Taking An IT System In-House, Post-Outsourcing

Copyright 2012 Gary Mohan
gary.mohan@plainprocess.com
www.plainprocess.com

This guide can be downloaded in PDF format, for free, at:
http://www.plainprocess.com/inhouse.html

Table of Contents

Introduction

Who is this for?

This is for managers in client organizations, attempting to take an IT system in-house after outsourcing.

What does it cover?

It covers the minimum steps involved with taking an IT system in-house.

Why bother reading this?

Client managers are frequently intimidated about removing IT consultancies, taking in-house the maintenance of IT systems. IT consultancies play on client fear, hoping to retain their position. This guide can help client managers understand and mitigate the risks involved in this process.

The Playbook

Prior to discussing the steps involved with taking a system in-house, the problem of handling a hostile transition needs to be understood. If your relationship with the vendor is good, the transition should be professional and amiable. This is especially if the vendor is a product-based company, with an integration consultancy as one of its business units. Their business model is focused around licensing, customizing for clients and providing a support service for client software engineers who maintain custom code particular to that client.

If the vendor is a "body shop" IT consultancy, they may have deliberately underbid on the development phase of a new system, hoping to make money on the maintenance contract. If they suddenly discover that this revenue is gone, the transition could prove hostile, with them desperately attempting to "lock in" the client. Vendor staff could turn nasty.

Timing It Just Right

Depending on the level of trust you have in the vendor, you may need to be very careful about timing exactly when you inform them of your decision to take maintenance in-house. Mistiming this step could facilitate sabotage of the transition.

Generally, the point in time when the vendor is least able to effect sabotage is in the 24 hours after the final candidate build of the system has gone into the test environment. The system has not yet been accepted, meaning that the vendor has uncertainty about being paid. The vendor project manager will be focused on turning around test faults. Informing him or her of the decision at this point in time means that they literally have no time for sabotage. Also, vendor engineers will have been working hard to remove bad implementation code, making source code easier to pick up. Ruthless as this approach may seem, your ultimate responsibility is to your own users and your own team.

Your New Best Friend

For most systems, the gap between delivery of the final candidate build and deployment into the live environment is three or four weeks. If the transition is likely to be hostile and the vendor is unable to "lock in" the client, the standard IT consultancy playbook is to then play on client fears. During these weeks, a mixture of very senior and surprisingly junior vendor staff will suddenly become your "new best friend".

Very senior consultants will appear, wanting to go out for lunch to “talk issues over”. If you have already made the business decision to take the system in-house, what is there to discuss? The idea of this meeting is to simultaneously instill a sense of confidence in the consultancy (as a pitch for the support contract) whilst insinuating that you lack the capacity to take over the system, creating a sense of paranoia. This is done under a facade of “reassurance”. It will be discussed below in this guide that taking a system in-house involves an element of risk. There are standard risk mitigation strategies for this, including extra training for your own people, hiring one or two contractors with niche skills and indirectly finding a legal way of hiring vendor staff.

A particularly nasty consultancy trick is to use junior consultants to bitch about your engineers’ abilities directly to users. The most innocent sounding people will be used to say the worst possible things. Some of it will be subtle, some of it not so subtle.

Really, their people are geniuses and your people are idiots? If you gave them the support contract, do you imagine that the “geniuses” would hang around or be replaced with recent graduates “learning on the job”? To what extent does this situation resemble a bad heist movie? Watch out for the scene where vendor staff all disappear simultaneously for an off-site meeting and return in a bad mood.

It is important to brief users about this potential problem and have them report bitching directly to you. It helps if users keep written notes of these conversations. These notes can be useful as leverage later in the transition and will make them look appalling in front of a judge, if it ever gets to legal action.

Do not make the mistake of either you or your people becoming angry at this nonsense. Stay focused on the transition.

The Minimum

At an absolute minimum, after transitioning the system in-house, you need to be able to:

- Build the system.
- Deploy the system.
- Regression test.

None of these are optional extras. Addressing them mitigates the highest risks involved in the transition.

