Software Development: Why the Traditional Contract Model Is Not Fit for Purpose

Susan Atkinson
Keystone Law
London, England
susan.atkinson@keystonelaw.co.uk

Gabrielle Benefield
Evolve Beyond Limited
London, England
gbenefield@evolvebeyond.com

Abstract

Many IT change initiatives involving the development of software fail, and the scale of the failures can be large. We believe that the traditional contract model for software development is generally responsible for these failures. Even if an IT project is resourced internally, the organisation applies similar management practices to the IT project as if it were outsourced to a third party supplier.

The contract model contains three fundamental elements, all of which are seriously flawed in the context of software development. In any IT project the contract model increases the risk of failure, and leads to a suboptimal design and poor return on investment. In this article we examine some of the ways in which this happens. We also consider an alternative approach, based on the principles of complexity theory.

1. Introduction

In 2007 the UK Department for Communities and Local Government (the DCLG) entered into a contract with European Air and Defence Systems (EADS, now known as Cassidian) to deliver an IT system that would underpin the FiReControl project. The FiReControl project aimed to improve the UK Fire and Rescue Service by replacing 46 local control rooms with a network of nine purpose-built regional control centres using a national computer system to handle calls, mobilise equipment and manage incidents. The project was expected to be completed in October 2009, and the DCLG's original estimate of the project costs was £120 million. By 2009, two years into the project, what was meant to take two years was then expected to take four years, and the anticipated total project costs had increased by more than 500% to £635 million. In 2010 the contract was terminated. The DCLG had wasted at least £469 million as a result of the failure of the project to deliver [1].

In 2003 the Levi Strauss Company entered into contracts with SAP and Deloitte to migrate its fragmented and archaic IT system to a single SAP system. Analysts estimated that the project would cost USD 5 million without consulting fees. When the new system was rolled out to the US market in 2008, the three US distribution centres of Levi Strauss went offline for a full week and Levi Strauss was unable to fulfil orders. These shipping problems, combined with other economic issues, caused the company's profits in the second quarter of 2008 to fall to a miserly USD 1 million from USD 46 million in the year-earlier quarter [2]. A project that was forecast to cost USD 5 million ended up costing Levi Strauss nearly 40 times that amount in terms of lost sales.

These are sobering stories of large IT projects spiralling out of control. But they are not isolated incidents. Indeed, about two thirds of all software projects are delivered late or fail outright [3]. Not only that, but one in six IT projects has a cost overrun of 200% on average and a schedule overrun of almost 70% [4]. It seems that no organisation is immune from these risks. There was a common belief that out of control IT projects were the preserve of the public sector, but recent studies show that the private sector does not fare any better in comparison. Organisations in the private sector are simply less publicly accountable and so have greater ability to conceal IT disasters.

Software is now intrinsic to so many aspects of an organisation, that it is inevitable that in the next few years virtually every organisation will need to update its existing IT systems or to develop new IT systems. Not all organisations have deep enough pockets to weather a project that is delivered late or fails outright, and very few organisations can survive a project that experiences a cost overrun of around 200% and a schedule overrun of around 70%.

Auto Windscreens is an example of a successful

1530-1605/12 $26.00 © 2012 IEEE
DOI 10.1109/HICSS.2013.492

company that was forced into bankruptcy as a result of a failed IT project. In 2006 Auto Windscreens was the second largest automobile glass company in the UK, with 1,100 employees and £63 million in revenue. However, by the fourth quarter of 2010, a combination of falling sales, inventory management problems, and spending on a failed IT project resulted in its downfall.

So why are IT projects so high risk, and how can this risk of failure be mitigated?

There is a huge disconnect in the world of software development. In theory, the legal and management functions should sit in between, on one side, the business function from which the need for the software arises and, on the other side, the practice of software development. The role of the legal and management functions should be to structure and coordinate the relationship between these two different areas. However, the legal and management functions are quite removed, both in language and in values, from both the business function and the practice of software development. The lawyers aim to make the relationship as precise as possible and to regulate every possible eventuality. However, both the business function and the software development practitioners are operating in an increasingly complex, dynamic and inter-connected environment.

The legal and management functions have, by and large, not adapted their practices or values in recent years to take account of the challenges faced by the business functions and software development practitioners. Indeed, the contract model for software development (the "Contract Model") and the management practices that surround it have barely changed in the last thirty years. Much of the thinking underlying the Contract Model is rooted in the Industrial Revolution and the practices at that time as developed by Henry Ford and Frederick Taylor.

