Selling Software by the Pound

In 2003, Methods & Tools published the article "ASP Tools for Software Development", where ASP stand for Application Service Providers. This was the "acronym of the day" for companies that were offering hosted software tools. Most of the companies mentioned in this article are still active today, but there is a new marketing label attached to their activity. The media prefer to speak about SaaS (Software as a Service) or give to this phenomenon the nickname "The Cloud". Even if the concept remains unchanged, the market of hosted software development is evolving towards a broader range of tools offering. The historical market is still focused on project management or bug tracking functions, but the proposed services are now extended to all software development aspects. Amongst of the most recent initiatives for software development tools are the Amazon SimpleDB. It provides a simple web services interface to create and store multiple data sets. We are also transitioning from user-based fee model ($/month by user) towards usage fees ($ by gigabytes used). For its SimpleDB product, Amazon will bill you according to machine utilization, data transfer and data storage.

How will this change the life of software developers? I think that the main impact of service-based software development will be on architecture. Modularization has been here for a long time, but there is still a large tendency for coders to mix every aspects of the application (interface, logic, data access) in the same program. Having to deal with exterior services should put more pressure on a better separation of functionalities. This will also change the way you consider deployment. Instead of having to deal with internal operation guys, you will talk to a guy from Amazon or Google. Will he be nicer? I don't know, but sometimes I know it will not be very difficult :) The "virtuality" of Cloud computing bring the promise to solve all scalability and performance problems. I admit being skeptic about this claim. There have always been performance problems when developing applications. Even with the best infrastructure, you cannot prevent developers from making bad performance choices. Will we have the tools to monitor the Cloud? One positive aspect is that people developing for the Cloud should be able to have the same testing environment than the production environment. Many of us have suffered the fact that setting a testing environment seems to be difficult for the infrastructure guys. Testing in a slightly different software infrastructure could often create bad surprises when the application is transferred in the real production environment. Finally, the Cloud approach will increase the shift towards the user for the responsibility of estimating the budget for running applications. All operating costs could now be more clearly defined and related directly towards user activity and application scope. Most suppliers promise that the Cloud solutions work with current technologies and require only small adaptations to run on their infrastructure versus internal servers. This mean that changes in the programming part could be minimal. This new technology should however favor the launch of Web start-ups that need scalable infrastructure and variable costs for their new applications. Thus the main change for software developers could be the availability of more interesting jobs.
UML vs. Domain-Specific Languages

Mark Dalgarno, mark @ software-acumen.com

Matthew Fowler, matthew.fowler @ nte.co.uk
New Technology / enterprise, http://www.nte.co.uk/

Introduction

The software industry has a big problem as it tries to build bigger, more complex, software systems in less time and with less money. With C++ and Java failing to deliver significantly improved developer productivity over their predecessors it’s no surprise that around 40% of developers [1] are already using or are planning to use code generation approaches to tackle this problem.

There are by now many cases studies of the successful application of code generation tools and technologies and it is our view that allowing developers to raise the level of abstraction over that supported by general-purpose programming languages is the best bet for development organisations wishing to address the productivity problem noted above. Although there are other approaches to raising the level of abstraction – such as framework evolution, or even programming language evolution, code generation, because it is flexible and fast, has the advantage of being able to adapt to new environments relatively quickly.

In this article, we consider the two most popular starting points for code generation:

• UML for program modelling, part of the OMG’s Model Driven Architecture (MDA) approach [2], and
• Domain-Specific Languages (DSLs), little languages that are created specifically to model some problem domain.

As well as introducing both approaches, our aim is to offer advice on their usefulness for real-world development. We also ask whether they are mutually exclusive or if in some circumstances it can make sense to combine them.

UML and MDA

Experience of using UML as a modelling language is widespread and so using UML to express what is required in a system and generating code from that is acceptable for many organisations.

"Code generation" in UML originally meant a very low level of generation - converting classes on a diagram into classes in C or Java. Experience has shown that this level of modelling does not give any business benefit when applied to complete systems. However, by using more specialised or abstract modelling elements it is possible to increase the amount of generation, as we shall see. This approach was adopted by the OMG in 2001 as part of its MDA standard.

MDA was developed to enable organisations to protect their software investment in the face of continually changing technology "platforms" (languages, operating-systems, interoperability solutions, architecture frameworks etc.). If the design and implementation is tied to the platform, then a platform change means a complete rewrite of a software system.
To avoid this, MDA proposed to separate "the specification of system functionality from the specification of the implementation of that functionality on a specific technology platform". The specification of system functionality is a Platform Independent Model (PIM); the specification on a particular platform is a Platform-Specific Model (PSM). The PSM can be annotated by developers to provide advice or guidance for the final code "text" generation step - which creates the source code and configuration files.

To reap the business benefits of this approach, the PIM must survive platform change and be reusable across platforms. The implications are that:

- models become first-class artefacts in the development process, rather than being ignored after a certain point: if you change the PIM, the functionality of the delivered system will change
- code generation becomes important: mapping the PIM to the PSM by hand is costly and error prone, whereas automatic mapping to the PSM can significantly reduce the cost of a transition to a new or upgraded platform.

MDA defines a set of standards for transforming models that was finally completed in 2007. These standards are well supported in the telecom and defence sectors, where there is a history of investing in development tools as part of large projects. In the commercial world, the lack of standards led to companies supporting the "model-driven" approach (MDD - development, MDE - engineering etc.) using a variety of tools to transform UML models into working systems - "pragmatic MDA", as it was called.
The industry position of UML also means that developers can choose from a wide variety of vendors for their MDA tooling. Furthermore, vendors typically provide additional products based on the MDA approach, reducing the investment for an individual company to adopt MDA.

However, there are some issues in the use of MDA. First is the expression of detailed business logic. While 90-95% of a commercial information system can be generated from a UML model, there is a point where the business logic is not general and so not amenable to a code generation solution. There are two approaches to expressing the business logic. The "purist" approach is to model the business logic; one of the MDA specifications covers this approach. The "pragmatic" approach is to leave holes in the generated application for the hand-written business logic; this is most popular where there is a rich, standardised development environment, like Java or C#/.NET.

Another issue is the low level of UML and the looseness (or generality, to put a positive slant on it) of its semantics: a common criticism is that UML is too big and vague to be effective. This assumes that the only "code generation" possible is the very low-level code generation described earlier - the assumption is that UML can't express more abstract or specialised concepts.

But this criticism ignores UML’s profile feature. "Pragmatic MDA" vendors use this to specialise UML. To do this, they define profiles so developers can create models with a more specialised terminology and associated data. On top of that, vendors add their own validation to tighten up the UML semantics. The result is a domain-specific subset of UML if you like.

Using UML profiles gives as much expressive power as DSLs: stereotyped classes typically equate to the DSL terminology (the 'nouns' - see sidebar) and stereotyped relationships are the same as for relationships in graphical DSL terminology. In other words, either approach can express concepts of arbitrary levels of abstraction.

There are two main problems with using UML with profiles to define new modelling languages:

- With current UML tools it is usually hard to remove parts of UML that are not relevant or need to be restricted in a specialised language
- All the diagram types have restrictions based on the UML semantics.

For example, NT/e is in the process of building a graphical DSL for a novel middleware product. The key to this is being able to model methods as first-class model elements. In theory we should be able to do this using action diagrams, but in practice there is too much other baggage that drags along with it. As we will see below, the DSLs are built from the ground up, so the modeller is not confronted with extraneous UML semantics or modelling elements.

Despite this, defining a high-level UML profile has historically been the best commercial approach to realising MDA. To produce a new profile is relatively cheap. On the marketing front, the installed base of UML tools and the understanding of the practice and benefits of modelling mean MDA products can be positioned as 'add-ons' rather than a completely new paradigm.
Domain-Specific Languages

Introduction

Although DSLs and DSL tools have been around for a while now it is only in the past few years that interest in this area has really taken off – partly in response to Microsoft’s entry into this space with its DSL Tools for Visual Studio.

As noted above DSLs are little languages that can be used to directly model concepts in specific problem domains. These languages can be textual, like most programming languages, or graphical. Underpinning each DSL is a domain-specific code generator that maps domain-specific models created with the DSL into the required code.

One way to think of how to use a (graphical) DSL is to imagine a palette containing the boxes and lines that correspond to key concepts and relationships in your problem domain. Modelling with this palette involves selecting the concepts you wish to model and ‘painting’ them onto your canvas. ‘Painting’ different types of lines between these concepts can then create different types of relationships between the concepts. An advantage of the DSL approach is that the modelling environment can constrain and validate a created model for the domain's semantics, something that is not possible with UML profiles.
What about the 'domain' in 'Domain-Specific Language'

The first description of DSL's I (MF) heard said, "you use the concepts from the target industry as your modelling types". I was confused as to how that related to modelling and programming. Thinking about nouns and verbs helped me understand what's going on here. Let me try to explain...

Computer languages have a syntax (e.g. ';' to terminate commands), a general semantics (e.g. expressing conditional clauses, loops) and a way of defining types for your problem domain (e.g. Customer, Order, Product). This is precisely what we do with day-to-day language:

- we invent new nouns to describe new types that we encounter
- but we don't change the syntax or underlying semantics of English to write about a particular problem industry or technical domain.

Most people now recognise that the syntax is not worth arguing about - most syntax expressions are transformable to each other. It's more interesting to start defining types as extensions in the language:

- For a DSL in a technical domain, these types can be things like 'page', 'entity', 'service', which can be mapped onto a specific technology domain;
- For an industry-specific DSL, such as for the insurance industry, we have types like 'policy', 'cover' or 'party' etc.
So, first of all a DSL can give you a range of new nouns, drawn from the domain's standard terminology. So far, this can be done without changing the general semantics of the language. But then we get to the "verbs" - the relationships between the nouns. For a domain-specific language, these relationships are the active part of the domain - they're what the players do to each other! The end result is that the nouns and verbs you use to describe industry-specific situations end up forming their own language within a language, as any programmer's spouse will tell you. The DSL approach is a formalisation of this natural process.

The ability to express the relationships between concepts of the domain is what makes a DSL specific, and potentially very powerful. In basic UML - without profiles - the relationships tend to apply generically. For example, a UML association can relate any type of class. But in a DSL, relationships can be constrained - e.g. an insurance policy can only be comprised of fire/theft/damage etc. - not people. Furthermore, industry-specific information about the relationship can be added into the model. This can produce a graphical modelling language that is industry-specific, precise and powerful. For domains with limited variations, the relationships may have enough information needed to generate all the implementation code.

If this makes the whole process sound very simple, it is ... up to a point. The complexities start when you implement those 'nouns' and 'verbs' in a real implementation platform: then you have to use existing components built in another language (as for a software product line for example), or use another layer of code generation. But when these implementation issues are resolved, the DSL approach, particularly graphical DSLs, gives a working presentation of a system that is appealing and understandable to business stakeholders.

Tool Support

Tools to support the definition of DSLs and domain-specific code generators have been around for a while now but have been far less commonly available than MDA-based toolsets, with only one or two vendors offering mature products. Given this, many developers using DSLs have chosen to go down the road of implementing their own generators with varying degrees of success due to the complexity of this type of work.

This is now changing with the increasing availability of tooling to support DSL and generator creation from companies such as MetaCase, Microsoft and as part of the Eclipse Modelling Framework. To some extent these have reduced the skill levels required to create DSLs and domain-specific generators.

Which to use?

Given that both approaches now have momentum behind them in the form of vendor support, successful case studies, and increasing industry awareness the question arises for developers of which approach to adopt (assuming developers are completely open-minded!). Perhaps the first thing to note is that developers in organisations, or supply-chains, where use of UML or Microsoft technologies is mandated may find it politically difficult to choose a 'competing' approach. Modelling and code generation is just one part of the software life cycle, albeit an important part, and must fit in with the rest of the organisation’s tooling and processes.

Similarly, in industry sectors such as real-time systems engineering where intensive work has already been undertaken to support the particular modelling needs and constraints of the sector (e.g. with development of the SysML customisation of UML [3]) developers may not find it cost-effective to create their own unique UML profiles or DSLs that don’t take advantage of this prior work.
As noted above, a basic DSL can be produced using UML profiles and this will often be a viable and relatively quick approach for a first-time code generator. However, the baggage that UML brings to the problem can confuse novice modellers; to avoid this, generator developers may choose to directly proceed to building their own DSLs – either with tool support or in a completely bespoke manner.

It’s also worth mentioning that in many cases software systems can only be implemented with multiple modelling languages and code generators addressing different aspects of the overall problem. There’s nothing to stop developers, who on the whole are a pragmatic bunch, from using a hybrid approach that combines UML with DSLs to create solutions that draw on the strengths of each approach, and indeed this is what some organisations, such as NT/e, have done very successfully.

**Conclusion**

So what is the outlook for the industry? It’s our belief that as a basis for modelling for code generation, UML tools - in their current form - will gradually lose "market share" to DSLs: DSLs can be more direct, appealing and easier to use for a wider range of users. However, UML vendors, with their strong background in code generation approaches, can compete by adding a non-UML modelling and meta-modelling ‘surface’. Combined with their tool's existing UML features, this would make an attractive combination product for many companies.

**References**


[4] [http://www.codegeneration.net/tiki-read_article.php?articleId=81](http://www.codegeneration.net/tiki-read_article.php?articleId=81)


  Leading experts including the OMG’s Andrew Watson, Microsoft’s Steve Cook and this article’s co-author Matthew Fowler discuss the question of ‘UML vs. Domain-Specific languages’.

