# Chapter 3

### In which we consider ICT as a product

Test image 

In this chapter we consider the broad topic of how systems come to be. What sparks the need for a system or a system change? How and who makes the decisions on what systems get commissioned? How are system requirements decided? How are systems built? How are they put into place in an organisation? How are they cared for during their life-span and how are they retired?

This chapter is about more than building new systems. It is also about NOT building a system when a new system is required, but rather acquiring it through other channels. And it's about how organisations decide whether to buy, build, rent or commission the building of a brand-new system.

## What sparks the need for a new system in a typical organisation?

The on-line Merriam-Webster Dictionary defines *process* simply as "a series of actions that produce something or that lead to a particular result." Recall our discussion of a simple process in Chapter 1. *Input* **-->** *Process* **-->** *Output*. Simple and clean. When you stop to think about it, everything that happens in any organisation (and in your life) is part of a process. Something spurs action (you feel cold in your room - this is input) so you either turn up the thermostat, or put on a sweater or close an open window (this is the process part). From this process action, some output is produced (something is changed - more warmth or less cold but in the end, you feel more comfortable). The process *improvement* piece is another matter. I argued in chapter 1 that improvement is measured at the margins: either the input is less costly or less voluminous, or the output is better, faster, more plentiful. So in this example, an improvement could be in making the sweater easier to put on (less effort at input) or more insulating (better outcome or quicker time to value) or in the thousands of other potential improvements including automating the closing of a window or in better insulation or a more efficient hear source. At any rate, in this book, we talk about process improvement (progress) being measured at the margins where we measure input and output.

It's the same in an organisation. Something acts as an input (profits are falling; employees are calling in sick in record numbers; a competitor releases a new product or it's simply time for a strategic review - the ticking of the clock has led to a milestone being hit, triggering a review). This input bubbles through the organisation until it's recognised by somebody or some monitoring process, causing someone or some group to start mounting a response.

One of those possible *somethings that could be done* is a change to an existing system or the recognition that a system should be scrapped and/or that an entirely new system is required. This decision would be made using one of the decision models discussed in Chapter 4, where we also discuss strategic alignment. For now, let's just assume that a decision was made to respond to the input stimulus through the tactic of introducing (what is often called *sourcing*) a new (meaning new *to the organisation*) system.

## Sourcing a new system

What are the choices when sourcing a new system? There are several different variations, leading to eleven distinct possible paths plus (just to make a long story even longer) several possibilities for combining one or more of the eleven.

We will first make the distinction of whether the system will be: A) custom build using some combination of existing components and/or brand new code, or; B) if the system is sourced from the marketplace of existing software systems that any organisation could purchase or rent on the open market. Purchased commercial software is referred to as COTS (for *Commercial Off-The-Shelf*) software (such as Microsoft Excel), while rented software is referred to as *Software as a Service* or SaaS. A further wrinkle is produced in that *Open Source Software* (or OSS) can be had for literally no initial capital outlay. Furthermore, systems can be pieced together using components sourced from any or all of the possibilities, creating a *mashup* that defies categorisation.

So as not to muddy the already turbid waters, the four groups and the eleven total possibilities are described in general below, followed by a table outlining pros, cons and including a high-level determination of cost, time to value and overall quality.

Here then are the eleven routes:

**Figure ZZ. Software Form Factors: Eleven paths to a new system**

## Eleven paths to a new system - the software form factor

**A: Custom Developed Software (CDS)**

A1. Custom-built system built using the in-house expertise of a dedicated systems development functional area (so called *in-sourcing*).

A2. New custom-built system using expertise from outside the organisation (so called *outsourcing*).

A3. New custom-built system using in-house expertise and built by the actual future users of the system whose job is not systems development but rather in the functional area in which the system will be used (so called *self-sourcing*).

**B: Open Source Software (OSS)**

B1. Custom-built system using code “borrowed” from the open-source community on the internet and built using the in-house expertise of a dedicated systems development functional area (a form of *in-sourcing*).

B2. New custom-built system using code “borrowed “from the open-source community and built using the expertise from outside the organisation (a form of *outsourcing*)

B3 New custom-built system using code “borrowed “from the open-source community and configured/built using in-house expertise of the actual future users of the system whose job is not systems development but rather in the functional area in which the system will be used (while this scenario would be relatively rare, it is a form of *self-sourcing*).

**C: Commercial Off-the-shelf Software (COTS)**

C1. Purchasing a COTS solution and customising it in some fashion to better match the requirements of the functional area in which the system will be used. Think Microsoft Excel with some specific functionality programmed in macros or custom VBA code. The customisation in this case is carried out by a dedicated in-house systems development function (a form of *in-sourcing*).

C2. Purchasing a COTS solution and the customisation is carried out using expertise from outside the organisation (a form of *out-sourcing*).

C3. Purchasing a COTS solution and the customisation is carried out by the end-users of the system (a form of *self-sourcing*).

C4. Purchasing a COTS solution and implementing it with no changes whatsoever (think Microsoft Excel here -- everybody gets a copy on their desktop with the management directive "Now go to it!")

**D: Software as a Service (SaaS)**

D1. SaaS involves essentially *renting* software from a software service provider. If the organisation wants a big enterprise system and doesn’t want the cost or responsibility of buying, installing and maintaining its own copy of the software, said software can be *rented* and provided over the internet. There is little or no customisation here. You take what is offered. If any customisation is available, it's done on a outsourced basis by the service provider.

As you might imagine, there are pros and cons to each approach, a summary of which appears below. The cost, time and quality assessments are made using graphical *thermometers*. The higher the green temperature (for quality), the higher the general quality of the solutions in this category. For cost and time, the higher the temperature, the more costly and time-consuming are the solutions in this category. So ideally, we're looking for the least amount of red and the most green. That's the ideal solution, but *perhaps not the optimal* in every situation. To get very high quality, for example, you might have to pay and wait... but it might be worth it.

**Table X: Sourcing a system**

| Source type | Advantages | Disadvantages | Assessment |
|---|---|---|---|
| | By far the most costly option, but you get what you pay for. CDS provides systems that are specifically optimised to deal with exactly the challenge or opportunity faced | The old adage that “nothing is free” is nowhere more | |

| | | | |
|---|---|---|---|
| A. CDS | by the organisation, and are built upon (hopefully) rigorous analysis, testing and monitoring in ways that are specific to the situation. The organisation isn't forced to shoehorn into a one-size-fits-all solution and compromise functionality, flexibility and efficiency for the sake of cost. Furthermore, the evolution of the software is under the organisation's control. The asset value of the intellectual property (IP) embodied in the code is internally owned and thus controlled. Finally, the organisation retains full control of any data that is implicated in the process. Plenty of upside here. | appropriate than here. While surgical solutions to critical issues are the ideal for every organisation, they come at a price. Here, the price is the two-headed monster of real dollar cost and a long time to value (a metric that indicates the elapsed time from spend to reward). There is a risk of obsolescence as well and, as the organisation owns the system, it must maintain it. When you own the stack you are subject to all the risk associated with the development right through to retirement. Finally, all tech and user support is your responsibility. This can be expensive. Big bucks and a lot of time are the main drawbacks here. If cheap and quick are important, look elsewhere. | |
| B. OSS | An increasingly worthy consideration in the systems development mix is the use, whether exclusively or as one of more components of a solution, of open source software. OSS is software that is made available *free from initial cost* by developers in the development community. The code is free to *fork* (copy), modify (create an organisation's own local version) and implement. Knowing what we do about the time value of money (it's better to have your money now than at any time in the future) and especially if capital availability is low and/or the cost of capital is high, this is indeed an intriguing opportunity as there is little or no cash outlay at onset. Solutions can be built from existing code or modified as required. Code (at least initially) is supported by a community of developers, providing many eyes on the implementation (but this can be risky as many eyes can also see potential vulnerabilities). Another potential advantage lies in not being tethered to a particular vendor such as SAP or Oracle or Microsoft. OSS is open, and open means freedom to choose. | Again, nothing is free, even free stuff. Gartner (accessed February 12, 2015) maintains that a sobering 92% of the total cost of systems ownership is accounted for by the maintenance phase (the period during which the system is in use) *ergo* the initial savings are not as enticing as expected, especially when trying to do maintenance on code that was not written from scratch to be maintainable and is not supported by a software vendor but rather by a vague *user community*. Moreover, like COTS solutions (see below), such generic code, if unmodified, confers no competitive advantage on the organisation. If it's a strategic priority to use a system that is tailored to a particular value-generating process, then OSS is not the way to go. In addition, the issue of solution integration must be addressed. A system can't simply be dropped into place and begin to work seamlessly in the organisation. Larger commercial vendors such as Google or Microsoft will provide such integration service as part of the development costs if you hire an integrator to produce and install a CDS. With OSS, it's on you to make all the pieces of the organisation perform like a symphony. Penultimately, there just might not be any OSS available to address the challenge at your organisation. There's no guarantee that anything will fit, and forcing a solution on a problem just because it's the best solution available poses risks similar to buying COTS (see below). Finally, mixing and matching OSS with other solutions such as CDS and COTS might raise governance issues with open source licensing. *Beware and read the agreements carefully.* | |
| | By far the least costly option, and that's always a valid consideration. But paradoxically there's a price to be paid for being cost conscious (see disadvantages). Additionally, it's likely that industry *best* | Given the nature of such software (commercially available, generic systems to solve generic problems), it is unlikely that an organisation's challenges would be efficiently or effectively addressed by such an all-purpose tool. Think Excel. Great tool, tons of power, massive scope, not at all specific to an industry or a functional area. You can do some dentistry with a pair of pliers; question is, would it be efficient or effective? The organisation buying COTS also exposes itself to the risk of vendor viability. What if you buy a system from XYZ Co. and suddenly they go bust? Or are bought out and all contracts up for renegotiation? Additionally, much like OSS, a solution might not exist right | |

| | | | |
|---|---|---|---|
| C. COTS | Additionally, it's likely that industry best *practices* (efficient and generally accepted, near universal ways of doing things) are adhered to and enforced in commercial software. This should be encouraging to the operations people at your organisation. Finally, it's likely that the technology is up-to-date. | off the shelf. So your problem could become framed in the context of the available solution. (For example if a specific solution for your problem is not available, you might find yourself picking a solution to a similar problem because there's a solution available to solve that *sort of the same* problem. Think of having a piece of wood that you need to cut into two, equal-length pieces. Obviously a saw would work best. However if a saw is not available but a hammer is, the job can be done, but with considerable expense of time and effort and with a much less accurate outcome.) This is obviously not optimal. Finally, retaining competent developers on your staff will likely be difficult. After all, if you're buying software off the shelf, why would you need developers? Developers develop. Buyers buy. Different people. | |
| D. SaaS | This is a fairly expensive proposition, despite there being no development costs at all. The organisation will pay a price for essentially transferring all the application risk to an outside provider, which houses and maintains the software at remote locations. There is no need for the organisation to have local hardware or software other than an internet connection and display devices for users. System maintenance (upgrades, bug fixes, etc.) are all the responsibility of the service provider. Service Level Agreements (SLAs) provide guarantees of uptime and availability and the hefty penalties attached to such agreements guarantee that the organisation consuming the SaaS are protected from loss of business arising from unforeseen circumstances (such as outages, system failures, etc.). SaaS offers a very short time to value, includes codified best practices and service providers provide state-of-the-art, up-to-date software, made available on all platforms and devices (from mainframe to smartphone). Finally, the software is available anywhere you have internet access, so distributed systems and a scattered workforce (such as the cottage in summer, for example -- oh joy!) are all supported. | Of all the software form factors, SaaS has the highest vendor viability risk. Putting all your eggs in one basket with one vendor can be dangerous. Much like COTS, what if the vendor (service provider) were to suddenly close its doors or go offline? The risk of data lock-in is also high. There is little opportunity to specify the optimal data model (data architecture) for the organisation. You take what you get. This also impinges on data confidentiality issues – how does your organisation protect its IP in terms of its own data when the data is processed and stored off-site? Finally, usage-based pricing could become costly over time if the organisation's use of the software scales up and is locked-in to the SaaS solution. It is expected that SaaS will grow rapidly in proportion to other forms of system acquisition. | |
| 1. In-source | More likely to meet user requirements since the organisation's developers, who will build the system in the case of CDS or configure/modify it in the case of OSS and COTS, are familiar with the organisation's business model and processes. Your organisation owns the code and the solution. | On the downside, it's costly to support such a systems development function and unless the organisation is itself a software house, systems development will not be a core competence. | |
| | | Less likely than in- or self-sourcing to meet user requirements as the requirements must be provided to an | |

