Legacy: Old technology that frightens developers (part 1)

(Original creator: bolarotibi)

By Clive Howard, Principal Practitioner Analyst, Creative Intellect Consulting To developers the term legacy is often a dirty word meaning old software that is a pain to work with. Ironically, of course, it is the software that developers spend most of their time working with and developers made it what it is. The question all developers should ask is why legacy software is generally considered to be bad and what can be done to avoid this situation in future? After all an application released today will be legacy tomorrow. Development teams do not set out to create bad software that will become difficult to maintain, support and extend. When allowed by their tyrannical masters architects and developers put a lot of work in upfront to try and avoid the typical problems of bloat, technical debt and bugs that they fear happening later. For some reason over the years these problems seem to have become inevitabilities. The type of issues that make developers fear working with legacy include: technologies that are no longer fit for purposes; bloated codebases that are impossible to understand; different patterns used to achieve the same outcome; lack of documentation; inexplicable hacks and workarounds; and a lack of consistency plus many, many more. Most of these have their roots in a combination of design and coding.

Design theory does not always reflect reality

Architects aim to design clean, performant, scalable and extensible applications. Modern applications are complex involving multiple “layers” often distributed from a hardware perspective and including third party and/or existing legacy applications. Different components will frequently be the responsibility of different development teams working in different programming languages and tools. For some time now the principle of separation has been applied to try and avoid the tightly coupled client/server applications of the past that were known to cause many legacy issues. This has gone under many guises, “separation of concerns”, n-tier, Service Orientated Architecture (SOA) and so on. They are all variants of the same concept that the more separated out the components of an application are the more flexible, scalable, extensible and testable that application will be. For developers having an application made of smaller parts makes it more manageable from a code perspective. One of the classic scenarios is the interchangeable database idea. An application might start life using one database, but later on it needs to change to another. The concept of ODBC meant that it was easy to simply change a connection string in code and providing the new database had the same structure as the previous everything would continue without a hitch. The problem has been that what looks good theoretically doesn’t hold up in reality. In the example of changing the database the reality often meant that there were a number of stored procedures, triggers or functions included in the database. Changing from one database to another meant porting these and that in itself can be a significant task. The time and therefore cost of such an activity resulted in the old database continuing. Hence today we find so many applications running unsuitable databases such as Access or Filemaker. A developer then has the frustration of having to work with inherently limiting and non-performant code.

No immunity from separatist design strategies

If we move forward to many of today’s architecture patterns such as SOA we still see similar problems. The concept of SOA is that components of an application become loosely coupled and so different parts of the application are less wedded to one another. Unfortunately within the separate services and consumers the same problems as outlined above can apply. Worse than that is many service providers do not version their services. Google Maps will often bring out a new version of their service and clients calling the previous version will continue to function. However many others (social networks take note) do not follow this practice and frequently push out breaking changes to their service. This introduces a whole new problem into legacy applications whereby developers have to regularly go back into code and update it to work with the changes to the service.