Build and Deploy: The Unified Support Document

During transitions, managers become concerned with the vendor's engineers producing an adequate level of documentation for internal engineers to be able to take over system maintenance. This can be a distraction. There should already be various documents available, explaining the basic system architecture and at least something about the data model. Your engineers can look at these documents and ask for detail to be added.

Of much greater relevance is that your engineers use the transition to produce their own unified support document, explaining end-to-end how to build and deploy the system. It should be stressed that they will need to produce this anyway for end-to-end disaster recovery purposes. The document is usually about 20 pages in length and should not be difficult to write, however, it may take several days to gather the information together. It should cover:

- A short summary of the system's business purpose, with reference to where functional specification documents can be found.
- Architecture overview.
- Physical Databases
    - Database product (e.g., Oracle) and version number (i.e., saying "Oracle" is not enough, it should be something like "Oracle 11g Release 2, 11.2.0.3").
    - All information needed for a desktop client to connect to the development database (or databases). Typically, this includes server names, port numbers, instance names, usernames and passwords.
    - Depending on your security and support policy, this information may also be captured for test, acceptance testing and training environments.
    - Where in source control a SQL script (or scripts) can be found to re-create a blank development or test database. Typically, this will recreate all tables, all indexes, all stored procedures, all

    triggers and load user defined reference data. This rebuild process needs to be tested end-to-end, with an instance of the system pointed at it and a sample of regression tests run on it.
  - A list of SQL scripts used as workarounds for administration screens not yet developed or other missing functionality. Whilst far from ideal, this needs to be documented. These scripts should be under source control.
- Physical Application Servers
  - Product name and version number.
  - Physical installation and deployment locations. If in development it is deployed locally, what are the conventions around installation and local deployment? If a shared instance is used, where it can be found on the network and how to deploy to it (e.g., “Copy the built file to network location X”).
  - Differences between development and live configurations (e.g., port numbers).
  - Log file locations.
  - Depending on your security and support policy, URL locations for the application server’s administration user interface for test, acceptance testing and training environments.
  - Are security certificates required? How are new certificates issued and old certificates replaced?
  - Licensed third party libraries. Which ones and how are they used?
- Operating Systems
  - If developing on one operating system and deploying to another, what are the differences?
  - Environment variables, including differences between development and live environments.
  - Relevant operating system logs.
  - Non-standard operating system configuration changes. For example, it may be that development and test environments use different DNS servers, compared to the live environment.
- Build
  - Build tools used, with version numbers for these tools.
  - Build command (or commands) run locally.
  - Does the command change for particular deployment environments (e.g., acceptance testing)?
  - Are build scripts under version control? Where can they be found in source control?
  - Does the build environment automatically build a project file for the Integrated Development Environment (IDE)? If not, how should a developer setup a project in the IDE?
- Deployment
  - Development deployment.
  - Differences with live deployment. For example, does the build need to be copied to a staging area?
  - Release instructions for the live environment

        - Which backups need to be taken before deployment?
        - Which parts of the build are deployed where, in what order?
        - Is there a particular start-up sequence?
    - Live infrastructure used in the development environment. For example, for whatever reason, testing if the system sends out email alerts may involve using the live SMTP server. Whilst these situations are far from ideal, they need to be documented.
    - Security rules. Have firewalls or other security products been configured in a particular way for this system?
- Separate End-User Client Application
    - Especially if a tailored version of the application is available for mobile devices.
    - How is this application built, deployed and distributed to end-users?
    - Are virtual machines or emulators used on developer PCs to carry out this work? Are licenses needed? Is there a standard approach to installing and running the emulator?
    - How is this application tested?
    - Are there particular support requirements for this application?
- Source Control
    - The process under which you can request or add yourself to source control.
    - The tools and steps needed to access source control and check out locally.
    - Which branches are present? Which are the current development and live branches?
    - Is source control tagged per-build or per-release? What are the conventions around this?
    - IDE integration with the source control database.
    - Has the source control database been exhibiting performance and stability problems? Does it need to be recreated from scratch?
- Batch Processes
    - Does the system run batch jobs?
    - Is the scheduler internal or external to the system?
    - What command does the scheduler run? How is this run locally, in the development environment?
    - Do user interface screens exist for batch jobs?
    - What are the consequences of a batch job failing?
    - Alerting framework used to notify of failures.
    - Impact of Daylight Savings Time changes.
    - Has a particular batch job been implemented in system code that could easily be replaced with standard operating system commands or off-the-shelf tools? Why? Is there a functional or technical reason for this approach? Be particularly wary of jobs that perform Extract, Transform & Load (ETL) functions. Input from system administrators can help with determining if a batch