| | | | |
|---|---|---|---|
| 2. Outsource | Frees up the organisation to focus on its core competence. There is more certain cost control (through performance metrics and contractual obligations) and external technical specialists are likely to use state-of-the-art tools and procedures along with industry best practices. Finally, systems development houses have well-trained staff as technology solutions are *their* core competence. | outside team with no particular expertise or knowledge of the organisation's business model or value chain. This route can be very costly and time consuming. The organisation has no control over the external entity in terms of its survival, potential sale or even going out of business, thus exposing the organisation to significant vendor risk. Moreover, the advantage of doing your own technology in-house is lost to the organisation. Penultimately, this development route is less likely to produce a sustainable competitive advantage as the driver is external to the organisation. Finally, unless specifically guaranteed by contract, the firm that creates the solution (that's the outsourcer) also owns the code – so the IP is lost. | |
| 3. Self-source | This route is very likely to meet user requirements as the end-users – who are actually creating a solution based on their own needs - are quite familiar with their own processes. Furthermore, the organisation owns the code and the solution, thus the IP remains in house. | Very costly in terms of diverting attention and resources away from the actual work of the end-users (they aren't doing their real job if they are developing a system). Often there is no attention to organisational standards in terms of software tools, protocols, connectivity to larger, enterprise-wide systems or, especially, **security**. Systems development is not a core competence of functional specialists in, for example, the Accounting department, thus systems built by those whose training is in Accounting or Finance or Marketing will not be optimal. Finally, to reiterate, system-wide connectivity, security and privacy are often the most serious issues. And heaven help the organisation if the employee who did the work were to leave for any of the various reasons that people move on, to say nothing about sabotage perpetrated by a disgruntled employee on whose home-made system the organisation has become dependent. Documentation is almost never attempted, let alone completed, in self-created systems. Finally, as Abraham Maslow famously wrote "Give a small boy a hammer, and he will find that everything he encounters needs hammering." [Interested?](Abraham H. Maslow, 1966. The Psychology of Science. p. 15.) Re-written in its more familiar aphorism form, we often see or hear "If the only tool you have is a hammer, everything looks like a nail." If the functional analyst building the system is familiar with Excel, then Excel will be her hammer and will be the solution to everything, whether or not it's the best tool for the situation. | |

## Buy or Build?

We will consider *Rent* to be a subset of *Buy*. That being said, let's start with the *Buy or Build* decision. That's the distinction between commissioning a brand new system using either the in-, out- or self-sourcing option, compared to the decision to buy or rent an existing commercial product and either customise it or not.

We begin by posing the question "Should we start from scratch here, rent something or should we buy off the shelf?" There are actually two distinct ways to look at the *B-or-B* question in terms of what is described above in the pros and cons table. We can consider the broad *Buy* category to be comprised of buying or renting anything from outside the organisation, whether it's a custom, ground-up application or an OTS solution that is either customised through outsourcing or not customised at all. So this one broad category involves not using internal resources in any way.

The other broad category is, obviously, the opposite -- doing everything in-house, whether building from scratch or customising a COTS system using the organisation's resources and personnel.

A useful tool to understand the context of this choice is the Outsourcing Decision Matrix. This is a strategy tool, useful in a wide variety of situations

but not necessarily in the B-or-B software systems arena. We'll use the tool to give us an appreciation for the decision process in B-or-B.

The Mindtools website (see the Interested? link below) poses a series of questions much like these: "Which activities should we outsource, and which tasks should we do in-house?" For instance, imagine that you are in the health care industry and let's say you are the administrator of a small, rural primary-care hospital. Should you outsource your cleaning staff, or hire in-house cleaners? Would the decision be the same for a furniture manufacturer in a large, urban setting? If you worked for an airline, would you outsource your in-flight meal preparation, or would you hire cooks directly and maintain kitchens? What if you managed a luxury hotel?" As I write this chapter, I have just returned from my local *Ikea* store (unpaid plug). My wife and I made a stop in the cafeteria (shopping at Ikea makes for hungry work) and I found myself wondering if the cooks and serving personnel providing the meals were Ikea employees or outsiders occupying Ikea-owned space and using their own or Ikea's kitchen tools to provide service under contract to Ikea. You get the idea.

So do you do it yourself or get others to do for you? This is a good question indeed. Interested?

Before we launch into examining this important question, consider that all modern organisations are struggling to become *digital, data-driven organisations* because, well, that's the hottest thing in the near future. Gartner had the following to say early in 2014 about the current trend to outsourcing application development to integrators (Deloitte, Accenture, etc.) and large software houses (SAP, Oracle, IBM, Microsoft, etc.). They say that senior management should at least think twice about giving up in-house application development expertise. Here's what they wrote [I have *highlighted* the important points and added commentary as well]:

"The discipline of application development involves methods and technologies for custom application development. Traditional enterprises often treat internal application development groups as cost centers to be optimized. This type of *traditional internal development organization often struggles to support business needs under the pressure of limited budgets, skills and technologies.* System integrators and consulting firms threaten the existence of the development organization *because they can often deliver better applications, more quickly and more inexpensively*. [So you can see the allure to the cost-conscious executive committee.]

"But enterprises pursuing the digitalization of their business often find that *application development is a competitive differentiator too valuable to outsource*. Digitalization almost always requires new, unique applications. The applications and the staff that build them *can become valuable competitive assets and valuable intellectual property that are best protected in-house*.

"Transitioning application development from a cost center [like paying for such things as photocopying service, for example - it's just a service that needs to get done] to a business catalyst [something that can become a competitive differentiator and actually add value for the enterprise] is not easy, but is crucial for every enterprise looking to keep pace in the digital age."

Interested?

**Figure x. Outsourcing Decision Matrix (symmetrical)**

Figure X shows the matrix in a symmetrical configuration. The idea of this tool is to envision a process or activity undertaken or proposed by a particular organisation and decide what to do with it. We get a measurement on the two dimensions (Contribution to Performance and Strategic Importance) and then plot the process on the grid based on the values of the two dimensions. Simple.

We will use as illustration here a strategic review process with which we should all be able to relate: that of a University considering the value of an existing academic programme. Universities are increasingly numbers driven (bums in seats) as government funding is increasingly tied to enrolment numbers. *Ergo* schools are required to be agile and to engage in continuous programme review.

Let's say that the programme under consideration is a degree in Airborne Fulfillment Logistics – better understood as *delivering stuff using drones*. The school in question currently offers the programme as a joint initiative of the Aerospace Engineering department and the Supply Chain Management people in the Business faculty.

Now we need to work towards deciding if and how to proceed with the decision of what to do with this *drone degree*. This is where the tool comes in handy.

**Figure Z. Outsourcing Decision Matrix with proposed process**

As a high-level illustration, let's assume we've got our two metrics (we will discuss these two metrics in detail below but for now, just play along). Now we need to locate the degree programme in the decision matrix based on its measurements on the two axis variables to assist in making the decision on how to proceed. Figure Z shows the metrics on the degree (little blue box), locating it in the upper, right quadrant of the matrix based on it being assessed a score of ~75% on both variables. This locates the process squarely in the "Retain" quadrant, meaning that it's important to competitive advantage and to organisational efficiency. So the school should decide to retain the programme as is. Now, some detail.

Upon measurement of the two dimensions, the programme under consideration here will most likely land in one of the four quadrants, (this is guaranteed if, in measuring the two dimension variables, a value of exactly 50% on both is avoided). Note the outcome (strategy) that is represented by landing in a quadrant (clockwise from top left we have: Strategic Alliance, Retain, Outsouce and finally Eliminate). Let's look in more detail.

First, examine the horizontal and vertical axes, which measure two important variables and range from Low (0%) to High (100%). The vertical axis represents *Strategic Importance*. A process is considered strategically important if it impacts competitive advantage. Recall from Chapter 1 that competitive advantage is a superiority gained through providing the same value (usefulness or problem-solving ability) as its competitors but at a lower price, or increased revenue through charging higher prices supported by providing greater value through differentiation. Definition (accessed February 26, 2015). Also note how close this is to our dictum of progress being defined as *less input* or *more output*.

In our example, if administration determines that the strategic importance of offering a drone programme is high (administration thinks that it provides an edge in attracting not just good students but specialist faculty and research dollars and that ever-important intangible *prestige*), then they would rank the drone initiative above the 50% level on importance. This signifies that administration feels that, on balance, the programme is a net contributor to fulfilling the school's strategy. The discussion of exactly how this would be measured is beyond the scope of this text, but must be done in the real world in order to use this tool.

To move forward, let's assume we have some measure of the strategic importance of the programme on a scale from 0 to 100% where higher is more strategically important.

So now we move on to the horizontal dimension, where the impact of the process on operational performance must be evaluated, again on a scale from 0 to 100%. This variable measures the impact on the organisation of the efficiency of the programme: if a program is operationally important but runs poorly, it will impact the organisation in a negative and important way. So the systems involved in supporting and running the programme itself must be efficient and effective.

Think of the process for fuelling a fleet of buses in a large metropolitan city. If unreliable, slow or otherwise inefficient, the integrity of the city economy could be jeopardised as nothing would run on schedule. On the other hand, a system with little operational significance will not materially affect the organisation's performance one way or the other. Think of the process or washing the exterior of those same buses. Clearly whether a bus is clean or not is not nearly as impactful as having a full fuel tank.

Back to our university example. Management must come up with a metric to measure the impact of the degree programme on the overall efficiency of the university. Considerations include whether facilities to house faculty and administration, offices, classrooms, labs, examination rooms, etc. are at a premium or do not exist. In addition, the average salary of professors in the field (professors in rare fields can attract better compensation than others) is high, then this might be factored in. If the programme is more expensive, pound for pound, than other programmes, then it might be decided that overall operational impact might be negative. If, on the other hand, significant synergies (the whole being of greater value than the sum of the pieces) are being achieved and/or if space is being efficiently utilised and perhaps if other faculties are finding ways to leverage the drone group, then the scale could be tipped in favour of the degree programme.

These are complex issues, but here again we assume that a metric exists to measure such impact.

Let's look at what would happen (what strategy would be followed) from the strategic review according to various hypothetical score combinations. First, a high score on Strategic Importance, coupled with a high score on Contribution to Performance would locate the programme squarely in the Retain quadrant in the upper right. In this case, the university would decide to not only continue offering the programme, but to offer it on campus with full staffing and facilities, keeping it all in house. This programme is a star.

On the other hand, scores coming in low on both dimensions (less than 50% on both measures) would locate the programme in the Eliminate quadrant, making it a prime candidate for elimination altogether.

