Legacy System Modernization Strategies

Robert C. Seacord
Santiago Comella-Dorda
Grace Lewis
Pat Place
Dan Plakosh

July 2001

TECHNICAL REPORT
CMU/SEI-2001-TR-025
ESC-TR-2001-025

Carnegie Mellon
Software Engineering Institute
Pittsburgh, PA 15213-3890

Legacy System Modernization Strategies

CMU/SEI-2001-TR-025
ESC-TR-2001-025

Robert C. Seacord
Santiago Comella-Dorda
Grace Lewis
Pat Place
Dan Plakosh

July 2001

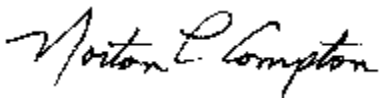
COTS-Based Systems

This report was prepared for the

SEI Joint Program Office
HQ ESC/DIB
5 Eglin Street
Hanscom AFB, MA 01731-2116

The ideas and findings in this report should not be construed as an official DoD position. It is published in the interest of scientific and technical information exchange.

FOR THE COMMANDER



Norton L. Compton, Lt Col, USAF
SEI Joint Program Office

This work is sponsored by the U.S. Department of Defense. The Software Engineering Institute is a federally funded research and development center sponsored by the U.S. Department of Defense.

This work was created in the performance of Federal Government Contract Number F19628-00-C-0003 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. The Government of the United States has a royalty-free government-purpose license to use, duplicate, or disclose the work, in whole or in part and in any manner, and to have or permit others to do so, for government purposes pursuant to the copyright license under the clause at 252.227-7013.

For information about purchasing paper copies of SEI reports, please visit the publications portion of our Web site (http://www.sei.cmu.edu/publications/pubweb.html).

Table of Contents

List of Figures

List of Tables

Abstract

Modernization of legacy enterprise systems introduces many challenges due to the size, complexity, and frailty of the legacy systems. Size and complexity issues often dictate that these systems are incrementally modernized, and new functionality is incrementally deployed before the modernization effort is concluded. This in turn requires that legacy components operate side by side with modernized components in an operation system—introducing additional problems.

In this report we discuss some alternative development approaches for incrementally modernizing legacy systems, including consideration of the advantages and disadvantages of each approach. These development alternatives can be mapped against the peculiarities of a particular modernization effort to recommend an appropriate approach.

1 Introduction

Modernization efforts for legacy information systems are often large, multiyear projects that pose significant risks. Information systems are critical to companies, and making a single deployment of the modernized version is too risky to be admissible. Additionally, a modernization effort of a large system requires a significant investment in terms of money and time; projects of this magnitude are strongly pressured to demonstrate early benefits. Figure 1 illustrates a typical example in which the modernized version of the system is developed and deployed incrementally over a six-year period.

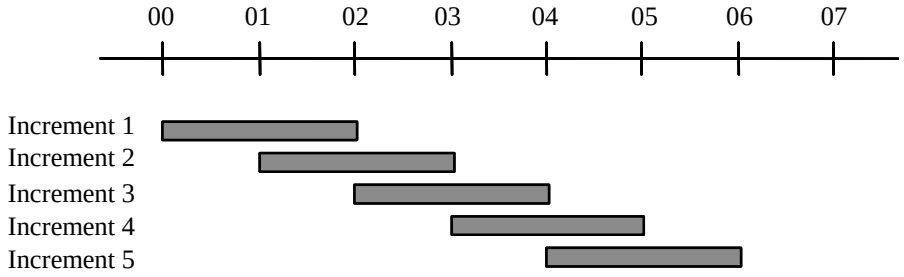


Figure 1: Development Increments

Incremental modernization of a legacy system is illustrated in Figure 2. Initially, the legacy system consists completely of legacy code. At the completion of each increment, the percentage of legacy code decreases while the percentage of modernized code increases. Eventually, the system is completely modernized.

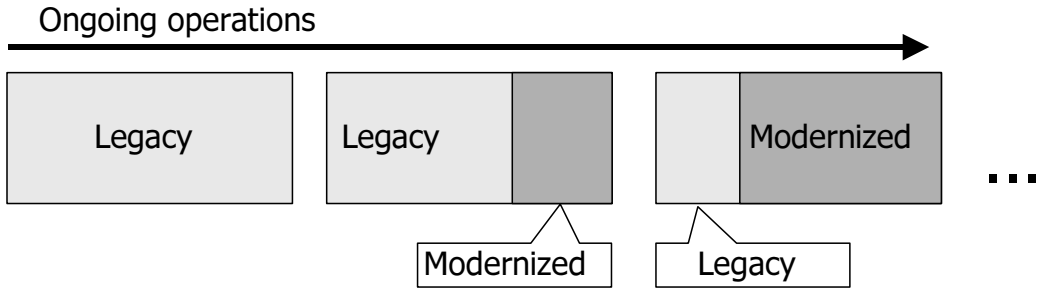


Figure 2: Legacy System Modernization

Since modernized components are being deployed prior to the completion of the entire system, it is necessary to combine elements from the legacy system with the modernized components to maintain the existing functionality during the development period. Adapters and other wrapping techniques may be needed to provide a communication mechanism between the legacy system and the modernized system as shown in Figure 3.

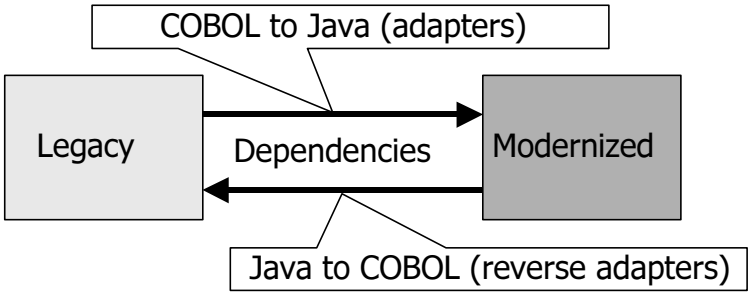


Figure 3: Interoperation of the Modernized and the Legacy Systems

An incremental modernization effort strives to keep the system fully operational at all times while reducing the amount of rework and technical risk during modernization. To balance these conflicting requirements, a modernization effort needs to be carefully planned. Planning a modernization effort involves more than creating a budget and setting up a few milestones: a modernization plan must also contain the order in which the functionality is going to be modernized, along with information describing the scaffolding code that must be created to keep the system operational at all times.

The technical aspects of the modernization plan are what we call the modernization strategy. In this report we describe a case study that involves the creation of such a strategy for the modernization of a large Retail Supply System (RSS).

The RSS consists of approximately two million lines of COBOL[1] code running on a mainframe. The overall architecture of the system has remained largely unchanged over 30 years, resulting in a system that is extremely brittle and difficult to maintain.

[1] COBOL (COmmon Business Oriented Language) was initially developed by the CODASYL (Conference On DAta SYstems Languages) Committee in April 1960. Major revisions were made in 1968 (ANS X3.23-1968), 1974 (ANS X3.23-1974), and 1985.

The information system life cycle is illustrated by Comella-Dorda, et al. [Comella 00]. These system evolution activities can be divided into three categories: maintenance, modernization, and replacement [Weiderman 97]. In many respects, the RSS is a typical information system following a similar life cycle. While development of the replacement system is underway, both maintenance and modernization efforts are being performed in parallel. The modernization work is an effort to "prep" the legacy system for replacement. The practical implication for this is that the end state as well as the starting state of the replacement effort may be undefined.

1.1 Goals and Objectives

In addition to the overall goal of modernizing the legacy RSS, and the necessity of following an incremental development approach, there are other modernization drivers:

- minimized development and deployment costs. Fielding modernized components alongside legacy code requires the development of adapters, bridges, and other scaffolding code that will be discarded after the final increment. While necessary, scaffolding code represents an added expense, as this code must be designed, developed, tested, and maintained during the development period. Minimizing the development of scaffolding code is one way to minimize overall development costs.
- schedule. The modernization strategy should seek to minimize the time required to develop and deploy a modernized RSS. Additionally, the approach should allow the RSS to be developed on a predictable schedule.
- quality. There are two issues regarding quality. One issue is the quality of the final, end-state system, once the RSS modernization effort has been completed. The final system should be easy to maintain and implemented around technologies that are not already obsolete. The second issue is the interim quality of the system, after each increment is deployed. Given the length of time required to complete the modernization, there will be many opportunities for the program to lose funding, be redirected, or take on a new focus. It is important that fielded increments improve the overall quality of the system, since there is always the possibility that each increment will be the last, given the length of time required for the development effort and normal uncertainties about changing business practices and requirements.
- minimized risk. Risks occur in many different forms, and some risk is acceptable if properly managed and mitigated. Due to the overall size and investment required to complete the RSS development, it is important that overall risk is kept low. To this end, the RSS componentization strategy should apply tried and proven techniques when possible, and lower risk approaches when some risk is necessary to achieve overall system goals.
- system performance. The RSS is replacing an existing system, so users have expectations concerning performance. While RSS modernization includes the modernization of hardware as well as software components, it is easy to negate hardware performance gains with poorly designed software. The componentization strategy must ensure that user performance expectations are met or exceeded.

- minimized complexity. Depending on how it is counted, the RSS consists of up to 1.8 million lines of legacy COBOL code, developed over a period of 30 years. The size of the RSS is a major complexity factor by itself. As a result, it is critical that the componentization strategy seeks to minimize overall system and development complexity, so that development complexity is kept at a manageable level. Managing the complexity of the development approach, by itself, may be the single largest factor that dictates the viability of the overall RSS modernization effort.

2 Legacy System Structure

Understanding the structure of the legacy RSS code is a necessary prerequisite to developing a componentization strategy. The RSS currently runs on a Unisys 2000 platform and was developed in Unisys COBOL on top of the Unisys Data Management System (DMS) database.

The existing RSS consists of approximately 900 program elements, each containing on average of 3000 lines of COBOL code. Dependencies exist between legacy program elements, and between legacy program elements and the legacy data store.

Program elements invoke other program elements using the COBOL CALL statement. The CALL statement transfers control to another program in the executable image. The RSS also uses the COPY statement to provide a library of COBOL source elements that are accessible by referencing text names. The COPY statement is similar to an INCLUDE statement in C or C++ (or macros in many languages). The RSS also uses PERFORM statements to execute one or more paragraphs. Control is returned to the next statement after the PERFORM statement when the paragraph execution ends.

Program elements interact with the DMS database using one of four operations:

- STORE - stores a new record in the database FETCH - a combined FIND and GET operation that establishes a specific record in the database as the current record of the run unit
- MODIFY - changes the contents of specified data items in a database record
- DELETE - logically removes a record from a mass storage file

Figure 4 illustrates both data flow and control flow in the RSS. Program elements typically fetch records from the DMS database into common storage. Then these data records may be completely or partially transferred from common storage to working storage where they can be operated on by the application program elements. Modified data may be placed back in common storage, and control passed between program elements, using common storage as a way to pass data between program elements.[2]

[2] This is similar to the use of shared memory in System V UNIX.

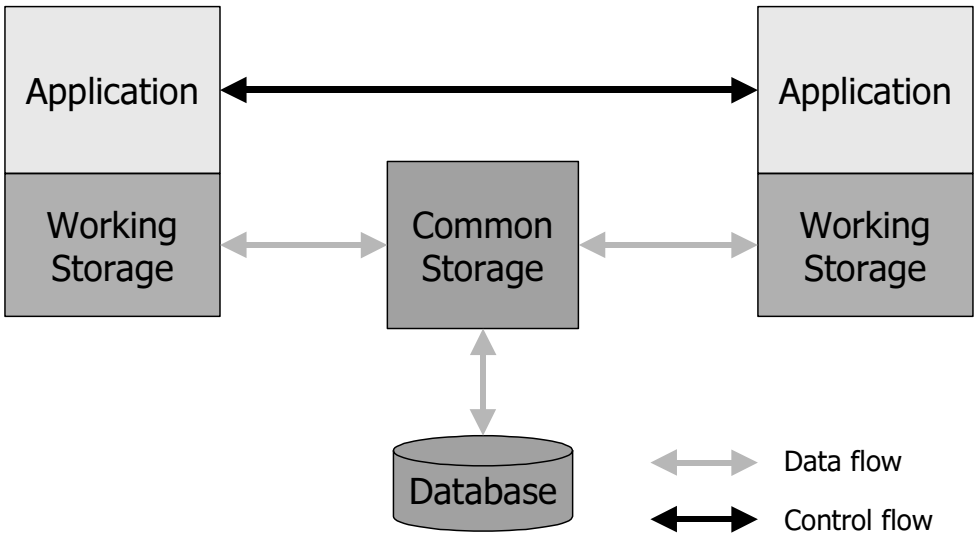


Figure 4: Data and Control Flow in the RSS

Eventually, data is written from working storage back to common storage. A program element may perform a STORE operation to create a new database record or a MODIFY operation to update an existing record.

Figure 4 may be misleading in some ways, since it is a fairly straightforward illustration of data and control flow in the RSS, when in fact there are many internal complications that make it more difficult to understand legacy data flows. Foremost among these is poor data encapsulation. Any program element can read and write to a database record. Many of these relationships can be easily identified by searching for STORE, FETCH, MODIFY, and DELETE operations in the source code. However, program elements can indirectly affect data stored in the database by modifying information in common storage that is then written to the database by a different program element. Understanding data flow in this environment requires considerably more in-depth and focused analysis.

Other complications include the use of FILLER space to hold data, and REDEFINES that allow the same common storage area to be accessed using different names. Both of these complications can make it more difficult to identify and manage data flow in the legacy system.

3 Adapters

In the introduction, we introduced the need for adapters to provide a communication mechanism between the legacy system and the modernized system. In this section, we provide a further analysis of the composition of these adapters.

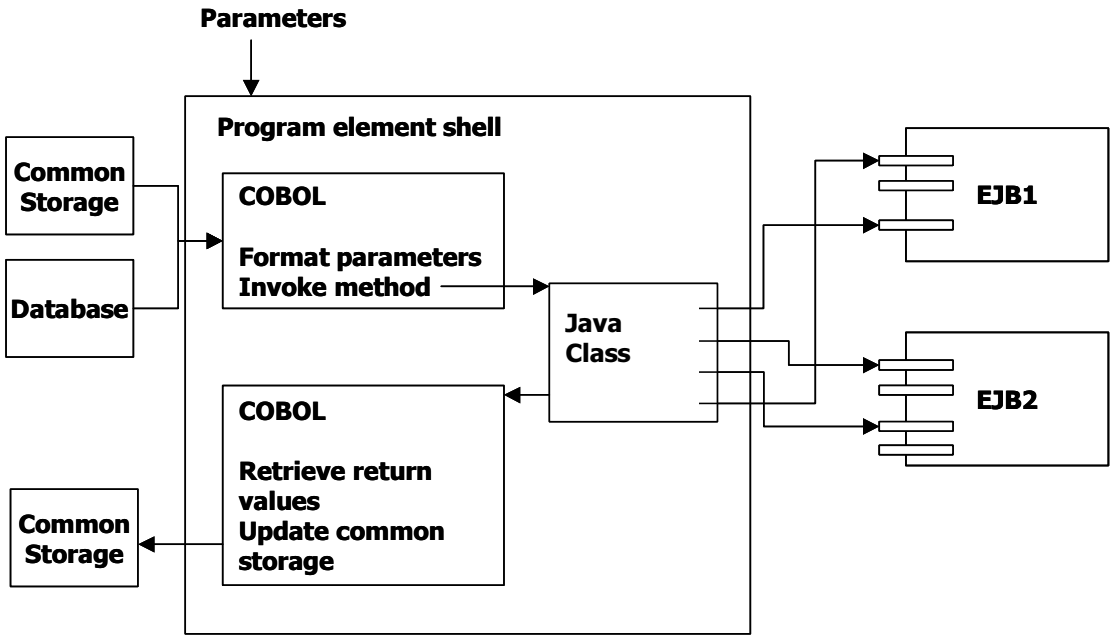


Figure 5: Legacy System Adapters

Figure 5 shows the composition of an adapter that is used to satisfy a dependency on a program element that has been re-implemented as Enterprise JavaBeans (EJBs). In broad terms, the adapter must satisfy the external requirements of the program element and provide a mapping between legacy and modernized functionality.