Our view is that the Contract Model compounds the effects of poor management, and that poor management is often based on the flawed thinking underlying the Contract Model. We have found that even if an IT project is resourced internally, the organisation applies similar management practices to the IT project as if it were outsourced to a third party supplier. Organisational policies often create contractual relationships between departments inside a single organisation that can produce the same effect as the Contract Model.

We do not believe that we will see any significant improvement in the success of IT projects until we change the basis of the Contract Model and the surrounding management practices. For the purposes of this article we use the term "IT project" as shorthand for any IT change initiative involving the development of software.

2. The fundamentals of the Contract Model

We believe that any contract for software development based on the Contract Model contains three fundamental elements, all of which are seriously flawed. We use the terms "supplier" and "customer" to explain the dynamics in an external relationship. However, as mentioned above, in many cases similar principles appear to apply even if the IT project is resourced internally.

The three fundamental elements to any contract for software development based on the Contract Model are as follows:

- ***Output-Based Requirements.*** The supplier is required to deliver output that possesses all of the requirements, as specified by the customer in the contract, by an agreed date. We use the term "output" to refer to the deliverables of the IT project. These may take the form of product (e.g. code, features, functions, attributes), documentation and/or services.
- ***Sequential Development.*** The software is to be developed sequentially, that is, using the waterfall model. Development is seen as flowing steadily downwards - like a waterfall - through the phases of conception, initiation, analysis, design, construction and testing. The supplier is required to complete each phase before starting the next phase, and the output of each phase provides the input for the next phase.
- ***Change Control Mechanism.*** The Contract Model mandates that any change to any Output-Based Requirement or to any other element of the contract must be regulated by the Change Control Mechanism. Broadly speaking, to initiate change the customer must submit a change request form to the supplier outlining the desired change. The supplier analyses the impact of the change request on the contract as a whole, including, in particular, the Output-Based Requirements, the price and the due date for completion of the IT project. On the basis of this, the supplier proposes an amendment to the contract. Following the agreement of the parties to the proposed contract amendment (which may involve lengthy negotiation of the commercial terms), a formal amendment is made to the contract to incorporate the requested change.

It is worth noting that we do not make any reference to the type of charging model in the Contract Model. If the three fundamental elements of the

Contract Model feature in a contract, the contract will be flawed regardless of which charging mechanism is adopted.[1] It makes little difference whether the price is fixed, target or based on time and materials, or whether there are bonuses or penalties.

On the face of it, and to a person that is not directly involved in the implementation of the IT project, the three fundamental elements of the Contract Model may appear to be eminently sensible. They create a sense of certainty and predictability regarding the IT project, and they provide a clear and understandable structure for the various activities involved in the IT project.

However, the combination of these three elements in a contractual relationship between a customer and a supplier appears to cause the failure of many IT projects that would probably otherwise succeed. In fact, we would suggest that those IT projects which do achieve success, do so in spite of the Contract Model and the surrounding management practices, and not because of them.

In any IT project the Contract Model increases the risk of failure, and leads to a suboptimal design and poor return on investment. In this article we examine some of the ways in which this happens.

3. An increased risk of failure

Any IT project is subject to risk, which can be categorised into three main types:

- ***Delivery risk.*** This is the risk that the IT project is not delivered on time, on budget and to the required quality.
- ***Business value risk.*** This is the risk that the IT project doesn't deliver the expected business value.
- ***Existing business model failure risk.*** This is the risk that the IT project damages the existing organisation.

The Contract Model does not address the second two categories of risk and actually increases the customer's exposure to all three categories of risk.

3.1. Delivery risk

The FiReControl Project is a classic example of the delivery risk de-railing the IT project. The UK National Audit Office noted that during the first two years of the contract there was little progress in delivering the IT system [5]. Indeed, the DCLG does not appear to have received any working software before the contract was terminated. We do not know the reason for this but we can hazard a guess.

An underlying rationale for the Contract Model is that if changes are made to the Output-Based Requirements during the development process, this can lead to serious delay and an escalation in costs. The Contract Model attempts to reduce this delivery risk by effectively ring-fencing the IT project and controlling the impact of any changes while the IT project is underway.

An important assumption underpins this thinking behind the Contract Model. This assumption is that the software can be finished before significant changes occur. If, on the other hand, significant changes do in fact occur while the software is being developed, over time it becomes increasingly difficult to ignore the impact of those changes on the IT project.

In the 1980s, which - we believe - is when the Contract Model was first used, it might have been a reasonable assumption that the software could be finished before significant changes occurred. But these days the pace of change is so fast that the assumption no longer holds true. Indeed, changes happen all the time and are to be expected.

Firstly, the internal dynamics of the IT project lead to changes. It is only natural that, as the customer learns more about the latest technology and its relative strengths and weaknesses, the customer revises its thinking on how best to take advantage of the technology.