These are the two extremes. Let's say the Operational Performance metric comes in at 75, but the Strategic Performance comes in at 25. This would represent a situation where the programme is efficient and contributing to overall performance, but was not measuring up in terms of the strategic direction of the school. Say the school has a strategy to be the preeminent university in Arts and Philosophy. A highly technical programme in remote drone supply chain fulfillment might not be what the school wants to be known for. So maybe, just maybe, the programme gets cut. Or maybe, just maybe, the school can outsource the delivery of the programme to the private sector. There are any number of private educational training firms out there. The school might want to keep the programme on campus, retaining the operational synergies and the capacity utilisation, but outsource the delivery to the private sector. This would be a difficult sell for faculty. And not likely to fly at all... but the times they are a changing.

The final possibility is the opposite of the above, where Strategic Performance is above 50%, but Operational Performance is below. So the school likes the programme because it fits with its overall strategy (maybe it wants to be a high-tech hub), but the programme is just not operationally feasible. In this case, a Strategic Alliance might work. The school could look to partner with another local university to deliver the content, perhaps offering to take the operations of an under-performing programme at the partner school in a trade.

When deciding on an Information System however, it's unlikely that a decision would be made to enter into a strategic alliance with a system solution provider or especially a competitor. Such alliances are formed between two or more organisations in order to solve a particular problem or to take advantage of a unique, perhaps non-recurring opportunity, where all parties to the alliance would benefit.

But before we throw away this possibility altogether, let's consider the scenario where the current market is small, but has the potential to grow. Remember that there are two ways to increase revenue in a market: A) take a larger share of the current market by attracting the customers of your rivals, or; B) grow the overall market. In the latter circumstance, partnering with a competitor on strategies that grow the market might not be a bad idea.

So the organisation could partner with another firm in the same market to *share* the development and ongoing costs of a system that would benefit both parties. This isn't as far-fetched as you might think. Consider the telecommunications marketplace. In the early years, it was not uncommon for competitors to cooperate on the building of cell tower infrastructure to expand cell coverage. All players benefited from the expanded availability and then competed on features and price to attract customers from each other to build their market share.

Interested?

While a strategic alliance is not likely to happen in the case of a B-or-B decision, this shouldn't dampen our enthusiasm for using the tool. It's still quite valuable as it illustrates the important considerations that need to be taken into account and highlights trade-offs that need to be made when deciding on how to commission a new system.

That's how it works in the ideal world. The real world is a bit messier and more complex.

What are the issues? First of all, we have chosen arbitrary cut points between decision outcomes, located at 50% on each scale. This is aesthetically quite pleasing (yielding four, nice, equally-sized quadrants), but it unlikely to represent reality in all but the most unusual cases. More likely is the organisation having different cultures, priorities, strategies and practices, thus requiring different cut-points.

**Figure Y. Decision Matrix examples with different variable cut points**

Examining Figure Y, we see organisations imbued with different realities. A particularly lean organisation would have a very small upper, right quadrant (the Retain area), preferring to outsource, partner or eliminate all but the most essential of processes. The cut points for both *strategic importance* and *contribution to efficiency* could be moved up to 75% or even 80 or 90%. Such organisations are sometimes referred to as *virtual organisations,* existing as only a core set of processes with no real physical space, having outsourced, partnered-off or cut everything not considered absolutely essential.

The recent and astoundingly swift rise of Web 2.0 communications and collaborative technologies, along with e-commerce, secure credit card transactions and teleworking, coupled with the rapid rise of the service sector, have led to the rise of such *virtual* (not bricks and mortar) organisations. This is how Amazon started way back when.

Don't confuse this sort of virtual organisation with a *virtual corporation*, which is really a form of strategic alliance. Some sources muddy the water a bit on this.

Interested?

Let's then take our new-found expertise in decision making and translate it into a more difficult situation which, while sharing some similarities with the outsourcing decision challenge, has more and varied inputs to the process.

To make the decision about how to commission a new system in an organisation, a series of determinations need to be made. The *consideration* column can be seen as the *characteristics* of the system or of the organisation considering the system.

**Table XX. Buy-or-build considerations**

| Consideration | Build if | Buy if |
|---|---|---|
| 1. System size and complexity | The requirement is for a small, simple,*ad hoc* system | You need a big, complex system, or if an adequate COTS solution exists and is cost-effective |
| 2. Strategic necessity | Not strategic and/or limited operational impact | Strategically important and/or significant operational impact |
| 3. Timeline | Small with plenty of time to get it right | Medium to large with a tight delivery timeline |

| | | |
|---|---|---|
| 4. Uniqueness | Unique or proprietary or might expose a trade secret or competitive advantage | Ordinary, garden-variety with no secret sauce |
| 5. In-house talent | Have a dedicated, well-funded and persistent systems development resources | No real competence in development |
| 6. System footprint | Small, restricted impact on a single function or process | Impacts a larger number of functions and/or needs integration with an enterprise system (such as SAP) and/or must be scalable and robust |
| 7. Lifespan | Short | Long |
| 8. Compliance | Not required to be compliant with external agencies | Requires external compliance such as SOX or Bill C198 among others |
| 9. Stability | Industry, market, competition and business practices are stable and glacial | Industry, market, competition and business practices are volatile |

There are plenty of compliance standards, and many agencies (especially governmental) have their own set of rules. [Interested?]

There's lots to consider when deciding to B-or-B. [Interested?]

# What makes a quality system?

Regardless of the decision to either buy or build, there are certain imperatives in system building. Let's examine them now.

Quality is an aggregate function of how well a system meets each of the following characteristics. Each is important in isolation, but the relative weight of each factor in determining overall quality is context dependent (and isn't *everything*?). Not to put too fine a point on it, but in one context at a certain point in time for a certain organisation, *efficiency* might take precedence over *security*, say. At another point in time, in another circumstance, the reverse might be true. So the relative weights of the system quality factors below can change over time, both within and between organisations and are sensitive to context.

Here are the factors:

- **Exhaustiveness** — A system is exhaustive if it *addresses all requirements* specified by analysing existing systems (if extant) to discover shortcomings, by examining the organisation's needs including their strategies, and by polling potential system users in the *Requirement Analysis* phase of a proposed systems development project (see later discussion on SDLC and others).
- **Reliability** — A system is reliable when it operates in *predictable and consistent fashion* no matter the demand load placed upon it. It is said to be *scalable*, both up and down, if it robustly responds to changes in demand such that there is minimal impact on other important metrics (such as accuracy and efficiency). "It takes a licking and keeps on ticking" to borrow a 1950-1960s and again in the 1990s *Timex* watch ad in which wristwatches survived various staged torture tests. Systems must have the capacity to handle peek workloads while ideally being able to scale back when capacity isn't required so that the organisation isn't paying for unused capacity. And they must scale without sacrificing reliably.
- **Accuracy** — A system is considered to be accurate when it produces *predictable and verifiably-correct outputs* when executing its required functions. So 2 + 2 is always 4. This requirement speaks to data and process integrity. See the discussion about accuracy in Chapter 1 of this text.
- **Efficiency** — A system is efficient if it produces outputs that are *more valuable than is the cumulative cost of the required inputs* to the system. The greater the spread between the value of outputs and the cost of inputs, the more efficient is the system. This is the basic and most important test of the value of ICT. A system must alter the relationship between inputs and outputs in a positive way by either shrinking required inputs or growing the volume or value of outputs. The spread between the cost and the benefit should be positive, non-zero and subject to continual scrutiny and improvement.
- **Security** — A system is considered secure if access to the system itself, to any required inputs, and in some cases the system outputs, are *protected from unauthorised access and tampering*.
- **Usability** — A system is considered to be usable if it allows users to complete their work with *a minimum of error, wasted effort or frustration*. This obviously impinges on the characteristic of efficiency. Much thought and effort has gone into the area of user experience design (or UxD) in the system design community of late (more on this later in this chapter).
- **Maintainability** — A system is maintainable if its program code is both written according to accepted industry standards and is well documented such that *errors can easily be detected and corrected* (see transparency below), and new features can be added (or existing ones deleted) with minimal impact on the metrics of other important system characteristics (such as efficiency). You must be able to fix what's broken.

- **Transparency** — A system is transparent if it is designed in such a way as to allow *metrics on its other characteristics to be tested and gauged*. Thus a system should afford testing of reliability, accuracy, etc.
- **Availability** — A system is considered available if it is *online and readily accessible* to do the job for which it was designed.
- **Recoverability** — A system is recoverable if it can be brought *back online quickly* and with minimal data loss following a system outage caused by any type of problem. This requirement demands that a robust *disaster recovery plan* be in place and enforced.
- **Interoperability** — A system is considered to be interoperable if it *plays well with others.* In other words, it fits well into the existing infrastructure and is a seamless player with whatever other processes or systems of which it is a part in the organisation. A good system should be virtually transparent – just quietly goes about doing what it is tasked with doing.

### What's at stake?

Large organisations spend a lot (a LOT) of their resources on systems development, training and maintenance. How much? One study published by McKinsey claimed that organisations spend around 50% (some report up to 60% in the US) of their total IT budget on application development. But even given the big bucks spent, "... the quality of execution leaves much to be desired." Furthermore, they offered that "A joint study by McKinsey and Oxford University found that large software projects on average run 66 percent over budget and 33 percent over schedule; as many as 17 percent of projects go so badly that they can threaten the very existence of the company." Yikes!

[Interested?]

## Building a system?

To the quality metrics introduced above must be added and separate but related dimension -- that of *timeliness of delivery*. Often, organisations face problems that demand immediate solution or are presented with opportunities the window to which is fleeting and fast closing. So solutions need to be timely.

A McKinsey/Oxford University study reported that "... 71 percent of large IT projects face cost overruns, and 33 percent of projects are around 50 percent over budget. On average, large IT projects deliver 56 percent less value than predicted."

So it's clear that there are real challenges in system development and in the related field of Project Management as it impacts large software development (more on this later). Clearly the software development community is aware of their quality and timeliness issues. More needs to be done, however, to improve on the underlying dimensions that give rise to these critical metrics.

How then is development actually accomplished? Whether the decision is to in-source or outsource, there are several paths that systems development can take. The general model of *systems* development, of which *software* development is a specific sub-species, will be discussed at length below. Here is the *Software Development life Cycle*.

Figure TTP. Software Development Life Cycle



The Software DLC illustrates the various stages that organisations go through in meeting their challenges or taking advantage of their opportunities. Like any good process, the software development process begins with a kick - something spurs it into action. In this case, it's a business requirement. Something bubbles up in the environment that leads to the decision to implement a system. So *business requirements* drive the process of determining *solution imperatives*, which, loosely translated, mean *what the system must do* in order to meet the requirements. This is still fairly high level. These imperatives, in turn, demand specific *solution features*, a refined set of properties that the solution must contain. From this set of required features, we can derive more specific *software requirements* which then allow very specific properties to be *designed*, then *coded*, *tested* and finally *implemented* in the organisation. Over time, changes in the environment (the context) will drive new business requirements which will drive the need for modifications to existing systems or entirely new systems. Such is the general *circle of life* n the business software arena.

Once we get the gist of the process, one of going from the very general need through a series of steps to refine and translate those needs into working code, then finally install the system and use it, we can move on the various ways in which this general model is implemented specifically.

## Systems development methods

As systems development evolved from an art to a science and increasingly came under the scrutiny of project management (seeking predictable structure) and cost accounting (focusing on ROI), methods were needed to ensure that systems were delivered on time, to specification, were efficient, accurate and maintainable, thus providing the longest possible period of trouble-free operation and thus a decent return on the significant investment put into them.

