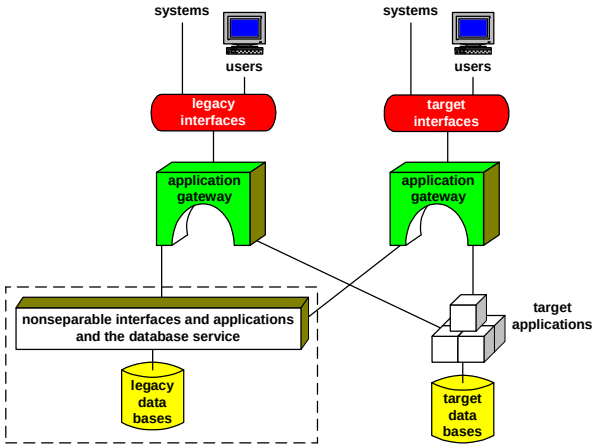
The interfaces connect to the applications via application gateways. An application gateway can route requests to either a target application or a legacy application. In some cases, the request may have to be broken up into subrequests that are routed to each.

As the migration proceeds, data sets will migrate from the legacy databases to the target databases. A particular request may need to access data items resident in both databases. Therefore, the legacy and target applications use database gateways, which will take a request in legacy or target format as applicable and access the data wherever it resides.



decomposable system

Semidecomposable



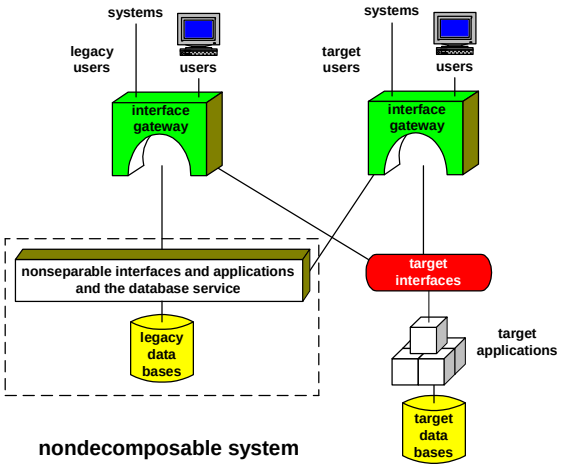
semidecomposable system

A semidecomposable legacy system is one in which the presentation layer can be cleanly separated from the applications. However, the applications and databases are still closely entwined. Therefore, they cannot be separated and must be treated as a monolithic whole.

For migration purposes, application gateways are used to route user requests from legacy and target interfaces to the appropriate legacy or target applications as required to satisfy the request. Responses returned to the gateways are reformatted to suit the issuing interface.

Nondecomposable

The interface, application, and database functions in a nondecomposable legacy system are so closely entwined that they cannot be separated. Therefore, the legacy system must be treated as a monolithic whole throughout the entire migration. An interface gateway routes requests from legacy users and target users to the appropriate legacy or target system. If a request needs to access application functions in both the legacy and target systems to order to be satisfied, the gateway is responsible for breaking the request into subrequests, submitting them to the appropriate system, and then combining the responses into an appropriate response for the legacy or target user.



nondecomposable system

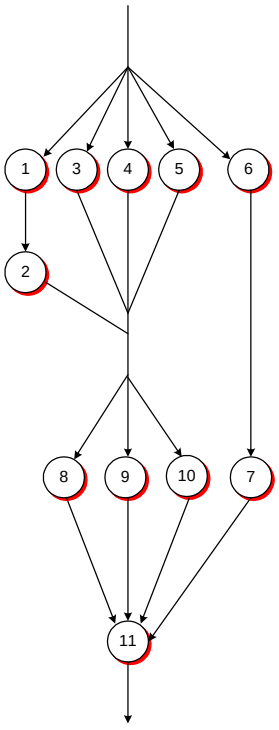
Hybrid

Of course, in general, any particular legacy system will comprise a range of nondecomposable, semidecomposable, and decomposable functions. As the system has been modified over the years, it is likely that newer functions have been implemented to be more and more decomposable.

In this situation, the appropriate combination of the above techniques may be used to perform the migration.

Migration Steps

The Chicken Little migration strategy utilizes an eleven-step process. Though these steps may be different for different levels of decomposition, they follow basically the same pattern. In their book, Brodie and Stonebraker go into great detail concerning what is to be performed in each step as a function of the level of decomposability.



The steps apply to each increment of migration and in general are as follows:

1. Incrementally analyze the legacy system.
2. Incrementally decompose the legacy system structure.
3. Incrementally design the target interfaces.
4. Incrementally design the target applications.
5. Incrementally design the target database.
6. Incrementally install the target environment.
7. Incrementally create and install necessary gateways.
8. Incrementally migrate the legacy database.
9. Incrementally migrate the legacy applications.
10. Incrementally migrate the legacy interfaces.
11. Incrementally cut over to the target system.

Case Studies

Brodie and Stonebraker detail as case studies two legacy migrations in which they were heavily involved. One migration was for a major global telephone company. This legacy system comprised hundreds of millions of lines of code and terabytes of data. The other migration was for a large cash management system.

Interestingly, neither of these migrations was completed. The global telephone company migration project was terminated when it became apparent that further business process reengineering was needed. The cash management system migration project was terminated after one year due to a merger in which the new management had a broader information services mandate. However, the Chicken Little process, which was used in both instances, was deemed to be a success for the portions of the projects that were completed.