"... [S]ystems requirements cannot ever be stated fully in advance, not even in principle, because the user doesn't know them in advance – not even in principle. To assert otherwise is to ignore the fact that the development process itself changes the user's perceptions of what is possible, increases his or her insights into the application's environment, and indeed often changes that environment itself [6]."

Secondly, external forces are at play. Technology is evolving at an ever increasing pace. The market or context in which the concept for the software was conceived continues to change. Hence, the opportunities or risks to be addressed by the software also change. For example, the emergence of disruptive technologies such as Facebook, Twitter or the touch screen IPad can have a huge impact on any existing plans for software development and distribution. The regulatory environment may also change.

If changes happen, whether as a result of the internal dynamics of the IT project or as a result of the external forces at play, it is inevitable that this will impact on the Output-Based Requirements. The customer will wish to revisit the Output-Based Requirements in the light of the recent changes.

[1] Sometimes the contract does not contain sequential development, but even if the contract only contains the Output-Based Requirements and the Change Control Mechanism it is almost as problematic.

As a result, these days the Output-Based Requirements are exposed to a large amount of change over the course of the IT project, and the larger the IT project, the greater the amount of exposure to such change. In a study conducted by Capers Jones in 1997, confirming the findings of an earlier study by Barry Boehm and Philip Papaccio conducted in 1988, it was found that in a typical software IT project 25-35% of the requirements changed over the course of the IT project [7]. More recently, Eric Ries has suggested that this figure may be as high as 100% [8]. This is very damaging for the IT project.

As an integral part of the contract, the Output-Based Requirements cannot be amended to reflect a change without a formal amendment to the contract as agreed by the parties in accordance with the Change Control Mechanism. The initial stage of the Change Control Mechanism is that the supplier analyses the impact of the change requested by the customer. The larger and more complex the IT project, and the greater the amount of work that is involved in the supplier carrying out this exercise. Sometimes, the impact of the change request is so complex that the supplier simply cannot work out how to incorporate the requested change into the existing IT project.

The process of analysing the impact of a change request can take so long and be so extensive that it has a destabilising effect on the IT project. The main team is only too aware that current work may be rendered redundant following the approval of the change request. The bigger the proposed change, the longer the hiatus while it is being analysed, and the more damaging the effects can be [9].

Any change request will inevitably cause delay to the IT project. It is unlikely that the original timetable builds in a buffer for the supplier's resources to be diverted to this activity and for any resulting additional work to be undertaken. If is for this reason that both customers and suppliers consistently cite changes to the Output-Based Requirements as one of the main causes of failure of an IT project.

To make matters worse, the Contract Model mandates Sequential Development. It is not until testing, late on in the IT project, that the customer has visibility of the software. Up until that point it is very difficult for anyone to assess whether the IT project is on track. The deliverables of all earlier phases are documents that are based on assumptions. It is only when the software is actually built that anyone can accurately assess whether the IT project is actually on course to meet the Output-Based Requirements. However, there is a long gap, often in the order of years, between the date when the customer collects the Output-Based Requirements and the date when the supplier makes the first delivery of working software. The longer this gap, and the more likely that significant change has occurred during the intervening period.

Recent studies, led by Al Goerner at the University of Missouri, Kansas City, demonstrate that the inherent value in Output-Based Requirements erodes exponentially over time. This rate of decay has been likened to the half-life of an unstable radioactive atom. The 'half-life' is the measure of the period of time it takes for the substance undergoing decay to decrease by half.

According to the studies carried out by the University of Missouri, the half-life of the value of the Output-Based Requirements has been rapidly decreasing. In 1980 this was around 10-12 years, by 2000 it had fallen to 2-3 years, and it is currently running at about 6 months.[2]

In other words, half of the Output-Based Requirements will become obsolete by the end of month 6, half of the remaining half (i.e. 1/4) will become obsolete by the end of month 12, half of the remaining quarter (i.e. 1/8) will become obsolete by the end of month 18, and so on. Hence, by the end of month 18, according to the University of Missouri studies, only 1/8 (i.e. 12.5%) of the Output-Based Requirements will still possess any inherent value.

If the University of Missouri studies are to be relied upon, the implications for IT projects are enormous. This would mean that if an IT project is running six months late, the likelihood of the supplier delivering what the customer actually wants is reduced by a half. Even if the IT project runs according to plan, if the IT project is scheduled to run for more than a few months the parties have to expect significant changes.