Indeed, methods were created, rigour was added and metrics were devised. The basic steps in systems development do not vary much from

method to method. Their arrangement, sequencing and whether they can be revisited once complete (or indeed if they are ever completed at all) is what creates the various flavours of systems development. It is on a tour of many such methods that we now embark.

There are two-and-a-half main *camps* in system s development. The first is referred to as *prescriptive*, meaning that what comes next is set in stone. When your physician writes a *prescription* for you, it's not to suggest that you might want to think about doing this or that. It's an order. Take this medication this number of times a day and with this food. Avoid this substance (alcohol usually) while on this medication. Do not double up if you miss a dose. And so on. These are strict rules to follow. Such is the predictive nature of the Systems Development Life Cycle (note this is *not* the *Software* development life cycle. Slightly different beast). This method is the first main type of challenge in an organisation - one that can be scoped out from beginning to end and all that is required is a set of rules and the tools to follow.

The *half camp* exists in the region between the SDLC and the Agile methods (below). THis is *prototyping. NOt quiteprescriptive and not quite adaptive, but shading towards the latter.

The second camp is the *adaptive* methods, of which there are several, under the *Agile Programming* umbrella. Adaptive means that nothing is completely set in stone, or *prescribed* as in the SDLC-type situation. In situations where the problem is not well enough defined, the rules cannot be set in advance but rather must be *discovered*. This is the second main type of challenge and demands a different approach to the creation of systems.

We will look at both, but first, the progenitor.

## The Systems Development Life Cycle (SDLC)

The first ever formalised method was the Systems Development Life Cycle, a so-called *waterfall method*.

The SDLC is referred to as a *waterfall method* since its steps resemble a waterfall; in order for water to reach the bottom, it must pass along the entire span between the top of the falls (system conception) to the bottom of the falls (system retirement). The water can't skip any of the distance along the way. Equally, once the water has reached any certain point, there's no going back. It's all one way, and the only way is to finish the plunge to the bottom.

Figure ZZ shows the Systems DLC (as opposed to the Software DLC introduced earlier (take me there)) illustrating the seven phases and accompanied by a whimsical estimate, through creatures, of the cost of finding and fixing errors or having to add new requirements at the various stages of the cycle. Image, if you will, the associated animal appearing suddenly, unannounced, in your home. What would it take to remedy the situation? The fly is a simple nuisance and nothing has really been expended on development at the *inception phase* of the SDLC. Plenty of ideas are thrown around and nothing is yet committed. So in the inception phase, you swat the fly. Image the structural havoc to your home of having to deal with a 6,500+ kg. bull elephant in your kitchen.

That’s the order of magnitude of issues faced by software developers who use the SDLC. The US space agency NASA estimates that, on average, it can take upwards of 100 times the resources to fix an error discovered after delivery (implementation) than if the error were discovered in the requirements or early design phase of development. Some estimates are up to 1,000 times.

Interested?

In fact, risk remains higher in SDLC-type (waterfall) projects than in agile-type (iterative) projects until the very end of the process. This is at least *partly* a function of the method itself, and partly a function of the type of project that matches with the method. Figure BCR shows the risk as measured in a procurement environment systems development *milieu*.

**Figure BCR. Risk comparison between SDLC and Agile development.**

We can clearly see that risk drops steadily for iterative method projects, while remaining almost unchanged until well into the construction (coding) phase for SDLC projects. This is strong evidence for looking carefully at how you develop systems.

Interested?

**Figure XCB. SDLC Phases**

Let’s take a quick look at what happens during each of these eight phases of the SDLC.

| Phase | Activities | Cost of |
|---|---|---|

| | | errors |
|---|---|---|
| Inception & feasibility | This is where the idea for a new system bubbles up from the organisation through innovation, error detection or challenges arising from competitors or regulatory or market changes. The possibility of a new system arises here where it gets a preliminary scan (see Chapter # where we discuss governance and decision making) and the general parameters of the organisational response are set. This is a gateway stage. The questions at this stage are “Here’s the situation. Can a system help us here?” Approval is critical. A no-go means end of discussion. | |
| Requirements analysis | Given a preliminary go at the inception stage, analysts of both types (business and systems) begin to gather the requirements for the system. What problem is the system supposed to solve? Who will the users be? What inputs are required and what are the expected outputs? Fairly broad, general requirements lead to quite specific details as analysis and fact finding progress. Two types of requirements are gathered in this stage: Functional (what the system must do), including: business rules, transaction corrections and adjustments, administrative functions, authentication, audit tracking, external interfaces, certification requirements, and reporting requirements, among others. The second set of requirements revolve around non-functional requirements (what qualities the system must have and to what standards is it subject - see discussion in this chapter entitled *What makes a quality system?*) Requirements here include: scalability, capacity, availability, reliability, recoverability, maintainability, serviceability, security, regulatory, manageability, environmental, data integrity, usability, interoperability and performance. Interested? | |
| Design | There are two distinct types of design occurring in the design phase: Logical and Physical. In the logical design sub-phase, analysts are concerned with high-level models and abstract representations of data flows and the inputs and outputs of the system. Data might just be referred to as “data” with no attempt to describe the characteristics at all. This phase if conceptual, as in going no deeper than the “concept” of a system and what it might broadly be tasked with in terms of the requirements gathered in phase 2. This is the “what” of system design – just broadly *what* a system will do, but not *how* it will be accomplished. The physical design activity, on the other hand, gets into the nitty-gritty of exactly *how* things will get done. Data and storage and process and input/output details are considered down to the most minute detail. Tools that analysts use in this process include Data Flow Diagrams (DFDs) and Entity-Relationship Diagrams (ERs or ERDs) and Unified Modelling Language or UML (Interested?) diagrams.. This is where the system is *scoped out* in preparation for building it, a phase that requires exact detail. Details such as exact inputs, process and output displayed and/or stored are carefully specified here. All the external actors who or which interact with the system are defined here, and their behaviour modelled. It is at this stage that user interfaces are designed. This is one of the most critical and creative pieces of software design. See the discussion in this chapter regarding Ux design. | |
| Development & coding | In this phase, the design emanating from the previous phase is programmed into actual software. Hardware such as computers, servers and even satellites, if required, is purchased. Service contracts for things such as cloud storage are negotiated and initiated. Software systems such as commercial databases are purchased, installed and configured while code is being written to access them. Requirements are put into action. Coding is accomplished using a language appropriate to the target environment (mainframe, desktop, Windows, Mac, mobile, Android, iOS, etc.) using any one or more of the many, many coding languages available. In addition, one of the two types of documentation is produced here – *technical documentation*. This documentation details how the system works on the inside. It provides such detail as object models (if using object-oriented design), data flow diagrams (DFDs), UML diagrams (UML is Universal Modelling Language), ERDs (Entity-Relationship Diagrams and other technical documents which would inform those doing maintenance on the system when it is implemented. | |
| Testing & verification | The testing phase measures the actual versus expected outcome of the system. The outcome expectations are based on the system requirements elicited in the requirements stage. The goal of testing is to find and fix unexpected outcomes when actually executing the system as built in the design phase. There are numerous types of testing, culminating in *beta testing* (where sometimes the public is invited to use a system for free and to report bugs to the developers) and *acceptance testing* (where the actual users of the system, which is in production-ready mode, are tasked with giving their final approval, or acceptance). While the SDLC (as a waterfall method) is no given to iterative development (test, find, fix, test…), testing is often conducted in this fashion. So quite unlike other phases of the SDLC, where discovering serious errors necessitates moving all the way back to the feasibility stage and a restart, testing often uncovers minor issues in coding, for example, that can be fixed quickly and without resetting the whole project. A specific type of testing, in fact, is dedicated | |

| | | |
|---|---|---|
| | to this iterative process. This type is *regression testing* wherein the system is re-tested to make sure that fixes to previous bugs have not introduced errors in previously error-free code. The beat goes on. The importance of testing cannot be overstated. It is at this stage that *user documentation*, the other important type, along with *technical documentation*, is created. User documentation is all about how to interact with the system from a user perspective including how to install, troubleshoot, maintain and use the system such that your goals as a user are satisfied by the system. Interested? | |
| Implementation & integration | In this phase, the fully-tested system is put into production in the target environment. The choice of how to *transition* to the (often) new system is an important one. See Table ZZZ for a comparison. Additionally, the new system must often be integrated with existing systems and workflows in the organisation. This is especially, though not exclusively, important when implementing a COTS system. All the pieces need to work together in order to create value. | |
| Maintenance & Evaluation | In this stage, the system is subject to monitoring to ensure that it continues to create value and meet the expectations of the organisation and the users of the system. Small, incremental additions, deletions, fixes and improvements are made on an ongoing basis. Recall that Gartner has reported that 92% or the total cost of system ownership is accounted for by activities in the Maintenance phase. | |
| Retirement | Finally, as with everything, the end eventually arrives. When a system cannot be further patched or tweaked and has stopped creating value (for whatever reason), it's time to retire it and move on. Note the line in Chart XX points directly back to Inception and Feasibility. Time to start again. At this point, several critical tasks must be undertaken, including securing the input and output data, both current and historical, involved with the system. Retiring a system shouldn't retire the data associated with it. | |

And here's Wally's take on implementation:

*Source: http://dilbert.com/strip/2015-04-22*

## System Conversion - putting the new system to work

This process involves the replacement of an existing system with a new one. Think of the new system in the broadest possible terms, as a new system might also involve changes to infrastructure, processes, networks and even personnel. Such deeply technical details would not likely be specified by business analysts but rather must be discovered and specified by systems analysts in the requirements analysis phase of the SDLC. Regardless of when or whom, all requirements must come together and be satisfied in the implementation and integration phase.

Generally, the conversion phase applies only to the situation where an existing system is being replaced by a new one. An entirely new-to-the-organisation system would likely be converted with the *plunge* method (see below).

Let's take a look at the four most common methods. Figure RM below shows the methods graphically.

**Figure RM. Conversion alternatives**

Let's examine each in light detail. The metrics in the rightmost column represent subjective assessments of the Risk associated with the conversion method (the column with the little traffic cone at the bottom) and the cost of the method (with the little $ in the bottom of the column). Recall from our discussion of quality metrics in the sourcing section of this chapter, the height of the colour in the column represents the absolute top of the range for risk or cost, while the density of the colour (always more dense near the bottom) represents the likeliest value of the measure. To calibrate, note that the least risky method is parallel , but it's also the costliest. The polar opposite is true for the plunge method.

It appears that there is a negative relationship between risk and cost. And that's true in general. As nothing is free (even free stuff), a reduction in risk to the organisation will cost money. You just can't escape. Willingness to accept more risk will be less costly… if things go well.

