TOWARDS PREVENTING SOFTWARE FROM BECOMING LEGACY: A ROAD MAP

Dillip Kumar Mahapatra\textsuperscript{1}, Tanmaya Kumar Das\textsuperscript{2}

\textsuperscript{1}(Head of the Department of Information Technology, Krupajal Engineering College/Biju Patnaik University of Technology, Rourkela, Odisha, India)
\textsuperscript{2}(Assistant Professor, Department of Computer Science & Engineering, C.V Raman College of Engineering, Bhubaneswar, /Biju Patnaik University of Technology, Rourkela, Odisha, India, Odisha, India)

ABSTRACT

Today’s organizations are faced with many challenges in their business environments. Legacy, outdated information technology (IT) systems and changing business processes are among these major challenges as companies address legacy system inflexibility, agility, lack of scalability, lack of wider data access, shortage of skills, high cost of maintenance and unreliability. Couple this with continually changing technologies, and organizations are faced with the need to assess these new technologies and adapt their infrastructures and applications to leverage those technologies as well. However legacy systems are typically the backbone of an organisation and are thus mission critical and their failure can have a serious impact on business. In fact, software is the system that significantly resists modification and evolution.

This papers highlights some major issues and focuses on few steps to prevent software products becoming legacy.

KEYWORDS: Legacy Software, Migration Planning Process, Migration issues, Legacy Application system, Database Centred system , Automated Migration tools.

I. INTRODUCTION

In information technology, legacy applications and data are those that have been inherited from languages, platforms, and techniques earlier than current technology. Most enterprises that use computers have legacy applications and databases that serve critical business needs. Typically, the challenge is to keep the legacy application running while converting it to newer, more efficient code that makes use of new technology and programmer skills. In the past, much programming has been
written for specific manufacturers' operating systems. Currently, many companies are migrating their legacy applications to new programming languages and operating systems that follow open or standard programming interfaces. Theoretically, this will make it easier in the future to update applications without having to rewrite them entirely and will allow a company to use its applications on any manufacturer's operating system (Ref. 02).

So in essence “Legacy software can be characterized informally as old software that is still performing a useful job”. Legacy software systems are programs that are still well used by the business community or have some potential inherent value, but, that were often developed years ago, using early versions of programming languages. Often these programs have been maintained and developed for many years by many hundreds of programmers, and whilst many changes have been made to the software, the supporting documentation may not be current, and the programming style archaic. These factors contribute to the staggering cost of maintaining these legacy programs or continuing their development. Consequently, there is an urgent need to find ways to make these programs more maintainable without disrupting their current use and to “safeguard the information they contain.” Making the programs faster, more reliable, and easier (and cheaper) to maintain.

This implies that it is the preferred solution to discard the software completely, and start again with a new system. Whilst this may be true for some business needs, this may not be appropriate in all cases, for example after:

- The software represents years of accumulated workforce experience, which is not represented elsewhere, so discarding the software will also discard this knowledge, however inconveniently this is represented.
- The software may actually work very well, and its behavior may be well understood by the workforce. A new replacement system may perform much worse, (at least in the early days,) be expensive, and, there are retraining and upgrading budgets to think of, hence, it may be worth recovering some of the “good” features of the legacy system, and even adding some better ones.
- It may not be acceptable to demand that users undertake a substantial upgrade for no discernible benefit. Therefore, it may be important to retain the interfaces and exact functionality of the legacy code.

In addition to moving to new languages, enterprises are redistributing the locations of applications and data. In general, legacy applications have to continue to run on the platforms they were developed for. Typically, new development environments account for the need to continue to support legacy applications and data. With many new tools, legacy databases can be accessed by newer programs.

II. WHY LEGACY?

Companies spend a lot of money on software systems and, to get a return on that investment, the software must be usable for a number of years. The lifetime of software systems is very variable but many large systems remain in use for more than 10 years. Some organisations still rely on software systems that are more than 20 years old. Many of these old systems are still business-critical. That is, the business relies on the services provided by the software and any failure of these services would have a serious effect on the day-to-day running of the business. These old systems have been given the name legacy systems. (Ref. 04)

These legacy systems are not, of course, the systems that were originally delivered. External and internal factors, such as the state of the national and international economies, changing markets, changing laws, management changes and structural reorganisation, mean that businesses undergo continual change. These changes generate new or modified software requirements so all useful software systems inevitably change as the business changes. Therefore, legacy systems incorporate a
large number of changes which have been made over many years. Many different people have been involved in making these changes and it is unusual for any one person to have a complete understanding of the system.

Fig.1: Legacy System Components

Businesses regularly replace their equipment and machinery with modern systems. However, scrapping legacy systems and replacing them with more modern software involves significant business risk. Most managers try to minimise risks and therefore do not want to face the uncertainties of new software systems. Replacing a legacy system is a risky business strategy for a number of reasons:

1. There is rarely a complete specification of the legacy system. The original specification may have been lost. If a specification exists, it is unlikely that it incorporates details of all of the system changes that have been made. Therefore, there is no straightforward way of specifying a new system which is functionally identical to the system that is in use.

2. Business processes and the ways in which legacy systems operate are often inextricably intertwined. These processes have been designed to take advantage of the software services and to avoid its weaknesses. If the system is replaced, these processes will also have to change with potentially unpredictable costs and consequences.

3. Important business rules may be embedded in the software and may not be documented elsewhere. A business rule is a constraint which applies to some business function and breaking that constraint can have unpredictable consequences for the business. For example, an insurance company may have embedded its rules for assessing the risk of a policy application in its software. If these rules are not maintained, the company may accept high-risk policies which will result in expensive future claims.

4. New software development is itself risky so that there may be unexpected problems with a new system. It may not be delivered on time and for the price expected. Keeping legacy systems in use avoids the risks of replacement but making changes to existing software usually becomes more expensive as systems get older. Legacy software systems which are more than a few years old are particularly expensive to change for several reasons:
   - Different parts of the system have been implemented by different teams. There is, therefore, no consistent programming style across the whole system.
   - Part or all of the system may be implemented using an obsolete programming language. It may be difficult to find staff that has knowledge of these languages and expensive outsourcing of system maintenance may be required.
   - System documentation is often inadequate and out-of-date. In some cases, the only documentation is the system source code. Sometimes the source code has been lost and only the executable version of the system is available.
Many years of maintenance have usually corrupted the system structure making it increasingly difficult to understand. New programs may have been added and interfaced with other parts of the system in an ad hoc way.

The system may have been optimised for space utilisation or execution speed rather than written for understandability. This causes particular difficulties for programmers who have learned modern software engineering techniques and who have not been exposed to the programming tricks that have been used.

The data processed by the system may be maintained in different files which have incompatible structures. There may be data duplication and the data itself may be out of date, inaccurate and incomplete.

III. LEGACY SYSTEM STRUCTURE

Legacy systems are not simply old software systems although the software components of these systems are the main focus of this chapter. Legacy systems are socio-technical computer-based systems, so they include software, hardware, data and business processes. (Ref.06) Changes to one part of the system inevitably involve further changes to other components. Decisions about these systems are not always governed by objective engineering criteria but are affected by broader organizational strategies and politics. Figure: 1 illustrates the different logical parts of a legacy system and their relationships:

1. System hardware In many cases, legacy systems have been written for mainframe hardware which is no longer available, which is expensive to maintain and which may not be compatible with current organizational IT purchasing policies.

2. Support software The legacy system may rely on a range of different support software from the operating system and utilities provided by the hardware manufacturer through to the compilers used for system development. Again, these may be obsolete and no longer supported by their original providers.

3. Application software As I discuss later, the application system which provides the business services is usually composed of a number of separate programs which have been developed at different times. Sometimes the term legacy system means this application software system rather than the entire system.

4. Application data These are the data which are processed by the application system. In many legacy systems, an immense volume of data has accumulated over the lifetime of the system. This data may be inconsistent and may be duplicated in different files.

5. Business processes These are processes which are used in the business to achieve some business objective. An example of a business process in an insurance company would be issuing an insurance policy; in a manufacturing company, a business process would be accepting an order for products and setting up the associated manufacturing process.

6. Business policies and rules These are definitions of how the business should be carried out and constraints on the business. Use of the legacy application system may be embedded in these policies and rules.

An alternative way of looking at these different components of a legacy system is as series of layers as shown in Figure 2. Each layer depends on the layer immediately below it and interfaces with that layer. If interfaces are maintained, then it should be possible to make changes within a layer without affecting either of the adjacent layers.
In practice, however, this simple encapsulation rarely works and changes to one layer of the system may require consequent changes to layers that are both above and below the changed level. The reasons for this are:

1. Changing one layer in the system may introduce new facilities and higher layers in the system may then be changed to take advantage of these facilities. For example, a new database introduced at the support software layer may include facilities to access the data through a web browser and business processes may be modified to take advantage of this facility.

2. Changing the software in the system may slow it down so that new hardware is needed to improve the system performance. The increase in performance from the new hardware may then mean that further software changes which were previously impractical become possible.

3. It is often impossible to maintain hardware interfaces especially if a radical change to a new type of hardware is proposed. For example, if a company moves from mainframe hardware to client-server systems these usually have different operating systems. Major changes to the application software may therefore be required.

The application software in a legacy system is not a single application program but usually includes a number of different programs. The system may have started as a single program processing one or two data files but, over time, changes may have been implemented by adding new programs which share the data and which communicate with other programs in the system. Similarly, the initial system data files are added to as new information is required. This is illustrated in Figure 3(a). Different programs share data files so that changes to one program that affect data inevitably result in changes to other programs. (Ref.03)

![Socio-Technical System](image)

**Fig.2:** Socio-Technical System

The different programs in the legacy application system have usually been written by different people and are often written in different programming languages or in different versions of a programming language. For example, the original software may have been developed in COBOL-72 but later programs implemented in a new version of the language, COBOL-80. Compilers and support software for all of these languages may have to be maintained. This all adds to the complexity of the system and increases the costs of making changes to it. While there are still legacy
systems that use separate files to maintain their data, a large number of business systems have centralized their data management around a database system (Figure 3(b)). The advantage of adopting this structure is that the data in the system is described using logical and physical data models. Redundancy and data duplication is less likely and it is easier to assess the impact of system changes which affect the system data. Databases also provide transaction processing capabilities where changes to the data can be made in a recoverable way. This allows interactive updates of the data to be made. There are two major legacy issues in database-centred systems:

1. The database management system which is used may be obsolete and incompatible with other DBMSs used by a business. Relational database management systems are now the most effective database management systems for business applications. However, many legacy systems rely on older database systems that are based on hierarchical and network models. These systems were designed to allow the performance of the system to be optimized rather than for simple data management. Modern hardware may make this performance optimization unnecessary but the costs of changing to a relational data model are very high.

2. The application which is used may have been designed for use with a particular database system and for mainframe hardware. Therefore, it may not be possible to use the same application with a new database. This part of the system may also have to be replaced and this increases the costs and risks of system change.

IV. WHY MIGRATION?

Legacy systems are successful and therefore mature, and likely have been in existence for a long period of time. A consequence is that legacy software is built using technologies available at the time it was constructed, as opposed to the most modern software technologies. Older technologies are more difficult to maintain, and this is a key point of pain for many legacy system owners. Often, an application may need restructuring to meet other business goals through a process so called Software Modernization or Software Migration from older requirements to new requirements. (Ref.05)

![Fig.4: Operational Activities on Legacy System](image)

There are many reasons that a migration of a legacy system may be needed:

- *Legacy languages are hard to support.* The legacy languages and development tools needed to support the legacy system are increasingly difficult or expensive to obtain. This is a very common occurrence with 4GLs popular in the 1970s.

- *People are scarce.* People that know the legacy languages are becoming difficult to find and retain. Younger staff are reluctant to learn "legacy" languages because it does not appear to advance their long-term career.

- *The underlying platform is hard to support.* Many legacy systems run on legacy hardware systems. Such hardware systems are becoming more expensive to maintain, and personnel that know these systems are also more difficult to find.
• **Legacy software does not integrate well with other IT systems.** The architecture of legacy languages often does not lend itself to building bridges to other IT systems that have grown up around it.

• **Outdated development methods used:** As the technology is changing day-by-day, the development methods which were used previously are obsolete.

• **Original developers not available:** The programmers or developers might have left the organization, which results in difficulties to manage/maintain the existing system.

• **Extensive patches and modifications have been made:** A lot of changes and modifications are made to the software system and can’t be modified further.

• **Missing or outdated documentation:** The documentation for the software might be outdated.

V. **MIGRATION ISSUES**

Legacy System migration essentially moves an existing, operational system to a new platform, retaining the legacy system’s functionality and causing as little disruption to the existing operational and business environment as possible. This is a significant challenge, and it could quite legitimately encompass numerous areas of software engineering, including program and database understanding, system development, and testing. (Ref.05)

Figure 6 shows important practical issues in migration, divided roughly according to those related to the legacy system and those related to the target system. Some migration issues are common to all software engineering projects and are widely researched and supported. These include target system development, testing, and database model selection. Other issues are specific to migration and have yet to be extensively researched. These include target system database population and cut-over with mission critical support.

Because a legacy system already meets some of the business and user requirements demanded of the target system, it is important to understand its operations and interactions. Poor system understanding can lead to incorrect target-system requirement specifications and ultimately to failed migration projects. Thus, to begin, engineers should have a good understanding of the legacy system data, interfaces, and applications that require tool support. Although some support is available, engineers may have to develop specialized tools to fit their legacy and target systems. They might also classify their legacy system by type and properties, and develop appropriate migration guidelines.

![Fig.6: Classification of Migration Issues](image-url)
5.1 Database population

To populate the target database with legacy system data, engineers first map the legacy system schema onto the target schema and work out the required transformation. Data must also be mapped at instance level before migration. Data can also be migrated in separate steps, by dividing it into independent fragments. If legacy system data is of poor quality, data cleaning might be required. If so, decisions must be made about which method to use and when to use it: before, during, or after migration.

5.2 Testing and functionality issues

Up to 80 percent of a migration engineer’s time can quite legitimately be spent testing the target system, which is an ongoing process during migration. Given the legacy system’s mission-critical nature, target system outputs must be completely consistent with those of the legacy system. Thus, it is inadvisable to introduce new functionality to the target system during the migration project. When functionality is the same, engineers can directly compare outputs to determine the target system’s validity. However, on a practical level, migration projects are often expected to add functionality to justify the project’s expense and risk. In this case, the legacy system should be migrated first. New functionality can be incorporated into the target system after the initial migration.

VI. WHAT CAN BE MIGRATED?

Migration can be instituted across a range of technology classes, including the following:

• Language or code migrations
• Operating system migrations
• Data migrations
• User interface (UI) migrations
• Architecture migrations, including migration to object-oriented programming (OOP)

As a result, enterprises can perform migrations whenever a greatly improved infrastructure is desired, including programming languages, operating systems, data, architecture, or any combination of these.

VII. CHOOSING MIGRATION STRATEGIES

The next step is to consider the strategies available for achieving migration. We will take a look at complete migration, iterative migration, limited migration, vertical migration, and horizontal migration. Organizations determine which of these strategies to use based on factors such as system qualities, manageability, training, and cost. (Ref.01)

7.1 Complete migration

In complete migration, all of the components in the application are migrated as a whole. This strategy does not allow for intermediate validation of the migration process or the business rules. The only way to know if the application is suitable for the business requirements is to evaluate it once the migration is completed. Complete migration requires significant effort and is potentially expensive and risky. This strategy does enable immediate enhancement with new and added functionality. For example,

VB 6.0 application functionality is improved when migrated to the .NET environment. Directly integrated into their new .NET environment, these applications gain new versatility, and if deployed on new hardware or new Windows operating systems, will not require further migration or
recompilation. Complete migration is an expensive strategy, but is also the most desirable strategy to adopt since it provides application adaptability to future needs. However, a comprehensive assessment of the application is necessary prior to opting for complete migration; this is discussed in section 4 of this paper.

7.2 Iterative migration

An iterative migration strategy allows for a more controlled migration process since the application is migrated component-by-component, with each newly migrated component rolled out as a phase. This strategy is only feasible if the existing application is composed of distinct components. Interoperability techniques are key, as the migrated and un-migrated components must function together. Iterative migrations provide an acceptable alternative to a complete migration, and are often the option for a large-scale legacy application migration. With this strategy, there is greater control over the cost and progress of the entire migration project. Each phased rollout, or iteration, minimizes risk since the application is returned to a stable production-quality state. Iterative migration also allows the flexibility to migrate only certain portions of the application that have immediate relevance to the business. Improved performance and scalability are immediately realized in the migrated components.

7.3 Limited migration

Limited migration is different from the iterative migration process in that only a component of the application is migrated. The migrated portion is then modified to interoperate with the unmigrated part of the application. Interoperability is the key issue in this type of migration. A business may not need to migrate the entire application, so this type of migration allows an organization to port only the components of the application those are actually required.

7.4 Vertical migration

Vertical migration differs from the other migrations in that the process is performed tier-by-tier. Vertical migration involves isolating and replacing a portion of an application through all n-tiers. The developer determines which component of an application has the least interaction with the other components and performs the migration. The migration is then completed on all tiers for a particular module prior to proceeding to the next module. This strategy is advantageous when portions of an application are well isolated from other portions of the same application. In these instances, the isolated components share little state information with the rest of the application and can undergo easy migration with minimal impact on the rest of the system. Vertical migration is also an effective option when ActiveX Data Objects (ADO) record sets are used between tiers. Many applications pass disconnected ADO record sets from the data and business tiers to the presentation tier. They then iterate through the record sets and generate HTML tables. This type of application is well suited to a vertical migration because migrating vertically minimizes the work involved in achieving interoperability with ADO.

7.5 Horizontal migration

Horizontal migration involves replacing an entire tier of an application without immediately migrating the other tiers. For example, a developer may choose to initially replace the ASP code within a Web-based presentation tier or replace the COM code within the middle tier as the initial migration step. For a business migrating to the .NET environment, the migration is performed a single tier at a time, taking advantage of the features of the .NET framework specific to a particular tier. In this instance, no application code is modified and no operations are affected on another
application tier. Horizontal migration is beneficial for infrastructures containing large numbers of servers, large amounts of shared code, heavy ASP application or session.

VIII. SIX STEP PROCESS FOR MIGRATION

The actual migration process is divided into six distinct phases: application assessment, application preparation, application migration, post-migration changes, and application testing (Ref. 07). However, the most important point to start with any migration process is migration planning and is shown in fig. 7.

Step 1: Assess the Application to be migrated

Once a business has determined that a migration is a viable solution, the first step is to ascertain which current applications continue to fulfill current business needs. Companies that omit this assessment process delay the inevitable need to retain or eliminate certain applications. This prolongs the duration of a project and reduces many of the key benefits of migration, such as improved and more efficient business processes. In determining the feasibility of current applications, businesses need to examine several decision drivers. These drivers include project priorities and goals, application business value, development environment and resource skills, application complexity and architecture, and quality assurance. To determine the application’s value to business, the following queries need to be answered,

- What functionality does the application possess that other applications or third-party tools cannot reproduce?
- What types of data and data transmission protocols does the application support?
- What are the application’s basic input and output types, different interface points, and external dependencies?
- Does the application handle legacy file formats or high value business transactions?
- How would removal of the application impact the organization?
- What is the current TCO for the application? Would TCO improve if the application was ported to the new environment?

The next step in the assessment establishes the application’s code quality in terms of design and source code. This step helps the migration team understand the code complexity and, in turn, helps determine the cost, effort, and schedule for migration. In addition to assessing code quality, other metrics for this stage of the assessment include examining the development environment and developer skill sets. To determine the application’s code quality, the migration team establishes the application’s size, usage, complexity, dependencies, and overall stability. The following are important queries to consider:

- What is the application’s size? How many lines of code, forms, user controls, and modules, classes, and data source types exist?
What are the application’s functions, properties, and types?
How complex is the application? What application features are not supported, resulting in a potential manual migration?
Does the application depend on other tools or use an internal mapping that generates internal functional dependencies?
Is the application currently undergoing enhancements or code changes?

The final parameters in the application assessment examine the development team's skills and the development environment. This helps determine if the migration adheres to estimates in terms of effort and schedule. Familiarity with the code base and new environment are essential to performing a successful migration. In the event the migration team is unfamiliar with the application, the extra time that would be required to gain familiarity with the code base needs to be factored in during estimation. Any lack of code level knowledge, insufficient documentation, and lack of development skills in the new environment increase the risks involved in a migration.

**Step 2: Prepare the Application to be migrated**
Once the assessment is completed, the second step prepares the application for migration. Three main initial conditions are required before migration commences:
- Provide all the relevant application documents and baselined source code to the migration team
- Supply functional experts to the migration team for accurate understanding of the project
- Provide application source code that has not undergone separate enhancements as the migration begins

With these conditions met, the application preparation phase begins. The entire application is rebuilt in the parent environment from the given source code. The migration team then executes it and runs the application test cases to determine that the application source code provided is the correct version for migration. With the correct source code version established, developers proceed to the migration.

**Step 3: Migrate the Application**
In the actual migration process, the prepared application is migrated to the new environment using migration tools developed specifically for this purpose. Migration tools provide many benefits, among them supplying the migration team with the ability to:
- Consider resource-consuming elements, constructs, and features
- Identify incompatible porting issues in the application’s code, build, and production environment
- Remove dead code and obtain recommendations for improved coding style
- Analyze application components and component relationships
- Shorten migration timeframes by eliminating manual rewriting of unsupported code
- Simplify the migration process via migration wizards

With the use of a migration tool, an upgrade report is generated which will identify what application features are not upgraded automatically; these result in the need for manual migrations. This is accomplished in the next phase of the migration process.

**Step 4: Perform the Post-migration changes**
Some applications simply cannot be migrated automatically and may require significant manual work. Based on the report findings from the prior step and the desired functionality in the new environment, the developer will need to change the code in the new environment. The objective is to write code for the new platform to obtain the same functionality in the migrated code as found in the original application.
Common situations that necessitate manual work are:

- Applications unable to update to their most current version prior to migration (e.g., VB 5.0 or earlier applications upgraded to VB 6.0 before migration to VB.NET)
- Certain features not upgraded automatically (e.g., DDE, OLE, DAO, and RDO data bindings in VB 6.0)
- Distributed n-tier applications with several layers of objects communicating through COM
- Web applications using DHTML, Web Classes, or ActiveX documents
- Projects using ActiveX controls or ActiveX DLLs

**Step 5: Test the Application**

In this final phase, the newly migrated application is subjected to rigorous testing using the same test cases applied earlier when validating the source code provided. Apart from functional testing, stress, volume, and load tests are carried out to ensure scalability and performance levels are achieved. Fine tuning and/or optimization is conducted after each round of testing to achieve the desired performance levels. Following successful testing, the application is released for production. Alternatively, further enhancements are executed based on the business needs for the application.

**Step 6: Post-migration Support**

Once the migrated application is deployed, additional business and user needs may be identified, requiring technical team, developer, and/or system support intervention. Issues may involve system configuration or optimization, or the application configuration parameters may require fine tuning. Ensuring close attention to this final phase and leveraging the knowledge capital of migration team members during this phase reduces risks during the field testing. It is an essential step to ensure mission critical application run smoothly.

**IX. CHOOSING AUTOMATED MIGRATION**

Legacy systems were built by manual methods, and it is what the IT organizations know how to do. So it is natural that a migration by manual methods is almost always proposed. These usually don't turn out well, for several reasons.

- **Cost of doing a manual migration**: The cost for manual code conversion is very and an assumption is that the migration is going between two relatively similar languages (e.g., not trying to go from COBOL to full object-oriented Java).
- **Time to do a manual migration**: Again, a legacy system with a million lines of code requires more man-years of labor to convert. With larger systems, one has larger teams. Larger teams require more interactions, slowing them down further. A 10 million line application simply doesn't have any practical manual migration due to time frames.
- **Scope creep**: Because manual migrations are long and expensive, it is very difficult to resist adding new functionality and requirements during the project. The migration task gets longer and riskier, and it is much more difficult to test the result for completeness, because it no longer does what the existing legacy system does.
- **Conflicting goals**: While the migration team is attempting to build a replacement, the legacy system must still continue to support the company. It must evolve to meet corporate needs, and so changes are continually made to it. Such changes are trouble for the migration team, which now has to go back and rework code already converted. This creates pressure from the migration team to stop legacy system evolution, which is not good for the company.
• **Inability to change course meaningfully:** No plan survives intact. During the migration, the company goals will change, the migration team will learn how to handle parts of the conversion better, and migration mistakes will be made and uncovered. The problem is that hand-converted code contains the assumptions that were valid, and the mistakes made, at the time of the hand-conversion, and it is painful to go back and change these modules. So, the migrated system either does not take in account new directions or better understanding of the task, or it takes longer to do at higher risk.