3.2. Business value risk

The FiReControl project highlights the importance of demonstrating from the outset how the ICT project will deliver the expected business value and of obtaining the buy-in of all those involved. The DCLG was criticised by the National Audit Office for not making sufficiently clear the case for a centrally-dictated standard model of emergency call handling and mobilisation, operating from new purpose-built regional control centres. From the start many local Fire and Rescue Authorities and their Fire and Rescue Services criticised the lack of clarity on how a regional approach would increase efficiency [10]. Unless the resulting software delivers tangible business value, it doesn't matter how state of the art or sophisticated it is,

[2] We have been unable to find out more details regarding these studies. Clearly the half-life for the specifications will vary for different sectors. However, there is anecdotal evidence to support the view that the half-life could be even shorter in the technology sector.

the intended end users may each simply decide not to use it.

The Contract Model does not address the possibility of business value risk. There is, instead, an assumption that if the supplier delivers software that meets the Output-Based Requirements, it will therefore deliver business value to the customer. However, this in turn assumes that the customer knows what it *needs*. What we have found instead is that, although customers are very good at stating what they want, far too often customers do not in fact know what they need. As a result, it is not uncommon for the customer to be disappointed with the resulting software, even if the supplier can demonstrate that the software meets the Output-Based Requirements.

It is a sad indictment of the current state of software development that one of the greatest risks is that the supplier builds *'the wrong product'*. This happens whenever the supplier successfully executes against the customer's specified Output-Based Requirements, but the resulting software does not add any real value to the customer. The software does not add value because it does not enable the customer to solve the problem that it had wanted to address.

This is best illustrated by the findings from the US Department of Defense (the DoD) [11]. The DoD analysed the results of its software spending, totalling an eye-watering $35.7 billion, during 1995. They found that only 2 per cent (2%) of the software was able to be used as delivered. The vast majority, 75 per cent, of the software was either never used or was cancelled prior to delivery. The remaining 23 per cent of the software was only used following modification. That would suggest that the DoD actually only received business value from $0.75 billion of its expenditure – nearly $35 billion of its expenditure did not result in software that delivered any immediate business value.

The reason why customers to date have derived so little business value from the software delivered by the supplier is that the Contract Model is not referenced to the target outcomes of the customer (that is, the results that the customer wishes to achieve and which will add value to the customer). Instead, the Contract Model is referenced to the Output-Based Requirements, that is, the requirements for the deliverables of the IT project which are intended to contribute to and facilitate the achievement of the target outcomes.

People buy a hammer to knock in a nail so that they can put up a picture – they know that they can achieve their target outcome (putting up the picture) with the acquisition of the hammer. Unfortunately, in the context of software development it is not as straightforward to make the link between the delivery of the output (the software) and the achievement of the customer's target outcomes. Many people simply don't even try. This creates a large risk that the supplier will only deliver what the customer asked for – a vague set of Output-Based Requirements – rather than what the customer actually needs, which is to achieve the target outcomes.

Software development involves the transformation of ideas into deliverables to achieve business objectives. The catalyst for the IT project is generally a business case. This justifies at a strategic and financial level the acquisition of the software. The anticipated cost of the software is justified by various assumptions such as the improvement of business processes, increase in market share, increased revenue, reduction of support costs and so on. Following internal approval of the business case, the Output-Based Requirements are then collected and assimilated from everyone at the customer's organisation who has an interest in the resulting software system.

So if a business case generally precedes the specification of the Output-Based Requirements, why is it the case that the resulting software will not necessarily meet the target outcomes?

Firstly, the business case is untested. In many business cases there are elements which are based on assumptions rather than facts. These assumptions have not been proved to be true, and it is probable that many of them are in fact erroneous. Ideally, those assumptions should be tested before significant resources are invested in building a software system that delivers against the business case. But that rarely happens.

Secondly, the business case is typically produced at a high level and with a view to obtaining funding or budgetary approval. It is not uncommon for the business case to be very ambitious in terms of what the software will achieve, as this provides a better argument for investing in the acquisition of the software. It is less common for the business case to play an active role in steering the direction of the IT project. It is even less common for anyone to revisit the business case in the light of the progress of the IT project or to measure the progress of the IT project with reference to the business case.

The implications of this cannot be understated. The DoD is one of the most sophisticated IT purchasers worldwide: it has a significant amount of leverage in negotiating contract terms because its annual spend its so large. And yet it derives practically no immediate business value from its investment in software development. Is it possible that other organisations, with less experienced procurement functions, actually derive less business value from their IT spend?

The most effective way for any organisation to reduce its IT spend is to ensure that only software

which delivers business value is built. We need to consciously connect the levels and clarify how the resulting software will deliver business results.

