Transformation Of Software Platforms From “The Edge”: Refactoring To An Outside-In Architecture

Thomas Winans & John Seely Brown

The emergence of cloud and other modern technologies provide opportunities to develop and deploy new software systems more rapidly and economically. The IT industry would like to modernize its collective software portfolio, though it is conflicted about just how to do such because of the need to balance its desire to start fresh with the necessity to reasonably extend the life of its past IT investments, with the accompanying needs to migrate users of large systems from old to new, with their considerable amounts of data, understand undocumented features/functions/integrations, and deal with technology heterogenity of the portfolio itself. Of course, the ownership of the actual code base of portfolio components merely exacerbates an already untenable situation. Nor is the complexity of market dynamics considered as these constrain opportunities for change.

Change in large platforms, with many users and lots of data, can require many years to implement, for just as many non-technical reasons as there are technical ones. We've learned from past software implementation failures that it is imprudent to take a big bang, bare metal up approach to replacing large platforms: requirements are not completely detailed; business rules buried deep within software components are not known often without careful sifting of code; reuse of code is not easily measured; and so forth. There usually is a tension between implementing a more modern version of existing capabilities and getting to new functionality that quite often drives change.

We've written elsewhere of how Rearden Commerce transformed its web application to a policy-driven personal digital assist service platform over a two year period. While Rearden's platform is not ERP-class, the method Rearden used to transform its platform comes in from its platform's outer edge, and this method scales to transforming ERP-class platforms. The transformation method can be stated as follows:

- Define public interfaces that must be used to access platform functionality by external systems, and that must be used between different platform modules.
    - The granularity of these interfaces *should be* coarse grained (to a human processible document level where possible) to minimize close coupling between modules and between the platform and external systems interoperating with it, i.e., to keep low level chatter down between platform components.
        - Ideally, granularity should be to a human processible document level. In our view, too much focus has been placed upon use of XML (implying *document orientation*) for enterprise application integration, and we have *not* questioned whether or not enterprise application integration (as we know it) would be needed at all *if* we properly defined "document". We have the opportunity to leverage XML effectively *if* we properly define "document" to align with how humans do work, and to include appropriate/complete content and context. Note the word "include" vs. the use of a phrase like "link to context". Object orientation might suggest "link to". But heading in that direction will take us to a discussion on use of fine-grained APIs: we've abused OO and had our heads in that sand for 15 years, and it is time to pull our heads out …
    - It might be necessary to implement software complying with these interfaces around legacy technology to encapsulate legacy and decouple its change from implementation of and interoperability with new functionality. Interfaces service as inviolate design contracts that

codify requirements and enable change within the platform *as well as* around the platform in parallel.

- Refactor and consolidate existing code, and replace it with modern technologies and methods as possible and expedient.
- Externalize key policy points, making it possible to configure the system to accommodate change without always having to develop code that is situation specific.

Rearden *could* have implemented a new system, from the ground up, and put it into production as a kind of shadow system to the original, cutting over at some point that it determined to be best. However, or design contract[1] perspective that Rearden took ultimately afforded it the opportunity to implement new functionality sooner, and in parallel with changing old.

The design contract approach, e.g. using programmatic interfaces to specify behavior and separate it from implementations of it, becomes more appealing as the size of the system to be replaced gets large and complex. Consider an ERP-class platform that has been in place for 15-20 or more years. Transformation of this platform from its current technology base to a modern one, externalizing policies so that behavior becomes more configurable, and migrating data in legacy structures to modern ones could take years to implement. New functionality (probably evolutionary in nature) could be implemented in the process, but software system authors likely would find themselves behind younger competitors unencumbered by legacy to maintain, and users of these systems would find themselves dissatisfied by the slower than hoped for pace of change vis-à-vis the revolutionary new capabilities they want. Well defined designed contracts permit software authors and software users to operate in parallel.

The reader might think that the contract-oriented design and document orientation messages are similar to messages given by software engineering experts in the past. Perhaps. However, it is worth saying twice that document orientation relates to structuring programmatic interfaces around documents that humans exchange, which are complete in both content and context – they are not context free and document segment oriented as we have seen in the past with enterprise application integrations that use XML. Resulting interface granularity is coarse, and coupling between platform components and external platforms is low. Platforms built using design contracts and document oriented interfaces are substantially easier to align to key business processes, and maintain or transform, than platforms that are not. Further, we observe that applying design contracts and cloud technologies to transform large systems can result in a paradigmatically different architecture that enables significant organizational change in an enterprise as well.

In large systems, like an ERP system, around which quite possibly many other application systems have been built, information used by various parts of “the system” has to be assembled through middleware-enabled integrations across systems of record of specific types of information. Integrated information sets are formed, business decisions based on them are made, and the integrated set of information is quite often then discarded until the next scheduled run of the integration application. What if these integrated information sets were persisted in a cloud, and they formed the basis of future applications to be developed? Over time, integration applications would be replaced by platform components that deal natively with the integrated information sets, obsoleting the integration applications in the process. Source of record systems could become subscribers to changes of the information for which they are the source of

[1] http://en.wikipedia.org/wiki/Design_by_contract

record so that they could continue to be used as reporting engines. But even this use of systems could be eliminated as reports are implemented as a function of natively integrated information sets and information necessary to run the business is migrated to the cloud.

Finally, it is interesting to consider the effects of this type of transformation. It has been true that when one internal system is replaced with another, change occurs within an organization that can be called evolutionary. Many business and system processes remain the same, or are at least similar to what they were. The convergence of the technology innovations labeled "cloud" is enabling change that is revolutionary … change that, once effected, will be difficult to revert.[2]

The process of refactoring and transforming an architecture described in this brief paper is more fully described in other of our papers ("Moving Information Technology Platforms To The Clouds" and "Demystifying Clouds") which explore key *outside-in* architecture characteristics necessary to effectively architect software systems for cloud and service grid deployments.

[2] As cloud and service grid platforms become platforms of choice, it is possible that volume of data, techniques to analyze data, and cloud and service grid services will become *platform primitives* that cannot be replicated in non-cloud and non-service grid platforms, at least without substantial effort.