The adapter satisfies the external requirements of the legacy program element in the *shell*. The shell is typically written in the same programming language as the remainder of the legacy system, and supports the same calls and accepts the same parameters as the legacy program element. The shell is also responsible for extracting required information from common storage and the database and formatting this information so that it may be passed as parameters to a Java class. The Java class invokes the necessary sequence of methods in the EJBs to execute the functionality extracted from the legacy system program element. The adapter al-

lows functionality in the modernized system to be decomposed differently than in the legacy system. The complexity of the adapter may vary, depending on how different the modernized architecture is from the legacy architecture. The adapter must absorb these differences for the modernized system to remain free of legacy constraints.

Once the Java class has invoked the necessary EJB methods, control is returned to the COBOL shell. This shell must now modify common storage according to the changes that *would have been made* by the replaced legacy program elements. Once the shell returns control to the calling program, common storage must contain exactly the same information it would have contained had the legacy program element executed. Fortunately, it is not necessary for the legacy component to update the Oracle database, as these modifications are made directly from the EJBs through the persistence layer in the EJB server.

In building adapters, one must also consider that the adapter is not only replacing the legacy program element, but also all the program elements upon which it was dependent. In other words, by the time the adapter returns control to the calling program, all database modifications and changes to common storage performed by the legacy program element, and all its dependent program elements, must still be accomplished. To some degree, the functionality and database modifications should be implemented through a logical decomposition of activity in the modernized system. However, as the EJBs have no knowledge of, or access to, common storage, these updates must be made by the COBOL shell upon return from the Java class.

4 Increment 1–Pre-Componentization

RSS is being modernized in two major increments. The first increment is designed to prepare the system for the latter, more comprehensive componentization effort. The initial phase primarily involves migrating the legacy code onto a modern hardware/operating system platform and migrating from the Unisys DMS database to a relation database management system. The second phase will involve converting the system to a modern architecture and programming language.

Considerable effort is being exerted to migrate the existing legacy system off the Unisys platform and onto more modern hardware. The completion of this work is urgent, because of excessive Unisys platform maintenance costs. As a result of this ongoing cost, the emphasis is on a quick migration to the modern hardware.

Since quick migration off the Unisys platform is a priority, several constraints are imposed on the Increment 1 development. Changes to the legacy system are minimized, and the database schema is changed as necessary to move from DMS to a relational database.

Increment 1 modernization is scheduled in four releases. Table 1 shows target goals for each release.

Table 1: Planned Increment 1 Releases

Web/GUI (Release 1)	Reports (Release 2)
• Web-enable all RSS screens. • Build one component of GUI functionality (e.g., Inquires).	• Data analysis output is used to migrate reports data to Oracle. • Reports module analysis defines requirements to be satisfied by Oracle tools. • Using Oracle or other COTS tools, build reports based on Oracle reports data. • Build interfaces and reports GUI.
Account & Financing (Release 3)	**Migrate Inline (Release 4)**
• Based on system inventory & analysis – remove accounting & financing code. • Add third GUI component. • Eliminate redundant code.	• Remove Unisys dependencies. • Migrate data and data access to Oracle/relational database. • Roll existing functionality to MicroFocus COBOL. • Port system to Solaris platform.

The primary goal of Increment 1 is to move off the Unisys platform as quickly as possible. To move the RSS off the Unisys mainframe it is necessary to isolate, eliminate, and replace functionality that is currently performed by the Unisys operating system or proprietary software on the Unisys platform. While evaluating candidates for replacement components, it is beneficial to understand the Increment 2 componentization strategy, since this strategy will invariably require deploying the release 4 system alongside modernized components. For example, how will control flow between the legacy and modernized systems? There are several possible solutions, including using MQSeries, CORBA, or direct COBOL-to-Java language bindings. How will data be synchronized between the legacy and modernized databases? How will the transaction context be maintained between the legacy and modernized systems?

Consideration of these questions may lead to the selection of components that are compatible with the modernized system and may reduce the effort required in Increment 2. Selection of these components has to be made with consideration as to how this will impact the Increment 1 schedule. Delaying the system migration from the Unisys platform to Solaris has an easily quantifiable Unisys maintenance cost. Using the wrong product may require an additional development increment between Increment 1 and Increment 2 to prepare the RSS for modernization. This will certainly increase development costs and lengthen development schedules.

Components that support the Increment 2 componentization strategy without adding time to the Increment 1 schedule can be adopted with little consideration. Components that support the Increment 2 componentization strategy but require extra time to complete must be evaluated on a case-by-case basis. To select a component in this case requires a convincing argument that development costs saved in Increment 2 would exceed costs incurred in delaying fielding of the 1.4 version of the system.

A desired artifact of Increment 1 development is to gain a greater understanding of the legacy system and to use this acquired knowledge to define the desired architecture of the future system. These benefits are lost, to some degree, if different individuals are involved in the Increment 1 and Increment 2 efforts, although some of this knowledge, if properly recorded, can be successfully transferred.

5 Development Plan

Complexity in the RSS program is not limited to the source code, but extends to the overall development plan. Before getting into details of the development and deployment approach, we will describe the overall context and define some process-related terms.

The overall RSS modernization effort is divided into two *increments*. The goal of the first increment is to quickly migrate the existing code base off the legacy, Unisys hardware platform and onto an open systems platform. This is critical for the overall program because of the high cost associated with maintaining the legacy hardware. Money spent on maintaining the existing system comes out of the overall budget, with the result that fewer funds are available for modernization.

Increment 2 is the modernization effort and assumes the completion of Increment 1. Increment 2 is divided into *phases*. Each phase represents a different emphasis or focus; for example, a focus on modernizing the structure of the database or on migrating code. The exact number of phases depends on the incremental development strategy option employed. Table 2 is an example of a three-phase development plan. In this example, the database is restructured first, followed by the code migration. In the final phase, the system is verified and then deployed as the operational system.

Table 2: Development Phases

Development Tasks	Phase 1	Phase 2	Phase 3
Restructure DB	restructure	restructure	restructure
Componentization	by transaction OR by application	by transaction OR by application	
Deployment	parallel ops OR operational	parallel ops OR operational	parallel ops OR operational
	Time		

Within each phase, the RSS will be incrementally developed and operationally deployed as shown in Figure 6. This means, for example, that there may be multiple, incremental releases during the code-migration phase as well as during the database-migration phase. Each of these releases is a candidate for deployment.

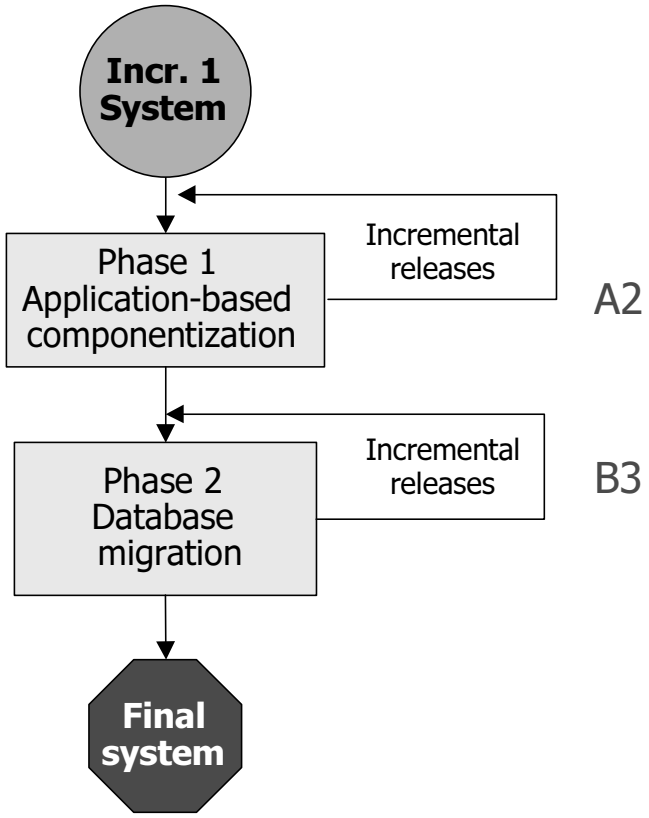


Figure 6: Incremental Releases

There are various advantages and disadvantages to incremental development and deployment. Incremental development should reduce overall program risks by allowing both users and developers to gradually understand the new system. Lessons learned from the first increment can be applied to prevent mistakes from reoccurring in later increments. Smaller steps should be easier to manage; it's also easier to evaluate progress against them. Smaller increments should also result in more focused effort.

Incremental development also has disadvantages. Theoretically, building a system in a single increment minimizes development and deployments costs, as it is only necessary to system test and deploy a single system. System testing and deployment have fixed costs that do not vary widely based on the size of the increment. These fixed costs are incurred for each increment. In practice, a single development/deployment cycle does not always result in lower development costs because of the added complexity of completing the entire project at one time.

Another problem with incremental development is that it is likely to cause the target architecture to resemble the legacy architecture. This is because the legacy system must be separated into sections that will be replaced. This initial separation of functionality is typically performed along the lines dictated by the legacy architecture. As the number of increments increases (and the corresponding size of each increment decreases), legacy and modernized architectures tend to converge.

Depending on the componentization strategy followed, an incremental development approach may also result in multiple restructuring of the database. Restructuring of the database has implied costs, including rework and the necessity to migrate the "new" legacy data that will be created after each incremental deployment.

5.1 Incremental Deployment Method

The RSS will be deployed over a series of increments, making functionality available to the user sooner than possible with a big bang development/deployment strategy. The goal is to break up the migration to the future system into small manageable steps, where the objectives of each increment are well defined.

Incremental deployment plans are driven foremost by complexity and technical feasibility. It is critical to ensure that the functionality, reliability, and performance of the system are not diminished after a deployment has completed.

Incremental deployment also offers opportunity for the organization to gradually begin sustainment of modernized components, easing the transition from legacy to modern technologies.

5.2 Incremental Deployment Options

There are many different approaches to incremental deployment. After some consideration, we determined that these approaches could be characterized by how they addressed the problems of code migration, database migration, and deployment. Table 3 lists these problem areas as well as potential solution strategies. Selecting a strategy for each problem area, Chinese-menu style, can form a componentization strategy. Of course once these answers are selected, they have to be considered in their entirety. Certain groups of answers have more cohesion when taken together and some have less. After examining the characteristics of each individual decision, we examine the combinatorial effects in Section 6.

Table 3: Incremental Deployment Options

Code Migration
A1 Based on transaction sets
A2 Based on existing program elements
Database Migration
B1 Before code migration
B2 During code migration
B3 After migration
Deployment
C1 Deploy each increment in parallel with the modified operational Phase 1 system
C2 Deploy each increment as the operational system

5.2.1 Code Migration

The first question that must be answered is the code migration strategy. This has a significant impact on how the modernization effort will proceed. Invariant in this selection is the target architecture, which is predefined based on a set of desired and objective system qualities.

Code can be migrated from the legacy system to the modernized system based on transaction sets or existing program elements.

Transactions, in this sense, are requests from users or external systems. The RSS defines a collection of transactions that are identified by two-letter codes. A series of program elements are then invoked to execute the transactions. Transactions may also spawn additional transactions within the system. The idea behind this code migration strategy would be to implement the transactional logic in the modernized system. At the same time, we could theoretically disable or remove the transactional logic in the legacy system. However, there are many issues to consider in taking both these steps.

The first of these issues is the database. The transaction we migrate is likely to read, create, or update records from one or more tables in the legacy database. Unfortunately, we cannot assume that this transaction is the only transaction that will modify these tables. That means that this data must exist and be accessible from both the modernized and legacy systems. Furthermore, changes that occur in one location must be propagated to the other.

The second problem is in removing code in the legacy system. It cannot be generally assumed that code that performs a function, as part of a user transaction, is not also executed as part of a different transaction. Therefore, analysis must be performed to ensure that this code is not required elsewhere. Removing code fragments or program elements may also introduce instability into the fragile legacy system.

Luckily, removing code is not a strict requirement of this approach. Legacy code can just be left in place and used as required to execute transactions that are still operating within the legacy system. Finally, after all functionality is moved to the modernized system, the legacy code can be discarded in bulk with no ill effects.

Transaction Sets

For the code migration to be successfully based on transaction sets, several conditions must be met. Most transactions must be localized to a small subset of program elements. If most transactions require most of the program elements to execute, it will be too difficult to migrate transactions in a single increment.

The second requirement is that componentization will not force the fragmentation of single program elements between COBOL and Java/EJB (i.e., the transition of only a part of an application to Java/EJB). Reengineering legacy components in this manner would add significant costs to the overall modernization effort.

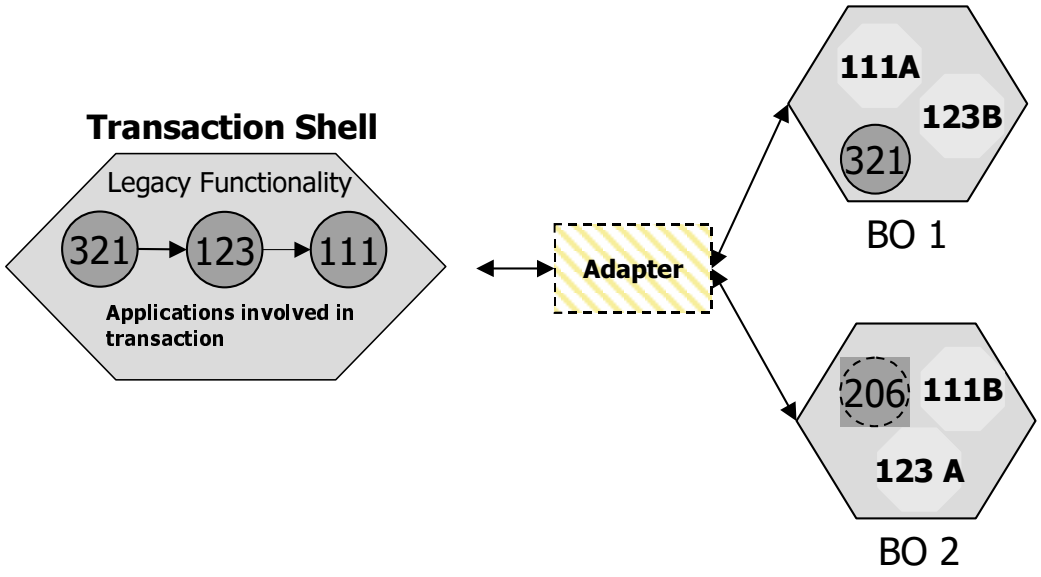


Figure 7: Transaction Set Code Migration

Figure 7 shows how code can be migrated using transaction set deployment.

