

Oracle Modernization

Oracle IT Modernization Series:
z/OS Mainframe for Dummies

An Oracle White Paper
January 2007

Oracle IT Modernization Series
z/OS Mainframe for Dummies

Introduction	3
In the Beginning ... There Was Batch	4
Introducing Third-Generation Languages	7
Everyone Can Access: The Introduction of Terminals	8
Improving Data Access Through Better File Systems and Databases	10
The Fourth-Generation Languages.....	12
So What's Wrong with This Picture?	14
Enter Modernization.....	15
Conclusion.....	15

IBM z/OS is a powerful environment that many organizations are looking to move away from because of its high cost, low agility, and reliance on hard-to-find skills.

INTRODUCTION

Mainframe technology is still very prevalent within the IT world. Although many pundits have predicted the death of the mainframe, there are over 15,000 mainframes running production applications today.

When mainframes were first produced in the 1950s, there were known simply as “computers.” The “mainframe” moniker was coined over time as minicomputers were developed, to differentiate the large systems from the newer, smaller systems.

In the early days of the mainframe, a number of companies produced hardware in this space to address the marketplace. IBM, Control Data Corporation (CDC), Honeywell, NCR, Burroughs, and Univac all produced mainframe computers. Today, IBM dominates the mainframe marketplace, with Unisys, Fujitsu, and Hitachi representing a much smaller share of the market.

In addition to selling hardware, each of these vendors also produced an operating system that drove the mainframe hardware. Since most mainframe computers had only one operating system, it has become common to equate the hardware name with the operating system name. One exception to this is that IBM now has two operating systems that run on its mainframe hardware: its legacy operating system, called z/OS (previously called MVS), and a version of Linux called z/Linux.

Because much of what IT personnel deal with relates to the operating system, it is important to think of the environment in terms of the operating system. Thus the legacy mainframe environment is an IBM z/OS mainframe, and the IBM z/Linux mainframe environment is the same computer running a version of Linux—a much more modern environment.

Even with the proliferation of increasingly faster microcomputers and personal computers, highly efficient mainframe applications are still heavily utilized throughout the Fortune 1000. At the same time, these legacy environments are costly and difficult to maintain, and organizations are looking to move away from them.

This white paper will introduce you to the legacy IBM z/OS mainframe and examine its history. It will help you understand the complex structure of the IBM

z/OS environment, and why its complexity is making organizations consider moving to other environments.

In the Beginning ... There Was Batch

The basic concepts that make up z/OS were first introduced in the 1960s. At that time, the only way to interact with a computer was to use punch cards as input and receive computer printouts as output. The concept of a terminal did not yet exist. To interact with z/OS, users gave the computer operator a deck of punched cards—called a job—containing the instructions for what they needed done. The computer operator then took a number of these jobs and combined them into a batch that was placed in a card reader and run through the computer. The computer operator then took a number of these jobs and combined them into a batch that was placed in a card reader and run through the computer.

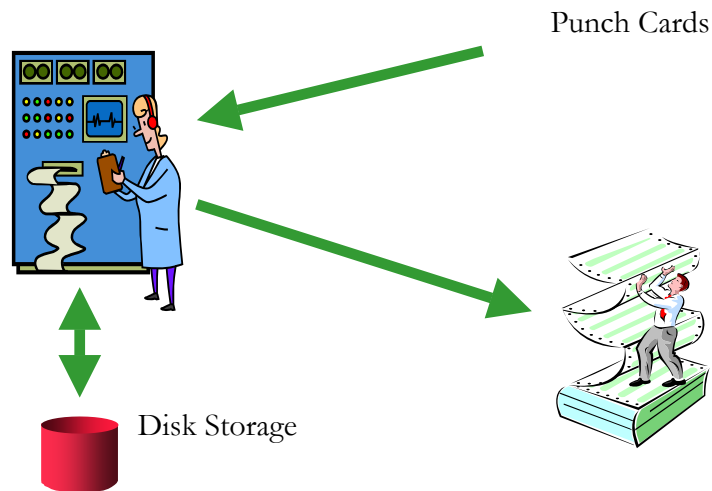


Figure 1—The Batch Environment

As a result of this batch environment—which is still heavily used even today—a number of terms developed in the 1960s that are still in use:

- *Batch* implies interacting with a computer without a terminal.
- Batches are made up of a number of *batch jobs*, or simply *jobs*, that operators submit to z/OS.
- A *batch job* executes a number of *job steps*, each of which executes one program. To control the program flow when no one is there, the operator can embed the programs in a scripting language called *JCL (Job Control Language)*.
- Many batch jobs run during off-peak hours, such as overnight, and perform tasks that must be completed before the next day. The *batch window* is the timeframe during which all overnight batch jobs must complete.