Advertisement – Knowledge Surfing - Click on ad to reach advertiser web site

Dive Into the Internet – Get More than Just Simple Surfing

The Internet is an ocean, a vast, deep reservoir of complex information currents with unsuspected cross connections. Equip yourself with a sophisticated tool to discover the global layers of knowledge and insights of the web.

Download for free: [www.p-navigator.com](http://www.p-navigator.com)
We Increment to Adapt, We Iterate to Improve

Jurgen Appelo, jurgen @ noop.nl
www.noop.nl

Introduction

Traditional views on iterative and incremental development are under pressure due to new ideas like Kanban development and continuous deployment. The original views were never solid in the first place, because the agile methods have not agreed on frequencies, cycle length, pipelines and other aspects of evolutionary rhythms in software projects. This article is my attempt to evaluate the many directions agile practitioners have taken in iterative and incremental development. I also propose some new definitions, as it is my firm belief that the existing definitions need to adapt to an environment that has changed significantly.

The Way Things Were

Software engineers and other living creatures share an important feature: they have rhythms. The formal study of biological temporal cycles, such as daily, tidal, weekly, seasonal, and annual rhythms, is called chronobiology. Despite the fact that the study of temporal cycles in the discipline of software engineering lacks a formal and nice-sounding word (chrono-software-engineering?), significant progress in this area has been made in the last couple of decennia.

Many authors have elaborated on the importance of both iterative and incremental development, particularly in the last ten years, though few of them have made serious attempts to explain what the differences between iterations and increments really are (Jeff Patton, 2007, Alistair Cockburn, 2008). Even worse, some methodologies – in particular the Unified Process and Scrum – have muddled the waters by making iterations and increments equivalent to each other (Per Kroll and Philippe Kruchten, 2003, Ken Schwaber, 2004, Mary and Tom Poppendieck, 2007), which renders it difficult for project managers and process managers to decide how to handle temporal cycles in their software projects. And communication among agile practitioners doesn’t get any easier when authors of the other agile methods – like Extreme Programming and DSDM – do not agree on the structure of the iterative and incremental cycles. The latest addition to the methodological ecosystem is Kanban development, which offers yet another perspective on iterations and increments (David J. Anderson, 2007). With so many methods beating the drum for very different life cycles, I believe it is hard to follow any suggestions in this cacophony of rhythms. Being almost deaf after having listened to everybody for a while, I think it is time to achieve some harmony.

Some often cited definitions of incremental and iterative development (Wikipedia, Alistair Cockburn, 2007 II), are as follows:

1. **Incremental development** is a staging and scheduling strategy in which various parts of the system are developed at different times or rates, and integrated as they are completed.

2. **Iterative development** is a rework scheduling strategy in which time is set aside to revise and improve parts of the system.

While these definitions have been very useful for some time, I now think that they have begun to break down under several changes in our software development environments in the last decade. Allow me to explain.
Another Fine Mess

First of all, the wording of the definition of incremental development still reflects the old way of treating the construction of big systems: break them down into manageable parts, schedule the parts to be built separately and then assemble the system from those parts. – I remember well being taught that this was the basic idea behind increments. It was in my youthful and more rhythmic days, back in the 80’s. – But nowadays, in the agile era, the purpose of incremental development is something entirely different (Craig Larman, 2004). We have learned to build better products by putting them into the hands of our users. Only the people who actually use our products are the ones who are in the best position to tell us what we did wrong and how we can do better. This was the whole idea behind evolutionary and adaptive software development, both of them popular terms in our field in the 90’s, until agile development became the norm. Therefore, the purpose behind incremental development shifted from the plan-driven idea of the decomposition and simplification of a big system to the user-driven idea of delivering business value to customers in an ever-changing environment. Incremental development itself has evolved into a very different creature!

The Unified Process and Scrum can be blamed for confusing the incremental and iterative strategies, but I believe that I understand the reason behind it. At the start of the whole evolutionary/adaptive/agile wave, iterative development was seen as a rework cycle within the (often bigger) cycle of increments (Alistair Cockburn, 2005 II). – Other methods, like Extreme Programming, still see iterations as part of a bigger release cycle. – But, due to the evolutionary/adaptive/agile approach, incremental delivery itself, just like iterative development, became another learning and rework strategy.
It appeared that users often preferred working with a functionally limited system – sometimes being only half correct and half finished – over having to wait endlessly for a system that was being tuned to perfection. The frequency of product releases to users was increased, and software developers needed to polish their rhythmic abilities so they could dance to both an iterative and an incremental rhythm. I believe this is the main reason why some experts simply decided to lump iterative and incremental development together: both strategies turned out to be associated with learning and rework.

The second significant factor in this evolutionary anarchy is that the Internet and other new technologies have flung us into a whole new era where releases can be delivered to users even faster than we are able to learn and adapt. The recent practices of continuous delivery (deploying each new feature to users immediately after its development) and Kanban development (decoupling iterations from increments) have further helped to overthrow the traditional idea that the iterative rhythm is performed inside the incremental rhythm (Corey Ladas, 2007, Brad Appleton, 2008). It is now clearer than ever that the iterative cycle is perfectly able to stand on its own.

Considering accelerated development rhythms and the several options for phasing incremental and iterative cycles, it seems to me that planning and scheduling activities are now under pressure. The original definitions explicitly align scheduling with the incremental and iterative cycles, which has led many agile practitioners to perform planning activities at the start of each new cycle (Mary and Tom Poppendieck, 2007, Ken Schwaber, 2004). But some people are suggesting these days that continuous planning is an even better way of dealing with increased project complexity (Corey Ladas, 2007). This is yet another evolutionary branch in an already crowded ecosystem of best practices.

It seems to me that software development follows the theory of punctuated equilibrium, where an ecosystem can be shaken up completely by sudden bursts of many rapid evolutionary changes and alternative forms. It is clear that we need to adjust our views on increments, iterations and planning, or we will never be able to understand which approach is best for our projects. This is my attempt at cleaning up this mess that we have gotten ourselves into.

We Increment to Adapt

For development of software systems in the 21st century the definition of incremental development might look like this:

- **Incremental development** is a staging strategy in which various parts of the system are developed, and delivered to the user, at different times, *with the intention to adapt to external feedback*.

The part in italics is what distinguishes this definition from the old one. The goal of an increment is not (in the first place) to break down complexity, but to expose a growing system as early as possible to the environment that it must be able to survive in. This creates a feedback loop with feedback from real users, real hardware, real infrastructure and real technical interfaces, which enables learning and adaptation. A software project is a complex system that must continuously seek an optimum value in a rough and challenging fitness landscape, formed by numerous quality criteria. These criteria – and therefore the performance system against which the project is continuously measured – are always subject to change. You will never know in advance that color-blind gnomes with arithmophobia are unable to read numbers with more than two digits from a screen, when printed in 8pt Arial Narrow. You need real feedback to know that. Biology has introduced reproduction and the five senses to allow for continuous
feedback. Learning, and adaptation to changing environments, takes place in the DNA of species, and in the brains of individual organisms. Software projects achieve this ability through increments (Figure 1). We increment to adapt.

Figure 1: A cycle of learning, adaptation, building, and releasing

Incremental development is about looking back and evaluating what has been delivered, and making changes accordingly. As this feedback is exogenous (originating from the outside) the incremental cycle needs to align itself with any relevant external cycles, just like an organism’s reproductive and consumptive rhythms may be aligned with daily or annual cycles in nature. The frequency of incremental cycles varies widely among software projects, from once every few months to several times each day. For example, regular delivery of a product to a big and bureaucratic organization is very different from continuous daily deployment of new versions of a popular web site. As an exercise, try to think of your project as a kind of animal. Maybe it’s an elephant: big and smart, but lacking in maneuverability. Or it might be a shark: fast and deadly, and with a keen sense of smell. Of course, it could also be an ostrich: clumsy and stupid, but quite tasty. The natural adaptive cycle for your project will depend on what kind of beast it is. But you would do well to set it to the highest frequency that the environment and your animal are able to cope with.
Besides enabling a project to be adaptive, an incremental cycle has several other advantages:

1. Early and frequent deployment reduces the risk of running into the unforeseen problems that are often associated with “big-bang” scenarios, which makes it a strategy for risk reduction;
2. Being able to see work under construction, and in production, usually helps winning trust and goodwill from the customer;
3. Urgent business needs can be more easily addressed using staged releases, which improves the business value attributed to the system;
4. A sense of accomplishment, experienced after each delivery, is a good motivational strategy in people management;
5. As a psychological effect, higher productivity can be achieved among team members when they are faced with regular and reliable delivery dates;
6. According to the principles of Lean Software Development, keeping the amount of work-in-progress at a steady low rate helps to improve throughput and reduce delivery times;
7. The incremental cycle can be used as a vehicle for any other regular activities, like reflection, revising of the process, planning and assessments of resources.

Note that, in my opinion, these are all secondary benefits. Though most are considered to be very important for successful software projects, they can be seen as distinct best practices that simply appear to coexist happily with the incremental strategy, in some sort of a piggyback ride. However, the main motive behind incremental development is frequent learning and adaptation. Even with a big-bang scenario, lack of customer trust, lack of team motivation, low productivity and low throughput, you might still be able to deliver your system successfully – though your project management skills would need to be of beastly proportions. Without adaptation to the environment, however, your software projects are simply doomed to fail.

**We Iterate to Improve**

After redefining incremental development the definition of iterative development might look like this:

- **Iterative development** is a rework strategy in which various parts of the system are revised at different times, with the intention for team members to **improve the system they are building**.

The iteration cycle is sometimes affectionately called the “heartbeat of the project”. The goal of having iterations is to learn by doing. The feedback in this case is from people’s own knowledge and experiences, from other members in the team, and from people representing the customer. In reworking requirements, the user interface, the data model, the architecture and any other parts of the product, the software project learns to improve itself and build a better system. You only really understand how you need to (re)build the interface to your GnomeFontCalibrator subsystem once your team mates are actually calling its functions. Note that feedback from (intended) users – the customer-on-site in XP – belongs in this category too. But in this case they will be users who are part of the project; acting as people who (hopefully) have a better chance at predicting how the product is going to survive once it is released in its real environment. These users might or might not know about the specifics of disabilities among gnomes. But if they do, they can help you build and tweak your font calibration subsystem, in multiple iterations, until everyone in the team is pleased with the results (Figure 2). Some people call this iterative process “refactoring” (Kevlin Henney, 2007), while others reserve that word for improvements at a lower (individual) level and prefer to use the term “iterative
development” for the learning cycles that take place at the team level (Mary and Tom Poppendieck, 2007). Refactoring is usually only associated with improving the design and quality of code without making any functional changes. Iterative development is more than that. It includes functional changes that improve the system as it is perceived by potential users. We iterate to improve.

![Figure 2: A cycle of improvement, from bad product to good product](image)

Iterative development is about anticipation, about looking forward at what needs to be delivered and how it will have to cope with external forces. As this feedback is endogenous (originating from within the project itself) the iterative rhythm is usually less concerned with any external cycles. Its frequency can often be set to any value that works best for the team and the project’s stakeholders. It can be aligned with the incremental cycle, but it can also be set to either a higher or lower value.

![Figure 3: Different cycles for different parts of the project](image)
As the iterative cycle is purely an internal rework strategy, it is possible to maintain different rhythms for different parts of the system (Figure 3). The nature of the beast will determine what works best. Whether it’s an elephant, a shark or an ostrich that you are working with, as a team member you must be comfortable with the iterations that the team chooses to follow. Quite often, an iterative cycle comprises all activities in the software engineering discipline, from gathering requirements to system testing, and everything in between. However, some approaches – with the DSDM methodology as a well-known example – prefer to group activities in different cycles (Jennifer Stapleton, 2003). All these multiple iterative cycles make it clear that the term “heartbeat of the project” is actually not a fitting description. Except for some rare extraterrestrial organisms, multiple heartbeat rhythms are a recipe for disaster. But in many terrestrial organisms the heart, the brain, the respiratory system, the digestive system, the nervous system, and several other body parts do maintain their own rhythms. Some of them are synchronized, some are not. You may find yourself in a situation where you need to implement something similar for different groups of people, or different groups of features, in your own project. You can even have each feature follow its own iterative cycle, independent of any of the other features. The same applies to a different cycle per team member. As long as you keep in mind that everything in the system should be designed and built iteratively, you are free to pick and try your own cyclic patterns.

The strategy of learning by doing has several advantages:

1. It relieves people of the pressure of needing to get things right the first time, which prevents what some authors have called “analysis paralysis”;
2. One is better able to find a dedicated slot in the schedule for the refactoring of existing code, an activity that is often overlooked with tough delivery deadlines;
3. It is easier to define synchronization points for inter-dependent teams working on different parts of a system;
4. The cycle is often used to impose a temporary freeze on the system requirements, so that team members can concentrate on stable requirements before discussing any changes;
5. Similar to the incremental cycle, the iterative cycle can be used as a vehicle for any other regular activities, like reflection, planning and assessments of resources.

When practiced together, iterative and incremental development address uncertainty by stepwise improvements and adaptation.

Planning: The Third Cycle

Increments and iterations deal with the phasing of work items over time. Planning is more than that: it includes the phasing of resources, assigned to carry out this work. It has been said that each iteration is a planning window (Alistair Cockburn, 2005 I). But this is also true for each increment. The discipline of planning is complicated enough to warrant a book of its own. I will barely scratch its surface with just a few paragraphs.