**Table ZZZ. Comparison of conversion methods**

| Conversion Method | Description | Good | Not good | Risk/Cost |
|---|---|---|---|---|
| | | | When time is of the essence and the new | |

| | | | | |
|---|---|---|---|---|
| Parallel | Both systems run at the same time for a period of time. | When the existing system performs a critical function in the organisation and you can't do without the output it provides and/or when there is an intolerable level of risk that the new system might fail. | system provides something that the current one cannot. Also when there just aren't enough resources available to do both jobs at the same time (i.e., if there aren't enough staff to prepare a critical input file in two ways). | |
| Pilot | The system is incrementally implemented in subsets of the organisation. | When the organisation does not critically need the output provided by the new system and more importantly if the system is large enough to allow it to be phased in. If the organisation is large enough to warrant an enterprise-wide system, and the system will be implemented in many functional areas, then a phased conversion could be carried out in one functional area at a time (for example, first in Accounting and then in Finance and then in Sales, etc.). Another scenario could see some members of one functional area using the new system while others use the old one. Finally, if the organisation is large enough to have multiple locations doing roughly similar work (a series of auto manufacturing plants, for example) then the conversion could be phased in one plant at a time, or one country at a time, etc. This type of conversion lets the organisation gauge the success of the conversion with less exposure to risk than would be the case under Plunge conversion, for example (see below). | When either the reach of the system or the size of the organisation do not allow for enough scenarios. In addition, if all areas or personnel or installations of the company work in tight unison it might be difficult to find the opportunity to differentiate the work process. | |
| Phased | Different pieces (modules, functions) of the overall system are implemented at different times. | When the system is large enough and the pieces are different enough to warrant splitting up the system. | When the benefits of the system (or even the integrity) would be compromised by piecemeal implementation. Some systems are all or nothing. In addition, it could be quite expensive to repeatedly integrate pieces of a new system with existing infrastructure and processes, let alone train users many times as new pieces become available. | |
| Plunge | Lights off, lights on. Out with the old, in with the new. Retirement party on Friday. Monday we | This is good when cost is critical or when it's not operationally feasible to run two systems at one. Also when the output of the old system is not needed in its current form and moreover, if time is of the essence – you need the new outputs ASAP or if there is a drop-dead date for implementation that cannot be changed (funds might disappear, for example, if the new system isn't implemented before the end of the fiscal year). Finally, it's good in situations where it's relatively easy to return to the old | When you need the output from the existing system, or when it's not clear what the outcome of the conversion will be (risky). | |

| | | | | |
|---|---|---|---|---|
| | start the new system. | system should the situation warrant. | | |

Regardless of what you might think of the SDLC in terms of its viability for creating systems, the stark truth is that all methods (more of which we are about to discuss) must engage in each and every phase of the SDLC in order to produce a software system. It is in the emphasis, sequence, frequency, timing and duration of time spent on each phase that is the major differentiator between the various methods of systems development.

Proponents of the SDLC and other waterfall variants (there are fewer and fewer of them all the time) will argue that adopting these methods have the advantages of forcing systems developers to fully understand the system requirements before embarking on a development project. Additionally they felt that there were savings in time, effort and resources (including money) to be realised from this rigid, un-adaptive process. Change was not to be tolerated. Figure it out at the start, then build it.

But there is an old adage among developers: “Often users don’t know what they want until they see what they don’t want.” This is a compelling argument for the Prototyping Method, presented next.

## Prototyping

Prototyping is the *half method* and takes a different view from the SDLC. The principle behind prototyping is that it allows the eventual users of the software system to get early and then frequent insights into the current design of the system by allowing users to actually use the it before completion, rather than having to either read about system features and processes or, in the worst case, to wait until they are tasked with acceptance testing of the finished product. By then, for all intents and purposes, it’s too late. The elephant is in the room. So it's *adaptive* in nature, meaning that the system can change based on feedback well into the development and coding stage, but it's really only superficial *features* that change. Prototypes are not scrapped and the process sent back to square one.

Prototyping, as an iterative (built in smaller pieces and shown to the users for approval) process, can thus better protect against the potential catastrophe attendant upon *finding the elephant*. But make no mistake. While more unlikely given the repeated, iterative exposure of the system to its eventual users, there still exists the slight possibility of finding that pachyderm at the end.

Furthermore, prototyping is also often used to *try out* improvements to a system that were not part of the original specification but which *occurred to* the developers (or users) in the process of building. In this sense, it is much closer to the Agile methods we will discuss later. So the prototyping method is a hybrid, thus the *half method* label. For now, back to prototyping.

**Figure LJ. The prototyping process**

Figure LJ provides a pictorial overview of the prototyping process as it relates to software development. First, note that there are no more bunnies, cats or dogs as a cost of finding errors. Prototyping replaces them with mice because there's not as much at stake at each iteration. The exposed and under-scrutiny part of the system is smaller, the changes more easily made and the investment in development much smaller. This is an improvement over the SDLC.

Next, note that the red lines connecting steps 3, 4 and 5, unlike in the SDLC, can be repeated (or iterated) as often as necessary until the evaluation at step 6 branches to step 7, effectively exiting the prototype loop when the system is ready for implementation. The obviously pivotal step is #6, where in addition to branching to step #7, the outcome of the prototype evaluation can branch to either a refinement of the current design (step #3) or to a revisit of the system requirements if the prototype is introducing a refinement (step #2). So it’s really all about the user feedback. This is an important advantage for prototyping.

### The prototyping process for software development

1. Project inception and feasibility stage is a given. We don’t just start producing a system out of nowhere. There must be demonstrated need and management approval if working in a large organisation.
2. Perform a basic requirements analysis, but not as in-depth as in the SDLC. Often, the riskiest bits of the system are modelled early. If you can do the hard stuff, the easy stuff will fall into place. So we get the nitty-gritty details; the make-or-break stuff. Often, difficult but manageable details such as security are ignored in the early stages.
3. A prototype is developed, often including only user interfaces in the earliest stages. This is referred to as Horizontal Prototyping (discussed below). In later stages, Vertical Prototyping is used to drill down deep into the system to model the full functionality of a feature or required process.
4. The prototype at whatever stage it’s currently at is shown to the clients (end users) to elicit feedback.
5. The feedback provided in step 4 is used to revise and enhance the prototype. New features or screens are added in an incremental fashion and we return to step 3, continuing in this fashion until a deliverable system is produced.

Usability Engineer Jakob Neilson described the various types of prototyping in his 1993 book entitled Usability Engineering (Academic Press Inc.). Two are relevant to us in this context:

**Horizontal**

A user interface prototype is referred to as a horizontal prototype. Such a prototype provides an overview of a complete system or a significant subsystem, with an emphasis on how the user uses the system rather than how the system processes user input. Such prototypes are useful to confirm the logic and flow of user interfaces and setting the broad parameters of what the system (or sub-system) is expected to do. Furthermore, they can help in developing some metrics around anticipated time and resources required to deliver the full system. Finally, such prototypes often have the benefit of securing *buy-in* from decision makers at the organisation as they can see and more tangibly grasp the overall scope, and thus value, of the proposed system.

**Vertical**

A vertical prototype is an in-depth elaboration of a single process, subsystem or function. Such prototypes are useful in discovering detailed requirements, such as the “risky bits” of a system and to demonstrate that they can be accomplished. The benefits of vertical prototyping include determining details of data modelling (database design)and in getting a handle on processing volume requirements (how many transactions, or basic units of work) a system is required to be capable of doing, Further distinctions are made between Throwaway and Evolutionary prototypes. The former is just as its name imp[lies, and is not as often used in software prototyping as the latter, which retains its basic structure and functionality throughout the process to become the final system.

**Prototyping Benefits**

Prototyping has several benefits: 1. Valuable *user feedback* is elicited early and often 2. *Requirements can be more easily verified*, tweaked, added or dropped 3. Metrics around *cost and time are brought into focus* iteratively 4. Users are *much more likely to buy into the system* and support its use when it goes into production as they had a stake and a voice in its development – they feel ownership of the system because they helped craft it and accepted it at each stage 5. *Training requirements for users are significantly reduced* they have been exposed to the system often and have trained themselves on its use and indeed directed the development in part 6. Prototyping is especially useful when systems involve *extensive interaction between computer and user* (have extensive Human-Computer Interaction or HCI)

Prototyping also has some drawbacks. These include, but are not limited to:

1. *Insufficient analysis*: The focus on a limited prototype can distract developers from properly analyzing the complete project.
2. *User confusion of prototype and finished system*: Users can begin to think that a prototype, intended to be thrown away, is actually a final system that merely needs to be finished or polished.
3. *Developer attachment to prototype*: Developers can also become attached to prototypes they have spent a great deal of effort producing; this can lead to problems like attempting to convert a limited prototype into a final system when it does not have an appropriate underlying architecture.
4. *User attachment to the features of a prototype:* Users might become accustomed or attracted to features that were included in a prototype for consideration and then removed from the specification for a final system. This can lead to misunderstanding and conflict.
5. *Excessive development time of the prototype*: A key property to prototyping is the fact that it is supposed to be done quickly. If the developers lose sight of this fact, they very well may try to develop a prototype that is too complex.
6. *Expense of implementing prototyping*: the startup costs for building a development team focused on prototyping may be high. Many companies tend to just jump into the prototyping without bothering to retrain their workers as much as they should.
7. Prototyping can lead to both *scope creep* and *feature creep*.

Interested?

Table RMA provides a comparison between SDLC and prototyping on some important dimensions.

**Table RMA. An evaluation of the SDLC and Prototyping**

| Condition | SDLC is | Prototyping is |
|---|---|---|
| When user requirements are poorly specified | Poor | Excellent |
| When developers are using unfamiliar technology | Poor | Better |
| When developers are using proven technology | Good | Better |
| With complex projects | Good | Good |
| When delivery deadlines are tight | Poor | Good |

| | | |
|---|---|---|
| When technology is changing rapidly | Poor | Good |
| When continuous management buy-in is crucial | Poor | Better |
| When user buy-in and support is critical | Poor | Excellent |
| When audit trails and multi-level sign-off are critical | Excellent | Poor |
| Where scope creep or feature creep need to be carefully managed | Excellent | Poor |

**Feature Creep and Scope Creep**

Any time clients/users ask developers to increase the amount of work a system will do, or to include features that were not specified in the original system requirements, * creep* is at work. Project managers and developers must be continuously on the lookout for such creeps, as they are everywhere. Each tiny little addition, without a commensurate increase of time allotted, resources allocated or quality expected, leads the project one tiny step closer to failure.

In the broadest sense, an example of scope creep might be if users were to ask the developers of Microsoft Excel (a spreadsheet) to also provide the capability to produce manuscripts (a word processing function). The scope of work is therefore much broader than what a spreadsheet is normally expected to do. Feature creep, on the other hand, could be illustrated by users asking Microsoft to include a feature in Excel whereby every time a user entered a valid email address in a cell, that email address is added to the user's contact list. Nice feature. Not in the original specs. Nothing to do with a spreadsheet's core functionality.

## Agile

The SDLC and, to a certain extent prototyping, represent the so-called *heavyweight*, waterfall-oriented methods, which critics have called ponderous (cumbersome), sclerotic (rigid) and over-managed (too many rules to follow). Such shortfalls led to the development of lightweight agile software development methods in the mid-1990s.

Early implementations of agile methods include Unified Process in 1994 (specifically implemented as the Rational Unified Process or RUP following IBM's purchase of Rational Software in 2003), Scrum in 1995, Extreme Programming (EP) in 1996, and others.

These methods are now collectively referred to as Agile Development following the publication of the Agile Manifesto in 2001.

**Figure NA. The generalised Agile process**

Note from Figure NA the emphasis on delivering working software at the end of each rapid iteration (it should take no longer than a month – and often much, much less time -- to traverse from top to bottom of one iteration in agile), then moving back to the beginning after evaluation by users. This is much closer to prototyping and clearly differentiates from the SDLC where stages are begun and finished and never revisited unless a catastrophic error causes a complete reset. This is the main contrast between the two camps. The SDLC is *predictive* – all is known beforehand and the process doesn't vary. Agile methods are more *adaptive and flexible*.

Let's take a moment with the RUP flavour of Agile and dig into what a project using this development method would look like.