There are both advantages and disadvantages to transaction set deployment. One of the advantages is that this approach increases the likelihood that the system will closely match the initial target architecture. Another advantage is that this approach can result in less development time, cost, and increments when compared to the application-based approach.

The disadvantages of this approach are that it can increase the complexity of incremental development (modifications will be more global in nature) and that it may come close to a one-step deployment by forcing the transition of large amounts of functionality in one increment. This in turn would provide less opportunity to refine the target architecture based on lessons learned (due to the possible transition of a larger amount of functionality at once).

Program Elements

Successful migration of the legacy code based on sets of program elements depends on the ability to link remaining legacy program elements to the new business objects while providing the same functionality. Program elements can be split across business objects, and business objects can be deployed while still incomplete—as long as the overall functionality of the system remains intact.

This approach may require some rework of business objects as the system evolves.

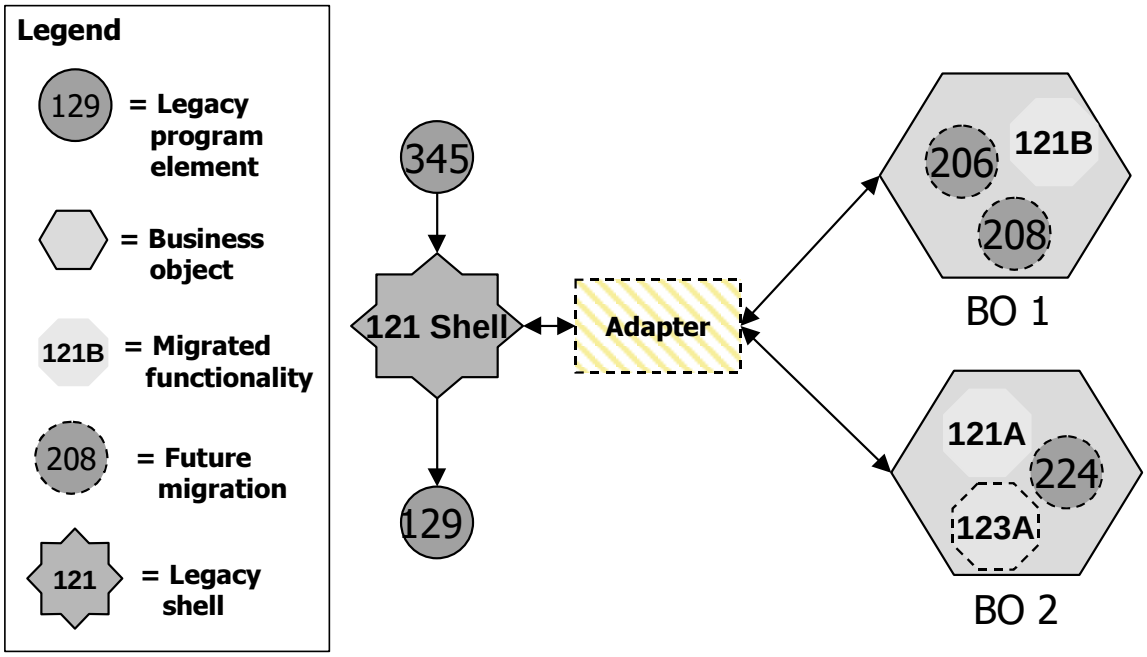


Figure 8: Program Element Set Code Migration

Figure 8 illustrates code migration by program element sets. A legacy component (121) is scheduled for modernization. Functionality performed by 121 is re-implemented as part of the modernized architecture as shown on the right. However, the 121 component is still invoked by the 345 program element and invokes the 129 component element, neither of which has been modernized in this example. In this case, it is necessary to develop a shell and adapter for the 121 program. The shell ensures that the external interfaces of the 121 program element are maintained. The adapter accepts requests from the 121 shell and invokes methods in the modernized components to implement this functionality. Results can then be returned to the 121 shell, which will use this data to satisfy its external requirements.

5.2.2 Database Migration

The second question to consider when deciding on a database migration strategy is when to modernize the database. Like other aspects of the legacy system, the database schema has evolved over time, and not necessarily in an optimal fashion. One of the goals of the modernization effort is to improve the representation of data in the database to eliminate redundancy, improve performance, reduce storage requirements, and reduce the potential for database anomalies.

In general, there are no guarantees about the structure of the modernized database. It is likely that some existing database tables will be split up, while others will be grouped together. New database tables will be created and existing tables eliminated. This may potentially result in a very complex relationship between database fields in the legacy and modernized systems.

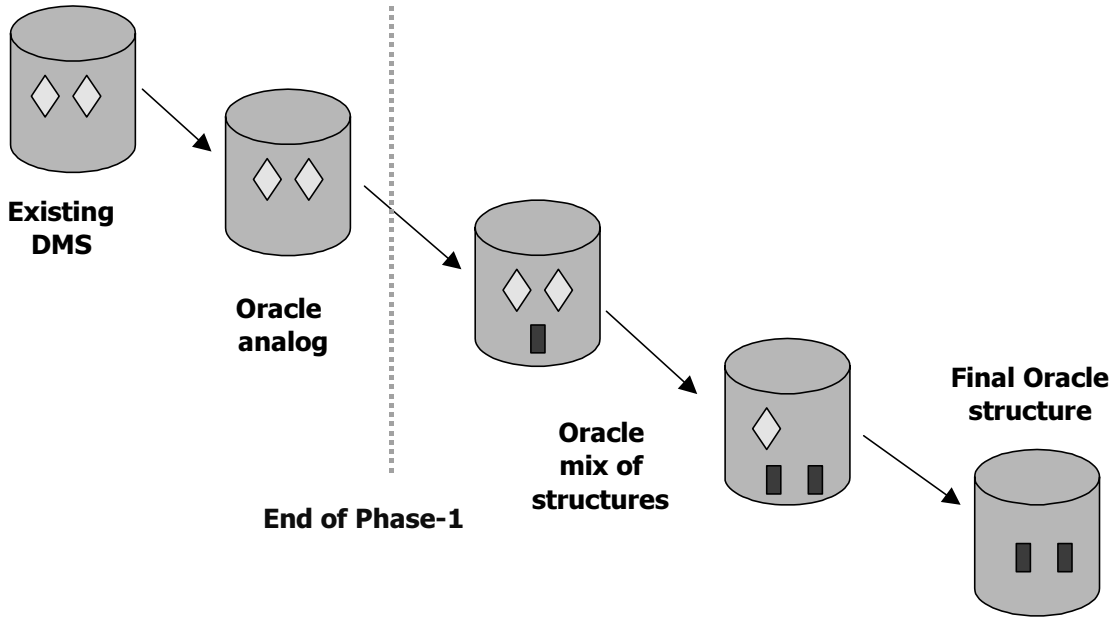


Figure 9: Database Migration

There are three options for database migration: before, during, and after the code migration. Regardless of which strategy is adopted, the database will pass through a series of states as shown in Figure 9. Initially, data is stored in CODASYL format[3] on the DMS. The first step in the database migration is to migrate this data to an equivalent relational form that can be maintained in the Oracle database. This translation alone requires significant modification to the structure of the data, as the modern relational model varies significantly from the CODASYL model.

The next step is to start to replace the database schema reflecting the legacy tables, with a modernized database schema. As the legacy system consists of close to 900 different database tables, this replacement must be viewed as a gradual process. This does not necessarily mean that a database consisting of mixed structures will be deployed. Eventually the entire database will be migrated to the modernized structure as shown in the lower right-hand corner of Figure 9.

Database Migration Before Code Migration

Database tables can be migrated before, during, or after the code is migrated. Migrating the database before the code is migrated has some advantages. Completing the database migration up front certainly simplifies the code migration. Modernized code can be developed to the target data architecture, and does not have to be mapped to legacy data elements. Migrating the database first is also clearly more of a focused effort with a single goal. Finally, migrating the database tables first reduces the risk of retaining the legacy architecture, since the eventual code migration will be based on the new database schema.

At this point migrating the database tables first appears to be a viable option, but unfortunately there are also numerous disadvantages to this approach. Migrating the database first may then require the restructuring of the legacy system to accommodate the modified tables. This is a major concern, as a principal reason the legacy system is being modernized is because of its lack of maintainability. Attempting a major restructuring of the legacy code to support the new database schema is extremely risky.

Migrating the database and restructuring the legacy code will consume considerable amounts of the schedule and resources available to the project. As a result, converting the database before beginning componentization can only be attempted if the target database schema is well understood and architecturally sound. This may be extremely difficult to validate given the magnitude of the system.

---

[3] a circa 1970 database model that, while antiquated by most measures, was the first to allow one-to-many relations

Since a large investment must be made to migrate the database up front, there will not be much latitude for further refinement of the database. This means that the project will more or less have to live with this initial assessment. Changes to the database will require more changes to the legacy system and possible restructuring of the modernized code. In general, this is a high-risk approach that depends largely on "getting it right the first time." If you do not have a high degree of confidence in your understanding of the data requirements for the modernized system, this may not be the best approach.

If the database is converted before code migration, the legacy system has to be rewritten to use this new database schema. In the end, this will result in less opportunity to evolve the architecture as necessary to support the system requirements.

Database Migration During Code Migration

Theoretically, this is the least expensive approach, since it requires minimal rework. This assumes, however, that performing both tasks simultaneously is not beyond the ability of the programming staff. This is because tackling both the data migration effort and the code migration effort simultaneously expands the focus of each increment and adds to the complexity of the effort. To modernize a component in this approach, for example, would require that you implement functionality in your modernized system, disable the corresponding functionality in the legacy system, implement a new database schema in the modernized system, migrate the data from the old to the new database schema, and update the legacy system to work with the new database schema and modernized code. This becomes particularly difficult when data elements or logic cannot be easily untangled from the legacy system. This approach can easily degrade to a "big bang" approach, where all legacy functionality is migrated in a single increment. This may not be feasible if you are under pressure to demonstrate progress by fielding increments before the entire system can be modernized.

Database Migration After Code Migration

Migrating the database after code migration has some interesting advantages. One advantage is that it provides additional time to refine the database schema. Of course, taking this approach requires that you construct modern components using the legacy database schema. Doing so is possible by using the persistence layer, which defines a mapping between state data in a component and the persistent store. In theory, a modernized system could be developed that only used component/object interfaces to access data elements. The persistence layer in these components can then map state data in the components to fields in the legacy system database. This works well in most cases, although there may be some cases where this breaks down, such as reports that need to directly access the database structure[4].

[4] Lewis, G., Comella-Dorda, S., Place, P., Plakosh, D., Seacord, R. *Data Architecture Guide for the ILS-S System*. Pittsburgh, PA: Software Engineering Institute, to be published.

Although isolating dependencies on the legacy database to the persistence layer can simplify the migration of the database after the code migration, code in the persistence layer will still require modification. This effort will involve replacing fairly complex code that needs to map between state data and fields in one or more legacy database tables with fairly straightforward calls that provide a direct mapping between state data attributes in the component and the database tables. The eventual mapping between component state data and the database schema should be relatively straightforward, because the database schema does not need to be fully specified until after the code migration has been completed.

The persistence code that maps to the legacy database structure may also be quite slow, as this code assumes a “to be defined” state in the system that must be emulated using the legacy system data structure. This should not be too big of a problem, as long as the interim performance of the system is acceptable and the development effort runs to completion (i.e., the database-migration phase is eventually implemented).

It may be possible to optimize this approach when a table is completely moved to the modernized system. However, this may be trickier than it seems. The first requirement is that no application elements that access this legacy table remain in the legacy system at the end of the increment (since the table will no longer be there). The state data maintained in the new component may have one of several relationships to the table being replaced:

1. It may contain the identical information (i.e., for every field in the legacy table the same field exists in the new table).
2. It may be a superset (i.e., it may contain all of the information plus some additional information).
3. It may be a pure subset.
4. It may contain a subset of information from the other table, plus some additional information.

In the first case it is not necessary to migrate the table, since it is already in its final form. In the second case, it may be possible to migrate to the new table, since all the data is moved. In the third case, you would have to make sure that the remaining information was also included in the increment, and in the fourth case you would need to make sure of that as well as mapping some of the new table back to a legacy table.

Eventually this line of optimization results in a modernization strategy where you are migrating the database at the same time as you are migrating the code. This option was discussed in the previous section.

5.3 Deployment Strategy

The RSS will be deployed in at least five increments. Each time new functionality is deployed to the field there is an operational risk that the system, including both modernized and legacy components, will not function properly once deployed. Deploying each increment in

parallel with the modified operational legacy system can mitigate these risks. Alternatively, these risks may be viewed as acceptable when considered against the additional costs and development risks introduced by parallel operations, as well as deploying each release directly to the field as an operational system. Each of these options is analyzed in the following sections.

5.3.1 Parallel Operations

One way to reduce operational risk is to continue to run the previous version of the system in parallel with the current release, as shown in Figure 10. In this approach, the modernized system is put into operation, but the legacy system is maintained as a “hot” backup. If the new system fails to function properly, control can be switched over to the legacy system. This solution provides a fallback capability that allows the new increment to be verified and tested online.

Parallel operations also provide the following benefits:

- allow users to compare both interfaces
- can aid in system verification
- minimize disruption to users

Of course, for this approach to be feasible the legacy system must have access to the latest data. Providing this access can be problematic, because the format and structure of the database tables may have changed between incremental deployments.

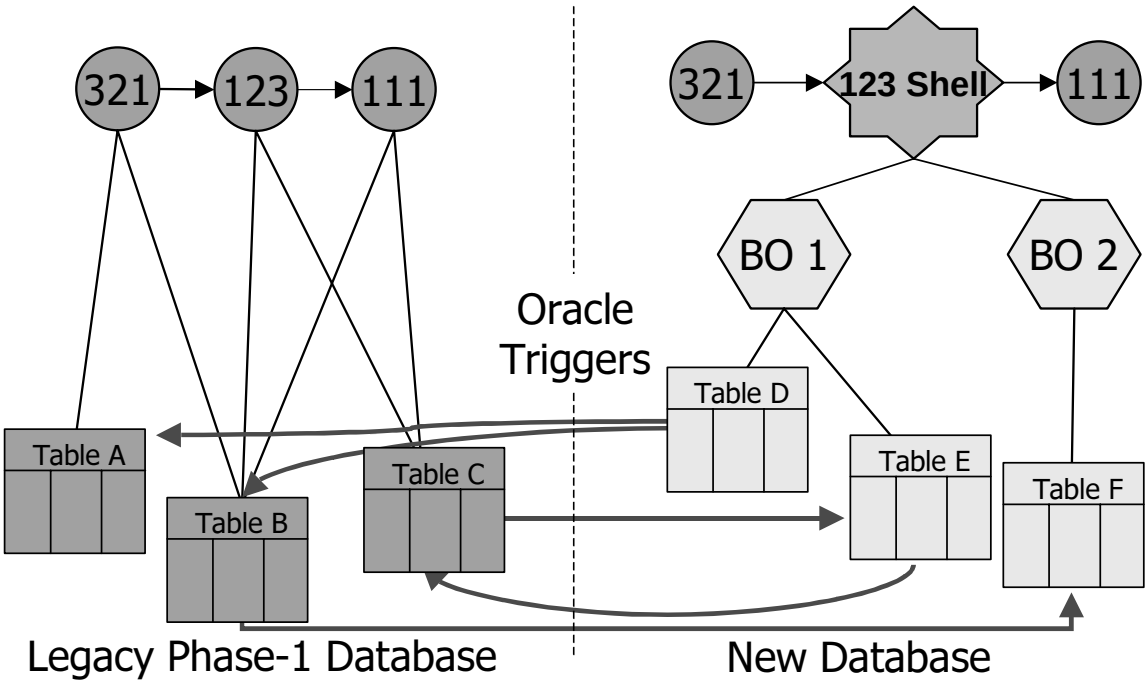


Figure 10: Parallel Operations

The databases can be synchronized through the use of data replication. Whenever the modernized system makes a change to a local table, synchronous row-level replication can be used to synchronously propagate the change to the legacy database using internal triggers [Bobrowski 97]. Other options may be available—such as loose synchronization through timed database conversions while copying.