For managing resources, cycles of work, dependencies and constraints the experts agree that the best approach is to use the principles of a rolling wave planning (Craig Larman, 2004) (Figure 4). The idea is that the level of detail in the planning should drop with time in the future. A project manager must focus on the details of both the current and the next increment and iteration, while trying to defer any decisions about the following cycles. It’s no use planning all activities of your Gnome Detonation HyperMatrix when you’re not going to start working on it for another year. The further away an activity is, the less time should be spent on its details.
Some people claim that planning nowadays is becoming more and more a continuous exercise (Corey Ladas, 2007). Planning frequently is better than planning rarely, but this doesn’t necessarily mean that you have to do this every day. In many projects the overhead incurred for each planning session requires us to figure out what the optimal frequency is, particularly when the customer and other stakeholders need to be involved in each session. This frequency can be matched with that of either the increments or the iterations, or both. But planning can also have its own rhythm. For example, with monthly iteration or increment cycles, you might still want to perform a weekly planning session, to check if your project is still on schedule. Add to this the possibilities of having multiple levels of planning, like daily planning, iterative planning, incremental planning and product planning, and you will see that you have lots of scenarios to choose from (Figure 5).

**Multiple Rhythms**

A software project can contain rhythms within rhythms, overlapping rhythms and parallel rhythms. Some combinations are depicted in Figure 6. In this figure all features (presented as boxes named A to L) are developed in up to three iterations each (with dotted, dashed and solid borders) and released in five consecutive releases (R1 to R5).
In the case of features A to E you can see that the team neatly iterated over them three times within each release. Such a sequence of iterations within a release is a well-known practice promoted by the Extreme Programming and Crystal Clear methodologies, among others (Kent Beck, 2005, Alistair Cockburn, 2005 II). Planning usually coincides with the start of each new increment (resulting in a Release Plan) and each new iteration (resulting in an Iteration Plan). You will want to follow this pattern when every increment you deliver to the customer must be of the highest quality. Your project may be like a horse or an antelope. As a prey animal it faces terrible risks and dangers when released in its environment. The project must always be in top-condition, and it should be very maneuverable. This pattern usually applies to off-the-shelf and mission-critical software.

Another possibility is to have iterations and increments follow the same cycle. The features F, G and H implement this pattern. It implies that features need not be fully matured before each next release, because in this case iterating over features coincides with releasing new versions. This pattern is followed by the Unified Process and Scrum, and it usually means that people use just one type of regular planning (Ken Schwaber, 2004). You may want to follow this pattern when users care most about delivered business value, and less about qualities. You are allowed to improve the quality of the system and patch up any problems in subsequent releases, if it means that users are able to start using some functions early. Your project may be like a tiger or an eagle. As a predator animal it is less concerned with risks and dangers. Its main concern is being very quick to respond to new opportunities as they come by. This pattern often applies to custom or bespoke software, created for specific customers.

The features J, K and L are iterated over in periods that span multiple releases. The idea here is to release whatever features happen to be completed at specific release points, optionally including any features that have sufficient business value without being fully mature. Though this practice has been around for ages, these days it has been adopted as part of Kanban development, a relatively new approach of decoupling iterations from increments and keeping work-in-progress down to the barest minimum (David J. Anderson, 2007, Amr Elssamadisy, 2007). Planning in this situation is sometimes continuous, but planning sessions can also simply follow the release cycle. You can adopt this pattern when availability of resources is unpredictable, or when the project can be treated as a never-ending maintenance/upgrade cycle. Your project can be like an elephant or a musk ox, usually considered neither predator nor prey. They feed and respond to the environment at their own pace and leisure, which is sometimes very often, and sometimes rarely. Many applications with large user bases, like open source projects and web sites, and projects using continuous deployment, belong in this category.

**Is Smaller Better?**

Biological rhythms are often in phase with external phenomena, like tidal, lunar and solar cycles. Similarly, the cycles in software projects may have to be aligned with external factors, like customer availability, hardware updates or seasonal market cycles. However, quite often there is only one external constraint: product releases must be carried out as soon as possible.

There is some disagreement among experts and methodologies about the best length of iterations and increments. Suggestions come in several sizes and ranges, like one week to three weeks (XP), four weeks (Scrum), and anything between two and eight weeks (UP and FDD). Other methods only advise to implement cycles, while refraining from making claims about optimal cycle lengths. A Google search on “iteration length” reveals noisy debates where agile practitioners are fending off each other, like animals in a crowded cage. You can even hear the whining and howling, if you listen carefully.
There are people who claim that shorter cycles are better (Corey Ladas, 2007), without balancing this view with the warning that increasing the frequency of a learning cycle may consequently lead to an increased burden in some other parts of the software project. In system theory feedback loops are applied to balance and stabilize complex systems. The shorter the feedback loop, the easier it is to steer them in the right direction. But the same theory also shows us that it is hard to predict the consequences of making changes, like reducing the cycle length. Some software developers think that there is enough evidence to support continuous deployment, and that perfection is a batch size of one. But this view, although it does have its merits, is too simplistic. Overhead costs tend to increase as delivery times decrease. It is true that cutting overhead costs will help you to speed up the delivery cycles, but the costs will never be zero.

Consider the analogy of organisms evolving continuously. Their biological cycles evolve with them towards any values that turn out to be optimal for the species, given the environment that they live in. A short reproductive cycle in a species enables it to adapt fast to a changing environment. But it comes at a cost: the organism cannot be too complex. So it follows that shorter is not always better, or else we would all be gastrotichs – a species of microscopic aquatic animals with the shortest lifespan on earth, which is about three days.

As for the variation in an iterative or incremental cycle, we can find much support in literature for steady rhythms. A regular cadence of iterations and increments in a software project helps people to maintain a level of high productivity for extended periods of time. Some positive reports have been published about minor frequency variations to accommodate temporary changes in the environment. One can imagine variation at the start and at the end of a project, for building up and winding down the system respectively (Alistair Cockburn, 2005 II). And sometimes circumstances might ask for cycles to be accelerated temporarily, right in the middle of a project. (I just spotted a number of gorgeous specimens of the human species out of the window of my living room and noticed several of my biological cycles accelerating rapidly. Rest assured that everything turned back to normal when they disappeared around a corner.)

The Pipeline Problem

Having handled planning, multiple rhythms, cycle length and frequency variations, it is time to turn our attention to the last and most controversial subject: the development pipeline. Let’s roll up our sleeves, because any fighting over this article is probably going to start here.

Many agile practitioners assert that all activities of software development, from requirements gathering and business modeling up to system testing and release management, should be performed within the context of one iteration (Ken Schwaber, 2004) (Figure 7). Though the experts admit that there is some intrinsic pipelining involved in the development of a software system (Mary Poppendieck, 2000) – a system tester can hardly do anything useful while the requirements are still being gathered – many claim we must try to have all team members in all disciplines work on the features simultaneously. The alternative, a pipeline with work moving from one discipline to the next, is considered a bad idea (Alistair Cockburn, 2007 I). Iterating over the same work leads to rework, and there is nothing that people hate more than redoing a lot of work because someone else further up the pipeline has changed his mind, again. Once work is passed along from specimen A to specimen B in a pipeline, specimen A must be finished iterating or specimen B will get very annoyed. Some growling and biting may even be involved. It is clear that iteration conflicts with specialization. There are three ways of dealing with this problem, neither of which is optimal for all projects.
Some people want us to solve this problem by getting rid of the specialists (Ken Schwaber, 2004). They have “cross-functional” teams that do not consist of business modelers, UI designers, programmers, database designers and testers. They simply have one role, called “team member”, and they do all of the work together. The ones analyzing the requirements are also the ones modeling the database and the ones testing the system. It is a team of generalists, who are likely to be good at creating solutions to problems that span multiple disciplines, but who are unlikely to be experts in all disciplines and every technological domain. This model (applied by Scrum and XP) is by far the easiest way to defeat the pipeline problem, because it gets rid of the pipeline itself (Kent Beck, 2005). It works fine, for small projects and small organizations. It also completely ignores everything society has learned since philosopher and economist Adam Smith pointed out in 1776 (in his landmark book *The Wealth of Nations*) that specialization leads to higher productivity and prosperity. Specialization is the reason why software developers do not bake their own bread, fix their own clothes or hunt for their own food, a few exceptions notwithstanding. The larger an economy or organization is, the more people will want to (and be able to) specialize in what they are good at. It is a mechanism that appears to work well, not only for individuals but for the whole as well. Scaling a generic role “team member” to a big organization would be like replacing all specialized animals in an ecosystem with a generalist species called “human”. Some would call this a destruction of biodiversity. Not a good thing at all.

If your organization is large enough to work with specialized people, you can try the second solution to the pipeline problem: overlapping the iterations (Craig Larman, 2004) (Figure 8). In this case you make sure that as many activities as possible are being performed in parallel – still in line with Scrum and XP principles – but you acknowledge the necessity that the Business Analyst might have to start a bit earlier with the next iteration than the Database Modeler and the UI Designer, who in turn may have to start a bit earlier than the Software Developer. And the Software Tester will join the iteration as soon as he has closed the curtain on the previous one. In this model, experienced by specialists following the Unified Process (Per Kroll and Philippe Kruchten, 2003, David J. Anderson, 2004), the tip of each new iteration will overlap with the tail of the previous one. Of course, you may be able to prevent overlapping of iterations.
in this case by having the Software Tester make fresh coffee for the Business Analyst at the start of each new iteration, and making sure that the latter returns this favor at the end. If you’re not concerned with productivity this will help to make your planning sessions somewhat easier.

A third way of dealing with the pipeline problem is to make sure that the pipeline consists of as few phases as possible, while ensuring that the work is iterated over a couple of times before being passed along to the next phase (Figure 9). For example, you might want to distinguish Design, Development and Testing as the three phases in a pipeline. Work is iterated in the Design phase first, with the Business Analyst, UI Designer and Database Modeler working together until some part of the system has been fleshed out sufficiently to hand it over to some other specialists in the Development phase, where, again, a number of iterations will take place. This model (promoted by DSDM) tries to combine the best of both practices: iterating over a part of the system, while still allowing for specialization in the pipeline (Jennifer Stapleton, 2003).

The DSDM method allows that work is taken back in the pipeline to any previous phase to re-iterate over it again. But then one runs the risk of annoying the people in the phases down the line. They will have to redo part of their work too. However, this may not be as bad as it sounds. In any dynamic environment you would do well to teach everyone involved that rework is part of a healthy iterative and incremental lifecycle, whatever its cause. After all, following an incremental release the environment has the final verdict on the delivered product and the customer decides what goes in and what goes out. Rework is inevitable. The need to work with separated internal and external environments means that, by definition, there is always a pipeline. It’s just a matter of trying to make it as short as possible.

**Conclusion**

I realize that some ideas in this article will be considered heretic by one half of the agile community, while other ideas will be hard to swallow by the other half. This might simply be the fate of anyone who attempts to bring conflicting streams of thought together. I also realize that, even after reading this long paper, it will still be difficult for a project manager or process manager to pick and choose the best project lifecycle, given all the alternatives for frequencies, multiple rhythms, planning levels and pipelines presented here. It’s a real jungle out there. And there are simply too many options to choose from to draw one single unified cyclic diagram for everybody.

Still, I hope you consider my contribution an informative and balanced view of iterative and incremental development. Rest assured that this view is subject to additional iterations and increments itself, so don’t be surprised to find an improved and adapted version next time we talk.
Note: I owe thanks to Mary Poppendieck, Alistair Cockburn, Brad Appleton, Jason Yip and Franco Martinig, who took time to review this article and give me some really valuable feedback.

**Sources and References**


Copyright

This work is released under a Create Commons License. You are free (and encouraged) to copy, distribute and transmit the work, and to adapt the work, provided you attribute the author. Please send it to your colleagues, friends, family, neighbors and local zoo.

http://creativecommons.org/licenses/by/3.0/
Acceptance TDD Explained

Lasse Koskela

This article is based on a chapter from the book
"Practical TDD and Acceptance TDD for Java Developers"
http://www.manning.com/koskela/

In the spacecraft business no design can survive the review process, without first answering the question—how are we going to test this thing?
—Glen B. Alleman, Director Strategic Consulting for Lewis & Fowler

In the previous chapters, we have explored the developer-oriented practice of test-driven development, covering the fundamental process itself as well as a host of techniques for employing TDD on a variety of Java technologies. In this chapter, we’ll take the core idea of TDD and apply it to the overall product development process.

TDD helps software developers produce working, high-quality code that’s maintainable and, most of all, reliable. Our customers are rarely, however, interested in buying code. Our customers want software that helps them to be more productive, make more money, maintain or improve operational capability, take over a market, and so forth. This is what we need to deliver with our software—functionality to support business function or market needs. Acceptance test-driven development (acceptance TDD) is what helps developers build high-quality software that fulfills the business’s needs as reliably as TDD helps ensure the software’s technical quality.

Acceptance TDD helps coordinate software projects in a way that helps us deliver exactly what the customer wants when they want it, and that doesn’t let us implement the required functionality only half way. In this chapter, we will learn what acceptance test-driven development is, why we should consider doing it, what these mysterious acceptance tests are, and how acceptance TDD integrates with TDD.

Acceptance test-driven development as a name sounds similar to test-driven development, doesn’t it? So, what’s the difference between the two? Surely it has something to do with the word acceptance prefixed onto TDD, but is there something else beneath the surface that differentiates the two techniques? What exactly are these acceptance tests? And what are we accepting with our tests in the first place? These are the questions we’re going to find answers to as we start our journey into the world of acceptance tests and acceptance test-driven development.

