Abstract

Most successful software systems outlive the people that use them, the hardware / platforms they run on and quickly outgrow the business requirements that they were originally implemented for. Once institutionalized they become critical to the business they serve. Hence it becomes important to plan reengineering exercises to ensure the system continues to deliver both expected and unexpected business benefits.

This whitepaper attempts to delineate the best-practices observed during various re-engineering exercises conducted at Infotech IT for both small and large software systems.

- Shiva Mantripragada
- Murthy Maddali

© 2014 Infotech Enterprises IT Services
Software is regularly maintained

There is a fundamental difference between software maintenance and software re-engineering. Maintenance is a technique of ensuring the software continues to meet stated goals (functional, performance, operational etc.). Maintenance has a short-to-medium term focus in adapting to newer requirements or focused technology changes. In short maintenance is a more tactical activity as far as IT investments are concerned.

In contrast, software re-engineering has a wider outlook. It is a more systemic change that takes a fresh look at the future and aims to open new vistas for the audience it serves. Re-engineering is strategic in nature and is often initiated as a measure to gain competitive advantage.

Unsuccessful software exhibits significant cost overruns and a conspicuous lack of business flexibility during the maintenance phases. The goals for re-engineering in such cases would be different (initially) from those for successful software.

Trigger for Re-engineering

Why re-engineer?

There are some very common reasons, both business and technical, why enterprises should take-up a re-engineering exercise. Each re-engineering has to counter one or more of these issues.

- Keep pace with the evolution of the current platform
- The current platform is no longer supported
- The current platform is no longer in vogue. It becomes difficult to keep skilled staff at hand
- The cost of change is high
- Business has changed
- The system has to align with organization's standard
- Business wants to use 'cutting-edge' technology to serve their markets
- Business wants to use 'bleeding-edge' technology to serve their markets
- Consolidation of systems or landscape changes due to mergers/acquisitions

When does it start?

A typical software evolution exercise looks like this,
The true-value zone for a software system is that period when the returns are proportionate to the investments made. While this is an objective assessment it requires a rigorous definition of what returns are and how to compute them. In their absence, a subjective and simpler assessment is when there are far fewer complaints (by any stakeholder, IT/ end user /finance / etc.) and a high activity rate on the system in question.

Most re-engineering exercises start somewhere between the later half of the true-value zone and the End-of-life zone. Those organizations that start early have a better opportunity and lead time to refine/ redefine their vision for the system and be able to execute the same.

The timing of re-engineering exercise is important. If too early, there may not be enough reasons to justify it. Premature reengineering is usually no more than a glorified maintenance exercise (with a few technology changes, feature additions etc.) It does not have enough experiences backing it to steer the system in the right direction. Delayed re-engineering has its pitfalls too. Usually it boils down to losing competitive edge. Organizations that aim to save costs by choosing to continue with outdated systems will eventually cease to be competitive.

Industry experience has shown that re-engineering is a collective decision voted by various stakeholders. Below are some parameters for attempting re-engineering from various stakeholders point of view.

**The end-user / The business**

The original system build will usually result in 10-15% of features which are either gold-plated (and consequently not used) or are mis-implemented. At the same time, an equal number of necessary features would have been missed out owing to lack of foresight. Also the business itself could have changed significantly warranting an overhaul.

**The operations team**

IT infrastructure is continually optimized to make it stay current and to ensure business continuity is achieved. If the IT systems do not improve at the same rate to take advantage of modernized infrastructure, the Operations team would not be able to provide a hassle-free experience.

**The engineering team**

There are several reasons why the engineering team votes for a re-engineering decision. It usually boils down to newer standards, newer technological possibilities that allow certain requirements to be more effectively implemented, lack of skills or unavailability of manpower, architectural corrections on a system based on lessons learnt from actual operation (such as performance, availability, scalability etc...), an improved design that allows for flexibility in the future.

**The finance team**

Rising costs of maintaining existing systems is usually a very strong motive for supporting a re-engineering decision. The cost-per-feature metric is closely evaluated by organizations to see signs of an aging system.
Some patterns and anti-patterns to a successful re-engineering exercise are presented below,

1) Re-engineering is not just technology replatforming: The good old idiom “Old wine in a new bottle” does not apply here. While replatforming can be an important exercise, it should not be the only candidate for a re-engineering exercise. Opinions need to be solicited from stakeholders as discussed before, and an appropriate consensus must be built on the scope of re-engineering.

2) Employing Bleeding-edge technology will delay the next re-engineering exercise: Surprisingly several high end IT organizations are a victim of this evil. This anti-pattern actually forces an early re-engineering. Since bleeding-edge technology is unproven / prone to failures and does not deliver academic promises, it is usually wiser to refrain from them. Depending on the desired longevity of the solution and the ability and impact of changing underlying technology, organizations would be better off choosing either Cutting-edge / Leading-edge technologies. IBM Web sphere Commerce Server, the leading Commerce solution, continuing to rely on Servlet/JSP technology despite availability of a plethora of java web/application frameworks is a case in point here.