**Figure MH. Tasks and timelines in a RUP project**

Time for a little bit of unpacking here. The RUP process (this is the same as UP, don't get confused) is divided into *Disciplines*, which are akin to SDLC stages and they are listed down the left axis. Specific work is done by specialists in each stage. Business Analysts are busy in the Business Modelling phase. Systems Analysts and Designers are busy in the Analysis and Design phase. Testers are testing when the time comes. And so on. So it's no different from any other development effort - just slightly different terminology. And note they map *roughly* onto the *Activity Phases* of the generalised Agile process map in Figure NA above. So we're getting somewhere...

So that's the horizontal dimensions defined, named *Disciplines*. Let's turn our attention to the vertical dimensions, or *Phases*. Highest level, there are four phases to development in RUP with Software Development Life Cycle stages to explain:

1. Inception - Problem definition and solution imperatives
2. Elaboration - Solution features, software requirements and analysis/design
3. Construction - Coding, database and testing

4. Transition - Implementation and system usage in production

At the next level down, below the four phases, we see the *iterations* named for the phase in which they occur. Thus we have the first (and only) iteration in the Inception phase named as *I1*. The two iterations in the Elaboration phase named as E1 and E2. Not rocket surgery at all. It's important to note that, depending on the project, iterations can be added to or subtracted from this basic model *but it's important to not stray too far from this template. If a project requires a great deal of modification from this *ideal* then perhaps it's time to rethink the scope (too big perhaps?).

What's really interesting about this model are the coloured bars running horizontally from the disciplines labels and crossing each of the phase and iteration boundaries running vertically. Overall, it's clear that there is only one discipline that has no work to do in all 9 iterations - and that's Analysis and Design, which ends at iteration 8. Otherwise, every discipline has work in every iteration. The height of the coloured are indicates the *quantity* of work being done in each phase/iteration. Check it out. Most of the Business Modelling is done early (makes sense) and most of the Deployment is done late. Again, this stands to reason. This simply reinforces the fact that requirements and solutions can be and are discovered late in the game and pose no real threat to the development. This is the strength of agile in any of its flavours.

The title of the diagram from the original work says it all. Business value can be discovered at any juncture by any discipline even during time-limited phases of development.

Interested?

Lean is a flavour of Agile, and has been extended or adopted by other movements in the business world. As Gaping Void designer/artist/guru Hugh MacLoed and business partner Jason Korman imply in the image below, it's always the destination that should drive the agenda, rather than how to get there. Going on a trip, you wouldn't say that you are "...going in a Toyota", but rather that you are "...on our way to Wonderland", for example. Here's the gapingvoid.com feature for April 10, 2015:

**Figure MC. The Lean process**

Here's what MacLoed writes in the accompanying text to the daily email from gapingvoid.com regarding the graphic:

"I never know from the start, what a drawing is going to look like when it’s finished. I just make it up as I go along. And that’s perfectly normal. Nobody ever asked Beethoven, “So when you get around to writing that Ninth Symphony, what notes are you going to use? How many times? In what order?” Of course not. All one has is a feeling that if you set off in a certain direction, something solidly good will happen. Maybe. And what’s true for symphonies is also true for businesses. The final project is never the same as when you first imagine it. The idea is changed the minute it comes into contact with execution. You just kinda know “what” is going to happen, without actually knowing what exactly is going to happen.

"This loose relationship with the word, “What” is at the core of our friend, Eric Ries’ “Lean Startup” movement. Not waiting around to know “Everything”, because that is impossible. Instead, you just start. And keep going. And stay light on your feet enough (i.e. low burn rate) in order to change directions on the dime (i.e. “pivot”) if you need to. Worry about the music, and the notes will take care of themselves."

Lean is much more than about systems development. If you get interested below, you'll note striking similarities to prototyping. Agile and Lean both owe some props to prototyping as they are descended from the general principle of providing frequent and rich feedback, and changing course based on the feedback, that is the foundation of prototyping.

[Interested?]

Agile has many devotees, at least partly because the principles of the Agile movement are simple, straightforward and compelling.

Interested?

The contrast with SDLC are added by the author in the SDLC reflection column:

**Table TB. Agile principles with comparative reflections on the SDLC**

| **Agile principle** | **SDLC reflection** |
|---|---|
| Our highest priority is to satisfy the customer through early and continuous delivery of valuable software. | SDLC delivers at the very end of the process. |
| Welcome changing requirements, even late in development. Agile processes harness change for the customer’s competitive advantage. | Change is discouraged given the high cost of returning even to the previous stage, let alone to the start of the process. |
| Deliver working software frequently, from a couple of weeks to a | |

| | |
|---|---|
| Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale. | SDLC delivers only as early as the testing phase, close to the end. |
| Business people and developers must work together daily throughout the project. | Business analysts are involved only early in the process to provide requirements but then not again until testing, late in the game. |
| Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done. | Systems developers are divorced from the business process and develop in a black box without continuous feedback from the business side. |
| The most efficient and effective method of conveying information to and within a development team is face-to-face conversation. | Little communication between developers and developers and even less between developers and business analysts. |
| Working software is the primary measure of progress. | Adherence to milestones, cost certainty and incremental approval are measures of success. |

Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely. | Pave of development is dictated by external sign-off and resource allocation. | | Continuous attention to technical excellence and good design enhances agility. | Parameters are determined at project outset and not reviewed continuously. | | Simplicity -- the art of maximizing the amount of work not done- is essential. | Steps are undertaken because they must be followed in order to get sign-off and are divorced from agility and/or improvement. | | The best architectures, requirements, and designs emerge from self- organizing teams. | Teams are organised around resource efficiency and cost certainty and not product excellence. | | At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behaviour accordingly. | Little internal reflection; emphasis is on conformity and hitting milestones on time, at or under budget. |

This might seem a stinging indictment of the SDLC. Indeed in many ways, the SDLC has outlived its usefulness, given modern technology and the growth of the internet as a medium for delivering applications. There are obvious differences between the monolithic applications popular in the heyday of the mainframe era.

But many organisations, especially those that are large and bureaucratic in nature (governments, for example), find comfort in the rigorous rules and milestone reviews and sign-offs. We mustn't sell short some of virtues of the SDLC. This being said, let's return to Agile.

Here is what the Agile Alliance says about its values:

Agile Values:

1. **Individuals and interactions** over processes and tools
2. **Working software** over comprehensive documentation
3. **Customer collaboration** over contract negotiation
4. **Responding to change** over following a plan

They close with "That is, while there is value in the items on the right, we value the items on the left more."

Interested?

Let's now take a brief look at Scrum, a popular agile software development method model. "Agile and SCRUM are related but distinct. Agile describes a set of guiding principles for building software through iterative development. Agile principles are best described in the Agile Manifesto. SCRUM is a specific set of rules to follow when practising agile software development."

Interested? Following is from the book entitled *Scrum: a Breathtakingly Brief and Agile Introduction*, 2012, Dymaxicon, ISBN 10: 193796504X. This overview of roles, artifacts and the sprint cycle is adapted from *The Elements of Scrum* by Chris Sims & Hillary Louise Johnson, 2011, Dymaxicon, ISBN 10: 0982866917.

The authors write that "Scrum is a lightweight framework designed to help small, close-knit teams of people develop complex products. The brainchild of a handful of software engineers working together in the late 20th Century, scrum has gained the most traction in the technology sector … A scrum team typically consists of around seven people [7 +/- 2] who work together in short, sustainable bursts of activity called sprints, with plenty of time for review and reflection built in. One of the mantras of scrum is "inspect and adapt," and scrum teams are characterized by an intense focus on continuous improvement— improvement— of their process, but also of the product."

Interested?

A *sprint* is a development session (or *iteration* in standard Agile parlance), traditionally lasting anywhere from two weeks to a month (but never longer), during which time the steps in the agile iteration process (see Figure NA) are executed in one sequence from top to bottom. Scrum sprints are increasingly shorter now, many lasting only a week. The value in short sprints is that deliverable and value-creating software is output at the

conclusion of each sprint, and adding value quickly is a good thing.

Scrum is simple in its organisation and recognises only three what they call “roles”, *viz:* Product owner, Scrim master and Team member. What’s the responsibility of each?

The *Product Owner* is responsible for maximising the ROI from the investment in the system. They do so by actively directing activities of the Scrum team towards ROI-enhancing activities and equally actively away from non-ROI-enhancing activities (such as much of the bureaucratic process inherent in the SDLC). They do so by controlling the *priority* of activities on the team’s *backlog* (more on this soon) and by ensuring that the team clearly understands the requirements – requirements than can and do evolve, change, morph and appear/disappear as the product matures. The owners accomplish this partly by recording the requirements in the form of what are called *user stories* in the form of “As a {role}, I want feature {a feature} so that I can {accomplish something}.”

Such user stories are added to what is called the *product backlog* (discussed below) which might be construed as a type of requirements list in the sense of the SDLC, but are much less rigorous while at the same time much more targeted towards directly adding value to the system.

In a nutshell, the role of the Product Owner can be summarised as follows: The Product Owner:

- Holds the vision for the product
- Represents the interests of the business
- Represents the customers
- Owns the product backlog
- Orders (prioritizes) the items in the product backlog
- Creates acceptance criteria for the backlog items
- Is available to answer team members’ questions

The next role to consider (although the authors of the book from which this brief introduction is taken actually outline the Scrum Master’s role next – I feel this is out of order and take responsibility for this shift in sequence) is the of *Team Member*. The authors have this to say about the role:

“The role of each and every team member is to help the team deliver potentially shippable product in each sprint.” The *sprint* is best understood as one complete vertical sequence in Figure NA, from Requirements to Evaluation. This “sprint (or iteration) is designed to deliver a working system to the client, albeit mostly incomplete until the project nears completion.

Microsoft, for example, has begun to use Agile to deliver its Visual Studio programming environment. Interested? Make sure to read the whole article!

The Scrum authors continue “Often, the best way for a team member to do this is by contributing work in their area of specialty. Other times, however, the team will need them to work outside their area of specialty in order to best move backlog items (aka user stories) from *in progress* to *done.*”

In a nutshell, the role of Team Member can be summarised as follows. The Team Member:

[Is] responsible for completing user stories to incrementally increase the value of the product Self-organizes to get all of the necessary work done Creates and owns the estimates Owns the “ how to do the work” decisions Avoids siloed “not my job” thinking

The final Scrum role to consider is that of Scrum Master. The authors write that “While a team’s deliverable is the product, a scrum master’s deliverable is a high-performing, self-organizing team. The scrum master is the team’s good shepherd, its champion, guardian, facilitator, and scrum expert.”

Further, they offer that “The scrum master is not— we repeat, not— the team’s boss. This is a peer position on the team, set apart by knowledge and responsibilities not rank.”

The role of the Scrum Master can be encapsulated as follows: The Scrum Master is a/an:

- Scrum expert and adviser
- Coach
- Impediment bulldozer [love this one :)]
- Facilitator

The Scrum uses various tools (called *Scrum Artifacts* by practitioners).

## Scrum Artifacts

Artifacts (defined as: ”*any object made by human beings, especially with a view to subsequent use* (dictionary.com)) include the aforementioned

Product Backlog, about which the authors write: “The product backlog is the cumulative list of desired deliverables for the product. This includes features, bug fixes, documentation changes, and anything else that might be meaningful and valuable to produce. Generically, they are all referred to as “backlog items.” While backlog item is technically correct, many scrum teams prefer the term “user story,” as it reminds us that we build products to satisfy our users’ needs.”

The Scrum Master sorts the backlog in order of descending priority. The stuff at the top of the list gets done first.

Furthermore, “Each item, or story, in the product backlog should include the following information:

- Which users the story will benefit (who it is for);
- A brief description of the desired functionality (what needs to be built);
- The reason that this story is valuable (why we should do it);
- An estimate as to how much work the story requires to implement;
- Acceptance criteria that will help us know when it has been implemented correctly.”