We’ll begin by describing a lightweight and extremely flexible requirements format called user stories. After all, we need to know how our system should behave, and that’s exactly what user stories tell us. From there, we’ll continue by exploring what acceptance tests are and what kinds of properties they should exhibit. By then, we will know what our requirements—the user stories—might look like, and we will know how our acceptance tests should look, which means we’ll be ready to figure out how we work with these artifacts or what the process of acceptance TDD looks like.

An essential property of acceptance TDD is that it’s a team activity and a team process, which is why we’ll also discuss team-related topics such as the roles involved and who might occupy these roles. We’ll also ask ourselves why we should do this and respond by identifying a number of benefits of acceptance TDD. Finally, we’ll talk about what exactly we are testing with our acceptance tests and what kind of tools we have at our disposal.
But now, let us introduce ourselves with a handy requirements format we call *user stories*.

### 1 Introduction to user stories

User stories are an extremely simple way to express requirements. In its classic form, a user story is a short sentence stating *who* does *what* and *why*. In practice, most stories just tell us who and what, with the underlying motivation considered apparent from the context. The reason a story is typically only one sentence long (or, in some cases, just one or two words that convey meaning to the customer and developers) is that the story is not intended to document the requirement. The story is intended to *represent* the requirement, acting as a *promise of a future conversation* between the customer and the developer.

#### 1.1 Format of a story

A number of people have suggested writing user stories that follow an agreed format such as “As a (role) I want (functionality) so that (benefit).” However, I and a number of other proponents of user stories for requirements management recommend not fixing the format as such but focusing on the user story staying on a level of detail that makes sense, using terms that make sense to the customer. This is not to say that such a template would be a bad idea—which it isn’t; it’s just that one size doesn’t fit all, and the people in your organization might feel differently about the format than I do [1].
On the physical format of user stories
In part because user stories are concise, many co-located teams keep their user stories written on small index cards or sticky notes, managing them on a whiteboard or other task board. This method is beneficial because communication is clear and progress is immediately obvious. Therefore, more and more multisite projects are also adopting such physical story cards for managing their work locally. After all, the benefits often far outweigh the hassle of relaying the status from the task board to electronic format for distribution. The use of index cards on an early XP project gave birth to the mnemonic of 3 Cs: card, conversation, confirmation.

1.2 Power of storytelling

Hannah Arendt, a German political scientist, has said, “storytelling reveals meaning without committing the error of defining it.” This particular quote eloquently communicates how user stories focus on meaning without stumbling on nitty-gritty details.

Inside every story is another one trying to come out
Just like there are multiple solutions to most computing problems, there is always another way to write a given user story. Indeed, it might make sense to take a closer look at a user story before rubberstamping it as a technical story. There is usually a way to express the story in a way that conveys the underlying value - the rationale - of the story. If you can’t figure out that value, try again.

User stories are in many ways a form of storytelling, which is an effective medium for transferring knowledge. For one, people like listening to stories. Storytellers are good at keeping our attention - a lot more so than, say, structured documents of equal volume - and it’s not just audible stories that have this advantage; prose with a storyline and context is far more interesting reading than a seemingly unconnected sequence of statements.

Let’s see how much this property shows through in practice by looking at a couple of examples of user stories.

1.3 Examples of user stories

To get a better idea of what user stories look like, here are some examples of the kinds of user stories I personally tend to write:

- “Support technician sees customer’s history onscreen at the start of a call”
- “The system prevents user from running multiple instances of the application simultaneously”
- “Application authenticates with the HTTP proxy server”

These user stories express just enough for the customer to be able to prioritize the feature in relation to other features and for the developer to be able to come up with a rough effort estimate for the story. Yet these stories don’t burden the developer by prescribing the implementation, and they don’t drown the team with excessive detail.

The first story about a technical-support application doesn’t tell us what the screen will look like; and the second story doesn’t talk about desktop shortcuts, scanning process listings, and so on. They convey what provides value to the customer—not how the system should provide that value. The third story is a bit different. It’s clearly a technical user story, not having much to do
with business functionality. It does, however, have enabling value, and it expresses that need in a clear manner. Furthermore, although harder to quantify, some technical stories might create value for the customer through lower total cost of ownership.

That’s about all we’re going to say about user stories for now. For a more in-depth description of user stories as a requirements management and planning tool, a great pick would be Mike Cohn’s book *User Stories Applied* (Addison-Wesley, 2004).

As we already mentioned, the format of a user story doesn’t matter all that much as long as it communicates the necessary information—who, what, why—to all involved parties, either explicitly or implicitly. In fact, just like the format of a story isn’t one-size-fits-all, using stories as a requirements-management or planning tool isn’t in any way a requirement for doing acceptance test-driven development—it’s a natural fit.

Now that we know what the mysterious stories are (or what they can be), let’s figure out what we mean by *acceptance tests*.

2. Acceptance tests

Acceptance tests are specifications for the desired behavior and functionality of a system. They tell us, for a given user story, how the system handles certain conditions and inputs and with what kinds of outcomes. There are a number of properties that an acceptance test should exhibit; but before taking a closer look, let’s see an example.
2.1 Example tests for a story

Let’s consider the following example of a user story and see what our acceptance tests for that particular story might look like. I present you figure with 1.

![Figure 1. Example of a user story, written on a story card](image)

The functionality that we’re interested in is for the system to obtain and display the customer’s history of records when a call comes through the customer support system. I might, for example, think of the tests for this story that are scribbled down as figure 2.

These three tests would essentially tell us whether the system behaves correctly from the perspective of a user—conditions of satisfaction. They tell us nothing about how the system implements that behavior.

Now, with these example tests in mind, let’s look at some essential properties of acceptance tests, starting with who owns them and who writes them.

![Figure 2. Example tests for the story, written on the back of the story card from figure 1](image)

2.2 Properties of acceptance tests

So far, you’ve probably deduced that acceptance tests are typically short and somewhat informal. There’s more to the nature of acceptance tests, however, and next we’re going to look at some general properties.
To make a long story short, acceptance tests are

- Owned by the customer
- Written together with the customer, developer, and tester
- About the *what* and not the *how*
- Expressed in the language of the problem domain
- Concise, precise, and unambiguous

Let’s expand these sound bites one by one and see what they mean.

**Owned by the customer**

Acceptance tests should be owned by the customer because their main purpose is to specify acceptance criteria for the user story, and it’s the customer—the business expert—who is best positioned to spell out those criteria. This also leads to the customer being the one who should ideally be writing the acceptance tests.

Having the customer write the acceptance tests helps us avoid a common problem with acceptance tests written by developers: Developers often fall into the pit of specifying technical aspects of the implementation rather than specifying the feature itself. And, after all, acceptance tests are largely a specification of functionality rather than tests for technical details (although sometimes they’re that, too).

**Written together**

Even though the customer should be the one who owns the acceptance tests, they don’t need to be the only one to write them. Especially when we’re new to user stories and acceptance tests, it is important to provide help and support so that nobody ends up isolating themselves from the process due to lack of understanding and, thus, being uncomfortable with the tools and techniques. By writing tests together, we can encourage the communication that inevitably happens when the customer and developer work together to specify the acceptance criteria for a story.

With the customer in their role as domain expert, the developer in the role of a technical expert, and the tester in a role that combines a bit of both, we’ve got everything covered. Of course, there are times when the customer will write stories and acceptance tests by themselves—perhaps because they were having a meeting offsite or because they didn’t have time for a discussion about the stories and their accompanying tests.

The same goes for a developer or tester who *occasionally* has to write acceptance tests without access to the customer. On these occasions, we’ll have to make sure that the necessary conversation happens at some point. It’s not the end of the world if a story goes into the backlog with a test that’s not ideal. We’ll notice the problem and deal with it eventually. That’s the beauty of a simple requirement format like user stories!

Another essential property for good acceptance tests ties in closely with the customer being the one who’s writing the tests: the focus and perspective from which the tests are written.
Focus on the what, not the how

One of the key characteristics that make user stories so fitting for delivering value early and often is that they focus on describing the source of value to the customer instead of the mechanics of how that value is delivered. User stories strive to convey the needs and wants—the what and why—and give the implementation—the how—little attention. In the vast majority of cases, the customer doesn’t care how the business value is derived. Well, they shouldn’t. Part of the reason many customers like to dictate the how is our lousy track record as an industry. It’s time to change that flawed perception by showing that we can deliver what the customer wants as long as we get constant feedback on how we’re doing.

Let’s look at an example to better illustrate this difference between what and why and how. Figure 3 shows an example of acceptance tests that go into too much detail about the solution rather than focusing on the problem - the customer’s need and requirement. All three tests shown on the card address the user interface - effectively suggesting an implementation, which isn’t what we want. While doing that, they’re also hiding the real information - what are we actually testing here?” - behind the technology.

![Figure 3. Acceptance test focusing too much on the implementation](image)

---

Concerned about the Quality of Your Web Application or Website? eValid™ Web Testing & Analysis Suite is a Complete Quality Solution:

- Functional Testing with AJAX Support
- Real Server Loading Driven by Multiple Browsers
- Automated Regression Testing
- Site Analysis of Website Tree With 3D-SiteMaps

Free product evaluation download and info available at [www.e-valid.com](http://www.e-valid.com)

---

Advertisement – Visual Use Case - Click on ad to reach advertiser web site
Instead, we should try to formulate our tests in terms of the problem and leave the solution up to the developers and the customer to decide and discuss at a later time when we’re implementing the tests and the story itself. Figure 4 illustrates a possible rewrite of the tests in figure 3 in a way that preserves the valuable information and omits the unnecessary details, which only clutter our intent.

![Figure 4. Trimmed-down version of the tests from figure 3](image)

Notice how, by reading these three lines, a developer is just as capable of figuring out what to test as they would be by reading the more solution-oriented tests from figure 3. Given these two alternatives, which would you consider easier to understand and parse? The volume and focus of the words we choose to write our tests with have a big effect on the effectiveness of our tests as a tool. We shouldn’t neglect that fact.

There’s more to words, though, than just volume and focus. We also have to watch for our language.

Use the language of the domain

An important property of acceptance tests is that they use the language of the domain and the customer instead of geek-speak only the programmer understands. This is the fundamental requirement for having the customer involved in the creation of acceptance tests and helps enormously with the job of validating that the tests are correct and sufficient. Scattering too much technical lingo into our tests makes us more vulnerable to having a requirement bug sneak into a production release—because the customer’s eyes glaze over when reading geek-speak and the developers are drawn to the technology rather than the real issue of specifying the right thing.

By using a domain language in specifying our tests, we are also not unnecessarily tied to the implementation, which is useful since we need to be able to refactor our system effectively. By using domain language, the changes we need to make to our existing tests when refactoring are typically non-existent or at most trivial.

Concise, precise, and unambiguous

Largely for the same reasons we write our acceptance tests using the domain’s own language, we want to keep our tests simple and concise. We write each of our acceptance tests to verify a single aspect or scenario relevant to the user story at hand. We keep our tests uncluttered, easy to understand, and easy to translate to executable tests. The less ambiguity involved, the better we are at avoiding mistakes and the working with our tests.
We might write our stories as simple reminders in the form of a bulleted list, or we might opt to spell them out as complete sentences describing the expected behavior. In either case, the goal is to provide just enough information for us to remember the important things we need to discuss and test for, rather than documenting those details beforehand. Card, conversation, confirmation—these are the three Cs that make up a user story. Those same three Cs could be applied to acceptance tests as well.

Remember the acceptance tests we saw earlier, for the story about a customer support system? Take another look at them, back in figure 2.

Would you say these tests are simple and concise? Perhaps you would. Perhaps you wouldn’t. Personally, I’d say there are some things in these tests that could be safely omitted while still preserving enough information to carry the original intent, and some things that shouldn’t be there. Figure 5 shows a revamp of the same tests shown in figure 2.

![Figure 5. Revamped acceptance tests from figure 2](image)

Notice the difference in conciseness? Notice how the developer would still know to test for the right things, provided they can ask the customer for the details, such as what kind of a message should be displayed for a non-existent account number or when the number is omitted altogether? The tests in figure 5 can be considered more accurate than the tests in figure 2 because they omit details that could change by the time we get around to implementing the story.

Obviously, some prefer to have more details written down for the acceptance tests than do others. Whether you’re into more text or less, or whether you prefer sketching little diagrams and UI designs as part of your user stories and the accompanying acceptance tests, it’s up to you and your team to decide. It’s all good as long as you remember to keep your acceptance tests simple and concise, and as long as you avoid writing down ambiguous things that can be interpreted wrongly at a later stage. Specifically, avoid writing down details that are easy to find out later and that don’t add crucial information for estimating the size of the story.

The last property of acceptance tests that we’ll list here has more to do with automating the tests than the way or form in which they’re written.

2.3 Implementing acceptance tests

Yet another common property of acceptance tests is that they might not be implemented (translation: automated) using the same programming language as the system they are testing. Whether this is the case depends on the technologies involved and on the overall architecture of
the system under test. For example, some programming languages are easier to interoperate with than others. Similarly, it is easy to write acceptance tests for a web application through the HTTP protocol with practically any language we want, but it’s often impossible to run acceptance tests for embedded software written in any language other than that of the system itself.

The main reason for choosing a different programming language for implementing acceptance tests than the one we’re using for our production code (and, often, unit tests) is that the needs of acceptance tests are often radically different from the properties of the programming language we use for implementing our system. To give you an example, a particular real-time system might be feasible to implement only with native C code, whereas it would be rather verbose, slow, and error-prone to express tests for the same real-time system in C compared to, for example, a scripting language.