3.3. Existing business model failure

What is unusual about the IT project failure of Levi Strauss in the US is that the company had already installed SAP successfully across its Asia-Pacific region. Although commentators can only guess, it is thought that perhaps the failure of the rollout in the US region was due to unexpected complexity in the financial environment at the company headquarters or due to project management issues unique to the US rollout [12].

The Contract Model does not address the possibility of existing business model failure risk. There is simply no recognition of the fact that when a new software system is launched, this may impact on the existing business processes.

Perhaps, back in the 1980s when the Contract Model was first used, software systems were fairly discrete and limited in terms of their operation. However, today software systems are used for virtually every business function of an organisation. There may be end users at multiple levels of the organisation – for example, the finance director, accounts department, marketing director, marketing team and sales team may all need to use the same software system. This software system may interlink with other software systems within the organisation which are used by other end users. The software system may also interface with software systems of other organisations – such as the clients or the suppliers of the organisation.

In light of the many business processes that may be impacted by a new software system, it is essential that the transition to this system is managed in a way that contains the risk of existing business model failure to a minimum. However, the Contract Model generally requires that all of the Output-Based Requirements are delivered as a single batch. The larger the IT project and the larger this batch will probably be.

For many organisations it is simply not realistic to attempt to assimilate a software system of this scale and complexity into their existing business processes all at once. The risk of any of those existing business processes falling down under the enormity of the change are huge. It would be much better if the transition to the new system was managed in smaller launches, with an emphasis on the quality of the user experience throughout the transition.

4. Suboptimal design

It is inevitable that the Contract Model will lead to suboptimal design. The Contract Model mandates practices that are directly at odds with what is currently regarded as best practice for creating high quality software design.

Sequential Development is a flawed and discredited development methodology. It is no longer used by many of the top software development organisations which favour incremental development with fast and iterative feedback. One of the most serious failings of Sequential Development is that all aspects of the design must be finalised before development starts.

That is like trying to design a bike without being able to build it or test-ride it. Every attribute of the bike needs to be considered in the greatest of detail without the knowledge of how the totality of those attributes will perform as a whole. It is possible that the combination of those attributes as described by the designers might not even look like a bike as we know it. How will the combination of those attributes perform under stress? If the lightest of titanium bike frames is used, how will it perform at speed? If the thinnest of tyres is chosen, how will they perform when a heavy cyclist rides against a strong headwind on a wet and greasy road? It is simply not possible for the designers to know the answer to all these questions. As a result, the design is premised on both fact and assumptions, and it is not clear from the design which is which.

To compound the problem, the design deliverable is a highly technical document which is probably unintelligible to many customers. The Contract Model requires the customer to approve the design deliverable before it is handed over to the programmers. However, it is not uncommon for the contract to provide that this does not in any way commit the customer to the design deliverable. The supplier must still ensure that the finished software complies with all of the Output-Based Requirements.

In other words, contractually speaking, the design process does not advance the position of either the customer or the supplier. There is no opportunity for either party to exploit any new insights or information that is gained during the design process. The contract literally treats the design process as a document-writing exercise.

Sequential Development, as mandated by the Contract, effectively encourages the supplier to structure its resources such that the designers and programmers are in separate teams. With the programmers physically and often geographically removed from the designers, it is likely that the only information that the programmers have from which to

base their build of the software is the design deliverable. They have little insight into the business objectives of the customer, and they are unlikely to have access to any representatives from the customer to work this out. By the time the programmers start work, the team of designers has probably been disbanded and the various individuals assigned to other IT projects.

There is a significant risk of misunderstandings and misinterpretation of the design deliverable by the supplier's programmers. It is also inevitable that when the programmers start coding, they will find things that question the design. How can the programmers resolve those questions effectively in such circumstances?

High quality software design involves an interplay of many factors. The system's central concepts must work together as a smooth, cohesive whole. There is a fine balance of the various attributes of the software, such as the flexibility, maintainability, efficiency and responsiveness. Plus the users' overall experience of the software needs to be taken into account. How intuitive is it to use? How well does the software deal with the idiosyncrasies of the users? How well does it keep up with changes in the domain? How well does it solve the users' problems? It is virtually impossible to strike the appropriate balance when each of the customer, the designers and the programmers are kept at arms' length.

The development process is initiated with the Output-Based Requirements, which are specified in the contract. By definition, these are collected and written by the customer before the project starts. At this point in time, the customer's knowledge and understanding of the ultimate solution is at its least well formed. It is therefore counter-intuitive for the customer to decide what it wants at a time when it is least well equipped to do so. Yet by incorporating the Output-Based Requirements in the contract and calling them 'requirements', it becomes mandatory for the supplier to deliver software that meets all of these so-called 'requirements'.