A second artifact is the *Sprint Backlog*. The authors write: “The sprint backlog is the team’s to do list for the sprint. Unlike the product backlog, it has a finite life-span: the length of the current sprint. It includes: all the stories that the team has committed to delivering this sprint and their associated tasks. Stories are deliverables, and can be thought of as units of value. … Each story will normally require many tasks.”

The sum of these tasks for a story can be considered the *scope* of the story – how much detail or how many processes are involved in the story.

A further Scrum artifact is the *Burn Chart*, which shows how much of the scope (tasks required per story) have been covered by the team over the specified period of time. A progress chart if you like.

Next comes the *Task Board*, which is visible to all team members, the simplest form having just three columns: 1) To Do; 2) Doing; and 3) Done. Elegant. Simple. Transparent.

Finally (for our purposes) is the notion of *done*. It might seem simple to you or I. If something is done, it’s done. Not so in the world of development. There are conflicting realities and stages of *doneness*. The authors write” “A programmer might call something done when the code has been written. The tester might think that done means that all of the tests have passed. The operations person might think that done means it’s been loaded onto the production servers. A business person may think that done means we can now sell it to customers, and it’s ready for them to use. This confusion about what “done” means can cause plenty of … trouble, when the salesperson asks why the team is still working on the same story that the programmer said was done two weeks ago! In order to avoid confusion, good scrum teams create their own definition of the word “done” when it is applied to a user story. They decide together what things will be complete before the team declares a story to be done.”

Scrum includes a great deal of communication with all affected parties in the software development process. A *sprint cycle* (akin to an iteration in Agile-speak) consists of the following meetings, often called *ceremonies* in Scrum-speak:

1. Sprint planning
2. Daily scrum
3. Story time
4. Sprint review
5. Retrospective

Ceremonies, as they are all about communication, are at the heart of Scrum.

Interested?

To sum up, Scrum is a straightforward, lightweight method for building software where user requirements can change and useful, value-adding software needs to be produced quickly. Scrum is a collaborative method, focused on continuous improvement of not only the end product (software) but also on the actual process of making the software. Interested?

### Challenges with agile (in general)

Increasingly, software is produced by system development shops who specialise in creating solutions for clients, using the most up-to-date technology and best practices. These software *houses* are referred to as *vendors.* The organisations requiring the system to be built are *clients.* So for the client, what this means is *outsourcing*. The client is outsourcing the creation of a system to a vendor. Increasingly, however, solution vendors (and almost everyone else) are *also* outsourcing their work to geographically-distributed software *authors*, i.e., independent or at least very small, teams of coders and specialists who live in China, India and increasingly, Latin America while the clients and solution providers are predominantly in North America and Europe. And this is a problem for agile. It's a problem because agile, according to its own manifesto, relies on real-time, face-to-face and frequent client/vendor and development team interaction. The vendor needs to be intimately immersed in the culture and business process of their client in order to fully understand the *user stories* that will drive the solution. Furthermore, agile requires (as

illustrated by daily *standup meetings* in Scrum) that the development team meet in order to prioritise, knowledge share and schedule their sprints. But how do this this effectively in teams that are distributed across the planet and don't work the same hours, speak the same language and/or understand each others (let alone the client's) cultural norms? This is a new and increasingly researched area of systems development termed *agile global outsourced software development (AGOSD)*.

Interested?

### UML, Use Cases and Use Case Diagrams - a light intro

https://www.youtube.com/watch?v=nN7ITDWKP6g - Business analysis Use Cases

The *Unified Process* (UP) or the *Rational* Unified Process (RUP) is an *object-oriented* (OO) systems development process developed by the three *godfathers* of the OO approach, the famous Booch, Rumbaugh and Jacobson. Their pioneering enterprise was named Rational Software, which was subsequently bought by IBM. UP is tailored to organisational and project needs, meaning is is less *presdictive* than SDLC and more on the *adaptive* side, *a la* Agile. It is use-case driven, using a design tool known as *Unified Modelling Language* or UML.

Let's interject here in the Scrum process and backtrack to the origin of the requirements that are being coded by the Scrum teams. What are these things that are being prioritised? They are *user requirements* (UR) and are the result of an intense exercise to elicit exactly what the system is meant to accomplish. Designers, architects and programmers need at least a general idea of what the system is meant to do, even if the exact details will emerge through an agile process of discovery. Unlike an ancient explorer who just *kept heading west* to see what they could find, our modern-day explorers need a good compass heading. We need to know exactly where we are meant to end up, even if the exact route is unknown. Such is the goal of user requirement discovery.

The process is not exactly set in stone and can itself proceed along a path of discovery, but in the end, certain things must have been documented and agreed upon. They are the *Business Requirements* often shortened to BR# where # refers to a sequential number that can later be referenced when particular *Functional Requirements* (FR) are listed. Each FR must map back to a BR. The FR are the top-level things that the system must allow and accomplish. In our auction system, there are certain things that absolutely need to be accomplished. Examples are:

- BR1: Professors must be able to log into the system
- BR2: Students must be able to log into the system
- BR3: Students must be able to buy products
- BR4: Sales of products must be recorded along with other contextual variables
- BR5: The system must be able to decide who gets bonus marks and who doesn't
- BRn ...

There are often many others.

There needs to be a comprehensive list of such requirements. Note that *how* those things would be accomplished or *what* the interface would look like are not a priority at this stage. These are simply the highest level requirements possible. After assumptions such as "students must have a wi-fi enabled device", and "the classroom must allow wi-fi authentication" and "the internet must be available." Such things are often assumed, but can be stated in an even higher-level document, or as the very highest level precursors to a successful system in the requirements document. Sometimes, such high-level requirements are labelled *System Assumptions* (SA) and are listed and numbered in the same way as BRs. We are talking about *solution features* here.

Once all RAs and BRs are listed, the Scrum team *Product Owner* has the responsibility to prioritise the requirements. Let's refresh some of the PO's responsibilities. The Product Owner:

- Represents the customers
- Owns the product backlog
- **Orders (prioritizes) the items in the product backlog**
- **Creates acceptance criteria for the backlog items**

I've added the **bold** where the requirements document directly affects the roles of the Scrum teams. So what does that look like? Figure TM shows a mock-up of a *Use Case* in a *Requirements Document* that a Scrum team might use to help prioritise, create the system and remain accountable. There could be literally hundreds (or even thousands depending on the size of the system) or these to prioritise in the Scrum:

**Figure TM. Use Case sample**

So what exactly is a Use Case? The nice folks at *TechTarget* offer this:

"A use case is a methodology used in system analysis to identify, clarify, and organize system requirements. The use case is made up of a set of

possible sequences of interactions between systems and users in a particular environment and related to a particular goal. [...] The [sum of all] use case[s] should contain all system activities that have significance to the users. A use case can be thought of as a collection of possible scenarios related to a particular goal, indeed, the use case and goal are sometimes considered to be synonymous."

So a Use Case is simply a story. A story about how actors (anything external to the system being built) interact with the system in order to get something done (a goal). These *things* are in fact *requirements* and are what Scrum POs prioritise. Some things must logically be done before others can be attempted. A user must be able to log into the system before they can print a document, for example. But each thing is important in its own right and each has a story to tell. Use Cases are those stories and must relate back to a requirement either directly or indirectly. And such stories can be told in pictures as well, as we see below in UML (Use Case) diagrams.

Interested?

**Figure MM. UML (Use Case) diagram**

The Unified Modelling Language (UML) tool for visual storytelling is the Use Case Diagram (UCD). Here is part of the definition provided by *agilemoddeling.com* (see the Interested link below):

"Use Case: A use case describes a sequence of actions that provide something of measurable value to an actor and is drawn as a horizontal ellipse.

"Actor: An actor is a person, organization, or external system that plays a role in one or more interactions with your system. Actors are drawn as stick figures.

"Associations. Associations between actors and use cases are indicated in use case diagrams by solid lines. An association exists whenever an actor is involved with an interaction described by a use case. Associations are modeled as lines connecting use cases and actors to one another, with an optional arrowhead on one end of the line. The arrowhead is often used to indicating the direction of the initial invocation of the relationship or to indicate the primary actor within the use case. The arrowheads are typically confused with data flow and as a result I avoid their use."

Figure MM (above) models the online auction system as a series of Use Cases. There are too many individual cases and actors and associations in this diagram (I created it just to show scope) but it nonetheless gives an overview of the tool used in agile programming of all kinds to make sense of requirements. Note the relationships between the *Use Case* and the set of diagrams. The UC itself is brought to life in the diagram. A better way to understand this tool is to look at the UCD detail in Figure VP, below.

**Figure VP. UML (Use Case) diagram detail**

The detailed view of just the Student actor's interaction with the system highlights two powerful aspects of UML, the *Extends* and *Includes* extensions. An *includes* is used where a particular process, byu its nature, *must* use another, independent process in the course of doing what it is supposed to do. The the Student actor's activity of *buy item* is a discreet activity. It involves a student playing the auction game observing a price on the interface and then clicking a button if they decide to buy t that price. But whether the player likes it or not, the sale must be recorded in the database in order for other things (such as the awarding of bonus marks) to occur. So we model this as "player decision to buy also *includes* the activity of recording that sale n the database". There is no other way to get to this saving behaviour *except* through the *buy* action. The player can't just decide to *record something to the database*. Furthermore, the recording of a sale to the database also *includes* updating the player's individual statistics to the interface.

The other important extension is *extends.* The functionality in an *extends* only happens *sometimes* and not every time as in an includes. For example, the *Professor* actor in the use case shown in Figure MM has a use case named *Start/stop/pause* auction. Only in *certain cases* does this use case result in *Calculate bonus distribution* (when the simulation has ended for example, and not when it is simply paused. So the calculation of the bonus mark distribution is *optional* and only carried out when the game has officially ended in a normal way (time has expired).

The way to distinguish between an *includes* and an *extends* is that includes are *always* and extends are *sometimes*. Clean and simple.

Interested?

## Summary

Here's a great summary of everything we've presented here, and more. It's not a long read, and it's recommended to get the broadest overview of the various development methods along with recommendations on when to use each. Highly recommended.

Interested?

# Which to choose?

When it comes down to the choice of development methods between the traditional waterfall method and the more contemporary (but not by much) agile methods, the choice is often made for you by the nature of the problem or opportunity the system will address and by how well you can scope out the requirements in advance. Specifically, there are two broad categories or types of requirements: the *shall* and the *should*. This folds nicely into our discussion of *creep*.

*Shall* requirements are *prescriptive*, as in *must be done*. These are *shall requirements*. *Should* requirements, on the other hand, are less *prescriptive* in nature and more *nice to have*. They are to be included in the system if things go well, or if they fit into the final model or if they can be afforded when all is said and done. These requirements are more likely to arise in the *process* of building the system as outcomes are discovered and contingencies are explored. These are the kind of requirements that prototyping and agile methods uncover.

In general then, if you can absolutely positively spec each requirement in advance at a very granular level, then the waterfalling SDLC is for you. Here's what to do, go do it. If things are more fluid, if the target is moving, if the real nature of the problem or challenge is evolving or is poorly understood, you're better off with Agile. Discover as you go, but make sure you have a target and a resource cap else the process will be never-ending.