    - job should be replaced. They may already have a preferred ETL tool licensed.
- Unit Testing and Continuous Integration
    - Unit testing framework, including mocking environments.
    - The steps needed to introduce a new unit test.
    - Continuous integration server setup and administration.
    - Are builds run on a scheduled basis or in reaction to a developer checking in new code?
    - What is the alerting framework used to notify of failed builds (e.g., emails)?
- Proprietary Test Harnesses
    - Was a proprietary test harness developed for integration testing? This is especially the case with messaging systems.
    - How is the test harness built, configured and used?
    - How can the test data used with the harness be changed?
- Fault Finding In The Live System
    - Logs. Which logs to look at and how to interpret them? Warnings that can be ignored.
    - Database. How to work with a DBA to understand deadlocks in the live system.
    - Certificate expiry. What faults would this cause in the live system?
    - Thread dumps in the live runtime environment.
    - Network I/O monitoring.
- Known Issues and Bad Implementation
    - Source code containing hardcoded SQL and physical configuration parameters. The search for this code can partly be automated by using a text editor able to traverse subdirectories, looking for “SELECT”, “UPDATE” and “INSERT” as search terms.
    - Parts of the system that were developed and abandoned but the source code remains.
    - Parts of the system where testing requires reference data different from that in the live environment.
    - Browser version and client platform incompatibilities.

Remember, you need this document anyway for end-to-end disaster recovery purposes. Whilst writing it, if engineers spot problems with the system, bug reports and change requests should be created as they go along. These change requests will help later with writing a remediation plan. If particular source code modules are incomprehensible, they should either be explained or completely replaced.

Regression Testing

If you can build and deploy the system and are unable to test it, you need not have bothered with the effort involved in creating a unified support document for engineers. Regression test scripts matter, as these provide your test staff with the means to determine if the system is acceptable. Later implementation of remediation changes, such as fixing bad data architecture, will be very difficult without these scripts.

It is important to request a copy of regression test scripts from the vendor. Your people should perform some level of due diligence on whether the scripts provide adequate functional coverage. Not requesting and not checking regression test scripts is one of the most common reasons for a transition to fail.

Some organizations have a regression testing policy requiring that all tests defined for system testing should be repeated. Often, regression testing does <u>not</u> test every possible permutation and combination of system functionality. Other organizations adopt a risk-based approach, focusing on high-risk areas:

- Functions defined by users as being critical to business continuity.
- Functions most frequently used by users on a day-to-day basis.
- Complex functionality requiring detailed test data (e.g., actuarial calculations).
- Fragile functionality with a known history of test failure.
- Interfaces to third party components (e.g., a rules engine or a messaging system).

The Remediation Plan

During the transition, a number of change requests and bug reports may have been raised that (for whatever reason) will not be fixed by the vendor but taken in-house. Users will want a plan, with estimates of how much each change will cost. These can be collated together as a remediation plan, split between functional and non-functional changes.

Functional

“Functional” changes refer to changes end-users “see” as part of the live system, explicitly stated in requirement documents. This can include data changed as a result of a batch process. Typically, the vendor will have left a package of functionality unimplemented. Users need to produce a prioritization list, ranking requirements, balancing importance with urgency.

For software engineers, an early priority is indentifying functional code not adequately covered by unit testing. Code coverage tools can help to automate the detection of “orphan” code. If the amount of remediation needed is significant, estimates should be produced for this work.

A classic mistake when transitioning an IT system in-house is to ignore a denormalized data model, not seeing it as a functional issue. Schemas are part of the functional system. Denormalized schemas bloat algorithmic source code in the system’s business layer, increasing fragility.