Deploying in parallel can reduce operational risks, but care must be taken not to corrupt the legacy system while "wiring" the two systems together. The introduction of complex trigger mechanisms, for example, could easily inject defects into the legacy system. In general, changes to the legacy system should be minimal and non-pervasive. Another concern is the performance overhead incurred in invoking triggers to update multiple database tables as a result of each update.

Figure 10 shows the operational system after the initial deployment of new functionality. However, after the second release there could be a new problem if the "operational" system is the previous version of the new system deployed in the first increment (which consists of both legacy and modernized components) instead of the original legacy system. This situation should be avoided, because it adds significant complexity and risk to the development effort.

In most cases, it is better to keep the existing legacy system with every new deployment instead of having the system from the previous deployment become the legacy system.

After the modernized system has been deployed and run in the field for some time, and its operations have been validated, the legacy system and modernized system can be decoupled, and the modernized system can be allowed to run on its own.

While parallel operation can reduce operational risk, it can also increase development risk, degrade performance, and significantly increase maintenance costs. Difficulties may arise in data synchronization and locking between the modern and legacy systems that can increase development costs and impact the schedule.

When deploying in parallel, each incremental release of the system is deployed alongside the legacy system. This is true until the operational status of the final release is verified and the backup system can be stood down. This has several implications for the overall life cycle of the system. First, it is necessary to maintain two separate databases from the time the initial system is incrementally deployed until the backup system is stood down, increasing maintenance and support costs over the life of the project. Finally, code and database changes must be removed from the completed system.

Parallel operations often make sense when the cost of any downtime in the modernized system is substantial and are most feasible when the following assumptions hold true:

- Resolving any database locking and synchronization issues is not very complex and is technically feasible.
- No major code changes or restructuring of the legacy system is required.
- The performance degradation to the legacy system will not significantly impact users.
- Both systems can be maintained simultaneously.

5.3.2 Non-Parallel Operation

Another deployment strategy is to deploy each development increment as the operational system, as shown in Figure 11. In this approach, the deployed system consists of modernized and legacy components.

Non-parallel deployments typically provide the following benefits:

- reduce cost and development time
- force all users to use the new system immediately (may increase acceptance)
- do not inject additional technical and software development risks

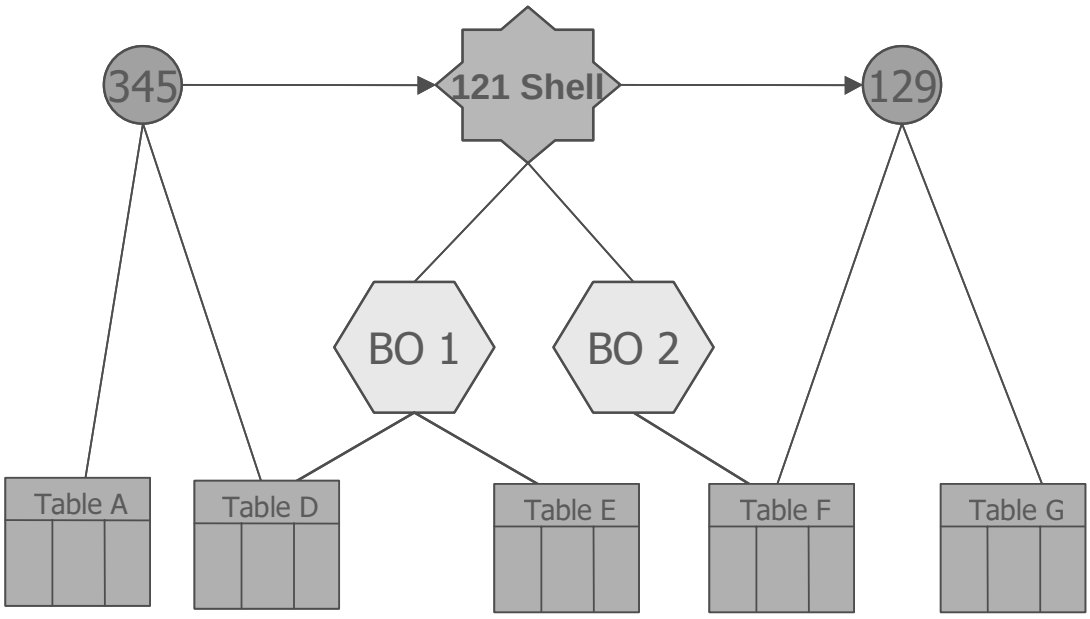


Figure 11: Non-Parallel Deployment

The major disadvantage to this strategy is that there is no fallback mechanism in the event of a system failure. Therefore, this approach requires that the software quality is sufficient for deployment and that those development increments that are candidates for deployment are completely verified and tested prior to deployment as the operational system.

5.3.3 Comparison of Options

Next we present a comparison of the different modernization options presented with respect to the following qualities:

- cost
- schedule
- risk
- performance
- complexity
- interim quality

A comparison chart is shown in Table 4. The up, down, and sideways arrows indicate how each option within a particular group will most likely affect each of the characteristics described. An up arrow indicates an increase in a quality; likewise a down arrow represents a decrease in a quality; and the sideways arrow represents no change. An up or down arrow can be good or bad depending on the quality. For example, an up arrow in cost would be considered to be bad, while an up arrow in performance would be considered to be good. Characteristics with more than one arrow are significantly impacted and should be given more weight.

Table 4: Summary of Modernization Options

Option	Characteristics					
	Cost	**Schedule**	**Risk**	**Performance**	**Complexity**	**Interim Quality**
A1 Develop components based on transaction sets.	⇩	⇩	⇧	⇔	⇧	⇧*
A2 Develop components based on existing applications.	⇧	⇧	⇩	⇔	⇩	⇩*
B1 Migrate database formats before componentization.	⇔	⇧	⇩	⇔	⇩	⇧
B2 Migrate database formats during componentization.	⇔	⇩	⇧	⇔	⇧	⇩
B3 Migrate database formats after componentization.	⇧	⇧	⇩	⇔	⇧	⇩
C1 Deploy each increment so that it runs in parallel with the modified operational Phase 1 system.	⇧⇧	⇧	⇩⇩⇧ +	⇩⇩	⇧⇧	⇩
C2 Deploy each increment as the operational system.	⇔	⇩	⇧	⇔	⇔	⇔
Symbol Meaning ⇧ Increase ⇩ Decrease ⇔No Change						

* Indicates that the increase or decrease in the particular quality is marginal at best and should not be heavily weighed.

+ Indicates a decrease in operational risk but an increase in developmental risk.

6 Modernization Trail Maps

A trail map is a time-phased modernization approach, consisting of up to three phases, that implements a set of the modernization options enumerated in Section 5.2. For example, the trail map in Table 5 shows componentization by transaction (A1), restructuring of the database after componentization (B3), and parallel deployment with the operational system (C1). The appendix contains the twelve trail maps used to evaluate the various combinations of options.

Table 5: Sample Trail Map

Development Tasks	Phase 1	Phase 2	Phase 3
Restructure DB	restructure	restructure	restructure
Componentization	by transaction OR by application	by transaction OR by application	
Deployment	parallel ops OR operational	parallel ops OR operational	parallel ops OR operational
	Time		

A group was convened to evaluate the modernization options outlined in the appendix with respect to cost, schedule, risk, and complexity. The trail-map characteristics summary shown in Table 6 was used as an initial starting point for discussion.

The group eventually reached general consensus on Trail Map 12, shown on page 61. This option calls for componentization by application followed by a restructuring of the database. At the end of each phase, the system is deployed as an operational system.

Table 6: Trail-Map Characteristics Summary

Trail	Characteristics					
	Cost	**Schedule**	**Risk**	**Complexity**	**Interim Quality**	**Interim Performance**
T1 - (A1 B1 C1)	⇧	⇧	⇩	⇧	⇧	⇩
T2 - (A1 B1 C2)	⇩	⇩	⇧	⇔	⇧	⇔
T3 - (A1 B2 C1)	⇧	⇩	⇧	⇧	⇩	⇩
T4 - (A1 B2 C2)	⇩	⇩	⇧	⇧	⇔	⇔
T5 - (A1 B3 C1)	⇧	⇩	⇩	⇧	⇩	⇩
T6 - (A1 B3 C2)	⇔	⇩	⇧	⇧	⇔	⇔
T7 - (A2 B1 C1)	⇧	⇧	⇩	⇔	⇩	⇩
T8 - (A2 B1 C2)	⇧	⇧	⇩	⇩	⇔	⇔
T9 - (A2 B2 C1)	⇧	⇧	⇩	⇧	⇩	⇩
T10 - (A2 B2 C2)	⇧	⇩	⇧	⇔	⇩	⇔
T11 - (A2 B3 C1)	⇧	⇧	⇩	⇧	⇩	⇩
T12 - (A2 B3 C2)	⇧	⇧	⇩	⇔	⇩	⇔
Symbol Meaning ⇧ Increase ⇩ Decrease ⇔No Change						

7 Conclusions

A workshop for developing a componentization strategy for the RSS modernization effort was convened at which participants considered different strategies for modernizing the legacy system that would support an incremental development and deployment approach and achieve the goal of implementing a modern software architecture.

The group eventually agreed on Trail Map 12 as the best option for modernization. In it, componentization by application is followed by a restructuring of the database, and the system is deployed as an operational system at the end of each phase.

The modernization strategy represented by this trail map can then be further developed. In particular, it is necessary to

- identify and prioritize the development deployment cycles (based on technical considerations first, and functionality second)
- identify groups of components that can be isolated and modernized during each incremental development and deployment cycle
- identify the number and types of wrappers that must be developed, and use this information to revise the cost estimation

The fact that the RSS team selected Trail Map 12 does not mean that it is the only appropriate modernization strategy. Each trail map, like each modernization effort, has its own unique characteristics. The purpose of this report was to evaluate a number of different options, and furthermore, to suggest a strategy for solving this and similar problems. The strategy is simple—when choosing between several paths where the selection is not obvious, walk down each path a hundred yards before deciding on the most appropriate route.

References

[Bobrowski 97] Bobrowski, Steve & Smith, Gordon. *Oracle8 Replication.* Oracle Corporation, December 1997.

[Comella 00] Comella-Dorda, Santiago; Seacord, Robert C.; Wallnau, Kurt; & Robert, John. “A Survey of Black-Box Modernization Approaches for Information Systems,” 173-183. *Proceedings of the International Conference on Software Maintenance.* San Jose, California, October 11-15, 2000. Los Alamitos, CA: IEEE Computer Society, 2000.

[Weiderman 97] Weiderman, Nelson H.; Bergey, John K.; Smith, Dennis B.; & Tilley, Scott R. *Approaches to Legacy System Evolution* (CMU/SEI-97-TR-014). Pittsburgh, PA: Software Engineering Institute, Carnegie Mellon University, 1997. Available WWW <URL: http://www.sei.cmu.edu/publications/documents/97.reports/97tr014/97tr014abstract.html> (1997).

Appendix: Trail Maps

A trail map is a time-phased modernization approach, consisting of up to three phases, which implements a set of the componentization options that we described in Section 5.2. In this section we present the twelve trail maps that allow us to evaluate the various combinations of componentization options.

Candidate Trail–T1 (A1 B1 C1)

In Trail Map T1, shown in Table 7, we first restructure the database and then we componentize based on transactions. During the incremental development Phases 1 and 2, we deploy the new system under development in parallel with the operational legacy system. Finally, once the new system is complete (in Phase 3) we deploy it stand alone as the operational system.

Figure 12 shows the system during Phase 1, with the operational system using the legacy database and the new system using the restructured database—updating each other using Oracle database triggers. During Phase 2 shown in Figure 13, functionality from the legacy system is being migrated to the new system based on transactions.

Table 7: Candidate Trail–T1 (A1 B1 C1)

Development Tasks	Phase 1	Phase 2	Phase 3
Restructure DB	restructure	restructure	restructure
Componentization	by transaction **OR** by application	by transaction **OR** by application	
Deployment	parallel ops **OR** operational	parallel ops **OR** operational	parallel ops **OR** operational
	Time →		

