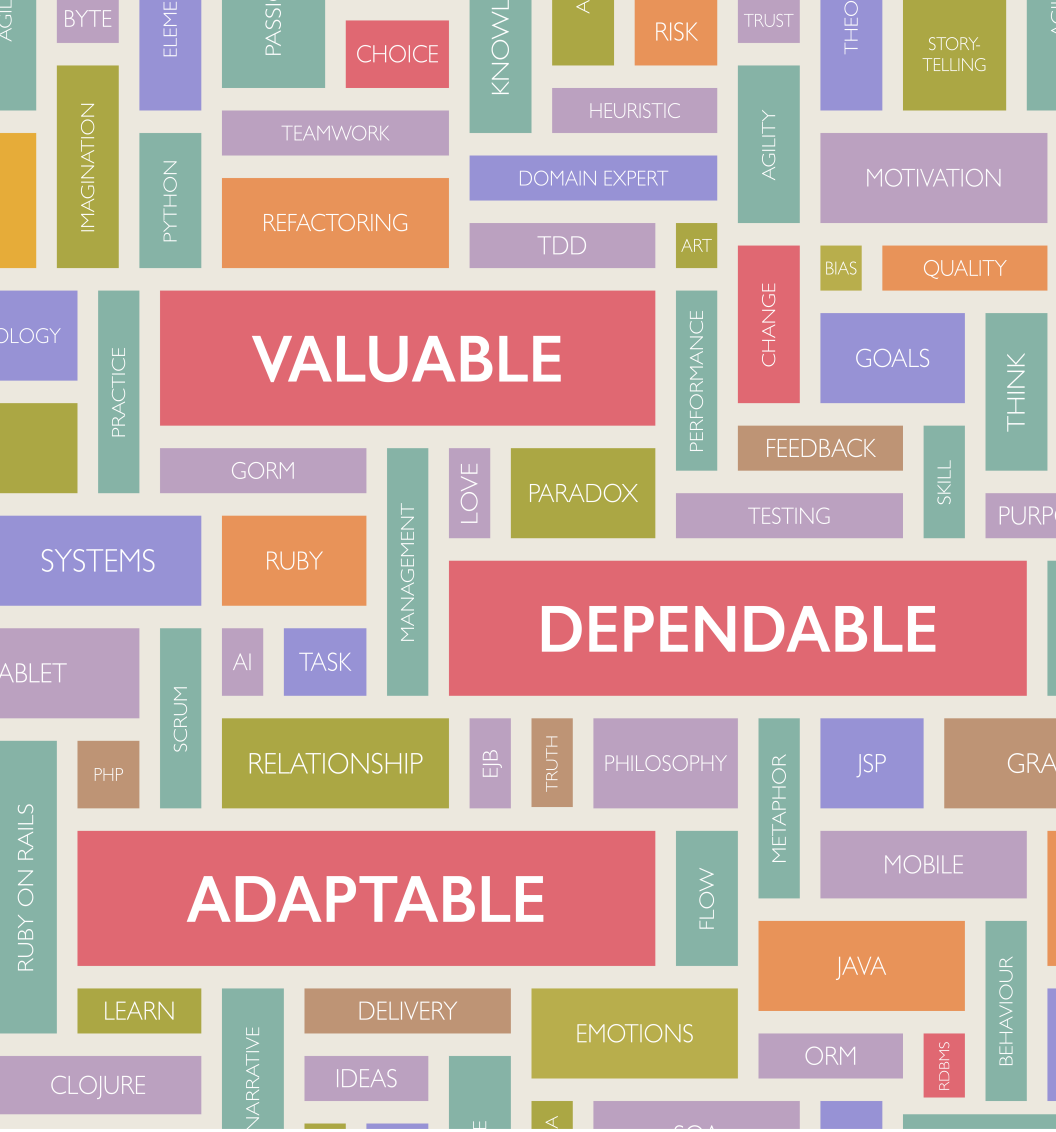

PRINCIPLES OF EFFECTIVE SOFTWARE DELIVERY

PAUL BOWLER

Valuable, Dependable, Adaptable

Principles of Effective Software Delivery

Paul Bowler

This book is for sale at
http://leanpub.com/effective-software-delivery

This version was published on 2014-02-11

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Contents

Introduction

> "Software is eating the world" ~ **Marc Andreessen**

Software touches so many aspects of all our lives. Every time we travel, buy goods or communicate remotely with each other we invoke an invisible cascase of software services.