As a general rule, modern enterprise applications should have no functional database stored procedures or triggers, given that these are notoriously difficult to test and can compromise stability. There are certain circumstances under which a DBA may decide to add a stored procedure or trigger, but this must be driven and controlled by the DBA. The longer you wait to fix the schema, the higher the cost and technical risk involved in making the change.

Non-Functional

“Non-functional” refers to aspects of the system not defined in a user requirement. These areas are: stability, scalability, security and system administration.

Stability was mentioned above in relation to the schema. The single most common cause of instability in a live system is the presence of a memory leak that was not detected during testing. Both stability and scalability problems can be observed through paying attention to the live environment and putting the system under performance testing.

If database transactions become deadlocked in the live environment, the DBA should be able to provide the underlying SQL statements that gave rise to the deadlock. The application server may also give a dump of which application threads are currently running. You might need to introduce extra logging. This information can guide where stability and scalability changes should take place.

Bad security is frequently ignored, with users paying lip service to the idea that security matters without allocating actual budget to addressing it. If you think there is a serious problem, you need to make the business case for the money. Similarly, system administration requirements are ignored. A typical example is an incomprehensible log file, which might as well not be there. If administrators have requirements, these should be addressed.

Risks and Risk Mitigation

Once the system has been transitioned, the biggest risk is not having appropriate people available to work on it. Extra training for in-house staff, bringing in contractors with niche skills and attempting legally to induce vendor staff to come work for you, can mitigate this risk.

If you identify a training need, budget for it early and expedite sending people on relevant courses. Contractors with niche skills can also help. For example, if you identify problems with the database schema that you lack confidence in addressing, bringing in a data architect may help.

A more vexed issue is attempting to induce vendor staff to come work for you (if this is something you want). Usually, the vendor will have had a clause in their supply contract preventing you from doing this. Some clients are able to get around this by having the vendor staff member come back in via another third party (e.g., an employment agency), making sure that they make no mention of where they intend next to work. If you need legal advice, get it.

For further discussion of project risks, a risk analysis manual for Agile environments is available here:

http://www.plainprocess.com/pdf.html

The First Three Months

Once the vendor is gone, the first few months in maintenance can be busy. Depending on your contract with the vendor, you may have some level of support agreed.

Organization Structure

In addition to setting up a normal team structure for engineers maintaining the system, other parts of the organization also need to be involved. Help desk staff and training staff rolling out the system will need guidance, attempting to fill in gaps in training courses and user help documentation.

Third level support also needs to be put in place, setting up an escalation process under which major problems with the live system can be sent to the engineering team for analysis. Out of hours support arrangements and on-call procedures may need to be sorted out.

It might help to introduce change control and release management processes (managing how changes are introduced to the system). A separate guide for this is available here:

http://www.plainprocess.com/change.html

The Incomprehensible Log File

System administration requirements are often ignored during the development of new IT systems; given that user acceptance testing ultimately determines who gets paid when. This leads to incomprehensible log files in the live system, useless for system administration purposes.

It is important to cooperate early with system administrators on improving the quality of logging messages. In many environments, administrators have third party log monitoring systems, automating the raising of system alerts. This approach can be much more effective than attempting to engineer alerting functionality into the system.

The 1% Batch Job

(Not the title of a bad heist movie.) If a batch job takes 1% longer to run each day, after one year it takes 37 times longer to run. To put context around this, if the job took 15 minutes on day one, it will take longer than nine hours to run on day 365.

Growths in batch job times are a notorious problem with new systems, often ignored until system administrators demand action (as a result of system stability problems). In the early life of a new system, it is important to pay attention to batch job times, attempting to anticipate problems before happening. Be particularly wary of jobs that perform Extract, Transform & Load (ETL) functions. Administrators may have an ELT tool already licensed and might prefer that the job be replaced by an ETL task. The earlier problems are identified, the earlier fixes can be made.

The Business Case: Offshore Development

Taking an IT system in-house can be particularly difficult if work is taking place offshore and your organisation has an entrenched management dogma that offshore development is “cheaper”. If management attitudes are so deeply entrenched that dialogue is not possible, it may help to make the business case directly to users, asking them to then overcome internal dogma. The business case is easier made if you concentrate on the <u>**hidden**</u> costs of offshore software development:

- Duplicated testing. This is the most concealed cost and is often introduced as a management “initiative” to “solve” quality issues “efficiently”. System testing occurs both offshore and onshore, increasing costs. It should be explained to users that they have paid for the same thing twice.
- It magnifies bad requirements, as requirements specifications are accepted as-is, without challenge. This leads to re-work, increasing costs.
- Various communication problems related to the fact that online video conferencing tools are not as good as the people using them think they are. Again, this ultimately leads to expensive re-work.
- Loss of context. The offshore team are even further removed from the business unit than an onshore team, leading to cost implications.
- Complex business requirements are easily lost or misunderstood. Often, the offshore team literally does not have the vocabulary to understand the issue, requiring language translation staff to be added to the bill. An example is in the insurance industry, where similar sounding vocabulary is used to refer to very different business concepts. A supplier may have finished work for an EU insurance client, believing that they can “cookie cut” the same solution for a US client, not understanding differences in use of vocabulary, let alone business practice and culture. Ultimately, bad business-alignment of IT is very expensive for business users.
- Change control is much harder, as this stuff really is more efficient face-to-face.
- The sheer volume of extra documentation that needs to be produced. How much is all this documentation work costing?
- Detachment from business risk. For projects where the business implications of non-delivery are very serious (e.g., regulatory fines), offshore development increases the risk. The option to ditch and re-write a large part of the system at the last minute is often infeasible.
- Scale. The larger the project, the less likely an offshore approach will work.
- Despite the advantage of having people work “overnight”, these efficiencies are lost through other inefficiencies (especially management overhead).

- The only reported metrics on efficiency are the ones that make the effort to take work offshore look good. This is due to the usual management pressure to make anything and everything look like a "success". Users are not stupid and can become quite angry at being misled in this way.
- The advertised model of onshore management and offshore development is, in practice, the worst model for running offshore development and is quickly abandoned in the early stages of a project. The "workarounds" introduced, such as putting some developers, testers and business analysts onshore increase costs.
- Job titles come into existence that had not existed previously. For in-house development, do you need an "Engagement Manager", "Onsite Coordinator", "Offsite Coordinator" or "Unit Test Executor"? How much are these people costing?
- To deal with the administrative hassle of running offshore development, offshore suppliers are often top-heavy with various middle-management layers. These middle managers need to be billed to the client, somehow. In many situations, economies of scale from offshore software engineering are an illusion.
- Hidden travel costs, where <u>users</u> are required to travel to the offshore site, explaining requirements and acceptance criteria. These costs are seen as "free", given that they come directly out of the business unit's own budget. They do not look "free" to users.

As soon as you start discussing these issues with users, you may be surprised at just how much of it they had already figured out for themselves. A small, in-house software engineering team (preferably co-located with the business unit that generates most requirements) is not only efficient; it is more responsive to users, as the various middlemen involved with offshore outsourcing disappear. Middlemen go out of their way to make sure that their margins are as non-transparent to clients as possible. The money that was spent on these margins can be put back into user requirements.

In 2005, Auerbach Publications published a book titled *Outsourcing Software Development Offshore* (following on from a 2002 conference). The fact that the subtitle of the book is *Making It Work* says quite a bit even before looking at the first page. The book makes various references to the "challenges" involved in offshore development and (despite the author's confident tone) can be read critically as a list of the risks associated with it. Really, you and your users need "challenges"? How much are these "challenges" costing? It can be a little embarrassing; pointing out that the emperor has no clothes.

About The Author

Gary Mohan is a senior enterprise architect, with experience working both for IT consultancies and end-user organisations. He has also worked for a mixture of private and public sector clients. Born in Northern Ireland and a dual British-Australian citizen, he is currently based in London and can be contacted at:

gary.mohan@plainprocess.com