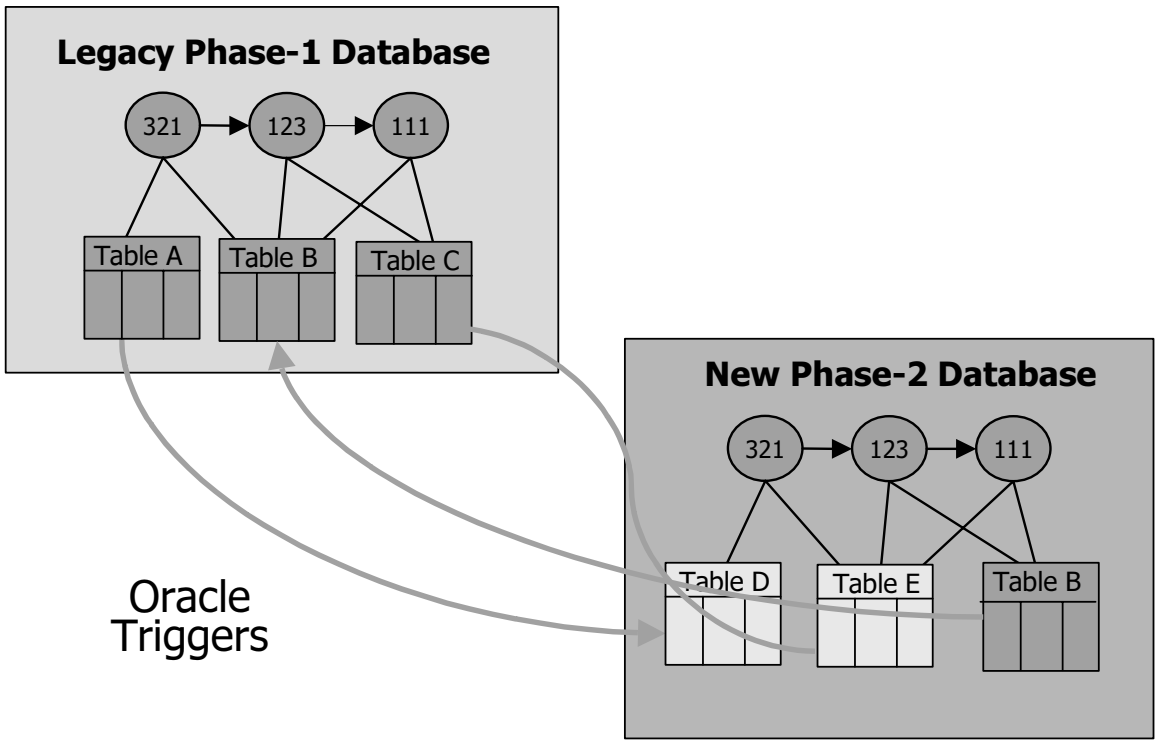


Figure 12: T1 During Phase 1

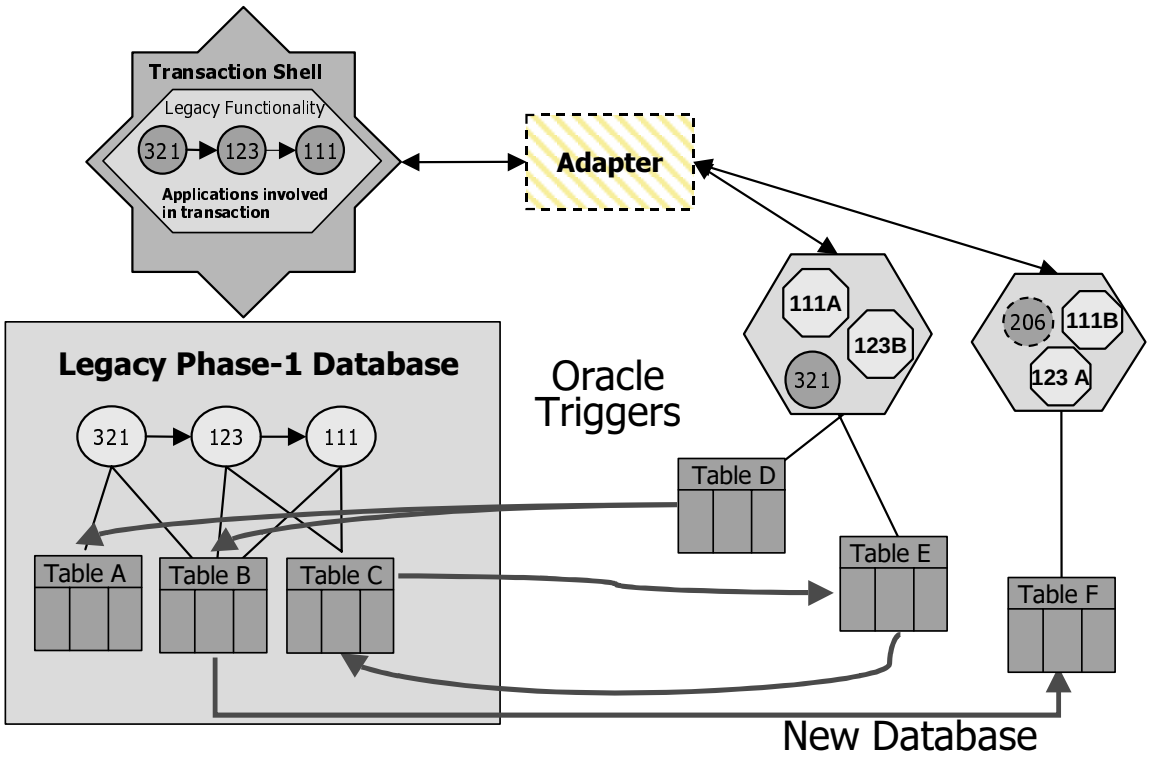


Figure 13: T1 During Phase 2

Candidate Trail–T2 (A1 B1 C2)

In Trail Map T2, shown in Table 8, we first restructure the database and then componentize based on transactions. During the incremental development phases, we deploy the new system as the operational system.

Figure 14 shows the system during Phase 1 with the database tables being incrementally restructured. During Phase 2, shown in Figure 15, legacy functionality is being migrated to the new system based on transactions.

Table 8: Candidate Trail–T2 (A1 B1 C2)

Development Tasks	Phase 1	Phase 2	Phase 3
Restructure DB	restructure	restructure	restructure
Componentization	by transaction OR by application	by transaction OR by application	
Deployment	parallel ops OR operational	parallel ops OR operational	parallel ops OR operational

Time

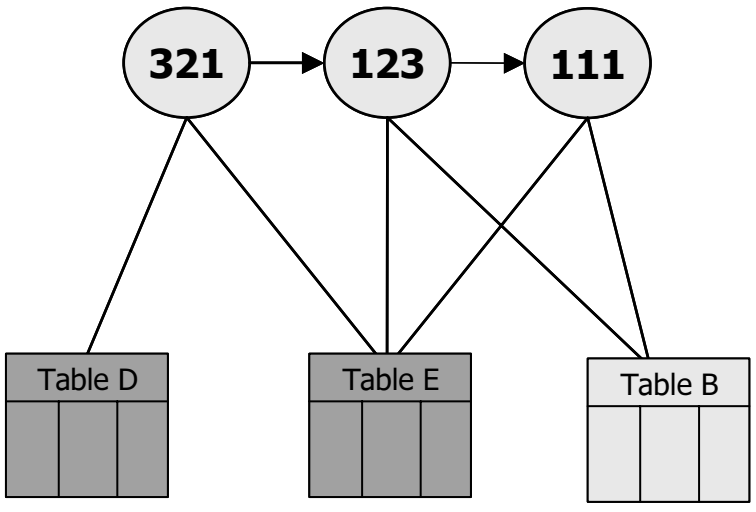


Figure 14: T2 During Phase 1

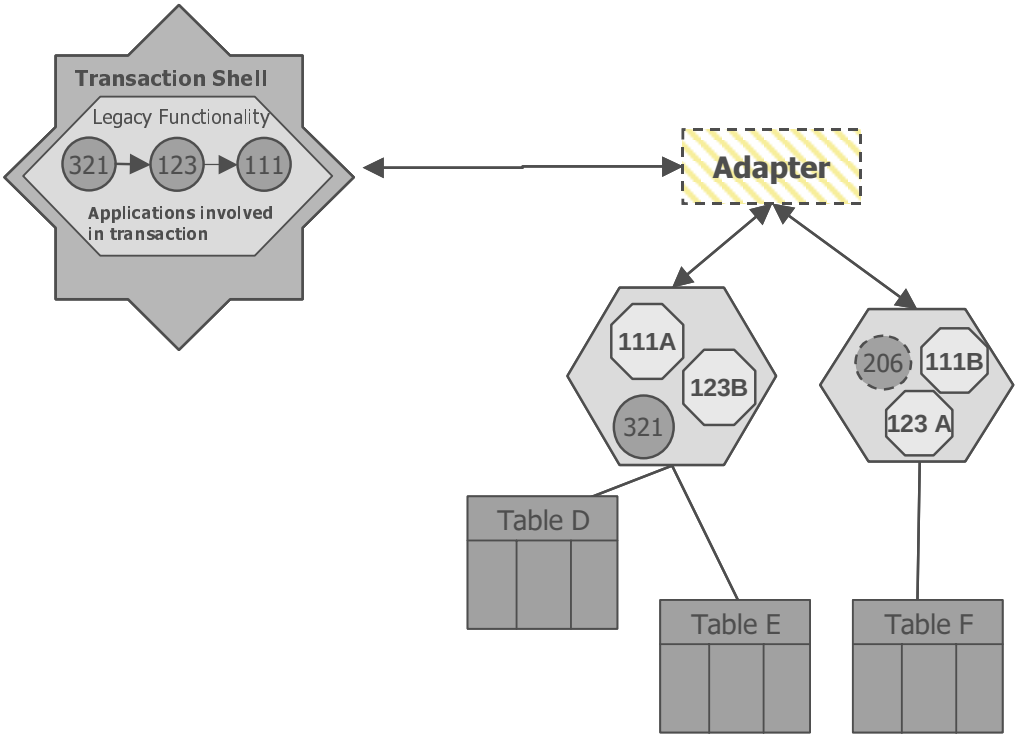


Figure 15: T2 During Phase 2

Candidate Trail–T3 (A1 B2 C1)

In Trail Map T3, shown in Table 9, we simultaneously restructure the database and componentize based on transactions. During the incremental development Phases 1 and 2, we deploy the new system under development in parallel with the operational legacy system. Finally, once the new system is complete (in Phase 3), we deploy it stand alone as the operational system. Figure 16 shows the system during Phase 1 with the operational system using the legacy database, the new system using the restructured database, and the migration of legacy functionality into the new system.

Table 9: Candidate Trail–T3 (A1 B2 C1)

Development Tasks	**Phase 1**	**Phase 2**	**Phase 3**
Restructure DB	restructure	restructure	restructure
Componentization	by transaction **OR** by application	by transaction **OR** by application	
Deployment	parallel ops **OR** operational	parallel ops **OR** operational	parallel ops **OR** operational

Time

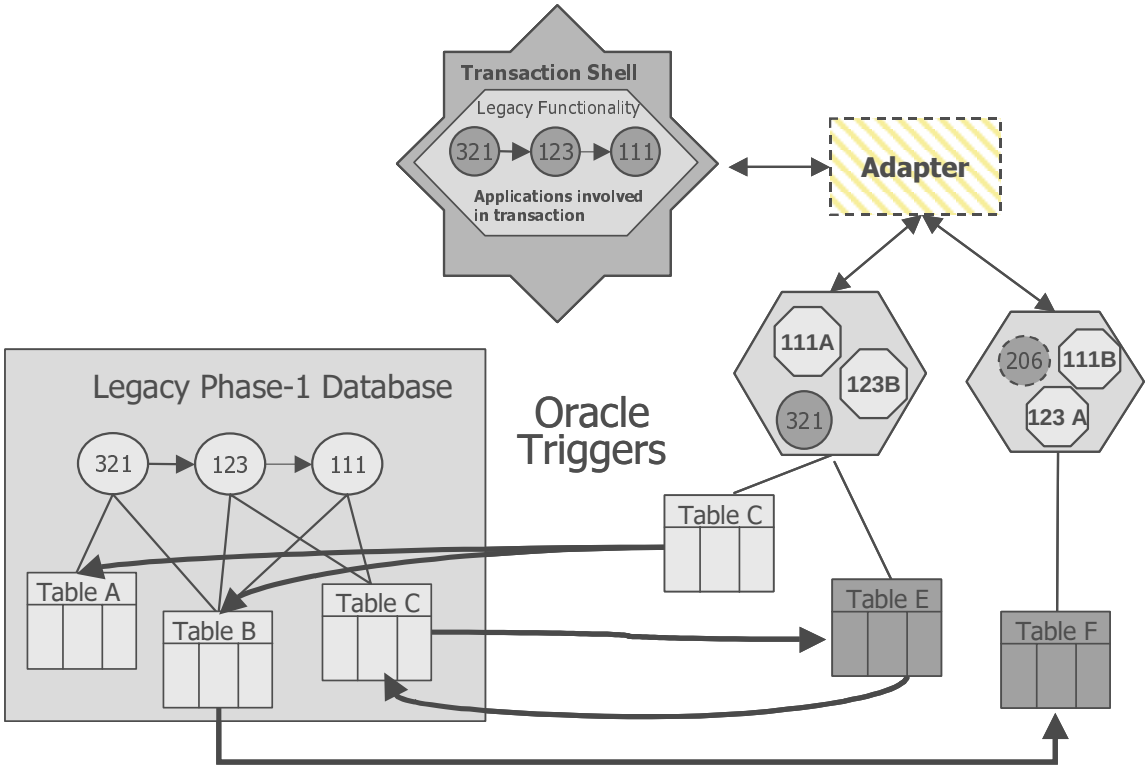
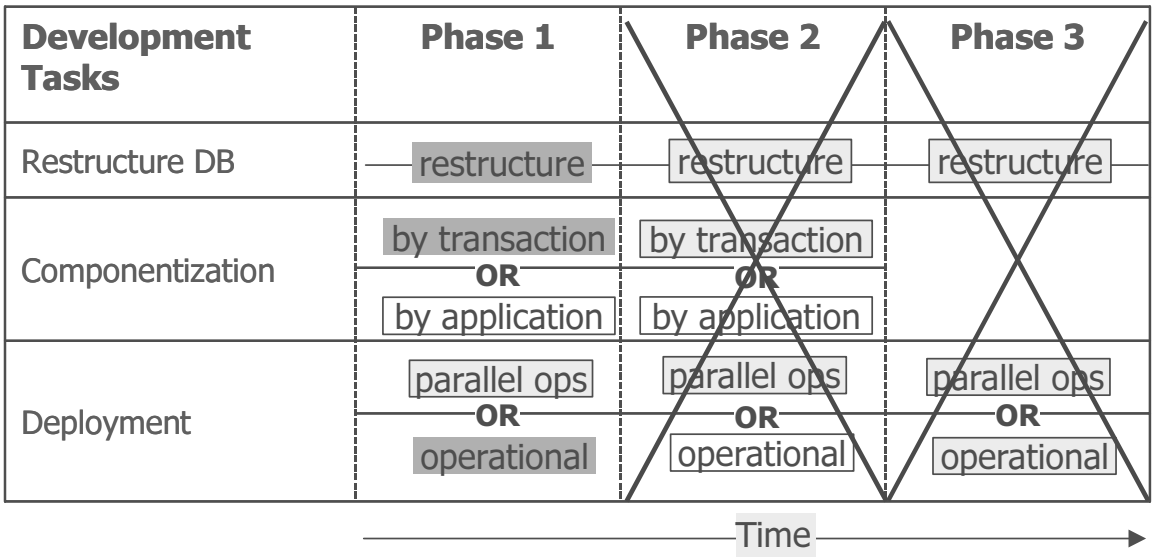


Figure 16: T3 During Phase 1

Candidate Trail–T4 (A1 B2 C2)

In Trail Map T4, shown in Table 10, we simultaneously restructure the database and componentize based on transactions, just as we did in Trail Map T3 with one exception: during the incremental development Phase 1, we deploy the new system as the operational system. Figure 17 shows the system during Phase 1 of construction, with the database being restructured and the migration of legacy functionality being moved into the new system.

Table 10: Candidate Trail–T4 (A1 B2 C2)



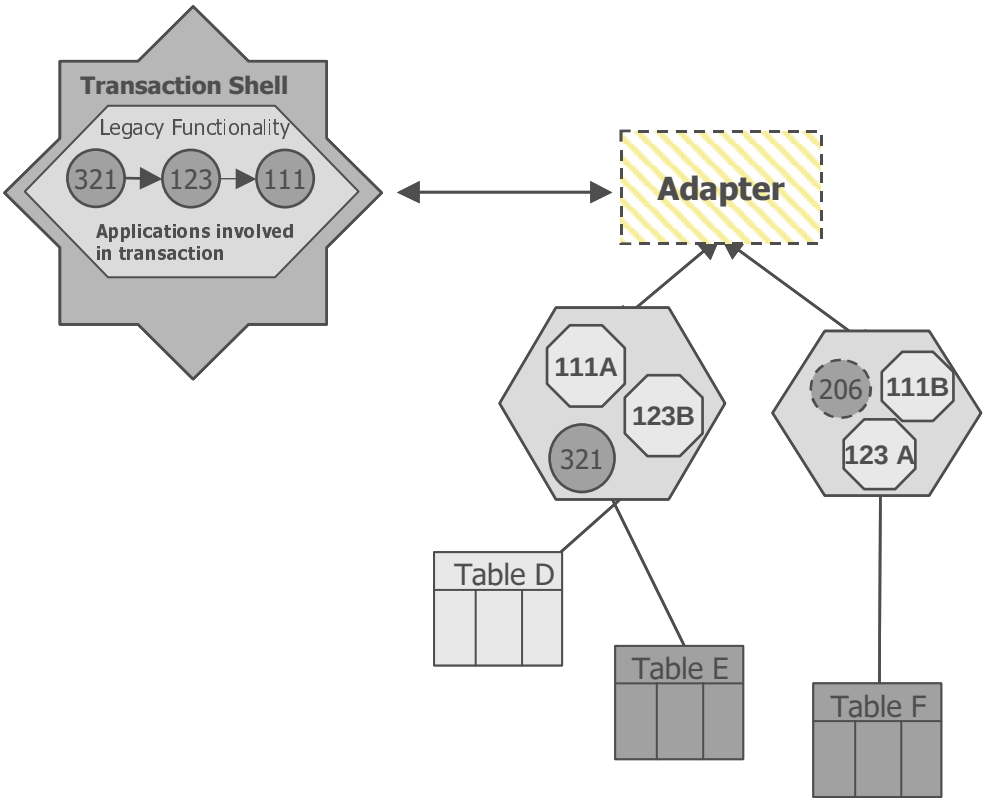


Figure 17: T4 During Phase 1

Candidate Trail–T5 (A1 B3 C1)

Trail Map T5 is shown in Table 11. In Phase 1, we componentize based on transactions, and then in Phase 2 we restructure the database. During the incremental development Phases 1 and 2, we deploy the new system under development in parallel with the operational legacy system. Finally, once the new system is complete (in Phase 3), we deploy it stand alone as the operational system. Figure 18 shows the system during Phase 1. During this phase, functionality from the legacy system is being migrated to the new system based on transactions, and updates between the two databases are performed using Oracle database triggers. During Phase 2, shown in Figure 19, the operational system is using the legacy database and the new system database is being restructured.