- Although batch jobs were initially run by production personnel working the night shift, over time many batch jobs came to be managed by *job scheduling systems* that determined when the batches should be run.

JCL is an important component of the z/OS batch environment. An easy way to understand JCL is to think of it as a workflow language for batch programs—a way to script which program runs next and under what conditions when there is no one around to manage the process manually.

```
//KEITHA JOB TS100,'PROGRAM1',CLASS=A,MSGCLASS=D,NOTIFY=KEITH
//S1 EXEC PGM=CBL001
//STEPLIB DD DSN=APG.LOADLIB,DISP=SHR
// DD DSN=SYS1.SCEERUN,DISP=SHR
//SYSOUT DD SYSOUT=A
//RDSOUT DD DSN=KEITH.A.JCLIB(REP001),DISP=SHR
//DDIN DD DSN=KEITH.A.JCLIB(SIMPLE1),DISP=SHR

//KEITHA JOB TS100,'PROGRAM2',CLASS=A,MSGCLASS=D,NOTIFY=KEITH
//S1 EXEC PGM=CBL001
//STEPLIB DD DSN=APG.LOADLIB,DISP=SHR
// DD DSN=SYS1.SCEERUN,DISP=SHR
//SYSOUT DD SYSOUT=A
//RDSOUT DD DSN=KEITH.A.JCLIB(REP002),DISP=SHR
//DDIN DD DSN=KEITH.A.JCLIB(SIMPLE2),DISP=SHR
```

Figure 2—Example of a Batch of Two Jobs Scripted Using z/OS Job Control Language (JCL)

Of course, no one said JCL was an intuitive scripting language.

Although JCL controls the execution of a set of programs, each program itself still has to be written in a specific computer language.

The very first computer language wasn't a language at all, but a technique—the user toggled the switches on the front of the machine itself. This toggling process is called *first-generation programming* and—thank goodness—is not used anymore.

Because toggling switches was an error-prone process, the concept of a computer language was introduced. The first of these were called *assembly languages* and formed the second generation of computer programming languages.

```

          PRINT NOGEN
          REGEQU
FMT001A1 CSECT
          STM   R14,R12,12(R13)
          LR    R3,R15
          USING FMT001A1,R3
          ST    R13,WSAVE+4
          LA    R14,WSAVE
          ST    R14,8(R13)
          LA    R13,WSAVE
*
          OPEN (DDIN,(INPUT))
          OPEN (RDSOUT,(OUTPUT))
          BAL   R10,GETREC          READ FIRST RECORD
          B     LAB110
LAB100   BAL   R10,GETREC          READ NEXT RECORD
          EX    R5,WCLC1          INDEX WORD CHNGED ?
WCLC1   CLC   WLAST(1),0(R4)
          BE   LAB120          NO.
          PUT  RDSOUT,WPRT      YES - WRITE PRINT
LINE
          MVI  WPRT,C' '          CLEAR PRINT
LINE
          MVC  WPRT+1(79),WPRT
LAB110  EX    R5,WMVC1          STORE INDEX WORD
WMVC1   MVC  WPRT(1),0(R4)
          LA   R8,WPRT          ADDRESS PRINT LINE
          LA   R8,2(R5,R8)      SKIP OVER
WORD
          B    LAB130
LAB120  MVI  0(R8),C','          ADD SEPARATOR
          LA   R8,1(R8)        INDEX OVER

```

Figure 3—Example of z/OS Assembly Code Segment

Since assembly languages represent the actual basic instructions a computer understands, each computer has its own assembly language.

Assembly languages are very low-level and—as is obvious from Figure 3—are only slightly better than toggling switches! However, because users code directly in the machine’s own instruction set, they can write code that runs very, very fast. For this reason, assembly language was often used to write routines that demanded high performance (although some companies have written entire applications in assembly language). Because programs written using assembly language are dependent on the machine architecture, any change of hardware platform necessitates eliminating any assembly routines; they cannot run on a different computer.

As computer hardware became faster, and better computer languages came about, the need for assembly languages vanished. However, it is not uncommon to find a number of assembly routines still “hanging around” as part of a legacy application.