The ideal syntax for expressing our acceptance tests could be a declarative, tabular structure such as a spreadsheet, or it could be something closer to a sequence of higher-level actions written in plain English. If we want to have our customer collaborate with developers on our acceptance tests, a full-blown programming language such as Java, C/C++, or C# is likely not an option. “Best tool for the job” means more than technically best, because the programmer’s job description also includes collaborating with the customer.

Now that we know something about acceptance tests and we have an idea of who’s writing the tests in the first place, let’s see how we use them to drive our development. What does acceptance test-driven development look like on paper?

### 3 Understanding the process

Test-driven development gives a programmer the tools for evolving their software in small steps, always certain of the software working as expected. This certainty comes from the programmer expressing their expectations in the form of automated unit tests. In acceptance test-driven development, this certainty is gained not on the level of technical correctness but rather on the feature level of, “does the software do what I want it to do?”

In other words, although in TDD we’re first defining the specific behavior we want our code base to exhibit and only then implementing the said behavior, in acceptance TDD we first define the specific user- or customer-valued functionality we want our system as a whole to exhibit and only then implement the said behavior, most likely using TDD as our vehicle of choice.

Because we know what acceptance tests look like, how about if we take a quick tour through the overall process of acceptance test-driven development and then broaden our view and look at what happens on the scale of a whole iteration? After that, we can go back and zoom in on the details of the more interesting bits.

---

### About the customer

You may have noticed that user stories as a requirements-management technique tend to stress having close and frequent interaction with the customer. If you’re worried about not having an on-site customer, or having a customer who’s not keen on having much to do with developing the software, you can stop worrying. There are ways around this obvious limitation, and we’ll talk about those later on. For now, just consider *the customer* as referring to a role rather than a specific person—a role that can be played by, say, one of the test engineers or developers who knows enough about the product’s domain to be able to make the kinds of decisions a real customer would make.
But now, I present you with the process of acceptance test-driven development, distilled into four small steps.

### 3.1 The acceptance TDD cycle

In its simplest form, the process of acceptance test-driven development can be expressed as the simple cycle illustrated by figure 6.

![Figure 6: The acceptance TDD cycle](image)

This cycle continues throughout the iteration as long as we have more stories to implement, starting over again from picking a user story; then writing tests for the chosen story, then turning those tests into automated, executable tests; and finally implementing the functionality to make our acceptance tests pass. In practice, of course, things aren’t always that simple. We might not yet have user stories, the stories might be ambiguous or even contradictory, the stories might not have been prioritized, the stories might have dependencies that affect their scheduling, and so on. We’ll get to these complications later. For now, let’s keep our rose-tinted glasses on and continue thinking about the simple steps outlined previously. Speaking of steps, let’s take a closer look at what those steps consist of.

**Step 1: Pick a user story**

The first step is to decide which story to work on next. Not always an easy job; but, fortunately, most of the time we’ll already have some relative priorities in place for all the stories in our iteration’s work backlog. Assuming that we have such priorities, the simplest way to go is to always pick the story that’s on top of the stack—that is, the story that’s considered the most important of those remaining. Again, sometimes, it’s not that simple.

Generally speaking, the stories are coming from the various planning meetings held throughout the project where the customer informally describes new features, providing examples to illustrate how the system should work in each situation. In those meetings, the developers and testers typically ask questions about the features, making them a medium for intense learning and discussion. Some of that information gets documented on a story card (whether virtual or physical), and some of it remains as tacit knowledge. In those same planning meetings, the customer prioritizes the stack of user stories by their business value (including business risk) and technical risk (as estimated by the team).
What kinds of issues might we have when picking stories from this stack of user stories? There are times when the highest-priority story requires skills that we don’t possess, or we consider not having enough of. In those situations, we might want to skip to the next task to see whether it makes more sense for us to work on it. Teams that have adopted pair programming don’t suffer from this problem as often. When working in pairs, even the most cross-functional team can usually accommodate by adjusting their current pairs in a way that frees the necessary skills for picking the highest priority task from the pile.

The least qualified person

The traditional way of dividing work on a team is for everyone to do what they do best. It’s intuitive. It’s safe. But it might not be the best way of completing the task. Arlo Belshee presented an experience report at the Agile 2005 conference, where he described how his company had started consciously tweaking the way they work and measuring what works and what doesn’t. Among their findings about stuff that worked was a practice of giving tasks to the least qualified person. For a full closure on their experience and an explanation of why this approach works, listen to Arlo’s interview at the Agile Toolkit Podcast website (http://agiletoolkit.libsyn.com/).

There can be more issues to deal with regarding picking user stories, but most of the time the solution comes easily through judicious application of common sense. For now, let’s move on to the second step in our process: writing tests for the story we’ve just picked.

**Step 2: Write tests for a story**

With a story card in hand (or onscreen if you’ve opted for managing your stories online), our next step is to write tests for the story. If you paid attention earlier in this chapter, we just learned that it’s the customer who should be writing the tests. So how does this play out?

The first thing to do is, of course, get together with the customer. In practice, this means having a team member sit down with the customer (they’re the one who should own the tests, remember?) and start sketching out a list of tests for the story in question.

As usual, there are personal preferences for how to go about doing this, but my current preference (yes, it changes from time to time) is to quickly scrawl out a list of rough scenarios or aspects of the story we want to test in order to say that the feature has been implemented correctly. There’s time to elaborate on those rough scenarios later on when we’re implementing the story or implementing the acceptance tests. At this time, however, we’re only talking about coming up with a bulleted list of things we need to test—things that have to work in order for us to claim the story is done.

We already saw a couple of examples of the kind of tests we’re referring to when we discussed the properties of acceptance tests. For example, you might want to peek back at figure 4, showing three tests on the back of a story card. That is the kind of rough list we’re after in this step.
On timing
Especially in projects that have been going on for a while already, the customer and the development team probably have some kind of an idea of what’s going to get scheduled into the next iteration in the upcoming planning meeting. In such projects, the customer and the team have probably spent some time during the previous iteration sketching acceptance tests for the features most likely to get picked in the next iteration’s planning session. This means that we might be writing acceptance tests for stories that we’re not going to implement until maybe a couple of weeks from now. We also might think of missing tests during implementation, for example, so this test-writing might happen pretty much at any point in time between writing the user story and the moment when the customer accepts the story as completed.

Once we have such a rough list, we start elaborating the tests, adding more detail and discussing about how this and that should work, whether there are any specifics about the user interface the customer would like to dictate, and so forth. Depending on the type of feature, the tests might be a set of interaction sequences or flows, or they might be a set of inputs and expected outputs. Often, especially with flow-style tests, the tests specify some kind of a starting state, a context the test assumes is part of the system. Other than the level of detail and the sequence in which we work to add that detail, there’s a question of when—or whether—to start writing the tests into an executable format. Witness step 3 in our process: automating the tests.

Step 3: Automate the tests

The next step once we’ve got acceptance tests written down on the back of a story card, on a whiteboard, in some electronic format, or on pink napkins, is to turn those tests into something we can execute automatically and get back a simple pass-or-fail result. Whereas we’ve called the previous step writing tests, we might call this step implementing or automating those tests.

In an attempt to avoid potential confusion about how the executable acceptance tests differ from the acceptance tests we wrote in the previous step, let’s pull up an example. Remember the acceptance tests in figure 5? We might turn those tests into an executable format by using a variety of approaches and tools. The most popular category of tools (which we’ll survey later) these days seems to be what we call table-based tools. Their premise is that the tabular format of tables, rows, and columns makes it easy for us to specify our tests in a way that’s both human and machine readable. Figure 7 presents an example of how we might draft an executable test for the first test in figure 5, “Valid account number”.

<table>
<thead>
<tr>
<th>Action</th>
<th>Parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>place call</td>
<td>555-1234, account 123456</td>
</tr>
<tr>
<td>accept call</td>
<td>555-1234</td>
</tr>
<tr>
<td>verify text</td>
<td>123456</td>
</tr>
<tr>
<td>verify text</td>
<td>Cory Customer</td>
</tr>
</tbody>
</table>

Figure 7. Example of an executable test, sketched on a piece of paper
In figure 7, we’ve outlined the steps we’re going to execute as part of our executable test in order to verify that the case of an incoming support call with a valid account number is handled as expected, displaying the customer’s information onscreen. Our test is already expressed in a format that’s easy to turn into a tabular table format using our tool of choice—for example, something that eats HTML tables and translates their content into a sequence of method invocations to Java code according to some documented rules.

Java code? Where did that come from? Weren’t we just talking about tabular formats? The inevitable fact is that most of the time, there is not such a tool available that would understand our domain language tests in our table format and be able to wire those tests into calls to the system under test. In practice, we’ll have to do that wiring ourselves anyway—most likely the developers or testers will do so using a programming language. To summarize this duality of turning acceptance tests into executable tests, we’re dealing with expressing the tests in a format that’s both human and machine readable and with writing the plumbing code to connect those tests to the system under test.

### On style

The example in figure 7 is a flow-style test, based on a sequence of actions and parameters for those actions. This is not the only style at our disposal, however. A declarative approach to expressing the desired functionality or business rule can often yield more compact and more expressive tests than what’s possible with flow-style tests. The volume of detail in our tests in the wild is obviously bigger than in this puny example. Yet our goal should—once again—be to keep our tests as simple and to the point as possible, ideally speaking in terms of what we’re doing instead of how we’re doing it.

With regard to writing things down (and this is probably not coming as a surprise), there are variations on how different teams do this. Some start writing the tests right away into electronic format using a word processor; some even go so far as to write them directly in an executable syntax. Some teams run their tests as early as during the initial authoring session. Some people, myself included, prefer to work on the tests alongside the customer using a physical medium, leaving the running of the executable tests for a later time. For example, I like to sketch the executable tests on a whiteboard or a piece of paper first, and pick up the computerized tools only when I’ve got something I’m relatively sure won’t need to be changed right away.

The benefit is that we’re less likely to fall prey to the technology—I’ve noticed that tools often steal too much focus from the topic, which we don’t want. Using software also has this strange effect of the artifacts being worked on somehow seeming more formal, more final, and thus needing more polishing up. All that costs time and money, keeping us from the important work.

In projects where the customer’s availability is the bottleneck, especially in the beginning of an iteration (and this is the case more often than not), it makes a lot of sense to have a team member do the possibly laborious or uninteresting translation step on their own rather than keep the customer from working on elaborating tests for other stories. The downside to having the team member formulate the executable syntax alone is that the customer might feel less ownership in the acceptance tests in general—after all, it’s not the exact same piece they were working on. Furthermore, depending on the chosen test-automation tool and its syntax, the customer might even have difficulty reading the acceptance tests once they’ve been shoved into the executable format dictated by the tool.

Just for laughs, let’s consider a case where our test-automation tool is a framework for which we express our tests in a simple but powerful scripting language such as Ruby. Figure 8 highlights...
the issue with the customer likely not being as capable of feeling ownership of the implemented acceptance test compared to the sketch, which they have participated in writing. Although the executable snippet of Ruby code certainly reads nicely to a programmer, it’s not so trivial for a non-technical person to relate to.

![Figure 8. Contrast between a sketch an actual, implemented executable acceptance test](image)

Another aspect to take into consideration is whether we should make all tests executable to start with or whether we should automate one test at a time as we progress with the implementation. Some teams—and this is largely dependent on the level of certainty regarding the requirements—do fine by automating all known tests for a given story up front before moving on to implementing the story.

Some teams prefer moving in baby steps like they do in regular test-driven development, implementing one test, implementing the respective slice of the story, implementing another test, and so forth. The downside to automating all tests up front is, of course, that we’re risking more unfinished work—inventory, if you will—than we would be if we’d implemented one slice at a time. My personal preference is strongly on the side of implementing acceptance tests one at a time rather than try getting them all done in one big burst. It should be mentioned, though, that elaborating acceptance tests toward their executable form during planning sessions could help a team understand the complexity of the story better and, thus, aid in making better estimates.

Many of the decisions regarding physical versus electronic medium, translating to executable syntax together or not, and so forth also depend to a large degree on the people. Some customers have no trouble working on the tests directly in the executable format (especially if the tool supports developing a domain-specific language). Some customers don’t have trouble identifying with tests that have been translated from their writing. As in so many aspects of software development, it depends.

Regardless of our choice of how many tests to automate at a time, after finishing this step of the cycle we have at least one acceptance test turned into an executable format; and before we proceed to implementing the functionality in question, we will have also written the necessary plumbing code for letting the test-automation tool know what those funny words mean in terms of technology. That is, we will have identified what the system should do when we say “select a transaction” or “place a call”—in terms of the programming API or other interface exposed by the system under test.

To put it another way, once we’ve gotten this far, we have an acceptance test that we can execute and that tells us that the specified functionality is missing. The next step is naturally to make that test pass—that is, implement the functionality to satisfy the failing test.
Step 4: Implement the functionality

Next on our plate is to come up with the functionality that makes our newly minted acceptance test(s) pass. Acceptance test-driven development doesn’t say how we should implement the functionality; but, needless to say, it is generally considered best practice among practitioners of acceptance TDD to do the implementation using test-driven development—the same techniques we’ve been discussing in the previous parts of this book.

In general, a given story represents a piece of customer-valued functionality that is split—by the developers—into a set of *tasks* required for creating that functionality. It is these tasks that the developer then proceeds to tackle using whatever tools necessary, including TDD. When a given task is completed, the developer moves on to the next task, and so forth, until the story is completed—which is indicated by the acceptance tests executing successfully.

In practice, this process means plenty of small iterations within iterations. Figure 9 visualizes this transition to and from test-driven development inside the acceptance TDD process.

As we can see, the fourth step of the acceptance test-driven development cycle, implementing the necessary functionality to fix a failing acceptance test, can be expanded into a sequence of smaller TDD cycles of test-code-refactor, building up the missing functionality in a piecemeal fashion until the acceptance test passes. The proportions in the figure should not be considered to reflect reality, however. Whereas the TDD cycle might range from one minute to a dozen, we might be chopping out code for a couple of hours or even the whole day before the acceptance test is passing.

While the developer is working on a story, frequently consulting with the customer on how this and that ought to work, there will undoubtedly be occasions when the developer comes up with a scenario—a test—that the system should probably handle in addition to the customer/developer writing those things down. Being rational creatures, we add those acceptance tests to our list, perhaps after asking the customer what they think of the test. After
all, they might not assign as much value to the given aspect or functionality of the story as we the developers might. At some point, we’ve iterated through all the tasks and all the tests we’ve identified for the story, and the acceptance tests are happily passing. At this point, depending on whether we opted for automating all tests up front (which I personally don’t recommend) or automating them just in time, we either go back to Step 3 to automate another test or to Step 1 to pick a brand-new story to work on.

It would probably not hurt to walk around a bit and maybe have a cup of coffee, possibly check out your email. Getting acceptance tests passing is intensive work. As soon as you’re back from the coffee machine, we’ll continue with a broader view of how this simple four-step cycle with its small steps fits into the bigger picture of a complete iteration within a project.

3.2 Acceptance TDD inside an iteration

A healthy iteration consists mostly of hard work. Spend too much time in meetings or planning ahead, and you’re soon behind the iteration schedule and need to de-scope (which might translate to another planning meeting…ugh!). Given a clear goal for the iteration, good user stories, and access to someone to answer our questions, most of the iteration should be spent in small cycles of a few hours to a couple of days writing acceptance tests, collaborating with the customer where necessary, making the tests executable, and implementing the missing functionality with our trusted workhorse, test-driven development.

As such, the four-step acceptance test-driven development cycle of picking a story, writing tests for the story, implementing the tests, and implementing the story is only a fraction of the larger continuum of a whole iteration made of multiple—even up to dozens—of user stories, depending on the size of your team and the size of your stories. In order to gain understanding of how the small four-step cycle for a single user story fits into the iteration, we’re going to touch the zoom dial and see what an iteration might look like on a time line with the acceptance TDD–related activities scattered over the duration of a single iteration. Figure 10 is an attempt to describe what such a time line might look like for a single iteration with nine user stories to implement. Each of the bars represents a single user story moving through the steps of writing acceptance tests, implementing acceptance tests, and implementing the story itself. In practice, there could (and probably would) be more iterations within each story, because we generally don’t write and implement all acceptance tests in one go but rather proceed through tests one by one.

Figure 10. Putting acceptance test-driven development on time line
Notice how the stories get completed almost from the beginning of the iteration? That’s the secret ingredient that acceptance TDD packs to provide indication of real progress. Our two imaginary developers (or pairs of developers and/or testers, if we’re pair programming) start working on the next-highest priority story as soon as they’re done with their current story. The developers don’t begin working on a new story before the current story is done. Thus, there are always two user stories getting worked on, and functionality gets completed throughout the iteration.

So, if the iteration doesn’t include writing the user stories, where are they coming from? As you may know if you’re familiar with agile methods, there is usually some kind of a planning meeting in the beginning of the iteration where the customer decides which stories get implemented in that iteration and which stories are left in the stack for the future. Because we’re scheduling the stories in that meeting, clearly we’ll have to have those stories written before the meeting, no?

That’s where continuous planning comes into the picture.

**Continuous planning**

Although an iteration should ideally be an autonomous, closed system that includes everything necessary to meet the iteration’s goal, it is often necessary—and useful—to prepare for the next iteration during the previous one by allocating some amount of time for pre-iteration planning activities. Otherwise, we’d have long-lasting planning meetings, and you’re probably not any more a friend of long-lasting meetings than I am.

Suggestions regarding the time we should allocate for this continuous planning range from 10–15% of the team’s total time available during the iteration. As usual, it’s good to start with something that has worked for others and, once we’ve got some experience doing things that way, begin zeroing in on a number that seems to work best in our particular context.

In practice, these pre-iteration planning activities might involve going through the backlog of user stories, identifying stories that are most likely to get scheduled for the next iteration, identifying stories that have been rendered obsolete, and so forth. This ongoing pre-iteration planning is also the context in which we carry out the writing of user stories and, to some extent, the writing of the first acceptance tests.

The rationale here is to be prepared for the next iteration’s beginning when the backlog of stories is put on the table. At that point, the better we know our backlog, the more smoothly the planning session goes, and the faster we get back to work, crunching out valuable functionality for our customer.

By writing, estimating, splitting if necessary, and prioritizing user stories before the planning meeting, we ensure quick and productive planning meetings and are able to get back to delivering valuable features sooner.
When do we write acceptance tests?

It would be nice if we had all acceptance tests implemented (and failing) before we start implementing the production code. That is often not a realistic scenario, however, because tests require effort as well—they don’t just appear from thin air—and investing our time in implementing the complete set of acceptance tests up front doesn’t make any more sense than big up-front design does in the larger scale. It is much more efficient to implement acceptance tests as we go, user story by user story.

Teams that have dedicated testing personnel can have the testing engineers work together with the customer to make acceptance tests executable while developers start implementing the functionality for the stories. I’d hazard a guess that most teams, however, are much more homogeneous in this regard and participate in writing and implementing acceptance tests together, with nobody designated as “the acceptance test guy.”

The process is largely dependent on the availability of the customer and the test and software engineers. If your customer is only onsite for a few days in the beginning of each iteration, you probably need to do some trade-offs in order to make the most out of those few days and defer work that can be deferred until after the customer is no longer available. Similarly, somebody has to write code, and it’s likely not the customer who’ll do that; software and test engineers need to be involved at some point.

We start from those stories we’ll be working on first, of course, and implement the user story in parallel with automating the acceptance tests that we’ll use to verify our work. And, if at all possible, we avoid having the same person implement the tests and the production code in order to minimize our risk of human nature playing its tricks on us.

Again, we want to keep an eye on putting too much up-front effort in automating our acceptance tests—we might end up with a huge bunch of tests but no working software. It’s much better to proceed in small steps, delivering one story at a time. No matter how valuable our acceptance tests are to us, their value to the customer is negligible without the associated functionality.

The mid-iteration sanity check

I like to have an informal sanity check in the middle of an iteration. At that point, we should have approximately half of the stories scheduled for the iteration running and passing. This might not be the case for the first iteration, due to having to build up more infrastructure than in later iterations; but, especially as we get better at estimating our stories, it should always be in the remote vicinity of having 50% of the stories passing their tests.

Of course, we’ll be tracking story completion throughout the iteration. Sometimes we realize early on that our estimated burn rate was clearly off, and we must adjust the backlog immediately and accordingly. By the middle of an iteration, however, we should generally be pretty close to having half the stories for the iteration completed. If not, the chances are that there’s more work to do than the team’s capacity can sustain, or the stories are too big compared to the iteration length.

Learning from our mistakes, we’ve come to realize that a story’s burn-down rate is constantly more accurate a source of prediction than an inherently optimistic software developer. If it looks like we’re not going to live up to our planned iteration content, we decrease our load.

Decreasing the load

When it looks like we’re running out of time, we decrease the load. We don’t work harder (or smarter). We’re way past that illusion. We don’t want to sacrifice quality, because producing
good quality guarantees the sustainability of our productivity, whereas bad quality only creates more rework and grinds our progress to a halt. We also don’t want to have our developers burn out from working overtime, especially when we know that working overtime doesn’t make any difference in the long run. Instead, we adjust the one thing we can: the iteration’s scope to reality. In general, there are three ways to do that: swap, drop, and split. Tom DeMarco and Timothy Lister have done a great favor to our industry with their best-selling books *Slack* (DeMarco; Broadway, 2001) and *Peopleware* (DeMarco, Lister; Dorset House, 1999), which explain how overtime reduces productivity.

Swapping stories is simple. We trade one story for another, smaller one, thereby decreasing our workload. Again, we must consult the customer in order to assure that we still have the best possible content for the current iteration, given our best knowledge of how much work we can complete.

Dropping user stories is almost as straightforward as swapping them. “This low-priority story right here, we won’t do in this iteration. We’ll put it back into the product backlog.” But dropping the lowest-priority story might not always be the best option, considering the overall value delivered by the iteration—that particular story might be of low priority in itself, but it might also be part of a bigger whole that our customer cares about. We don’t want to optimize locally. Instead, we want to make sure that what we deliver in the end of the iteration is a cohesive whole that makes sense and can stand on its own.

The third way to decrease our load, splitting, is a bit trickier compared to dropping and swapping—so tricky that we’d better give the topic its own little section.

**Splitting stories**

How do we split a story we already tried hard to keep as small as possible during the initial planning game? In general, we can split stories by function or by detail (or both). Consider a story such as “As a regular user of the online banking application, I want to optionally select the recipient information for a bank transfer from a list of most frequently and recently used accounts based on my history so that I don’t have to type in the details for the recipients every time.”

Splitting this story by function could mean dividing the story into “…from a list of recently used accounts” and “…from a list of most frequently used accounts.” Plus, depending on what the customer means by “most frequently and recently used,” we might end up adding another story along the lines of “…from a weighted list of most frequently and recently used accounts” where the weighted list uses an algorithm specified by the customer. Having these multiple smaller stories, we could then start by implementing a subset of the original, large story’s functionality and then add to it by implementing the other slices, building on what we have implemented for the earlier stories.

Splitting it by detail could result in separate stories for remembering only the account numbers, then also the recipient names, then the VAT numbers, and so forth. The usefulness of this approach is greatly dependent on the distribution of the overall effort between the details—if most of the work is in building the common infrastructure rather than in adding support for one more detail, then splitting by function might be a better option. On the other hand, if a significant part of the effort is in, for example, manually adding stuff to various places in the code base to support one new persistent field, splitting by detail might make sense.
Regardless of the chosen strategy, the most important thing to keep in mind is that, after the splitting, the resulting user stories should still represent something that makes sense—something valuable—to the customer.

References


TimeLine - Getting and Keeping Control over your Project

Niels Malotraux, niels @ malotaux.nl
http://www.malotaux.nl/nrm/English/

1 How do we get projects on time?

*Insanity is doing the same things over and over again and hoping the outcome to be different (let alone better)*

(Albert Einstein 1879-1955, Benjamin Franklin 1706-1790, it seems Franklin was first)

Many projects deliver late. If we don’t change our ways, projects will continue to be late. The only way to get projects on time is to change the way we do things. The Evolutionary Project Management (Evo) approach [1] is about continuously introducing small changes (hence evolutionary) in the way we do things, constantly improving the performance and the results of what we are doing. Because people can imagine the effect of the change, this evolutionary change can be biased towards improvement, rather than being random.

One of the techniques having emerged out of the Evo way of working is the TimeLine technique, which allows us to get and keep the timing of projects under control while still improving the project results, using just-enough estimation and then calibration to reality. TimeLine doesn’t stop at establishing that a project will be late. We actively deal with that knowledge: instead of accepting the apparent outcome of a TimeLine exercise, we have ample opportunities of doing something about it. One of the most rewarding ways of doing something about it is saving time.

An essential prerequisite of getting projects on time is, however, that we care about time.

2 The Goal or a project

Many projects treat requirements as sacred: they say: “This is what we have to do!” However, many requirements used in projects are not real Requirements, but rather wishes and assumed requirements. Let’s, as a driver for finding the real Requirements, define the following as an universal Goal for a project:

*Providing the customer with what he needs, at the time he needs it, to be satisfied, and to be more successful than he was without it …*

What the customer needs may be different from what he asks for and the time he needs it may be earlier or later than he asks for. If the customer is not satisfied, he may not want to pay for our efforts. If he is not successful, he cannot pay. If he is not more successful than he already was, why should he pay?

Of course we have to add that what we do in a project is:

*... constrained by what the customer can afford, and what we mutually beneficially and satisfactorily can deliver in a reasonable period of time.*

*What the customer wants, he cannot afford*

If we try to satisfy all customer’s wishes, we’ll probably fail from the beginning. We can do so many nice things, given unlimited time and money. But neither the customer nor we have unlimited time and money. Therefore: The Requirements are what the Stakeholders require, but for a project the Requirements are what the project is planning to satisfy.
If the Requirements aren’t clear (which they usually aren’t), any schedule will do

If the Requirements are unclear or incorrect, we will be spending time on the wrong things, wasting time and money. That’s in contradiction to the Goal of the project. And what use is a schedule that plans for doing the wrong things? Using the Goal as a top-level Requirement helps us to focus on what we have to do and what not. Understanding better what we should and shouldn’t do is one of the drivers for doing less, while delivering more. Continuous Requirements Engineering and Management is imperative for optimizing the duration of a project.