Table 11: Candidate Trail–T5 (A1 B3 C1)

Development Tasks	Phase 1	Phase 2	Phase 3
Restructure DB	restructure	restructure	restructure
Componentization	by transaction OR by application	by transaction OR by application	
Deployment	parallel ops OR operational	parallel ops OR operational	parallel ops OR operational
	Time →		

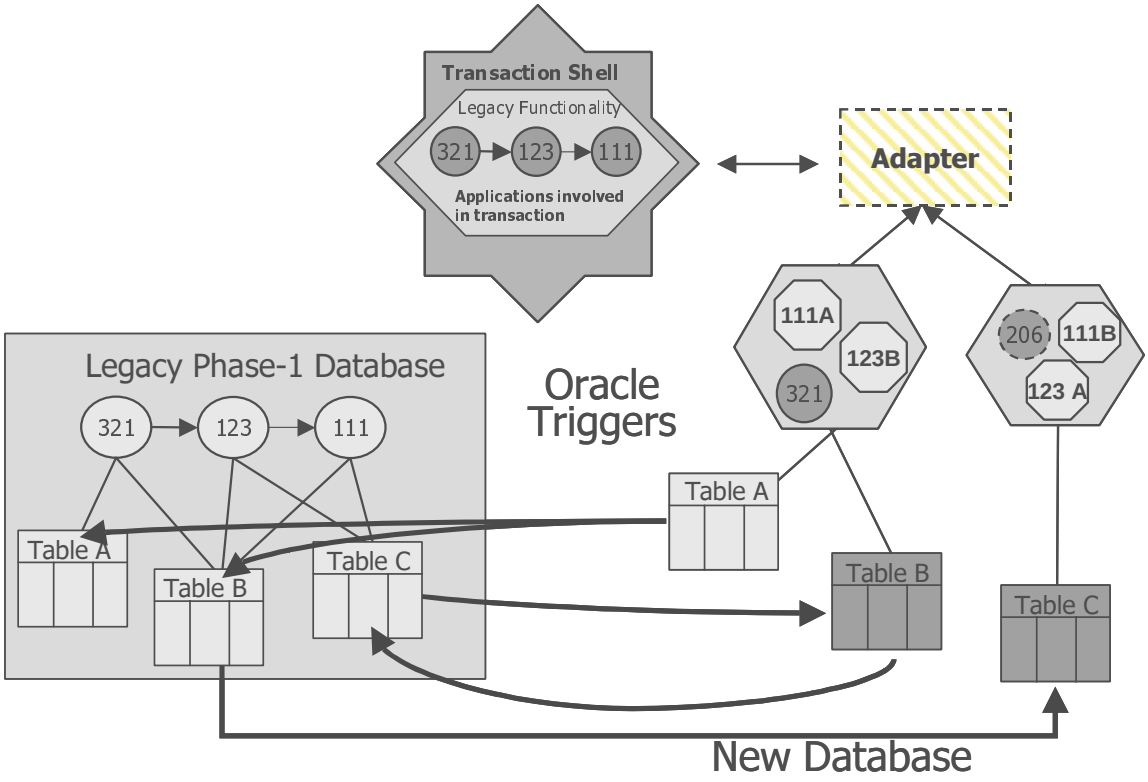


Figure 18: T5 During Phase 1

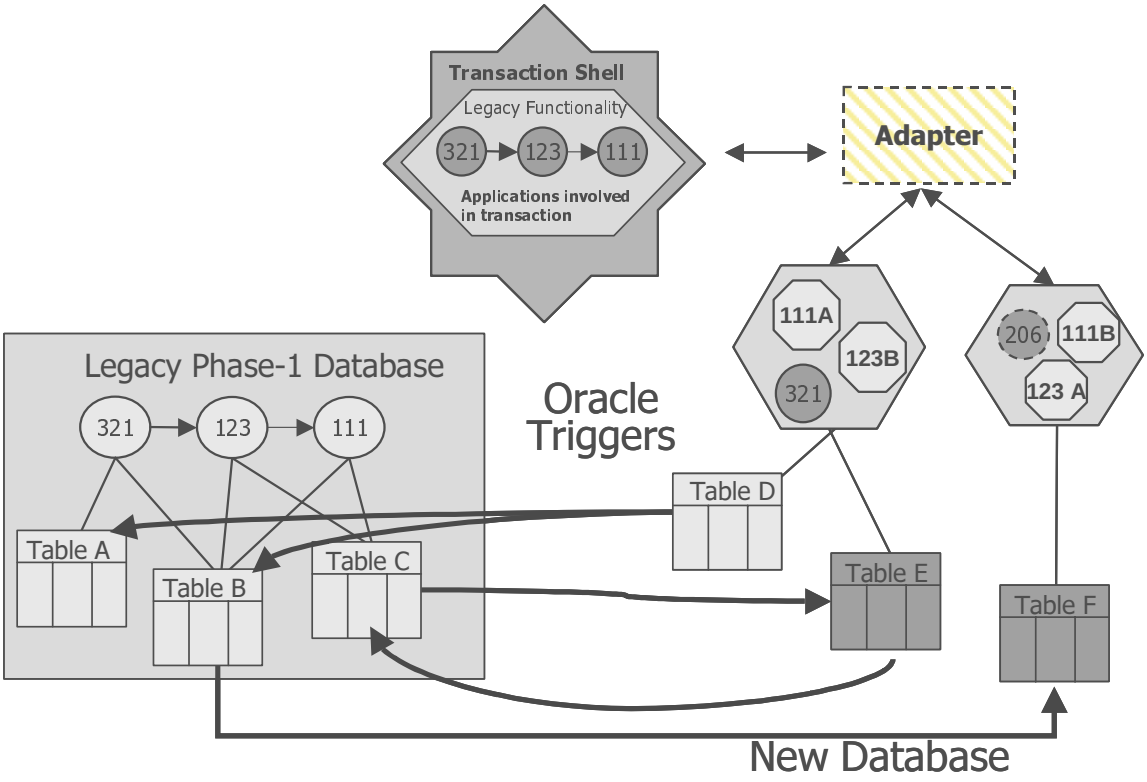
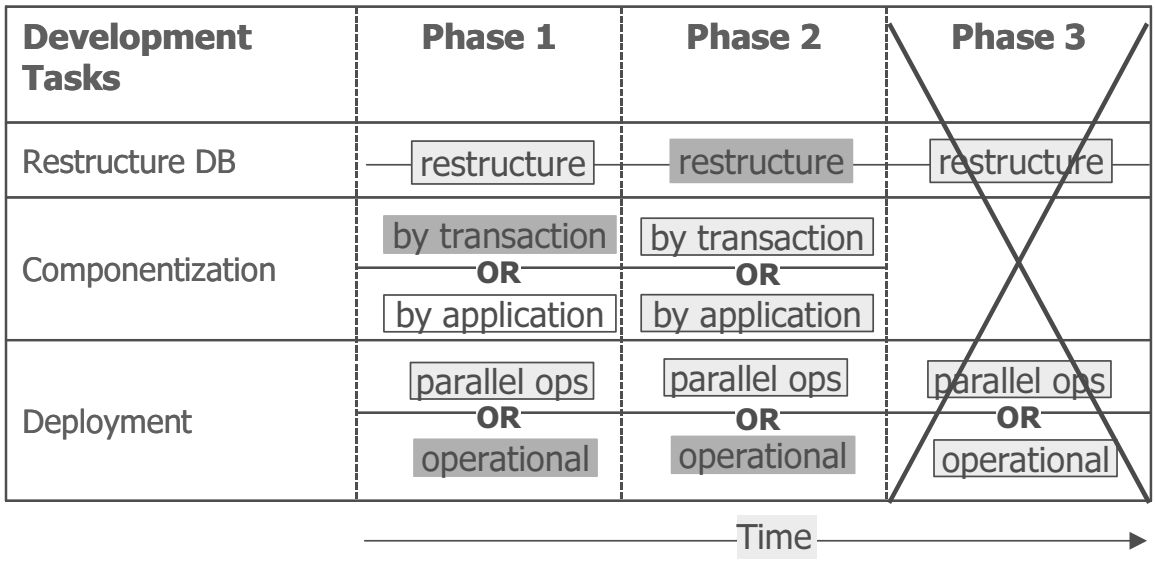


Figure 19: T5 During Phase 2

Candidate Trail–T6 (A1 B3 C2)

In Trail Map T6, shown in Table 12, we first componentize based on transactions, and then restructure the database. During the incremental development phases, we deploy the new system as the operational system. Figure 20 shows the system during Phase 1 when legacy functionality is being migrated to the new system based on transactions. During Phase 2, shown in Figure 21, database tables are being incrementally restructured.

Table 12: Candidate Trail–T6 (A1 B3 C2)



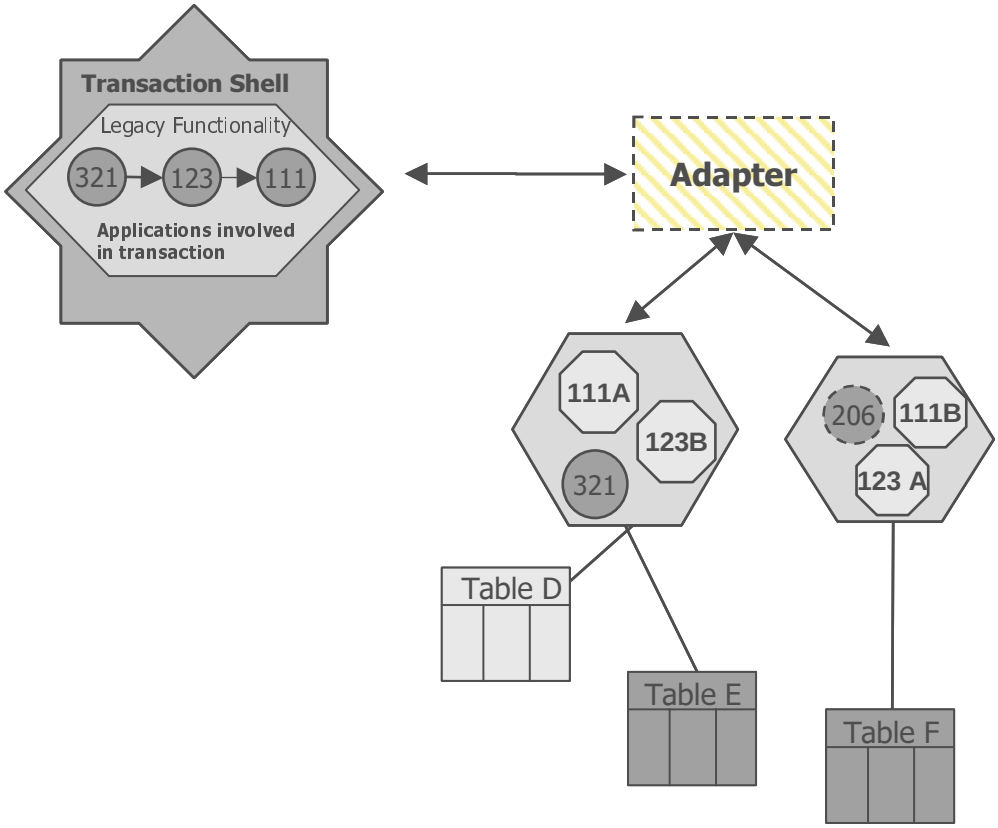


Figure 20: T6 During Phase 1

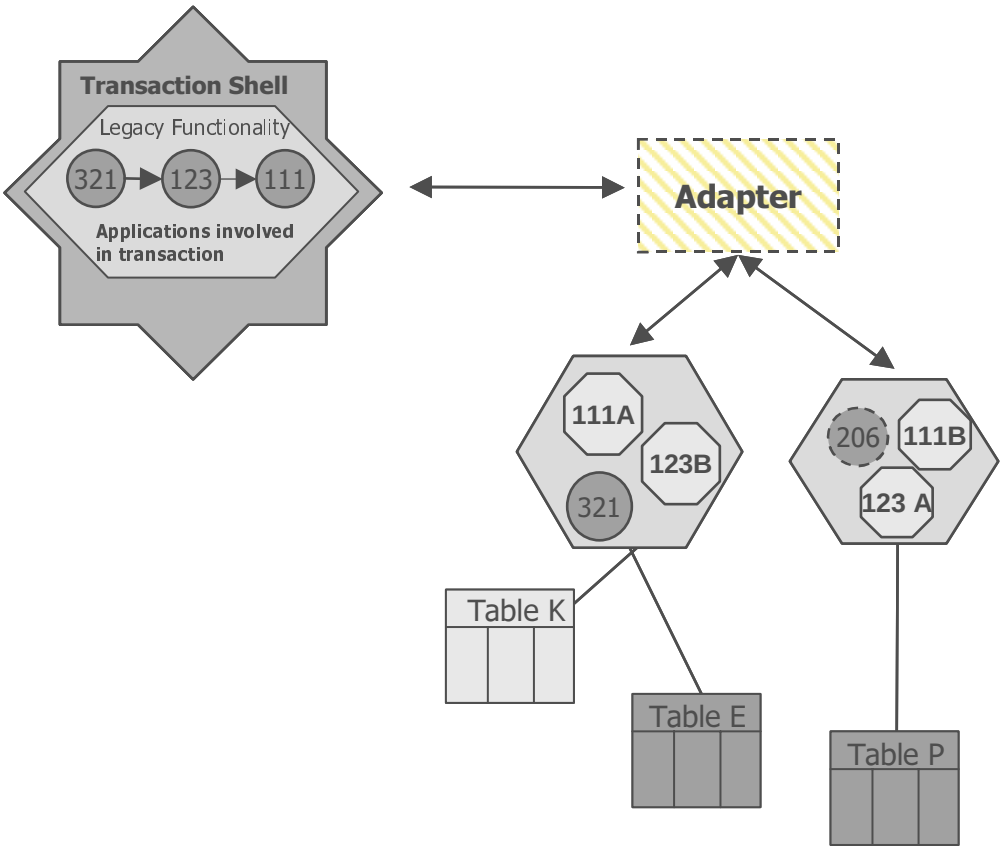


Figure 21: T6 During Phase 2

Candidate Trail–T7 (A2 B1 C1)

In Trail Map T7, shown in Table 13, we first restructure the database and then componentize based on applications. During the incremental development Phases 1 and 2, we deploy the new system under development in parallel with the operational legacy system. Finally once the new system is complete (in Phase 3), we deploy it stand alone as the operational system.