In the original z/OS batch environment, the user could also access disk storage. However, the access was at a low level, because data was stored in a file system much like the file systems used in desktop computers. Concepts such as databases were still a thing of the future.

The batch environment still exists today in all z/OS environments. One of the key issues in modernization is deciding what to do about these batch jobs that were written in legacy languages, access low-level file systems, and have done intensive “offline” processing, often overnight, for years.

Introducing Third-Generation Languages

If you examine the code in Figure 3, it quickly becomes apparent that it would be nice to use something other than assembly language for writing computer programs.

One of the greatest early advancements in computing was the introduction of third-generation (3GL) languages—languages in which the code is “higher level” and looks somewhat more like English. The first of these—and still the most common programming language in the world—was COBOL.

PROCEDURE DIVISION.

```
005100 MOVE 2 TO r3
005200 MOVE 0 TO WRETURN
005300 OPEN INPUT DDIN-DDNAME
005400 OPEN OUTPUT RDSOUT-DDNAME
005500 PERFORM GETREC
005600 IF exit-flag NOT = 1 THEN
005700 MOVE 0 TO fl-flag2
005800 PERFORM UNTIL fl-flag2 NOT = 0
005900 CALL "CBL001A1" USING WPRT TMP-FWORD-1
006000 MOVE r4 TO TMP-FWORD
006100 SET ADDRESS OF LINK-TMP-DATA-1 TO TMP-PTR
006200 MOVE TMP-FWORD-1 TO TMP-FWORD
006300 SET ADDRESS OF LINK-TMP-DATA-2 TO TMP-PTR
006400 MOVE LINK-TMP-DATA-1(1:(r5 + 1))
006500 TO LINK-TMP-DATA-2(1:(r5 + 1))
```

Figure 4—Example of COBOL Code Segment

Languages such as COBOL allow a programmer to express concepts at a more abstract level not related to the instruction set of the computer itself. To execute the program, the user must run the program through a *compiler* that translates the higher-level statements into machine instructions—with one statement in the programming language generating the many machine instructions needed to carry out the statement.

Another advantage of 3GL languages such as COBOL is that they are no longer related directly to the instruction set of any machine, thus you can move programs written in these languages from one machine to another. All you need is a COBOL

compiler for each machine that translates the COBOL into machine instructions and an agreement among all the COBOL compiler manufacturers to use exactly the same definition for COBOL (something that is not always that easy to achieve).

There are many 3GL languages that operate in the z/OS environment, but the most common is COBOL, with PL/1 running second. By some estimates, there are over 200 billion lines of COBOL still in production today. Although most organizations would not today use COBOL for a new application, the total number of lines of COBOL code continues to grow approximately 10 percent per year due to the maintenance and extension of existing applications. The vast majority of z/OS mainframe applications are written in COBOL.

Everyone Can Access: The Introduction of Terminals

Batch processing is very powerful, but very slow when it comes to interacting with users requiring users to wait for their jobs to be run before they can see the results. With demand building for more direct access, the concept of having users access a computer directly via terminals came about in the 1970s.

Interestingly, the handling of terminals was not incorporated directly into the z/OS operating system. Instead, terminals became the responsibility of an additional piece of software called a *teleprocessing monitor (TP monitor)* that runs on top of the core z/OS operating system.

IBM introduced two TP monitors for the z/OS environment: *CICS* and *IMS/DC*. Each z/OS site typically has one or the other. CICS is older and more common, but many large sites make use of IMS/DC (which IBM markets as more efficient for large sites). Although IMS/DC was first developed in the 1960s for NASA, today over 90 percent of Fortune 1000 companies still use the IMS/DC teleprocessing monitor along with its IMS/DB file system. Collectively called simply *IMS*, IMS/DC and IMS/DB are capable of processing more than 21,000 transactions per second on larger machines.

For simplicity, we will discuss only CICS. (For the purposes of this white paper, CICS and IMS/DC can be considered essentially interchangeable.)