In almost all projects the requirements are not really clear. Even if they seem to be, they will change during the project, because we learn, they learn and the circumstances change. If the requirements aren’t really clear and will change anyway, why spend too much time on very detailed estimation of the project based on what we only currently think we have to do? If whatever time needed to do the work cannot be exact, because our understanding of the work is not exact, any ballpark figure will do. “But they want more exact estimations!” Well, if you estimated the work to be between 800 and 1000 days of work, but they insist in a more exact number, give them any exact looking figure, like 893 days, or 1093 if you like. If that keeps them quiet, you don’t have to waste more time on determining a figure that isn’t exact anyway. Can I do that? Yes, you can. It saves you time you need for more important things. And we should spend our time only on the most important things, shouldn’t we?
3 TimeLine

The standard procedure of defining the time needed to arrive at an expected end date of a project is (Figure 1) adding up the time we think we need for all the things we think we have to do and then adding some time for contingency.

Usually, however, the customer needs the result earlier. The date the customer needs a result from the project, we call it the FatalDate, is a Requirement, so it has to be treated as seriously as any other Requirement.

If we really take the FatalDate seriously, we can define a more refined procedure, called TimeLine:

1. Define a deadline or FatalDate. It is better to start with the end: planning beyond the available time/money budget is useless, so we can avoid wasting time if we find out that what we have to do takes more time than we have. Often we count back from the FatalDate to see when we should have started.
2. Write down whatever you currently think you have to accomplish
3. List in order of priority. Note that priority is based on value contribution and hence is influenced by many things!
4. Write the same down in elements of work (don’t detail more than necessary)
5. Ask the team to add forgotten elements and add duration estimates (days or weeks of calendar time, depending on the size of the project)
6. Get consensus on large variations of estimates, using a Delphi process (see section 0)
7. Add up the duration of all elements
8. Divide by the number of available people
9. This is a first estimate of the duration

This technique can be used on any scale: on a program, a project, on deliveries, on tasks. The technique is always same.

If the estimate of the duration is longer than the time available before the FatalDate (Figure 2), we will first have to resolve this problem, as it’s of no use continuing the work if we know we won’t succeed.
STAR WEST
SOFTWARE TESTING
ANALYSIS & REVIEW
The Greatest Software Testing Conference on Earth
Anaheim,
California

www.sqe.com/STARWEST
Register Early and save $200!

September 29–October 3, 2008
Disneyland® Hotel

99.7% of 2007 Attendees Recommend STARWEST to Others in the Industry

KEYNOTES
- Testing Lessons from Springfield—Home of the Simpsons
  Rob Sabourin, Am.IBug.com
- Telling Your Exploratory Story
  Jon Bach, Quisrel, Inc.
- Six Thinking Hats for Software Testers
  Julian Harty, Google
- Branch Out Using Classification Trees
  Julie Gardiner, Grove Consultants
- Has the Time for the Adversarial Organization Passed?
  Gerard Meszaros, ClearStream Consulting
- Testing Microsoft Office: Experiences You Can Leverage to Drive Quality Upstream
  Tara Roth, Microsoft

As to Disney photos, logos, properties: ©Disney
We can discuss the TimeLine with our customer and explain:

- What, at the FatalDate, surely will be done
- What surely will not be done
- What might be done (after all, estimation is not an exact science)

If what surely will be done is not sufficient for success, we better stop now to avoid wasting valuable time and money, rather spending it on more rewarding activities. Note that we put and keep putting what we plan in strict order of priority, so that at the FatalDate we have done the most important things that could be done in the time available. Customers usually don’t really mind about the bells and whistles. Time to Market is more important. Because priorities may change very dynamically, we have to constantly reconsider the order of what we do and when.

Initially, customers can follow this “will be done - won’t be done - might be done” reasoning, but still want “it all”. Remember that they don’t even exactly know what they really need, so “wanting it all” usually is a fallacy, although we’d better not say that. What we can say is: “OK, we have two options: In a conventional project, at the FatalDay, we would come to you and tell that we didn’t make it. In this project, however, we have another option. We already know, so we can tell you now that we will not be able to make it and then we can discuss what we are going to do about it. Which option shall we choose?”

If we explain it carefully, the customer will, eventually, choose the latter option. He will grumble a bit the first few weeks. Soon, however, he will forget the whole issue, because what we deliver is what we promised. We assume that we are using the Evo TaskCycle [14] to organize the work, by which we quickly learn what we can promise and how to live up to our promise. This enforces trust. Note that many customers ask for more, because they expect to get less. The customer also will become more confident: He is getting Deliveries (way before he ever expected it. And he will recognize soon that what he asked was not what he needed, so he’s not asking about “all” any more. We also assume that we use Evo DeliveryCycles [14] to check the requirements and assumptions, delivering real results to Stakeholders for feedback.

At the very first encounter with a new customer we cannot use this method, telling the customer that he will not get “it all”. Our competitor will promise to deliver it all (which he won’t, assuming that we are not less capable than our competitor), so we lose if we don’t tell the same, just as we did before using the Evo approach. There’s no risk, because we apparently survived promising “all” in our previous projects. If, after we won the contract, we start working the Evo way, we will soon get the confidence of our customer, and on the next project he will understand and only want to work with us.

If the estimated duration is more than the available time, we first have to resolve this problem, before going into any more detail. If it fits exactly the available time, we’d better assume that we still won’t make it, as we probably will have forgotten some elements of work. If the
estimated duration fits easily the available time, then there is a good chance that we may succeed (Figure 3).

![Figure 3: Only if the estimated time is well under the available time, we may succeed](image)

**Setting a Horizon**

If the total project takes more than 10 weeks, we define a Horizon at about 10 weeks on the Timeline, because with the limited size of our skull we cannot really oversee longer periods of time. (Thinking of Dijkstra’s lecture “The Humble Programmer” [2]: “The competent programmer is fully aware of the strictly limited size of his own skull; therefore he approaches the programming task in full humility, and among other things he avoids clever tricks like the plague.”) We may set a Horizon once we have done three things to make sure that we’ll not be surprised when we start looking over the Horizon again.

The Timeline procedure continues:

10. Choose a Horizon (default: 10 weeks from now)
11. Determine when to look over the Horizon again as shown in Figure 4: a (default: halfway)
12. Determine the amount of work proportionally to the total work (Figure 4: b)
13. Pull tasks from beyond the Horizon that need earlier preparation not to get surprised later (Figure 4: c)

![Figure 4: Three things we have to do before we can set a Horizon](image)

Now we can, for the time being, “forget” about what’s beyond the Horizon and concentrate on a more limited period of time. A period of 10 weeks proves to be a good compromise between what we can oversee, while still being long enough to allow for optimizing the order in which we deliver results.
We don’t use a sliding window of 10 weeks, but rather define a 10 weeks period, and try to accomplish our plans within this TimeBox. When we decide the time is right, we move to the next 10 week window.

### 3.2 DeliveryCycles

Within these 10 weeks, we plan Evo DeliveryCycles [14] (Figure 5), each not more than 2 weeks in length: *What* are we going to deliver to *Whom* and *Why*. Deliveries are for getting feedback from appropriate Stakeholders. We are humble enough to admit that our (and their) perception of the requirements is probably not perfect and that many of our assumptions are probably incorrect. That’s why we need communication and feedback and that’s why we make many short DeliveryCycles: to find out about the real Requirements, which assumptions are correct, and to waste as little time as possible on incorrect requirements and assumptions, saving precious time. In order to get feedback, we have to deliver to *eagerly waiting* Stakeholders. If the appropriate Stakeholders aren’t eagerly waiting, either they’re not interested and we may better work for other Stakeholders, or they have to be made eagerly waiting by delivering what we call *juicy bits*.

![TimeLine summary: setting a FatalDate, a Horizon, Deliveries, TaskCycles, and then calibrating back](image)

The TimeLine procedure continues:

14. Put the work for 10 weeks in optimum order, defining Deliveries of 2 weeks
15. Work should be estimated in more detail now
16. Make a rather detailed description of the first one or two Deliveries
17. Check the feasibility of completing Deliveries in two weeks each, with the available resources

We don’t only design the product, we are also constantly *redesigning the project*. Defining Deliveries is about designing the project: in which order should we do things to find out what is really necessary for a successful result and what is *superfluous*. How can we make sure that at any time, looking back, we can say: “We weren’t perfect, but we couldn’t have done it better”.

**TaskCycles**

Once we have divided the work over Deliveries, which also behave like Horizons, we first concentrate on the first few Deliveries and define the actual work that has to be done to produce these Deliveries. This work is organized in TaskCycles of one week, every TaskCycle defining a weekly Horizon. In the TaskCycle we define Tasks, estimated in effort-hours (see for a more detailed explanation: [14]). We plan the work in plannable effort time which defaults to 2/3 of the available time (26 hrs in case of a 40 hr week). We put this work in optimum order, divide it over the people in the project, have these people estimate the time they would need to do the work, see that they don’t get overloaded and that they synchronize their work to optimize the duration.

If we take the example of Figure 6, we can see that if Carl doesn’t start Task-f about $6+9+11+9 = 35$ hr before the end of the Delivery-Cycle, he’s putting the success of the Delivery on the line. If this Delivery was planned for the coming week, we also see that John should start right at the beginning of the Cycle, otherwise Carl can’t start in time. It’s easy to imagine that if the work of this Delivery wasn’t *designed* this way, the Delivery probably wouldn’t be on time. *Designing* the order of work for the Delivery *saves* time.

![Figure 6: Planning serial and parallel Tasks to fit the available time](image)

Note: “6/9h”: 6 is effort hours (planned), 9 is duration (allowing for unplanned activities)

The TimeLine procedure goes on:

18. Determine Tasks for the first week
19. Estimate the Tasks, now in real effort (net) hours needed to 100% complete each Task
20. Select Tasks to fit the plannable time (default: 2/3 of available time) of the people in the team
21. Select only the most important Tasks, never ever plan nor do less important Tasks
22. Now we have the Tasks for the first week defined
23. Make sure this is the most important set of Tasks
24. Put the Tasks in optimum order, to see how to synchronize individual people’s work during the week, e.g. as in the example of Figure 6.

3.4 Calibration

Having estimated the work that has to be done for the first week, we have captured the first metrics to start calibrating the TimeLine. If the Tasks for the first week would deliver only about half of what we need to do in that week, we now can, based on this limited material, extrapolate that our project is going to take twice as long, if we don’t do something about it. Of course, at the start this seems weak evidence, but it’s already an indication that our estimations may be too optimistic. Putting our head in the sand for this evidence is dangerous. One week later, when we have the real results of the first week, we have even better numbers to extrapolate and scale how long our project may take. Week after week we will gather more information with which we can calibrate and adjust our notion of what will be done at any FatalDate. This way, the TimeLine process provides us with very early warnings about the risks of being late. The earlier we get these warnings, the more time we have to do something about it.

![Figure 7: Earned Value (up to week 4) and Value Still to Earn (from week 5)](image)

The TimeLine procedure now concludes with two more steps:
25. Calibrate the TimeLine estimations and take the consequence
26. Repeat every one or two weeks.

Let’s take an example of taking the consequence of the TimeLine:

At the start, we estimate that the work we think we have to do in the coming 10 weeks is about 50 person Weeks of Work (WoW; Figure 7, line a). With a team of 5 people this seems doable. After 4 weeks, we find that 15 WoW have been completed (line b), instead of the expected 20 WoW. We now already know that the project will probably be late!
If we keep working this way, we may expect that at the end of the 10 weeks, we’ll have completed $10/4 \times 15 = 37.5$ WoW (line c). This is 25% less than the original 50 WoW expected. Alternatively, the original 50 WoW may be done in 13.3 weeks, or 33% more time than originally expected (line d). In case the deadline is really hard, the typical reaction of management is to throw more people at the project. How many people? Let’s calculate:

The velocity (actual accomplished against planned effort; see also Cohn [3]) is $15/20 = 0.75$ WoW per week. With a velocity of 0.75, we will need for the remaining 35 WoW $35/0.75 = 46.7$ person weeks in the remaining 6 weeks. So we need 7.8 people instead of the original 5. Management decides to add 3 people (expecting line e).

But there is another issue: based on our progressing understanding of the work we found that we forgot to plan some work that “has” to be done (requirements creep?) to complete the result we planned for the 10 weeks period: now we think we have, in the remaining 6 weeks, 40 WoW to do instead of the 35 WoW originally estimated (line f). This would mean $40/0.75 = 53.3$ person weeks in the remaining 6 weeks, which makes management believe that they actually need $53.3/6 = 8.9$ people. So they decide to add 4 people to the project, because they don’t want the project to take almost 50% longer and they think they are prepared to absorb the extra development cost, in order to win Time-to-Market. Beware, however, that this is a solution to be used with utmost care, because it may work out counterproductive, as explained in section 4.1. Much overlooked, but most rewarding and usually quite possible, is doing things more cleverly (line g), as explained in section 4.6.

4 If things don’t fit

If what we think we have to do doesn’t fit the available time, or if we want to fit what we think we have to do into a shorter timeframe, there are several options we see being used in practice:

- To be used with utmost care: Adding people
- Deceptive options:
- Hoping for the best
- Going for it
- Working Overtime
- Adding time: Moving the deadline
- Most interesting to exploit, but mostly overlooked: Saving time
- Not doing things that later prove to be superfluous
- Doing things differently
- Doing things at the right time, in the right order
- TimeBoxing

4.1 Adding people …

A typical move is to add people to a project, in order to get things done in less time. Intuitively, we feel that we can trade time with people and finish a 12 person-month project in 6 months with 2 people or in 3 months with 4 people, as shown in Figure. In his book The Mythical Man-Month, Brooks [4] shows that this is a fallacy, defining Brooks’ Law: *Adding people to a late project makes it later.*
Putnam [5] confirms this with measurements on some 500 projects. He found that if the project is done by 2 or 3 people, the project-cost is minimized, while 5 to 7 people achieve the shortest project duration at premium cost. Adding even more people makes the project take longer at excessive cost. Apparently, the project duration cannot arbitrarily be shortened, because there is a critical path of things that cannot be parallelized. We call the time in which nobody can finish the project the *nine mothers area*, which is the area where nine mothers produce a baby in one month. When I first heard about Brooks’ law, I assumed that he meant that you shouldn’t add people at the *end* of a project, when time is running out. After all, many projects seem to find out that they are late only by the end of the project. The effect is, however, much worse: if in the *first several weeks* of a project we find that the development speed is slower than predicted, and thus have to assume that the project will be late, even then adding people can make the project later.