Figure 22 shows the system during Phase 1, with the operational system using the legacy database and the new system using the restructured database—updating each other using Oracle database triggers. During Phase 2, shown in Figure 23, functionality from the legacy system is being migrated to the new system based on applications.

Table 13: Candidate Trail–T7 (A2 B1 C1)

Development Tasks	Phase 1	Phase 2	Phase 3
Restructure DB	restructure	restructure	restructure
Componentization	by transaction **OR** by application	by transaction **OR** by application	
Deployment	parallel ops **OR** operational	parallel ops **OR** operational	parallel ops **OR** operational
	Time →		

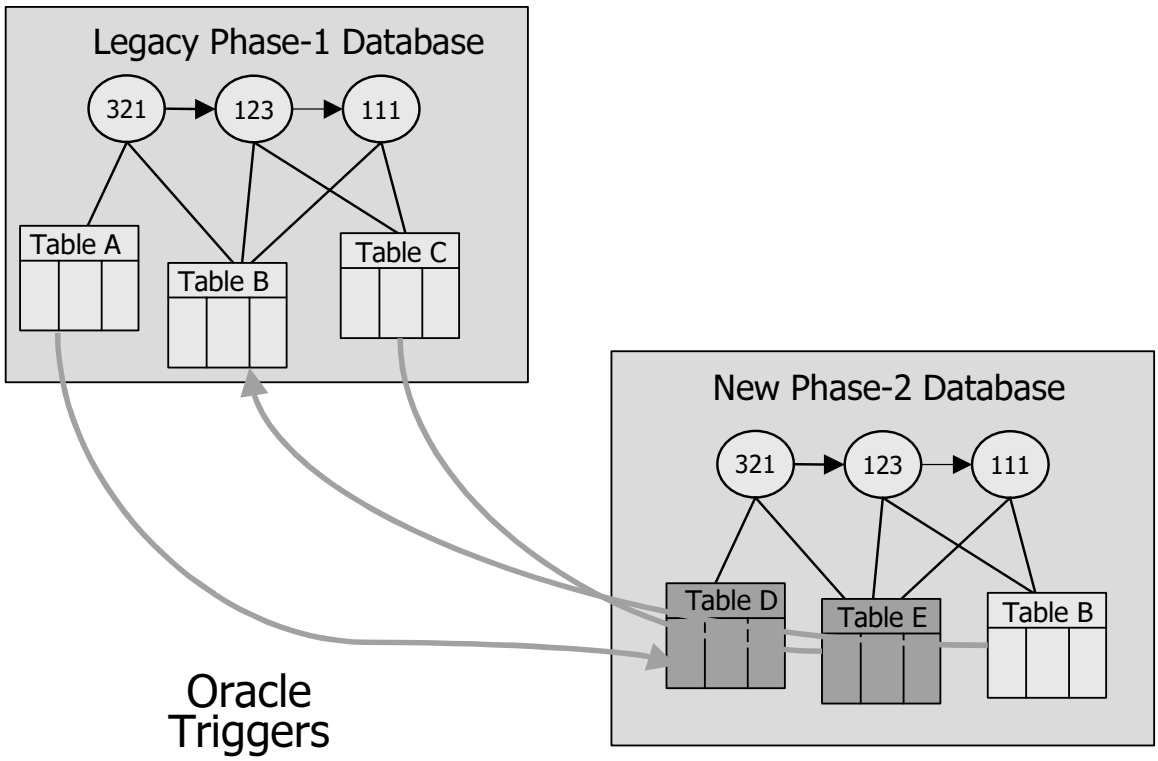


Figure 22: T7 During Phase 1

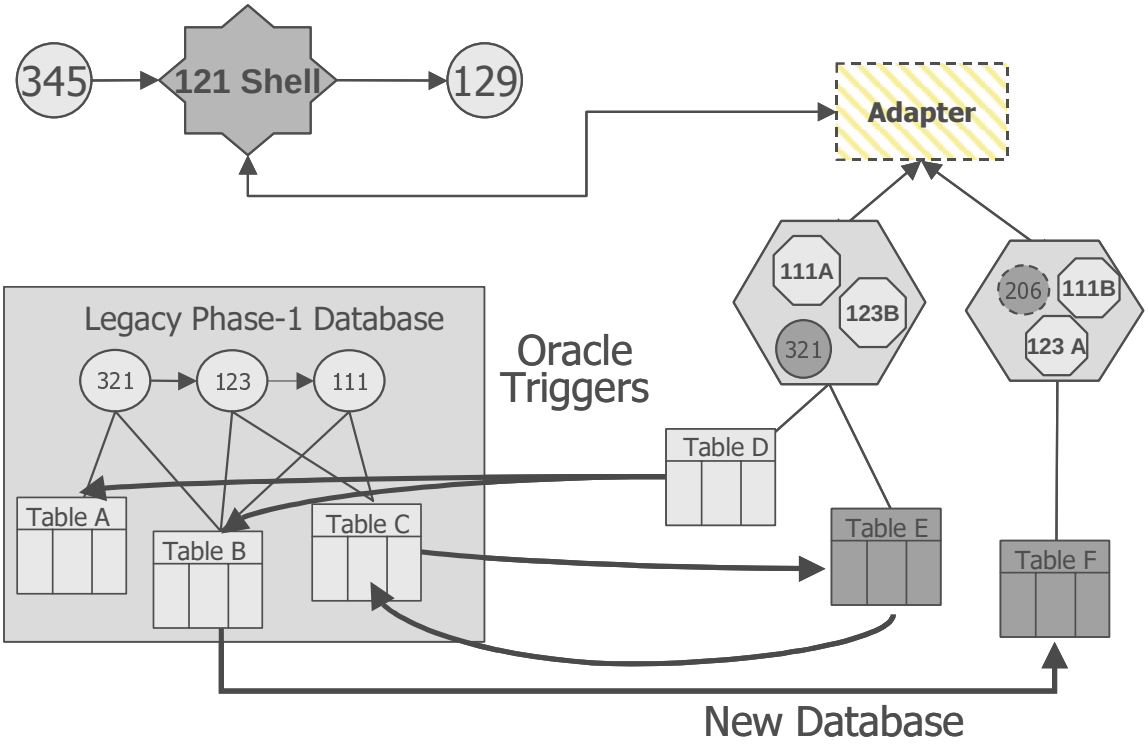


Figure 23: T7 During Phase 2

Candidate Trail–T8 (A2 B1 C2)

In Trail Map T8, shown in Table 14, we first restructure the database and then componentize based on applications. During the incremental development phases, we deploy the new system as the operational system. Figure 24 shows the system during Phase 2 as legacy functionality is being migrated to the new system based on applications.

Table 14: Candidate Trail–T8 (A2 B1 C2)

Development Tasks	Phase 1	Phase 2	Phase 3
Restructure DB	restructure	restructure	restructure
Componentization	by transaction **OR** by application	by transaction **OR** by application	
Deployment	parallel ops **OR** operational	parallel ops **OR** operational	parallel ops **OR** operational
	Time →		

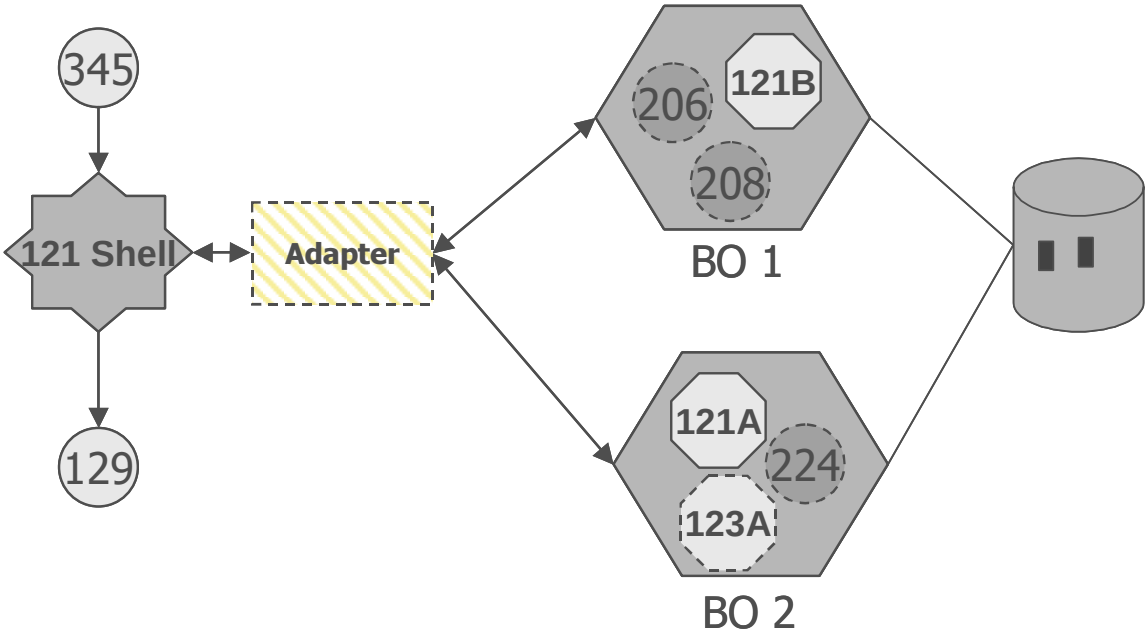


Figure 24: T8 During Phase 2

Candidate Trail–T9 (A2 B2 C1)

In Trail Map T9, shown in Table 15, we simultaneously restructure the database and componentize based on applications. During the incremental development Phase 1, we deploy the new system under development in parallel with the operational legacy system. Finally, once the new system is complete (in Phase 2), we deploy it stand alone as the operational system.

Figure 25 shows the system during Phase 1, with the operational system using the legacy database, the new system using the restructured database, and the migration of legacy functionality into the new system.

Table 15: Candidate Trail–T9 (A2 B2 C1)

Development Tasks	Phase 1	Phase 2	Phase 3
Restructure DB	restructure	restructure	restructure
Componentization	by transaction **OR** by application	by transaction **OR** by application	
Deployment	parallel ops **OR** operational	parallel ops **OR** operational	parallel ops **OR** operational

Time

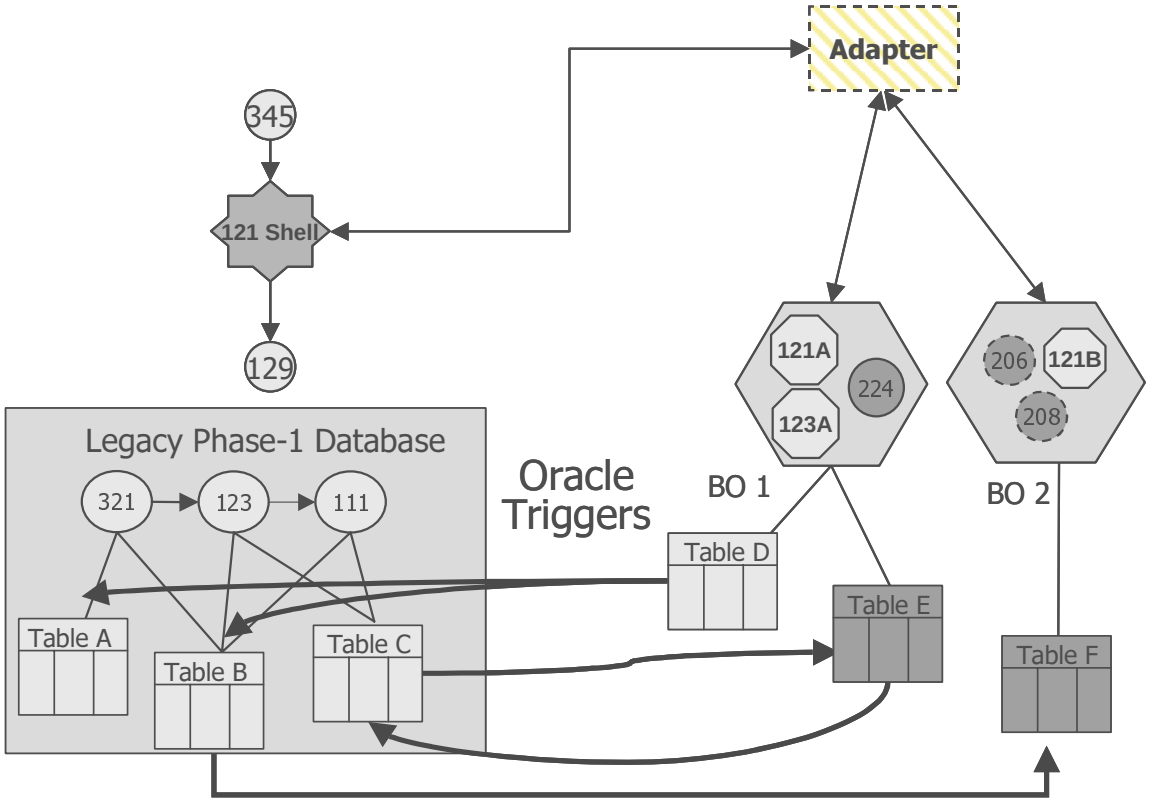
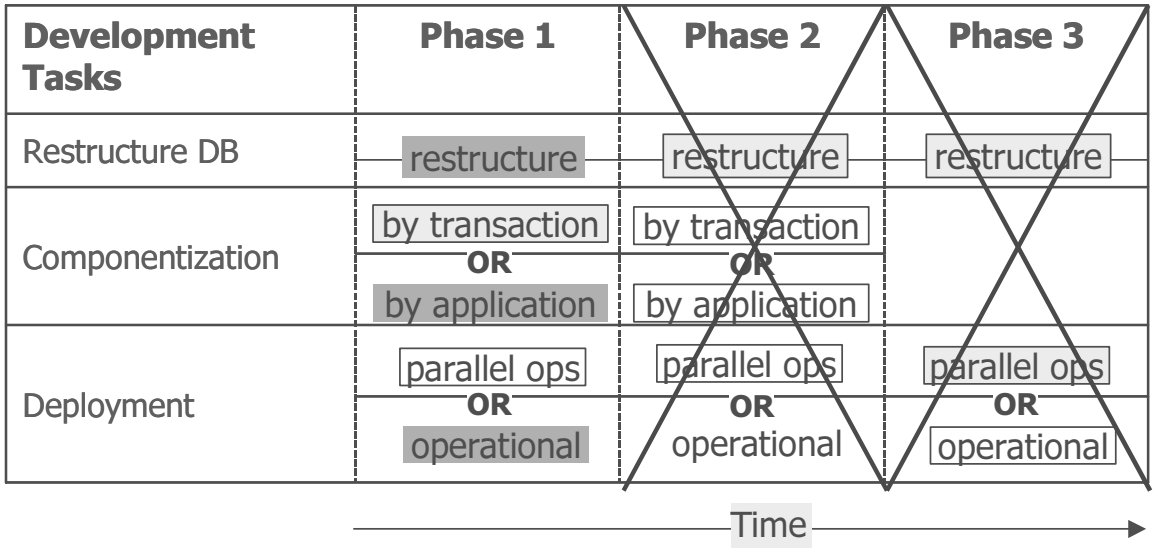


Figure 25: T9 During Phase 1

Candidate Trail–T10 (A2 B2 C2)

In Trail Map T10, shown in Table 16, we simultaneously restructure the database and componentize based on applications, just as we did in Trail Map T9 with one exception: during the incremental development Phase 1, we deploy the new system as the operational system. Figure 26 shows the system during Phase 1 of construction with the database being restructured and the migration of legacy functionality into the new system.