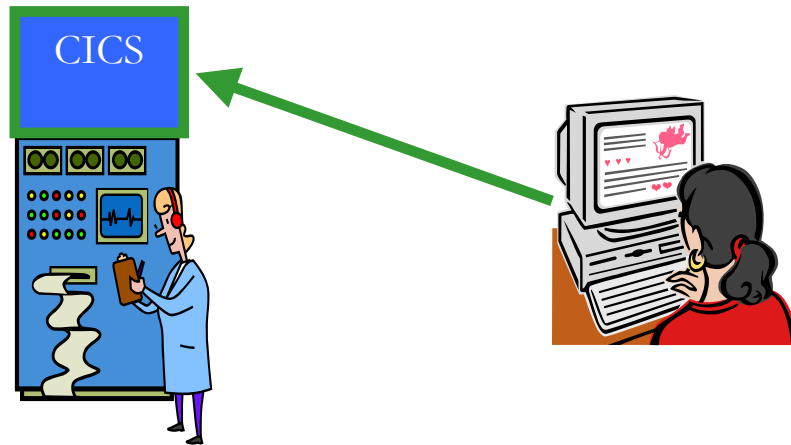


Figure 5—Adding Terminals to the z/OS Mainframe

TP monitors such as CICS and IMS/DC were created to service huge numbers of users for online workloads at a time when computers were physically large but small in capacity. For example, early mainframes might have 128,000 bytes of memory compared to the millions of bytes of memory in any laptop today. The 1960s and 1970s were the age when computer resources were expensive and people were cheap. Remember how many programs were written with two-digit dates in order to save two characters of space?

As a result of the need to preserve computer resources, a programming style called *pseudo-conversational*, used only by CICS and IMS/DC, was developed to minimize the amount of resources held by an application while it waited for a user to finish typing. To understand the impact this has on legacy applications, you must first understand a bit about conversational programming.

In conversational programming, the program displays a screen or a form, waits until the user has finished entering data, and then continues with the next statement in the same program.

- 1) Step 1
- 2) Step 2
- 3) Display Screen
- 4) Step 4
- 5) Etc.



Figure 6—Conversational Programming

Anyone who uses a PC is familiar with this model. For example, when you work on your PC, you often have a number of “minimized” applications along the bottom

of the screen. These applications stand by, waiting for you to do something with them, and then resume where they left off. In the meantime, they take up room.

This model is the “normal” model today given that today’s computers have a huge capacity and can easily keep a number of programs open at the same time. But given the small capacity of computers in the 1960s, using a conversational style would have dramatically limited the number of users they could support. Hence the creation of pseudo-conversational programming.

Pseudo-conversational programming techniques force the release of all resources while waiting for the end user—a period of time that the computer sees as “infinitely long” compared with the time it needs to actually execute code between user interactions.

To achieve this, when the program displays a screen to the user, CICS terminates the program, which releases resources and gives the user time to respond to the screen. When the user is finished, CICS either restarts the same program or starts another program—but always starts from the beginning.

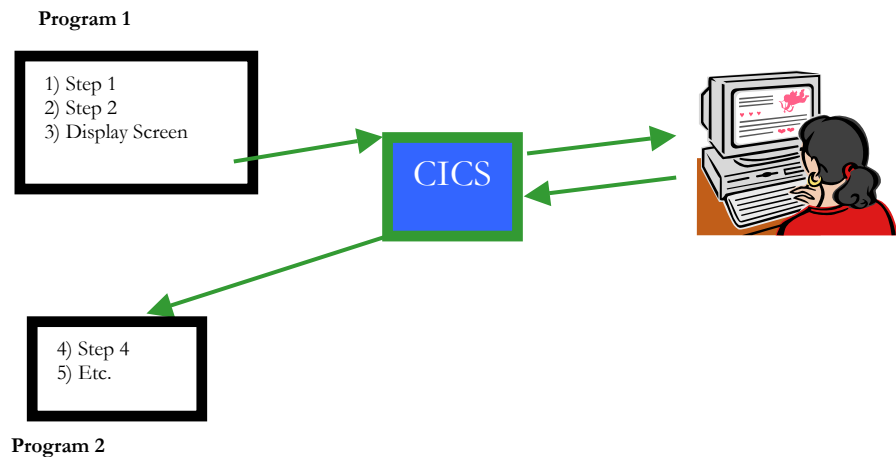


Figure 7—Pseudo-Conversational Programming

Pseudo-conversational programming results in programs that are “chopped up” into little pieces. These pieces send requests to CICS for pretty much everything they do—not just for handling terminals, but also for things like data access. This makes CICS (or IMS/DC) one of the hardest areas to modernize, because the architectural approach used is so different from modern programming approaches such that used by J2EE and Java.

Improving Data Access Through Better File Systems and Databases

During the 1960s, it was also recognized that easier access to data would speed up the programming process.

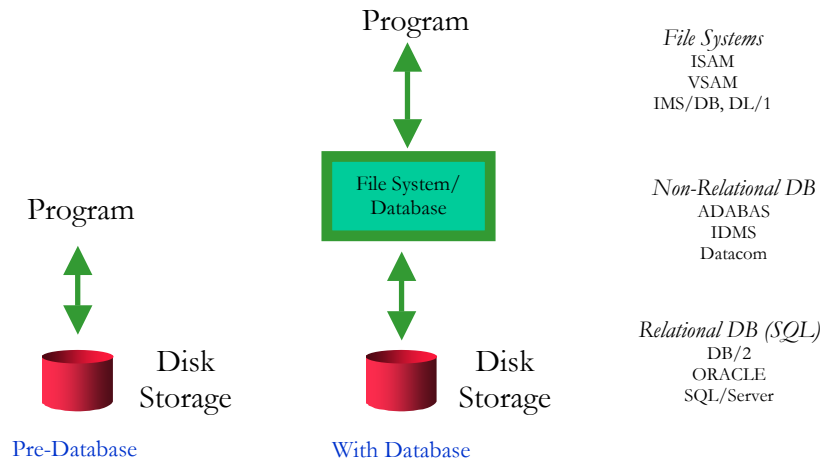


Figure 8—Better File Systems and Databases

One of the first improvements for accessing data was the introduction of *VSAM* (*Virtual Storage Access Method*), a file access method that uses keys to randomly access data (for example, find the customer whose customer number is 1234) or sequentially access data (for example, read a file from one end to the other). Prior to this time, only sequential access was possible.

VSAM files allow data to be accessed in a more structured manner, but VSAM does not divide the records within each file into separate fields. You can store anything you want in any given VSAM record—and programmers wrote programs that did just that. For this reason, applications that use VSAM can be difficult to modernize, as one has to “untangle” the data in the records.

Other mechanisms were also introduced in an attempt to improve data access. IMS/DB, created by IBM in the 1960s, is a hierarchical file system that stores data using a highly optimized parent-child relationship designed for speed.

Around this time, other 3rd party vendors began to introduce the concept of a true database that could store data in very efficient formats, handle the concept of transactions (where the database ensures that a unit of work that updates multiple records happens either completely or not at all), and guarantee that all records in the same file always had the same set of fields—unlike VSAM and IMS/DB, which viewed each record as a blob of data in which you could store anything.

Although there are a number of these legacy databases, the most common ones still used today are

- ADABAS, developed and sold by Software AG
- IDMS, developed by Cullinet and today maintained by CA
- Datacom, developed and sold by Cincom.

In the 1980s, a new concept emerged within the database world: the relational model for storing data and *Structured Query Language (SQL)* as a means to access the data. The relational model using SQL quickly became the prevailing standard.

Oracle was one of the first companies to market an SQL database, and Oracle Database 10g continues to adhere to the SQL standard.

In 1983—two years after Oracle released the Oracle relational database—IBM also released an SQL database on z/OS in the form of DB/2. Unlike VSAM- or IMS/DB-based data, SQL enables a common interface that keeps the data structured in a well-defined format with specific fields. This, in turn, enables users to do a number of things with greater ease, such as create ad hoc reports without having to write programs to extract and format the data. Even today, relational databases based on SQL are the preferred form of data storage for end-user reporting and form the basis for *data warehouses* – databases used specifically for reporting.

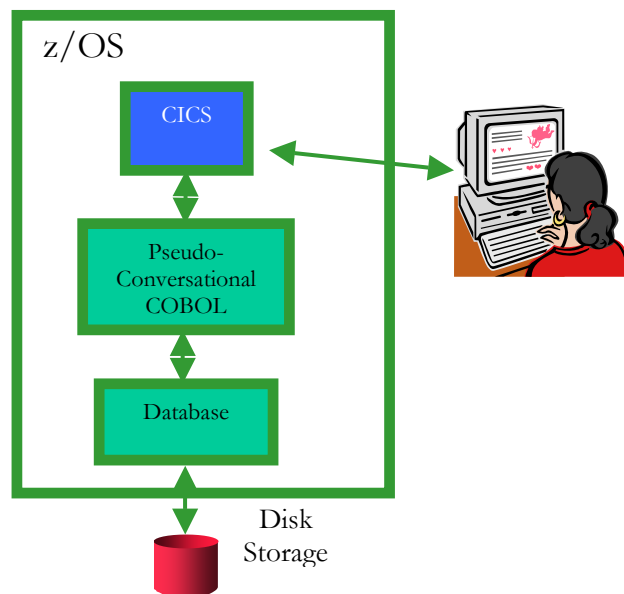


Figure 9—Typical 1970s–1980s z/OS Environment

Figure 9 shows what z/OS looked like by the 1970s–1980s. It was an environment created over time, in which users interacted via character based non-graphical terminals with a teleprocessing monitor (such as CICS) running pseudo-conversational programs written in a 3GL programming language such as COBOL, accessing data via a file system or database—all managed by the z/OS operating system. Getting complicated, but we aren’t done yet.

The Fourth-Generation Languages

As computers became both more powerful and less costly in the 1980s, the “computers are expensive and people are cheap” mantra began to shift to “computers are cheap and people are expensive.” Demand increased for environments that were easier and more intuitive to use than CICS or IMS/DC, and which therefore cost less to develop and maintain applications for them—even if they used more computing power.

This demand resulted in the introduction of *fourth-generation (4GL)* languages.

```
Read Customer starting from "A" sorted by City
print customer-name, address
at break of city
    print "number of customers in" old(city)
    "equals" count(city) //
at top of page
    print "City Report" //
End-read
```

Figure 10—Example of Code Segment Written in a 4GL (NATURAL)

Fourth-generation languages are more abstract than third-generation languages, and their syntax is even more like natural English. In addition, they usually allow programmers to write programs that interact with the user via a conversational—and thus more intuitive—programming style, making them easier for programmers to use thus reducing development and maintenance costs.

To make programming easier, fourth-generation languages also have their own environment that hides much of the z/OS environment away from the application. This 4GL environment usually runs on within the existing teleprocessing monitor, as shown in Figure 11. This is because even with fourth-generation environments, the only way to access terminals is through the teleprocessing monitor

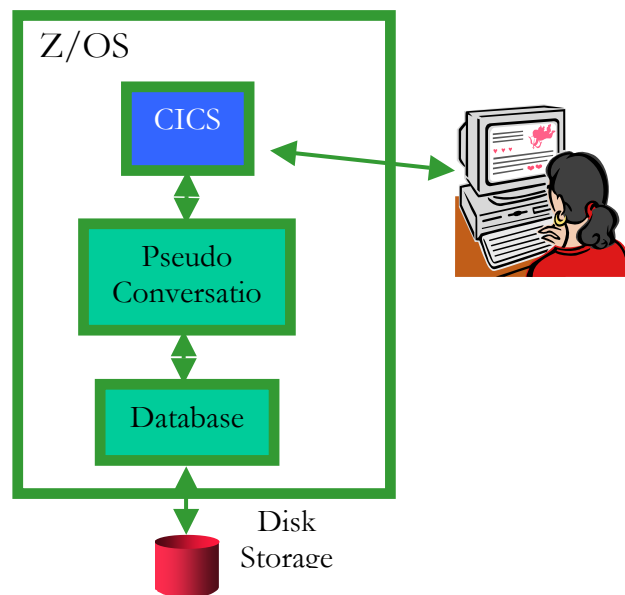


Figure 11—The 4GL Environment Incorporated into CICS and z/OS

SO WHAT'S WRONG WITH THIS PICTURE?

So why are legacy systems built on the IBM z/OS operating system getting so much criticism from the IT community? They're a bit complex, perhaps, but they do work.

Here's the problem: the picture shown in Figure 9, depicting the z/OS environment in the 1980s, is—with minor changes—is still the picture of the z/OS environment today. What caused the evolution of the z/OS environment to nearly stop?

The answer to this lies in the introduction of midrange computers and the now ever-present PC. These computers, which became prevalent starting in the 1980s, introduced a whole new paradigm of computing. All the “new” technology developments after 1990 have been largely in the environments that dominate these new computers, such as Linux/UNIX and Windows. Although hardware has become faster and faster in all levels of computing (mainframe, midrange, and PC), the numerous developments in programming languages and programming architectures since 1990—including graphical interfaces, object orientation, and new modeling environments such as the *Unified Modeling Language (UML)*—have been implemented primarily within newer operating environments such as Linux/UNIX and Windows.

As a result, the z/OS environment has become stagnant (a fact recognized even by IBM, which—in addition to z/OS—now offers a Linux environment called z/Linux that runs on its traditional hardware). Consequently, organizations are finding that to make maximum use of newer software technologies—and thus derive the greatest cost savings, most improved agility, and lowest reliance on old skill sets—they have to move their existing applications over to the newer environments.

The problems with legacy z/OS mainframes provide a good illustration of the four main drivers for modernization:

- 1) The need to reduce total cost of ownership. The z/OS environment is often expensive to maintain. In addition, many organizations already have a more modern environment such as Linux, and maintaining both z/OS and Linux environments just adds cost and complexity.
- 2) The need to restore agility into business. Legacy applications are complex and difficult to maintain. But today's emerging markets attract competition on a continual basis, and business owners cannot wait for weeks or months for new enhancements to be introduced into applications and deployed into production.
- 3) The need to reduce reliance on legacy skill sets. The skills required to maintain legacy environments are becoming harder to find, and therefore also increase the total cost of ownership. Universities don't train students in these legacy technologies and languages.

- 4) The need to meet compliance requirements. Compliance regulations are increasing the need to track processes and workflow within computerized applications. Many legacy applications don't track processes because regulations requiring such tracking didn't exist when they were written.

(For more information on these business drivers, see also the Oracle IT Modernization Series White Paper [“Why Modernize?”](#).)

ENTER MODERNIZATION

To solve the problem of moving to new technology, organizations could simply throw away their current applications and redevelop them in the newer technology environments. However, this is typically highly costly and impractical, in part because applications that have been developed over years on z/OS contain tremendous amounts of intellectual property that in some cases are not even completely understood outside of the application. The legacy applications may be cumbersome and costly to maintain, but they often work—and it is critical in adopting any new environment that the existing functionality be preserved.

To accomplish this, an organization needs to turn to modernization—the transformation of legacy applications to new technology concepts based on platforms such as Linux/UNIX and architectures such as the Oracle Modernization Framework. [For more information on the Oracle Modernization Framework, see also the Oracle Modernization Series White Paper [“The Oracle Modernization Framework \(OMF\)”](#).] With modernization, the existing application is used as the starting point for the modernization process.

Fortunately, various modernization techniques can be used to move applications from legacy environments into open systems while preserving portions of the original investments. Thus, organizations can move away from these burning platforms in a controlled and structured manner to increase quality and reduce risk.

For more information on these processes and how they can be applied, see also the Oracle IT Modernization Series white paper [“The Types of Modernization.”](#)

CONCLUSION

z/OS mainframes were once the best possible computing platform for critical business applications in the enterprise. These systems leveraged their high availability and centralization to become the de facto platform for serious business computing. IBM invested vast amounts of research and development to create a robust and secure platform that could service thousands of transactions per second.

However, the z/OS environment has been added to repeatedly over time by both IBM and other vendors, resulting in an environment for which development, maintenance, and production processes are no longer as effective and efficient as they could be. In the past 20 years, most of the improvements in application architecture that provide lower cost, greater agility, and less reliance on legacy skill

sets have not occurred in the z/OS environment, but in more modern environments such as Linux/UNIX.

As a result, legacy environments such as z/OS mainframes are now viewed by many as a major obstacle for organizations that are striving to stay relevant in a highly competitive world.

Since developing new applications is highly risky, the best approach for organizations looking to preserve the content of their current applications while taking advantage of new technology environments is to modernize them.



Oracle IT Modernization Series: z/OS Mainframe for Dummies

January 2007

Author: Ted Venemea

Contributing Author: Lance Knowlton

Contact: modernization_ww@oracle.com

For more information: www.oracle.com/goto/modernization

Oracle Corporation

World Headquarters

500 Oracle Parkway

Redwood Shores, CA 94065

U.S.A.

Worldwide Inquiries:

Phone: +1.650.506.7000

Fax: +1.650.506.7200

oracle.com

Copyright © 2007, Oracle. All rights reserved.

This document is provided for information purposes only and the contents hereof are subject to change without notice.

This document is not warranted to be error-free, nor subject to any other warranties or conditions, whether expressed orally or implied in law, including implied warranties and conditions of merchantability or fitness for a particular purpose. We specifically disclaim any liability with respect to this document and no contractual obligations are formed either directly or indirectly by this document. This document may not be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose, without our prior written permission. Oracle, JD Edwards, PeopleSoft, and Siebel are registered trademarks of Oracle Corporation and/or its affiliates. Other names may be trademarks of their respective owners.