The reason is a combination of effects:

- Apparently, the time needed to complete a development project is depending on more parameters than just the number of people
- It takes time for the added people to get acquainted with the project
- It takes time of the people already in the project to help the new people getting acquainted
- The new people are likely to introduce relatively more bugs during their introduction period, causing the need for extra find-and-fix time
- Having more people on the team increases the capacity linearly, but the lines of communication between these people increase much quicker, every n\(^{th}\) person adding (n-1) extra lines of communication

![Figure 8: The Mythical Man-Month](image-url)
• The architect who has to prepare and oversee what everybody has to do may become the bottleneck when there are more people to manage
• The productivity of different people can vary vastly, so people cannot simply be exchanged for time

So, adding people is not automatically a solution that works, it can even be very risky.

How can those mega-projects, where 100’s of people work together, be successful? Well, in many cases they aren’t. They deliver less and later than the customer expects and many projects simply fail, as found in numerous research, like the Standish reports [6].

The only way to try to circumvent Brooks’ Law is to work with many small teams, who can work in parallel, and who only synchronize their results from time to time, for example in bi-weekly DeliveryCycles. And yes, this adds complexity to the design of the project, for which the architect may become a bottleneck.

4.2 Hoping for the best

Most projects take more time than expected. Your past projects took longer than expected. What makes you think that this time it will be different? If you don’t change something in the way you run the project, the outcome won’t be different, let alone a better. Just hoping that your project will be on time this time won’t help. We call this ostriching: putting your head into the sand waiting until Murphy strikes again.

4.3 Going for it

We know that the available time is insufficient, but it has to be done: “Let’s go for it!” If nothing goes wrong (as if that is ever the case) and if we work a bit harder (as if we don’t already work hard) … Well, forget it.

4.4 Working overtime

Working overtime is fooling yourself: 40 hours of work per week is already quite exhausting. If you put in more hours, you’ll get more tired, make more mistakes, having to spend extra time to find and “fix” these mistakes (half of which you won’t), and you think you are working hard, but you aren’t working smart. It won’t work. This is also ostriching. As a rule, never work overtime, so that you have the energy to do it once or twice a year, when it’s really necessary.

4.5 Adding time: moving the deadline

Moving the deadline further away is also not a good idea. The further the deadline, the more danger of relaxing the pace of the project. We may call this Parkinson’s Law [7] (“Work expands so as to fill the time available for its completion”). Observation by Parkinson: “Granted that work (and especially paperwork) is elastic in its demands on time, it is manifest that there need be little or no relationship between the work to be done and the size of the staff to which it may be assigned.”) or the Student Syndrome (Starting as late as possible, only when the pressure of the FatalDate is really felt. Term attributed to E. Goldratt [8]). At the new deadline we probably hardly have done more, pushing the project result even further. Not a good idea, unless we really are in the nine mother’s area, where nobody, even with all the optimization techniques available, could do it. Even then, just because of the Student Syndrome, it’s better to optimize what we can do in the available time before the deadline. The earlier the deadline, the longer our
future is afterwards, in which we can decide what the next best thing there is to do.

We better optimize the time spent right from the beginning, because we’ll probably need that time anyway at the end. Optimizing only at the end won’t bring back the time we lost at the beginning.

4.6 Saving time

Instead of letting things randomly be undone at the FatalDay, it’s better to choose what we won’t have done, preferably those things that weren’t needed anyway. We know that we won’t have enough time, so let’s save time wherever possible!

There are several ways to save time, even without negatively affecting the Result of the project:

- Efficiency in what to do: doing only what is needed, not doing things that later prove to be superfluous. This includes efficiency in knowing why and for whom to do it. Because people tend to do more than necessary, especially if the goals are not clear, there is ample opportunity for not doing what is not needed.

- Efficiency in how to do it: doing things differently. We can probably do the same in less time if we don’t immediately do it the way we always did, but first think of an alternative and more efficient way.

- Efficiency in when to do it: doing things at the right time, in the right order. A lot of time is wasted by synchronization problems, like people waiting for each other, or redoing things because they were done in the wrong order. Actively Synchronizing [15] and designing the order of what we do (e.g. as in Figure 6), saves a lot of time.

In my experience, these are all huge time savers. And of course we can also apply these time savers if what we think we have to do easily fits in the available time. We don’t have to wait until we’re in trouble …

TimeBoxing provides incentives to constantly apply these ways to save time, in order to stay within the TimeBox. TimeBoxing is much more efficient than FeatureBoxing (waiting until we’re ready), because with FeatureBoxing we lack a deadline, causing Parkinson’s Law and the Student Syndrome to kick in badly. Note that this concept of saving time is similar to “eliminating waste” in Lean thinking, and already indicated by Henry Ford in his book “My Life and Work”, back in 1922.

5 Preflection, foresight and prevention

Because “hindsight is easy”, we can often use it to reflect on what we did, in order to learn: Could we have avoided doing something that now, in hindsight, proves to be superfluous? Could we’ve done it more efficiently? Reflection, however, doesn’t recover lost time: the time is already spent and can never be regained. Only with preflection we can try to foresee and thus prevent wasting precious time.

6 Estimation

There are several methods for estimation. There are also ways to quickly change from optimistic to realistic estimation. An important precondition is that we start treating time seriously, creating a Sense of Urgency. It is also important to learn how to spend just enough time on estimation. Not more and not less.
6.1 Changing from optimistic to realistic estimation

In the Evo TaskCycle [14] we estimate the effort time for a Task in hours. The estimations are TimeBoxes, within which the Task has to be completely done, because there is not more time. Tasks of more than 6 hours are cut into smaller pieces and we completely fill all plannable time (i.e. 26 hours, 2/3 of the 40hr available time in a work week). The aim in the TaskCycle is to learn what we can promise to do and then to live up to our promise. If we do that well, we can better predict the future. Experience by the author shows that people can change from optimistic to realistic estimators in only a few weeks, once we get serious about time. At the end of every weekly cycle, all planned Tasks are done, 100% done. The person who is going to do the Task is the only person who is entitled to estimate the effort needed for the Task and to define what 100% done means. Only then, if at the end of the week a Task is not 100% done, that person can feel the pain of failure and quickly learn from it to estimate more realistically the next week. If we are not serious about time, we’ll never learn, and the whole planning of the project is just quicksand!

6.2 0th order estimations

0th order estimations, using ballpark figures we can roughly estimate, are often quite sufficient for making decisions. Don’t spend more time on estimation than necessary for the decision. It may be a waste of time. We don’t have time to waste.

Example: How can we estimate the cost of one month delay of the introduction of our new product?

How about this reasoning: The sales of our current most important product, with a turnover of about $20M per year, is declining 60% per year, because the competition introduced a much better product. Every month delay, it costs about 5% of $20M, being $1M. Knowing that we are losing about $1M a month, give or take $0.5M, could well be enough to decide that we shouldn’t add more bells and whistles to the new product, but rather finalize the release. Did we need a lot of research to collect the numbers for this decision …?

Any number is better than no number. If a number seems to be wrong, people will react and come up with reasoning to improve the number. And by using two different approaches to arrive at a number we can improve the credibility of the number.

6.3 Simple Delphi

If we’ve done some work of small complexity and some work of more complexity, and measured the time we needed to complete those, we are more capable than we think of estimating similar work, even of different complexity. A precondition is that we become aware of the time it takes us to accomplish things. There are many descriptions of the Delphi estimation process [10], but also here we must be careful not to make things more complicated than absolutely necessary. Anything we do that’s not absolutely necessary takes time we could save for doing more important things!

Our simple Delphi process goes like this:
1. Make a list of things we think we have to do in just enough detail. Default: 15 to 20 chunks.
2. Distribute this list among people who will do the work, or who are knowledgeable about the work.
3. Ask them to add work that we apparently forgot to list, and to estimate how much time the elements of work on the list would cost, “as far as you can judge”.

4. In a meeting the estimates are compared.

5. If there are elements of work where the estimates differ significantly between estimators, do not take the average, and do not discuss the estimates, but discuss the contents of the work, because apparently different people have a different idea about what the work includes. Some may forget to include things that have to be done, some others may think that more has to be done than needed.

6. After the discussion, people estimate individually again and then the estimates are compared again.

7. Repeat this process until sufficient consensus is reached (usually repeating not more than once or twice).

8. Add up all the estimates to end up with an estimate for the whole project.

Don’t be afraid that the estimates aren’t exact, they’ll never be. By adding many estimations, however, the variances tend to average and the end result is usually not far off. Estimates don’t have to be exact, as long as the average is OK. Using Parkinson’s Law in reverse, we now can fit the work to fill the time available for its completion.

6.4 Estimation tools

There are several estimation methods and tools on the market, like e.g. COCOMO [11], QSM-SLIM [12] and Galorath-SEER [13]. The tools rely on historical data of lots of projects as a reference. The methods and tools provide estimates for the optimum duration and the optimum number of people for the project, but have to be tuned to the local environment. With the tuning, however, a wide range of results can be generated, so how would we know whether our tuning provides better estimates than our trained gut-feel?

The use of tools poses some risks:

- For tuning we need local reference projects. If we don’t have enough similar (similar people, techniques, environments, etc …) reference projects, we won’t be able to tune. So the tools may work better in large organizations with a lot of similar projects.

- We may have to start working for the tool, instead of having the tool work for us. Tools don’t pay salaries, so don’t work for them. Only use the tool if it provides good Return On Investment (ROI).

- A tool may obscure the data we put in, as well as obscure what it does with the data, making it difficult to interpret what the output of the tool really means, and what we can do to improve. We may lose the connection with our gut-feel, which eventually will make the decision.

- Use a tool only when the simple Delphi and 0th order approaches, combined with realistic estimation rather than optimistic estimation, really prove to be insufficient and if you have sufficient reasons to believe that the tool will provide good ROI.

Conclusion

TimeLine doesn’t solve our problems. TimeLine is a set of techniques to expose the real status of our project early and repeatedly. Instead of accepting the apparent outcome of a TimeLine exercise, we have ample opportunities of doing something about it.
We can save a lot of time by not doing the things that later would prove to be superfluous. Because people do a lot of unnecessary things in projects, it’s important to identify those things before having started, otherwise the time is already spent, and never can be recovered. By revisiting the TimeLine every one or two weeks, we stay on top of how the project is developing and we can easily report to management the real status of the project.

Doesn’t all this TimeLining take a lot of time? The first one or two times it does, because we are not yet acquainted with the various elements of the project and we have to learn how to use TimeLine. After a few times, however, we dash it off and we’re getting into a position that we really can start optimizing the results of the project, producing more than ever before. TimeLine allows us to take our head out of the sand, stay in control of the project and deliver Results successfully, on time.

Still, many Project Managers hesitate to start using the TimeLine technique for the first time. After having done it once, however, the usual reaction is: “I got much better oversight over the project and the work than I ever expected”, and the hesitation is over.

The TimeLine technique is not mere theory. It is highly pragmatic, and successfully used in many projects coached by the author. The most commonly encountered bottleneck when introducing the TimeLine technique in a project is that no one in the project has an oversight of what exactly the project is really supposed to accomplish. This could be a reason why Project Managers hesitate to start using the technique. Redefining what the project is to accomplish and henceforth focusing on this goal is the first immediate timesaver of the technique, with many savings to follow.

Reference


[10] www.iit.edu/~it/delphi.html


[12] www.qsm.com


White Paper: Keeping Pace with Changing Business Needs – Requirements Change Management with MKS Integrity. As the pace of change continues to increase, we are faced with the challenge of how to keep up with evolving and expanding business needs. How do we keep up with change when no sooner is a project underway, than the requirements change? This paper speaks specifically to these challenges and demonstrates how MKS Integrity’s workflow enabled support for requirements change management can help your organization better assess, react and respond to rapid and constant change. Download your copy:

http://www.mks.com/mtjn_reqs_change_mgmt

SoftDevLinks.com is a new general directory for software developers, testers and managers. If you have a blog, a web site, distribute a tool or work a consulting company related to software development, do not hesitate to add a (free) link in this directory.

http://www.softdevlinks.com

The Software Development Articles Directory. You are looking for articles on testing Java code? You want more information on creating use cases? You want to share an article you have written on improving database performance? Search and contribute software development articles on

http://www.softdevarticles.com

Advertising for a new Web development tool? Looking to recruit software developers? Promoting a conference or a book? Organizing software development training? This classified section is waiting for you at the price of US $ 30 each line. Reach more than 50'000 web-savvy software developers and project managers worldwide with a classified advertisement in Methods & Tools. Without counting the 1000s that download the issue each month without being registered and the 40'000 visitors/month of our web sites! To advertise in this section or to place a page ad simply http://www.methodsandtools.com/advertise.php