Table 16: Candidate Trail–T10 (A2 B2 C2)



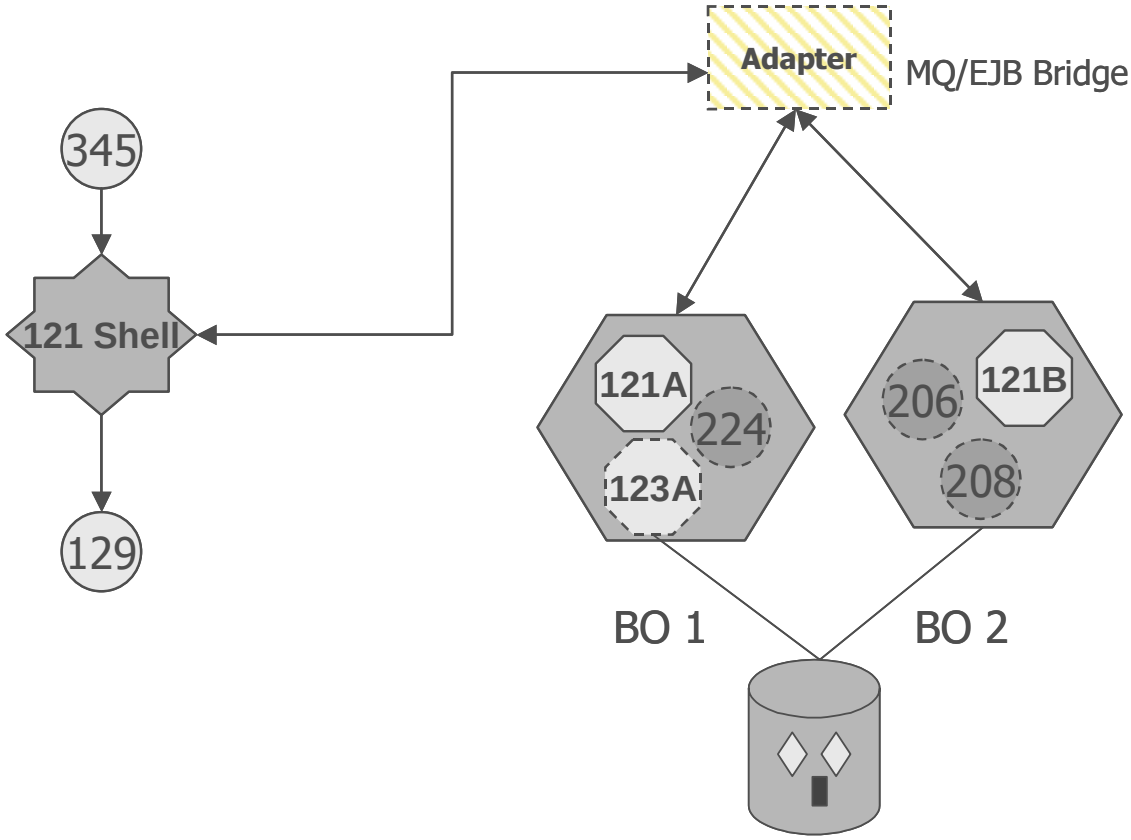


Figure 26: T10 During Phase 1

Candidate Trail–T11 (A2 B3 C1)

Trail Map T11 is shown in Table 17. In Phase 1, we componentize based on applications, and then in Phase 2, we restructure the database. During the incremental development Phases 1 and 2, we deploy the new system under development in parallel with the operational legacy system. Finally, once the new system is complete (in Phase 3) we deploy it stand alone as the operational system. Figure 27 shows the system during Phase 1. During this phase, functionality from the legacy system is being migrated to the new system based on applications, and updates between the two databases are performed using Oracle database triggers. During Phase 2, shown in Figure 28, the operational system is using the legacy database and the new system database is being restructured.

Table 17: Candidate Trail–T11 (A2 B3 C1)

Development Tasks	Phase 1	Phase 2	Phase 3
Restructure DB	restructure	restructure	restructure
Componentization	by transaction OR by application	by transaction OR by application	
Deployment	parallel ops OR operational	parallel ops OR operational	parallel ops OR operational
	Time →		

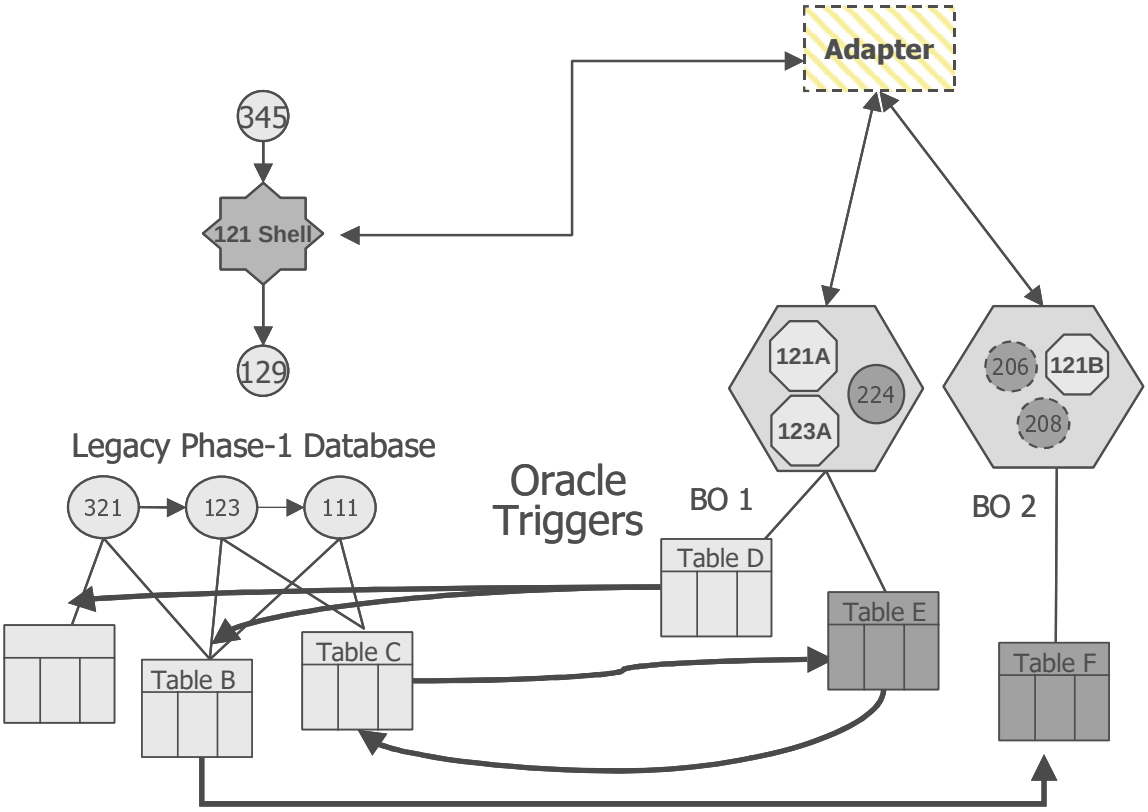


Figure 27: T11 During Phase 1

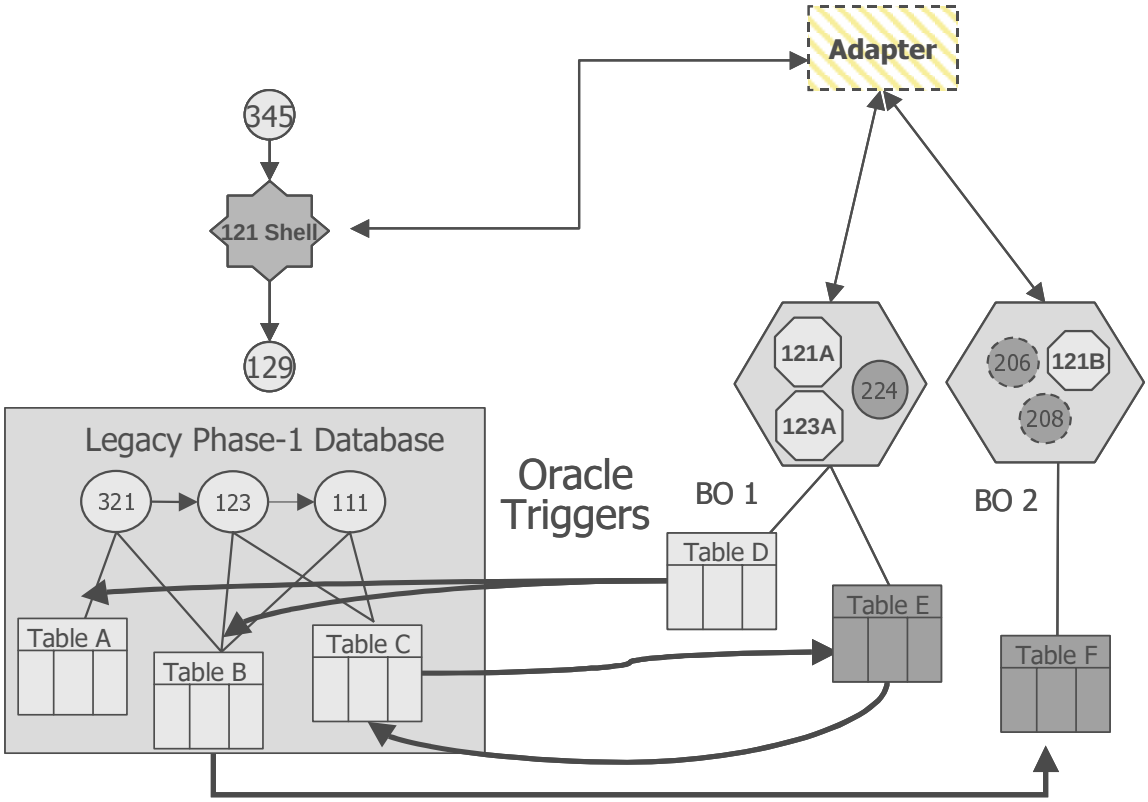


Figure 28: T11 During Phase 2

Candidate Trail–T12 (A2 B3 C2)

In Trail Map T12, shown in Table 18, we first componentize based on applications and then restructure the database. During the incremental development phases, we deploy the new system as the operational system. Figure 29 shows the system during Phase 1, when legacy functionality is being migrated to the new system based on transactions.

Table 18: Candidate Trail–T12 (A2 B3 C2)

Development Tasks	Phase 1	Phase 2	Phase 3
Restructure DB	restructure	restructure	restructure
Componentization	by transaction **OR** by application	by transaction **OR** by application	
Deployment	parallel ops **OR** operational	parallel ops **OR** operational	parallel ops **OR** operational
	Time →		

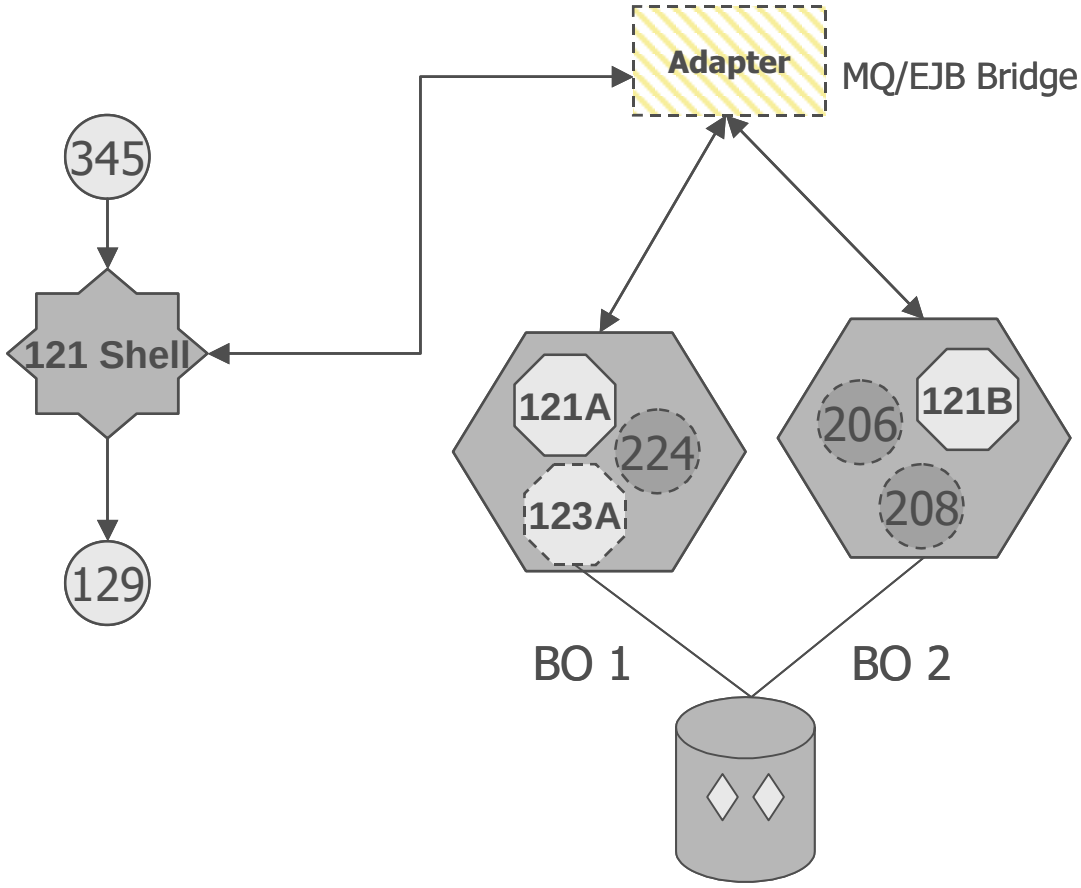


Figure 29: T12 During Phase 1

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
(Leave Blank)	July 2001	Final

4. TITLE AND SUBTITLE	5. FUNDING NUMBERS
Legacy System Modernization Strategies	F19628-00-C-0003

6. AUTHOR(S)

Robert C. Seacord, Santiago Comella-Dorda, Grace Lewis, Pat Place, Dan Plakosh

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
Software Engineering Institute Carnegie Mellon University Pittsburgh, PA 15213	CMU/SEI-2001-TR-025

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSORING/MONITORING AGENCY REPORT NUMBER
HQ ESC/XPK 5 Eglin Street Hanscom AFB, MA 01731-2116	ESC-TR-2001-025

11. SUPPLEMENTARY NOTES

12A DISTRIBUTION/AVAILABILITY STATEMENT	12B DISTRIBUTION CODE
Unclassified/Unlimited, DTIC, NTIS	

13. ABSTRACT (MAXIMUM 200 WORDS)

Modernization of legacy enterprise systems introduces many challenges due to the size, complexity, and frailty of the legacy systems. Size and complexity issues often dictate that these systems are incrementally modernized, and new functionality is incrementally deployed before the modernization effort is concluded. This in turn requires that legacy components operate side by side with modernized components in an operation system—introducing additional problems.

In this report we discuss some alternative development approaches for incrementally modernizing legacy systems, including consideration of the advantages and disadvantages of each approach. These development alternatives can be mapped against the peculiarities of a particular modernization effort to recommend an appropriate approach.

14. SUBJECT TERMS	15. NUMBER OF PAGES
Legacy system modernization, incremental development and deployment, incremental modernization	74

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL