Working Model for the Modernization of Legacy Systems

Erik Philippus
Improvemen**T** BV

Step 1: Free Resources

Most organizations facing severe legacy problems are trapped in a downward spiral of spending an increasing amount of resources on maintaining the obsolete software system, including continuous bug-fixing, preparation of hot fixes for urgent problems in the field, updating the old system with new features in a non-economical way, etc. Sooner or later the organization will find itself in sheer survival mode, spending nearly all the available development budget on corrective maintenance instead of innovation.

In this situation, the readiness to invest in structural modernization of the legacy system will be almost zero, even when there is a sound business plan that clearly demonstrate the business profit of modernization of the broken-down software system. Apart from the psychological and organizational aspects, there are just no resources available to do the necessary work. Not only lack of time and money, but predominantly personnel with key domain knowledge. Critical knowledge and experience are typically not explicitly available, and mostly in the heads of a few people – which are vital member of the 'fire brigade', aiming to keep systems running at client sites at all cost.

Hence, a prerequisite to create 'space' for dealing with the legacy problem in a structural way, is to free resources (time, money and/or people) in a direct and concrete way. In most cases, it will require an external intervention to interrupt the ongoing pattern. Measures to free resources can take several forms, e.g. offshoring maintenance and/or testing activities, removal of directly visible inefficiencies or waste, etc.

In conclusion, the major goal of this step is to free resources needed to make the next step, Depending on the magnitude and impact of the legacy problem, this phase can consume a considerable amount of persuasive power and time.

Step 2: Map the Problem

For most companies, the business value of the software system is only known roughly. When software is 'overhead', the cost & benefits of a particular software module are not explicitly known. This implies that the cost of maintaining a particular legacy module cannot be related directly to business benefits. This situation makes it difficult to prioritize maintenance activities, and to focus on aspects of the legacy system that are the most relevant from a business perspective. Hence, it is possible that a whole development team is spending a lot of effort on obsolete software modules which hardly contribute to the overall profit, and therefore could be sustained or even phased out without much risk. No concrete figures exist to underline implicit trade-off decisions.

Furthermore, the hectic situation often prevent an organization to do proper market research, to spot trends and unique selling points. This will have a negative influence on the quality of the product road map, and at the end could jeopardize the market position and image of the organization. This is a hidden side-effect of the legacy problem. Therefore, additional (fresh) market research is sometimes needed to identify the critical assets in the legacy software system, in order to make proper trade-off decisions. Spending effort on the maintenance or renovation of features which have no clear business relevance, is pure waste.

A legacy system is a system which lacks quality. For instance, if no specific measures have been taken to guarantee the extendability, the innovation speed of the system can drop significantly over time. Also other quality flaws can contribute considerabely to the legacy situation. Hence, it is essential to have an overview of the current quality-level of the software, e.g. by application of relevant metrics to the source code, to perform a trend-analysis of the software archive, to have a formal inspection of the design, etc. Inspection of the history of software modules can also give a clue where to look.

Furthermore, most legacy problems are caused by more then just pure technical problems. Often, organizational an process-related issues will play an equally important role here. The human factor cannot be left out. In general, dealing with legacy requires a multilateral approach, balancing the technical, organizational and process-related factors. The maturity of the applied development proces, the quality of the development environment, the competence-level of team members, organizastional culture as reflected in management style and communication patterns, etc, - the assessment must also take these factors into account.

A pure technical assessment might bring the immediate technical problems to the surface, but there is no guarantee that when other factors remain unchanged, that the organization will not run into legacy problems in the near future again. However, the technical part is of course an important point of departure, and advanced tooling can help here to chart all relevant technical aspects of the legacy problem, and to give the assessment a sound technical foundation.

In conclusion, to map the problem, an multidimensional assessment will be needed, taking into account relevant technical, organizational and process-related aspects of the legacy situation.

Step 3: Achieve Commitment

Modernization of legacy systems require a well-considered management resolution. Without strategic management support, it is nearly impossible to hold to a decision for renovation during operation. At the first headwind, the renovation team will be trimmed down in order to have the most knowledgeable engineers working on solving legacy-related problems. The fight for resources is inevitable without proper and explicit management support for modernization. Hence, achieving commitment for a underlined strategy for modernization is a necessary step in the proces. and to smooth the (psychological) path for system renovation.

It should also be mentioned that the resistance against structural modernization has a psychological side which should not be underestimated. Bringing the root cause of legacy issues to the surface can be quite confronting for engineers and managers involved. Therefore, the issue has to be presented carefully, and at all time, *blaming* has to be prevented. Radical change always evoke resistance, and management must have not only the vision, but als the guts to take unpleasant renovation measures that not immediately contribute to the operating profit.

In any case, it is vital to have a solid business case for the renovation effort. The backbone of this business case is a migration strategy based on the results of the assessment in the previous step. Since the goal of the modernization of to yield a system with the desired quality, the business case should explicitly address the return on investment for the renovation effort. The challenge however, is to come up with a strategy that is not intrusive, which means that the renovation work doesn't hamper the ongoing projects.

The renovation activity must be carried out in parallel with the ongoing work, and it must be possible to replace obsolete parts with new code at regular time intervals without too much interference. Nobody will not be pleased when inclusion of renovated source code will give unexpected problems in another part of the system. Hence, testing is a paramount part of renovation in all occasions. The result of this phase is a clear and outspoken support from higher management for the structural modernization of the legacy system, based on a sound business case.

Step 4: Plan the Work

Necessary resources are available, and commitment from the management is achieved, so it is time to turn the intention to modernize the legacy system into a real and concrete project. The renovation activities must be carefully planned. This work is challenged by (potentially) conflicting demands. At one hand, the renovated system must lay the foundation for future systems, but at the other hand it must not jeopardize the planned product releases. When backwards comptability is a requirement, the situation is even more complex. This requires a production plan which is carefully balanced with ongoing activities. In almost all cases, a 'big-bang' migration must be prevented – which calls for the strategy of branching with periodic merging.

Thus, in order to mitigate the involved risks, it is strongly recommended to use an incremental approach as foundation for the renovation plan. Incremental software renovation is the practice of reengineering legacy parts of a software system on a phased basis, including the phased re-incorporation of the reengineered parts into production. Major benefit of incremental reengineering is less risk and better recovery from (integration) errors. If an error occurs within a reengineered component, the entire project is not put at jeopardy. A failure within a particular reengineered component is easier to pinpoint than an error somewhere within an entire system.

But what is more important, experience learns that the organizational strategic direction is always dynamic: the external and internal pressures will alter the goals and direction of the overall reengineering project. This situation requires increased flexibility, and incremental reengineering allows the organization to better redirect the reengineering efforts in response to these changes.

Once a component has been reengineered, validated, and implemented, incremental reengineering better enables the introduction of functional enhancements. Thus, maintenance need not be postponed while the system is being fully reengineered, and parallel maintenance (of both the reengineered and legacy system) is not required thus easing the burden on limited resources. As technology changes, incremental reengineering can adapt more easily while minimizing the risk of losing the resources already invested in previous reengineering efforts.

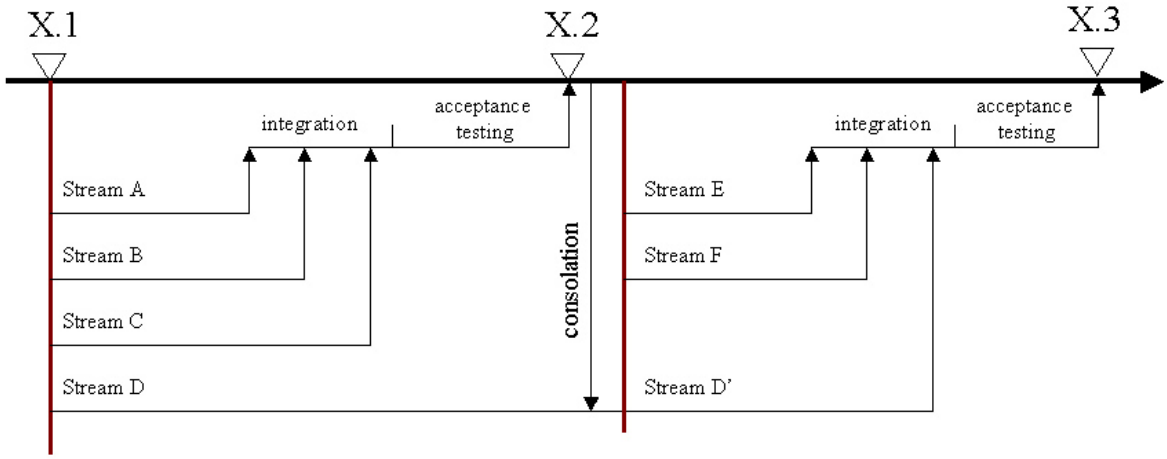
Incremental renovation, also known as "chicken little" approach, uses two methods:

Integrate
Reengineered modules are re-integrated into the legacy system. This is usually applied to small, localized software changes that do not seriously impact the rest of the legacy system.

Isolate
Reengineered modules are not re-integrated into the legacy system. Functionally cohesive components are identified, isolated, and (re-)engineered. The subsequent system now has both reengineered components and legacy modules that interface with each other through carefully defined mapping mechanisms called (forward and reverse) *gateways*.
In most cases, a pragmatic combination of these two methods for incremental reengineering will be applied. To solve minor legacy problems, the integrate method might already be applied, e.g. as part of the 'pizza list' method. However, application of the second method is necessary to address major legacy problems which affect larger parts of the code base.

Essential part of the plan is an explicit strategy as how to combine renovation and production. Introduction of non-trivial re-architecting work during ongoing development activities is not an easy task. One could compare this activity with changing a flat tire\ without stopping the car. From a business perspective, it is not acceptable to postpone releases of new features in order to roll out a completely new architecture. Production must go on – so one way or the other, these activities must be combined in a sensible way.

In most cases, software release are produced in an incremental way, using separate development streams for foreward engineering ('pizza list approach'), as depicted in the following figure:



concurrent software development using parallel streams

In the above example, streams A,B & C are started from the same archive, and combined for the next increment X.2. Stream D is a more labor-intensive feature, which is not included in increment X.2. Hence, a consolidation is needed after increment X.2 before streams E & F can begin with implementation of the features planned for increment X.3. After consolidation, these streams will progress parallel to stream D', and these three streams will end in increment X.3 after integration and acceptance testing.

This approach is quite suitable for relatively small refactoring work – a local 'clean-up' can be planned as a development stream, possibly spanning two or more increments (like stream D). This is in line with the 'integrate' method for incremental reengineering, in which refactored modules are directly re-integrated with the production code.

The situation will be different when the renovation requires the isolation of a new software component, especially when the reengineered components have many dependencies with the remaining legacy code. In other words: the pizza list approach implemented thus far might, might have drawbacks when the 'isolate' incremental reengineering method must be followed.

After all, when the isolation and reengineering of a new component is rolled out as an 'ordinary' development stream, the consolidation will be very cumbersome and time-consuming. The implementation of the forward/backward gateways for instance, will cause a lot of distortion and additional complexity.

More important, however, is the increasing pressure on delivery of the new component, especially when its development spans a number of increments. When there is not enough time to develop the new component in a decent way, there will a sacrifice on quality – and new legacy issues might arise. The conclusion is that the incorporation of the 'isolate' method for incremental reengineering requires a more specific approach.

The solution is to let a number of smaller, preparatory steps precede the reengineering effort needed to isolate a new component. The development of a new component can be prepared in many ways, for instance by re-grouping of functionality, refurbishing or re-routing existing interfaces, introduction of (temporary) adaptors, re-directing data/information flows, etc. These activities can be part of the regular pizza-list, planned in advance of the actual reengineering of the component. After completion of these modifications (using the 'integrate' method), the reengineering of the component (using the 'isolate' method) will be less cumbersome, having less impact and risk.

In other words: instead of 'going underground' for a considerable amount of time to completely reengineer the component, particular smaller (preparatory) tasks can be identified and split off as individual streams. The rationale of this approach is to minimize the off-line period needed for the actual reengineering of the intended component, in order to prevent undesired dependencies and to simplify the merging effort.

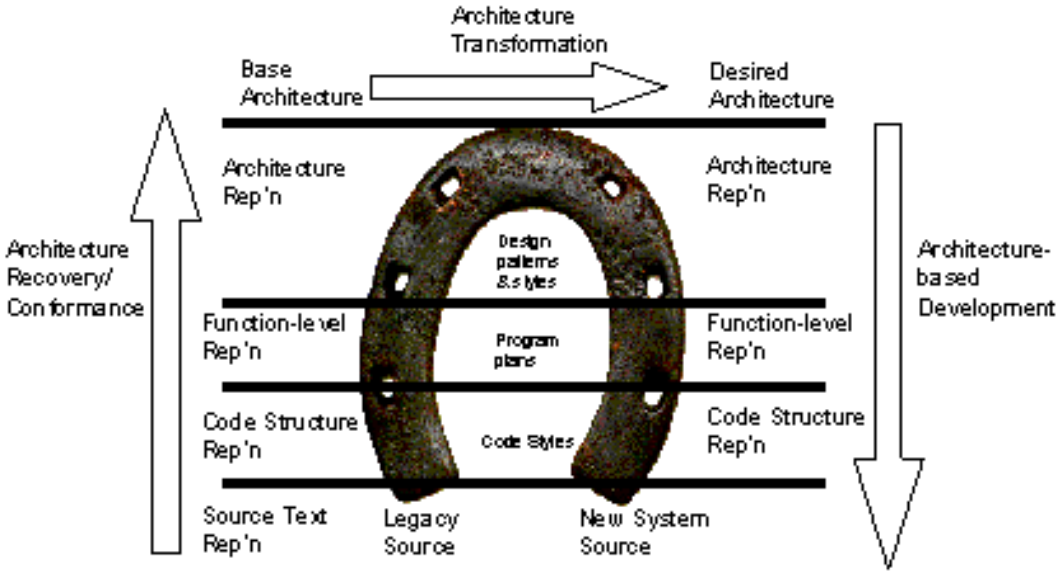
In some cases, the incremental approach can even yield business value before the renovation is completed. If for instance, a certain renovation activity would smooth the path to implement a new feature without much effort, it could be a wise strategic (and political) decision to combine renovation and new development this way. The visible pay-off for the organization will provide more buy-in for the renovation plan, perhaps bacly needed for more cumbersome steps in the future. The result of this phase is a well-considered and feasible plan, that will remove most risk from the renovation project. This is a prerequisite to keep the renovation project alive when the legacy systems runs into unforeseen problems.

Step 5: Renovate the System

The renovation of a legacy system involves in most cases a considerable re-architecting. In other words: renovation is not just code transformation! To integrate the code-level and the architectural renovation view, the "horseshoe" model is introduced. This conceptual model distinguishes different levels of reengineering analysis and provides a foundation for renovations at each level, especially for transformations to the architectural level.

In its most fundamental form there are three basic renovation activities:
1) analysis of the existing legacy system,
2) logical transformation, and
3) development of a new, renovated system.

These three activities form the basis of the "horseshoe", as illustrated below:



Horseshoe model for software renovation

In essence, the first activity recovers the architecture by extracting artifacts from source code. This recovered architecture is analyzed to determine whether it conforms to the "as-designed" architecture. The discovered architecture is also evaluated with respect to a number of quality attributes such as performance, modifiability, security, or reliability.

The second activity is architectural transformation. In this case, the "as-built" architecture is recovered and then reengineered to become a desirable new architecture. It is re-evaluated against the system's quality goals and subject to other organizational and economic constraints.

The third activity of the horseshoe is architecture-driven development to instantiate the desired architecture. In this process, packaging issues are decided and interconnection strategies are chosen. Code-level artifacts from the legacy system are often wrapped or rewritten in order to fit into this new architecture.

The horshoe model distinguishes three distinct levels. The first, or base, is the code level which includes the source code and artifacts such as abstract syntax trees and flow-graphs. The second is the function level which describes the relationship among a program's functions, data, and files. Third is the concept level in which clustering of both function and code level artifacts are assembled into patterns of architectural level components. The trip around the outside of the horseshoe represents an abstract form of renovation. In practice there are two additional shortcuts that cut across the horseshoe and that enable one to get from the "as built" system to the "as desired" system. These "shortcut" paths across the horseshoe can represent pragmatic choices based on technological, organizational, or process-related constraints.

An important contraint is the availability of appropriate *tooling*. Especially for the architecture reconstuction of complex systems, to reason about the reconstructed architecture, to motivate an architectural transformation with new architectural quality requirements, and realize this architectural transformation, advanced tooling is indispensable.

Another important, pragmatic aspect of renovation is the crucial role of *testing*. Without a proper testing environment, renovation activities are very inefficient, ans sometimes contraproductive.

A related issue can be the absence of a representative system at the development site which can be used for backwards compatibility testing. This can be caused by such a large variety of configurations or 'specials' that adequate is only possible at the client site, which situation can be a serious drawback for the renovation initiative.

Step 6: Adopt Best Practices

System renovation is a difficult process, with a lot of potential roadblocks. It is important that practical experiences are shared, in order to prevent the organization (or other organizations) from making the same mistakes again. It will be much easier to achieve commitment and motivation for structural modernization activities when there is a reference to succesfull practices applied in another, similar project.

Sharing and documenting best practices is also needed to secure valuable experience and knowledge, especially when dealing with legacy issues.

It is recommended to adopt Agile practices for the incremental reengineering, such as the agile Scrum method for project management. Retrospectives after each increment are a rich resource for gathering valuable information about the pragmatic aspects about renovation of legacy systems.

April 2008
Erik Philippus
***I**mprovemen**T** BV*
Erik.Philippus@improvement-services.nl