• **Uneven conversion quality:** A migration manually depends on the individual skills of the team members. Some are naturally better or worse than others. The resulting code quality will consequently vary, raising maintenance costs for the converted system.

A better approach is automated conversion. One needs strong automation to meet economic and time frame objectives simply choosing appropriate software tools.

X. **GUIDELINES FOR MIGRATION PLANNING PROCESS**

It has been identified for migrating legacy system for a new system, why reengineering projects fail even by utilizing sincere migration efforts. We derived few guidelines that need to apply during the migration planning process.

These guidelines include:

1. Analyze the needs of the affected stakeholders to determine migration schedules, training requirements, and operational cutover to the new system.
2. Develop quantifiable measures of success for the migration effort.
3. Initiate the migration planning effort at the outset of the project before the development and implementation approach is set in concrete. Clearly define lines of communication and authority and provide adequate resources. Do not treat the effort as an added task for assigned team members.
4. Involve customers and users in the migration planning effort.
5. Do not allow system implementation to begin until a migration plan is approved and the “buy-in” of the affected stakeholders is obtained.
6. Avoid a “big-bang” approach to migration. Break the problem into “doable chunks” that correspond to the planned rollout of new system solutions:
   - Make certain the migration effort is appropriately scaled and commensurate with the organization’s resources, skills, and current workload.
   - Establish priorities for migrating users of the existing legacy systems.
   - Give consideration to any organizational infrastructure improvements that can accelerate the migration effort or may be needed to overcome impediments in the current working environment.
   - Establish criteria to evaluate the level of difficulty of transitioning the user community corresponding to each legacy system.
   - Consider addressing high-risk migration issues first since their solution may have the greatest impact on the development effort and may determine the feasibility of migrating users of every legacy system.
   - Identify meaningful and measurable milestones to track progress.
   - Survey the user community as early as possible to obtain their insight into what they believe are the greatest impediments to moving to a modernized system.
   - Ensure that the scope of migration planning includes deployment, transition to full operational use, and phase out of any affected legacy systems.
7. Actively manage and monitor the migration effort:
   • Establish a goal-driven measurement program based on stakeholder needs to obtain visibility into the migration effort.
   • Schedule periodic reviews commensurate with the risks involved, organizational policies and regulations, vested interests of key stakeholders, and assurances sought by the sponsor and project manager.
   • Establish a tracking system to manage progress, problems, issues, and other action items that pertain to migration planning and execution.

XI. CHECKLIST FOR MIGRATION

The migration best practice checklist below provides an at-a-glance overview of the steps required to launch and drive a migration process.

   • Establish costs and benefits of the migration in advance
   • Evaluate current resources
   • Define the scope of the migration
   • Start with simple projects initially
   • Use applications best suited for the current operating environment “as is”
   • Understand how the application is going to be modified
   • Analyze the current application
   • Ensure a complete understanding of the migration tool that will be used
   • Prepare the code being migrated prior to the actual migration
   • Upgrade module-by-module
   • Test each module as it is being upgraded before continuing migration
   • Review the upgrade report generated by the migration tool
   • Use stored procedures as much as possible
   • Use tools like source code analyzers and compatibility test tools to identity issues in advance
   • Define technical changes needed due to migration
   • Write new test cases for the migrated applications to gauge existing functionality along with performance and scalability tests.

XII. CONCLUSION

Legacy systems are still alive because of their distinct characteristics and good pedigree. In the last 40 years, we have learned that it is neither practical nor affordable to migrate legacy codes into other technologies which were short lived. However, it is possible to either eliminate or integrate the legacy systems by effective migration strategy and appropriate migration tools. Current business and IT challenges—outdated IT systems and changing business processes—require proactive resolution. By empowering the rejuvenation of existing business systems and application uses, migration offers opportunities that both current and future technologies provide. Companies that carefully embrace and incorporate the six strategic steps outlined in this paper leverage the power of migration and drive the changes that equate to business success, now and in the future.

REFERENCES