Here's an example of *prescriptive requirements* that I created so that a developer could code up the *dynamic auction system* that is used in my class. This is a small subset of the specs, but they are of the *please do this* nature. The developer and I aren't *discovering* anything:

--- *Begin specs* ---

Ruleset 4: Run Game (Scenario C)

Processes:

a. Start game

```
i.  Set gPending = False in tGame

ii. Deny Student login

iii. Fetch and write Start time to tGame

iv. Calculate and write Total players to tGame
```

b. Pause running game

```
i.  Stop the timer

ii. Preserve all elements

iii. Function to resume running game
```

c. Running game (Client)

```
i.  Poll and show game status (running = green / stopped = red)

ii. Poll and Enable BUY! button as appropriate

iii. Show total game time in seconds

iv. Show time remaining (Start time + Duration – Current time) in seconds

v.  Show Min buy number

vi. Show Max buy number

vii. Show already bought this client

viii. Show average paid this client

ix. Show average paid all clients
```

```
x.  Fetch current price

xi. Show current price
```

d. Create bid records in tBids

```
i.  Fetch oID  from tLogin and write boID to tBids

ii. Fetch sCUID  and write bCUID to tBids

iii. Fetch IP from running game

iv. Fetch oIP form tLogin

v. Compare IP with oIP

vi. If same write IP to tBids else exit

vii. Get and write Player bid

viii. Get and write Time bid submitted to tBids

ix. Calculate total bids this player

x.  If total <= Max bids then Enable bid button else Disable
```

e. End game (duration override)

```
i.  Get and write Elapsed time to tGame

ii. Get and write Success (game came to resolution) to tGame

iii. Fetch and write Total bids entered to tGame

iv. For each CUID in tBids where oID = gID (current game)

    1.  Verify Sum of bids in tBids >= Min Bids and <= Max Bids

        a.  If within range

            i.  For each bBid in tBids

                1.  Increment counter

                2.  Store Bid price

            ii. Next

        b.  End if

    c.  Calculate Average Bid price this student

v.  Next

vi. Get and write gBonus = if bonus marks awarded based on this game to tGame

vii.    Sort Averages in descending order

viii.   For each CUID in tBids where Average bid > 0

    1.  If Average >= gCutoff_1 and < gCutoff_2 from tGame

        a.  Increment gActual_1

        b.  If bBonus in tGame = True then
```

```
            i.  Create record in tBonus

            ii. Write CUID from tBids

            iii.    Write gID from tGame

            iv. Write nBonus = gBonus_1 from tGame

        c.  End if

    2.  End If

    3.  Else if  Average >= to gCutoff_2 from tGame

        a.  Increment gActual_2

        b.  If bBonus in tGame = True then

        i.  Create record in tBonus

        ii. Write CUID from tBids

        iii. Write gID from tGame

        iv. Write nBonus = gBonus_2 from tGame

        c.  End If

    4.  End If

ix. Next

x.  Write gActual_1 to tGame

xi. Write gActual_2 to tGame
```

--- *End specs* ---

Complementary software development methods to systems development life cycle, along with a short description and pros/cons as appropriate, are:

- *Joint applications development (JAD)* JAD features a workshop-like methodology emphasising continuous feedback and iterative development. This is an agile method to be sure, but emphasises complete agreement between the business side and the tech side before development begins on any piece of a system. Interested?
- *Rapid application development (RAD)* RAD is an incremental method that makes good use of prototyping, with regularly-scheduled delivery of components for user feedback. Interested?
- *Extreme programming (XP)* XP is extension of earlier work in Prototyping and RAD. The core values of XP are: Communication (open, frequent verbal discussions between the business and the tech side and among developers on the tech side); Simplicity (in designing and implementing solutions - deliver only what is required is the most straightforward fashion); Feedback (frequent stocktaking on functionality, requirements, designs and code; Courage (I love this one - courage in facing choices such as throwing away bad code or standing up to a too-tight schedule), and finally Pair Programming where two programmers work intensely together side-by-side on particular elements of the code and test cycles of development.

Interested in comparing Agile, XP and SDLC?

# Why do systems development projects fail?

An Oracle (originally a database vendor but now a full-service application provider including, as we have seen, in the ERP space) developer with many years of experience wrote a sponsored piece in the Harvard Business Review about information systems delivery. He began with, "Information systems projects frequently fail. Depending upon which academic study you read, the failure rate of large projects is reported as being between 50%-80%. Because of the natural human tendency to hide bad news, the real statistic may be even higher. This is a catastrophe. As an

industry we are failing at our jobs." (Paul Dorsey, *Top 10 reasons why systems development projects fail* - accessed June 18, 2015). Add this to what we have already seen from McKinsey regarding failure rates (take me there) and ICT doesn't have the best track record for delivering systems as promised.

First of all, how do we define *failure*? That's important. The vast majority of systems are actually delivered, but what causes them to be deemed a failure is how well they measure up to the following metrics:

1. Was the system delivered on time?
2. Was the system delivered at or under budget?
3. Does the system meet the agreed-upon systems requirements?
4. Does the system provide acceptable quality output?
5. Does the system reduce input and/or augment output?
6. Will the net gain in #5 produce acceptable ROI over the system's predicted lifespan?

On-time and at-budget are the easiest to measure. There's a delivery date and there's a budget. It's black and white. I have no sympathy for either a developer or a client who cannot set and meet defensible time and resource goals. There's simply no excuse for missing such targets. As a developer, it's up to you to make sure you have a good set of requirements (whether specified beforehand or discovered through iterative means as the system is being scoped) and have estimated the time and resources required to meet those targets. If you are slipping, get back to the table and work it out. Be above board, flexible and, above all, willing to swallow your pride and work harder to meet the promised deadlines. If you are unsure of the requirements and are going Agile to discover the core need, then that must be accounted for in the time and resource scope. While *how long it will take to discover the requirements* might be tough to nail down, there must be a time limit on this phase. If you can't find your way in six months, let's say, then the project is cancelled. Fair enough. As a client, it's incumbent on you to keep the project scope in check, to not make unreasonable demands just because you think the developers will cave, and to have a critically sympathetic ear for changes and impediments and new developments that could never have been foreseen and are unavoidable. There will be challenges along the road. It's an awful lot like a marriage. Having been married for 40-some-odd years, I know what I'm talking about. You work on it every day.

Criteria numbers 3 and 4 are the most problematic since they are not as easily measured. In general, these three are about *expectations* and expectations are notoriously slippery and malleable. I know. I've been on both sides of the developer/client relationship. I've developed (architected, designed, coded and delivered) many, many systems, large and small, in teams ranging from myself alone to multiple specialists. And I've worked as a solutions architect and senior consultant, abstracted several layers from both the coders and the clients. And I've also been a client for many systems. I currently have three systems under development. I've been a senior developer and a senior manager. I've worn all the hats.

Criteria 5 and 6 are somewhat subjective (subject to opinion) but decent metrics can clearly be developed to measure whether or not the system can and is meeting expectations. Input and output are, for the most part, tangible and measurable. Agreements on what constitutes input and what counts as output can be made *a priori* and measured at any interval. The cost of capital and the alternatives to investing in a system are all clearly measurable and comparable to the current reality. So this stuff *can be done*. It's just doing it and not romanticising system development (the siren call of technology) or, on the other end of the spectrum, being driven by fear. The entrepreneur's goal is to instill *FOMO* (fear of missing out) in the hearts and minds of potential investors. This is *NOT* how systems should be conceived.

So yes, expectations are difficult to manage. Requirements are difficult to nail down, especially when clients *don't know what they want until they see what they DON'T want..."* (this is a real saying in the systems development world). Clients come and go. Client expectations change. Senior management shuffles responsibilities. Your development team membership will change. Technology evolves at an alarming pace. New people need to be *re-sold* on the virtues of the system. Budgets dry up or, maybe worse, budgets get pumped full of cash at fiscal year end and must be spent (talk about needlessly raising the bar on expectations). Business requirements (the context) change. Everything is in constant flux, so meeting expectations is like trying to hit a target that's moving multiple times faster than your ability to adjust your sights. So in many ways, it's surprising that *any* system gets delivered and considered a success.

That being said (and I feel better for having said it and defending my old friends in the business), there are some well accepted ground rules for developing systems that are timely, efficient, well received by users, that bring value and are a delight to use. They are:

1. The proposed system must be doable. So many cascading effects will kick in once you agree on a project and a delivery date. If you fail to deliver, anything that depends on your project being in place is put in jeopardy as well. Not a good way to make friends around the organisation.
2. Strong governance (the rules about how the rules are applied) is a must.
3. There must be a real, defensible business case for the system and clear ROI (return on investment) targets and time lines. When will the system start paying us back? What's the *time to value*? Not every (or even *many*) system will pay back starting on delivery day +1. Everyone must agree on this expectation.
4. System requirements must be clear to all concerned parties - everyone must know what's in, what's out and what's expected from the system. *Scope creep* must be managed. If things change, requirements and deadlines and allocations must be revisited and time milestones,

resources or both need to be adjusted. Clarity here is paramount. Protect your project scope as if it were the very life-sustaining oxygen you breath.

5. Projects must be carefully costed and budgeted. Skill in this area is critical. Estimates must take into account the expected project deliverables, carefully overlaid on the chosen systems development methodology. Agile methods, while they appear to have more fluid boundaries and more liberal expectations, are still measurable and predictable. The project estimation staff must be crystal clear on requirements and on how long it will take and with what resources to meet those expectations in order to accomplish the tasks given the chosen development method.
6. Senior management buy-in and a champion is required if you're working in a large organisation. Someone near the top needs to love the system and be willing to go to bat for it. It's a given that times will get tough on large projects. You need a champion who will fight for your vision and keep the buzz going even in the tough times.
7. You need a good team, and especially (but not exclusively) project managers. PMs will manage expectations, and expectations are your biggest challenge.
8. It's critical to have great technical people - people with vision, imagination and a willingness to make things work. It's hard to keep good techies. There is incredible demand for talented people who can make tech sing.
9. You need good tools. There is absolutely no substitute for having the right tools available to do the job.
10. Don't cut corners to compensate for a tight time line or a lack of resources. Get back to the table with the client if things are going off the rails. A good project manager will ensure this works.
11. Don't get seduced by cool, cutting-edge, shiny new technology. It could be here today and gone tomorrow. Make sure all the pieces fit together. Make sure the project can deliver on expectations. Don't do something because you *can*. Do it because it adds value.
12. Get end-user buy-in and acceptance early and maintain as close and as good a relationship with those who will ultimately use your system as possible. Many shiny new systems have failed on the shop floor when users refused to use them, once delivered. Prototyping with end-users is a great way to earn user respect and acceptance. And it reduces the need for training as well.

## UxD - User experience design

The final topic (I know!!! finally...) we need to look at is *User Experience Design*. UxD was the topic of an Interested? link in Chapter 7, but I wanted to close this chapter on development with an interesting article from the Nielson-Norman Group on the importance of the *platform* upon which you do business. A *platform* can be thought of as a *computing model* in the sense of the size of the device (a smartphone being infinitely smaller - though getting larger again of late - than a tablet, than a laptop, than a desktop) upon which business is being offered, as well as vaguely the operating system (Android presenting itself differently than Apple iOS than Apple OS than Windows, etc.). But thinking strictly of what are called *conversion rates* (as simple as simply the number of sales on an eCommerce site divided by the number of visitors multiplied by 100 or as medium complex as the number of abandoned shopping carts divided by the number of visitors or as a ratio of completed sales to abandoned sales, etc.) it is apparent that platform matters. If you're in Marketing, this will interest you. If you are in IS, this will interest you. If you are in Finance, this will interest you. If you are in Supply Chain, this will interest you. If you are in Managerial Accounting, this will interest you. It's just plain interesting. Check it out.

Interested?