3) Employing commodity (standard) technology will delay the next re-engineering exercise: Contrary to above point, this too can manifest as an anti-pattern. In a rapidly changing and highly competitive business environment, employing standard technology may not allow an organization to differentiate from its competition (let alone catch-up with its competition). In such scenarios certain amount of guarded risk must be employed in choosing volatile technology. Appropriate design care must be taken to ensure the system's core functions are sufficiently insulated from the volatile exterior.

4) Don’t change the UI: Some businesses (who have many end-user staff) are worried about retraining effort/costs incurred due to a reengineering exercise. Businesses perceive changes in UI as having a detrimental effect on the end-user's / operator's efficiency. While there is truth in this, this is a definite Anti-pattern. For example, some enterprises wish the (currently operational) desktop UI be migrated AS-IS to be web based scenario (including the key-combinations) for the same reason. Not only was it very difficult to implement, it also meant that new platform's (browser) native features were completely ignored. What should have been a simple retraining exercise (for a few months at the extreme) soon becomes a technology baggage that has to be painfully borne until the next round of re-engineering was planned.

5) Look at the current source code and build the new system: Some customers hint that all requirements are hidden in the current system and they need to be implemented for the next system. This is an anti-pattern. There are many things that are wrong with this approach. The source code may not be understandable / complete. The source code might be in a language (either programming language or a spoken language) which is completely alien to the implementation team or may not have enough / correct documentation. Effort must be spent in developing requirements early-on. The existing sources can serve at-best as a secondary reference.
6) **Build discrete services and Focus on integration:** This is a best-practice, we are living in a world of services era. This is the right time, to think in terms of services and not applications. Since the system will evolve over a period, focus on getting the integration right. This might cause certain changes to the surrounding applications, but will payoff many times over in the longer-run.

7) **Co-existence of old and new systems:** Some customers expect that the new system will be built in a manner that it will continue to co-exist with the older system. Features from the older system will be erased in phased manner as they are migrated to a newer system. The user-experience is expected to be consistent between the older and the newer systems. While this sounds as an agile best-practice (in some cases this may even be), it is not a mean technological feat to achieve. The older and newer systems might be incompatible in a number of ways ranging from platform incompatibility (say Mainframe v/s Java or VisualBasic v/s .NET or ASP to PHP) to database incompatibility (Oracle v/s Mongo) to data model incompatibility to process incompatibility (synchronous v/s asynchronous). One such business experience is to implement a similar setup between an older system built on AndroMDA / SpringMVC and a newer one which was being built on Wicket/Scala running on different data-models. The goal was to make the 2 systems under-the-hood appear as a single system to the end-user (including login, menu etc.) To keep both systems running data had to be synchronized between both data-models instantaneously (not even near-real-time) because a different user would expect to see data created in a older-app in a newer screen. This caused immense architectural problems.

8) **Codebase in a universal language such as English:** When re-engineering systems, it must be ensured that the codebase of the newer system is expressed in English (the original codebase can be in language such as German / Japanese) since this ensures that the code is understandable to a wider, trans-geo coding community. Similarly all documentation, operation runbooks etc. must be prepared in English.

9) **Old is not always Gold:** The features available in the older system must be debated on their applicability in the current scenario. Never copy feature to feature. This is an anti-pattern and creates problems. Such features force the data-model to remain consistent with the older data models, leaving little scope for improvement.

10) **Never mix Old and New code, unless you know what you are doing:** Reusing code is a good idea and must be encouraged. However reuse brings advantages only when the code to reuse is isolated. If there are additional dependencies (library version level, platform specific dependencies) then that unwanted baggage is also inherited that will be regretted later. Evaluate reuse carefully. If necessary copy the code (or recompile it) to use newer dependencies and get rid of the older ones.
11) Old and New systems continue to evolve: In long-drawn re-engineering exercises, the old systems continue to evolve as the newer system is being built. While this is inescapable, it must be understood that this causes the newer system to continually lag behind. Testing, rollout and maintenance efforts also duplicate as a result of this. While the top management / senior management views the new system as being critical (from a strategic standpoint), the user-staff/middle-management regard the current system as being important (from an operational standpoint). These discrepancies in perceptions cause problems in implementation. The business must be convinced that only the high-priority requests be implemented in the older software.

Conclusion

The paper attempted to uncover some of the best practices to be followed for Re-engineering software systems based on Infotech-IT’s substantial experience in such areas. It must be noted that, depending on the context some of these practices may appear to be in conflict with each other. Readers are encouraged to understand better the application case and choose the practices that seem most relevant for their scenarios. Please reach out to Infotech-IT for any assistance or evaluation at it.services@infotech-enterprises.com