On closer analysis many of the Output-Based Requirements are not in fact mandatory. Instead, many of them are in fact architecture, design, implementation and installation/configuration constraints that are unnecessarily specified as requirements. The customer often inappropriately specifies how to build the software system rather than what the software system should do or how the software system should perform. This happens because the customer incorrectly assumes that a common way to implement a requirement is actually the only way to implement the requirement, so they confuse the implementation with the requirement. By unnecessarily specifying constraints, the customer inadvertently prevents the supplier from selecting the optimal solution for the problem [13].

In many contracts the fees payable by the customer to the supplier are determined with reference to the Output-Based Requirements. These provide the basis on which the supplier arrives at a fixed price, or they determine the amount of resources that are in fact used by the supplier (for example, the fees are based on time and materials or the number of features, function points or story points). The more Output-Based Requirements that are detailed in the contract, and the more expensive the IT project is likely to be. If the supplier is rewarded for delivering output, it is incentivised to create more output.

Redundant system features can be damaging to the overall integrity of a product. Redundant features do not necessarily add value: they lead to greater complexity and potentially render the product less intuitive to use. For example, some digital watches have so many buttons and are so complex that it is virtually impossible to carry out one of the most basic requirements such as changing the time without referring to the user manual.

A combination of a single set of Output-Based Requirements together with Sequential Development means that the customer only gets one opportunity at the start of the IT project to describe its requirements, without paying a premium for additional requirements under the Change Control Mechanism. As a rational response to this situation, the customer tends to err on the side of caution in the contract by over-specifying the Output-Based Requirements. The customer is effectively encouraged to ask for anything and everything it might possibly need, as it generally doesn't know at the beginning of the IT project exactly what it does need.

This inevitably leads to an unnecessary padding of the features in the software. What is built is the 'biggest possible' product, not the 'minimal valuable' product. Not only are additional features requested that may not be needed, but each feature may be specified to an unnecessarily high standard, known as 'gold plating'.

A lot of the features in bespoke software are simply not used. Our own experience suggests that more than 40% of the features are redundant. The findings of the DoD suggest that as many as ninety eight per cent (98%) of the features could be redundant [14].

5. A poor return on investment

For too long software development has not been held to account as a business activity. Many organisations regard software development as a

necessary evil which places enormous strain on their finances. They treat the IT department as a cost centre rather than as a value creation centre. However, in most organisations there has to be a business case for acquiring software before the organisation will contemplate making the investment. Central to any business case is generally the desire to increase profits, to protect revenue or to reduce costs.

This in turn means that any IT project must be assessed by its economic impact. The cost of the resources invested by the customer in the software development activity should be directly referenced to the resulting increase in profits, protection of revenue or reduction of costs. It is only when the software development activity is analysed in these terms that the customer can assess whether it has achieved a good return on investment.

The Contract Model mandates a development process that has been proven to be neither efficient nor cost-effective. It is inevitable that the customer will only achieve a poor return on investment.

As discussed earlier, Sequential Development does not involve the expertise of the supplier's programmers until later on in the IT project. If the programmers who understand the details of the software system are not involved early in the IT project, the pitfalls will not become apparent until much later in the process, when their discovery often results in much more costly redesigns and cascading delays [15].

"The solution to this well-known problem is not to complete the entire design and get sign-offs. The solution is to involve those who will have to implement and live with the design early in the process and drill down as much as is necessary to be sure that lurking problems have been uncovered and addressed [16]."

Another unwelcome consequence of a combination of a single set of Output-Based Requirements together with Sequential Development is a lack of transparency for the customer of a break-down of costs incurred in the IT project. The customer is not provided with any indication of the relative cost of the individual Output-Based Requirements. If the customer was privy to this information, it may reassess whether it actually needs all of the specified Output-Based Requirements.

The customer also does not have transparency of the relative cost of the performance qualities of the resulting system as expressed in the Output-Based Requirements. For example, many customers do not appreciate the relative cost implications of increasing system availability from, say, 99.9% to 99.95%. This is not a linear progression. Instead, as system availability approaches 100%, the costs increase exponentially. In order to achieve a higher level of availability it may be necessary to run mirror systems. The costs of the hardware may be doubled or even quadrupled to ensure there are sufficient levels of redundancy.

The same is true for other performance qualities. As the performance quality approaches perfection, its associated cost increases exponentially towards infinity. If the customer is unaware of the cost implications, it may inadvertently ask for performance qualities that cannot be justified on a cost-benefit analysis [17].

Sequential Development leads to enormous waste. At each stage of the development process the Output-Based Requirements are worked on by a different team. The transfer of the Output-Based Requirements from one team to the next is known as a hand-off. Each hand-off leads to a loss of knowledge, some of which is never replicated and the rest of which has to be built up again by the next team:

"Each handoff of an artefact is fraught with opportunities for waste. The biggest waste of all in document handoffs is that documents don't - can't, really - contain all of the information that the next person in line needs to know. Great amounts of tacit knowledge remain with the creator of the document and never get handed off to the receiver. Moving artefacts from one group to another is a huge source of waste in software development [18]."

The Contract Model mandates that all of the Output-Based Requirements flow en masse through each of the gated stages of the development process. Large batches seem attractive because they appear to generate economies of scale that increase efficiency. The idea of large batches conjures up images of large numbers of car parts moving along the conveyor belt in a factory as they are assembled.

However, whilst it may be possible to achieve economies of scale in manufacturing, it is not the case in software development. In software development this efficiency gain is an illusion. Indeed, there can be strong diseconomies associated with large batches in software development. There are many ways in which a large batch adversely impacts the economics and performance of software development. Here we examine just a few of them.

The supplier's development team is presented with all of the Output-Based Requirements en masse. This often leads to the development team becoming overwhelmed by the scale and complexity of the IT project, leading to what is commonly referred to as 'analysis paralysis'. In a study conducted by Capers Jones in 2000 it was found that as the size of a IT project increases (measured in language-independent function points), the monthly productivity of staff decreases significantly [19].

The batch of Output-Based Requirements acquires the properties of its most limiting element. For

example, if one module of code is particularly challenging, it may potentially hold up the delivery of the entire IT project. It is possible that this particular module is not even a priority for the customer. The supplier's development team is unlikely to be given any steer on which of the Output-Based Requirements are the most important for the customer, so they are not capable of making an informed decision. In any event, if the supplier does not deliver software that meets all of the Output-Based Requirements, the supplier will be in breach of the contract. The Contract Model does not take into account the relative value to the customer of any particular Output Requirement, and it is hard to ascertain this from the contract.

Any performance quality required by the customer applies by default to all of the Output-Based Requirements. For example, if the customer requires the software to be 99.9% available, the entire software system must be 99.9% available. As explained earlier, it can be very expensive to achieve 99.9% availability. However, it may be the case that only the end-user interfaces need to be 99.9% available. For example, if an end user places an order to buy something on a website, an order confirmation should be generated 99.9% of the time, but it may not be necessary for the system to check immediately whether the requested item is in stock.

By breaking down the overall system into constituent modules, it may be possible to be more specific about which performance qualities should apply to which modules. This is an effective way of reducing the costs involved in building the software system.

If the supplier is rewarded for delivering output, it is incentivised to create more output. Unwanted and/or unused features in software lead to an enormous amount of waste. It takes longer to develop the software successfully and, as a result, the development process becomes unnecessarily expensive. Once developed, the overly-engineered software is more prone to error and costs more to maintain.

We consider it better practice to only develop those features which deliver immediate value to the customer. Further features should only be added as and when there is a proven business case that can be aligned to the return on investment. This keeps the customer's options open: they can choose whether to continue to invest in the software or whether to invest in another area which will provide a greater return on investment. In any case, redundant features and features specified to an unnecessarily high standard may further extend the schedule and increase the associated costs.

6. An alternative approach

We believe that the time has come to broaden the traditional approach to contracts and to form a new perspective based on complexity science. Over the last decade the principles of complexity science have been applied to governments and a broad range of industries, and their usage has been slowly extended in the field of project management.

The Stacey Matrix is a useful map for navigating through the concepts and field of complexity [20]. It provides helpful guidance on selecting the appropriate management style based on two dimensions: the degree of certainty and level of agreement on the issue in question. From these two dimensions four different contexts are identified: simple, complicated, complex and anarchy (also referred to as chaos).

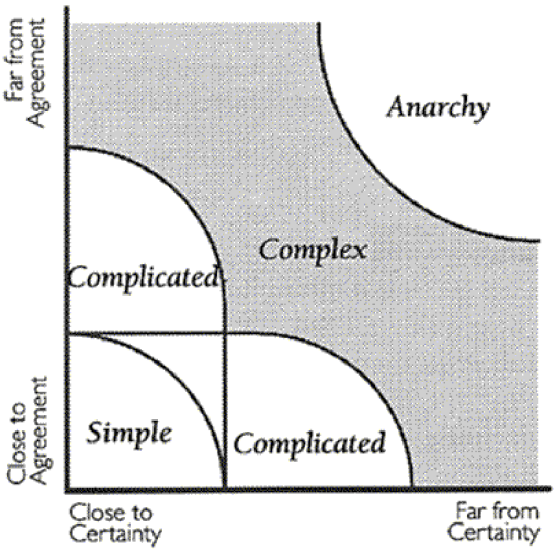

Figure 1. Stacey Matrix

Traditional contracts work well in simple contexts (the realm of *'known knowns'*) and complicated contexts (the realm of *'known unknowns'*). In each of these contexts there is relative certainty and stability, and there is a reasonably clear relationship between cause and effect (although in a complicated context there are multiple right answers and expertise is required to determine the right answer). So it is possible to create a plan and monitor performance in terms of conformance to the plan.

However, the context for software development is complex. This is the realm of *'unknown unknowns'* where unfortunately it is not possible to impose the *'right answer'*. Demanding certainty in the face of uncertainty is dysfunctional. Instead, complex problems require a more experimental mode of

management.

In a complex context the role of management is to set goals and constraints that provide boundaries within which creativity and innovation can flourish. Performance is measured in terms of movement towards the goals rather than assessing compliance to an earlier plan. Inspection of the IT project should occur frequently and often so that any unacceptable variances in the process can be detected. If the inspection highlights that one or more aspects of the processes are outside acceptable limits and that the resulting product will not achieve the stated goals, the process and/or the solution being developed must be adjusted.

This management style, which is appropriate for complex contexts, provides a theoretically sound basis for the structure of a contract model for software development. However, the enormity of moving to such a model should not be underestimated. Significant education is required and this would involve a large cultural shift.

7. Conclusion

Much research and many studies have been carried out on IT projects to try to understand why there are so many failures and why the extent of those failures can be so great. Yet to date the Contract Model has been largely ignored. We believe that the Contract Model is in need of a total overhaul. Indeed, a new model is required that is based on complexity science. With our increasing dependency on IT and escalating costs of IT spend, an overhaul of the Contract Model cannot happen soon enough.

8. References

[1] Report by the Comptroller and Auditor General of the National Audit Office, "The failure of the FiReControl project", 1 July 2011.

[2] Alexander Budzier and Bent Flyvbjerg, "Double whammy – How ICT IT projects are fooled by randomness and screwed by political intent", Said Business School working papers, Oxford, England, August 2011; Michael Krigsman, "Levi Strauss: SAP rollout 'substantially' hurt quarter", www.zdnet.com, 18 July 2008; John Sterlicchi, "Software disaster slows Levi Strauss's resurgence", www.guardian.co.uk, 14 July 2008.

[3] The Standish Group, "The CHAOS Report", 2011. Whilst the methodology and the conclusions of the CHAOS Reports have been called into question, these reports still remain the most comprehensive surveys to date, and in our experience anecdotal evidence is largely consistent with the findings of the CHAOS Reports.

[4] Infra. Reference [2].

[5] Infra Reference [1].

[6] Daniel McCracken and Michael Jackson, "Lifecycle Concept Considered Harmful", ACM Software Engineering Notes, April 1982.

[7] Capers Jones, "Applied software measurement; Global analysis of productivity and quality" McGraw Hill, 2008; Barry Boehm and Philip Papaccio, "Understanding and Controlling software costs", IEEE Transactions on Software Engineering, Vol. 14, No. 10., October 1988.

[8] Eric Ries, "'The lean startup: how today's entrepreneurs use continuous innovation to create radically successful businesses", Portfolio Penguin, England, 2011.

[9] Susan Atkinson and Gabrielle Benefield, "The curse of the change control mechanism", Society for Computers & Law publication, England, May 2011.

[10] Infra. reference [1].

[11] The results of the study were presented at the 5th Annual Joint Aerospace Weapons Systems Support, Sensors, and Simulation Symposium (JAWS S3) in 1999.

[12] Infra. Reference [2].

[13] 'Common requirements problems, their negative consequences, and the industry best practices to help solve them' by Donald Firesmith, Journal of Object Technology, Volume 6, No. 1, January – February 2007.

[14] 'Managing the design factory: a product developer's toolkit' by Donald Reinertsen, The Free Press 1997.

[15] Infra Reference [11].

[16] 'Leading Lean Software Development: Results are Not the Point' by Mary and Tom Poppendieck, Addison-Wesley (2010).

[17] 'Competitive Engineering' by Tom Gilb, Butterworth Heinemann (2007).

[18] 'Lean Software Development: An Agile Toolkit' by Mary and Tom Poppendieck, Addison-Wesley (2003).

[19] 'Software Assessments, Benchmarks, and Best Practices' by Capers Jones, Addison-Wesley (2000).

[20] 'Complexity and Creativity in Organizations' by Ralph Douglas Stacey, Berrett•]Koehler Publishers (1996).