These services, each created by teams of developers, analysts, testers and other professionals, form the backbone of our modern way of life, and without them we'd be plunged back into some simpler time - a technology ice-age - where the instantaneous automated transactions we're used to would slow time down to a crawl.

Given the importance of software in keeping going the fast-paced lifestyles we have become accustomed to, you'd think that we'd have mastered the process by which software is conceived, created and delivered. However, this is far from the truth: software has remained a complex, unruly beast which has proved very hard to tame no matter what management theories have been used to try to control it.

> "Thinking is very hard work. And management [fads and] fashions are a wonderful sub-

stitute for thinking." ~ **Peter Drucker**

As the utility of the software products and services increases, so does their size and subsequent internal complexity. With complexity comes risk, inertia and resistance to change and, in over twenty years of experience in the software industry, I've not come across one organisation that does not have a software problem - be it one of scale, quality or creativity.

So why is delivering valuable, dependable and adaptable software so difficult? What patterns can we see in the past seven decades of software development that could help us improve? What have we forgotten and need to relearn from the past? What lessons can the software industry take from other industries that are successfully managing the complexity problem? Beyond process, practice and tools, what are the founding principles for successful software delivery? This book aims to answer these questions.

Principle 1: Metaphor matters

So what is software? How should we think about creating it? What mental models lighten the cognitive load to help us talk about software, share ideas and bring people together to get the job done?

> Six blind men were asked to determine what an elephant looked like by feeling different parts of the elephant's body.
>
> The blind man who feels a leg says the elephant is like a pillar; the one who feels the tail says the elephant is like a rope; the one who feels the trunk says the elephant is like a tree branch; the one who feels the ear says the elephant is like a hand fan; the one who feels the belly says the elephant is like a wall; and the one who feels the tusk says the elephant is like a solid pipe.
>
> A king explains to them: All of you are partly right, but all of you are wrong. The reason every one of you is telling it differently is because each one of you touched the different part of the elephant.

> So, actually the elephant has all the features you mentioned.

Metaphors are useful platforms for transferring a large body of existing knowledge from one context to another and lay the foundation for how we think about the world. They capture a concept in a few words and help people grasp and retain a complex idea by relating it to something they already understand. However, we are often drawn into metaphors that create beliefs, modify behaviours and direct resources in directions we might not choose if we were paying closer attention to them.

There are many metaphors used for describing the job of software development and delivery. In the next section I'll discuss a few of the more prominent metaphors and show how these help us construct our mental models of the act of creating software, how they shape the work itself, how they demand we measure performance and success and the typical outcomes you could expect to get by using each one.

Software as Science

I first came face-to-monitor with software at school in the mid 1980's when home computing was just becoming affordable and practical, and government initiatives were helping get personal computers into the UK's educational

establishments. At this time, we talked about software as 'computer science'.

In schools and universities, computer science was an extension of mathematics where we learned the essential data structures such as queues, lists and arrays, the fundamental algorithms for sorting and searching, and communication protocols for sending and receiving data from and to other peripheral devices or across the wire.

At this low level of abstraction, the 'programming as a science' model is an appropriate one. However, today we are usually working at much higher levels of abstraction, far removed from the basic building blocks of computer systems, with altogether new problems of concurrency, parallelism and global connectivity.

We also are increasingly putting our software into the hands and eyes of unskilled users where emotional reactions to the machine interfaces we provide can make an enormous difference to an organisation's survival prospects. Here the science meme falls very short with the skills and techniques learned in computer science class become largely redundant. Now, complementary capabilities, such as human psychology, design-thinking and aesthetics become far more valuable.

As a test of this, ask any developer today to compare and contrast bubble-sort with quick-sort and they'll probably stare at you blankly - sorting is now a solved problem, it's solutions embedded within the libraries or frameworks we use to build code on top of, not something we should be

concerning ourselves with unless we really need to.

Generally speaking then, software as a science is no longer appropriate. Having said that, there are still areas where computer science still rules. If you're creating a new programming language or compiler then you'll need to delve down into the depths of your virtual machine's bytecode or into the supporting operating system's kernel. There are also industries that have grown up around advances in computer science, such as high frequency trading algorithms for investment banks and hedge funds, cloud computing services and 'big data' number crunchers. These industries only exist through cutting edge computer science research and development.

So, software as science has its place for new platforms and special applications, but perhaps not for the vast majority of software developers.

Software as Engineering

The engineering metaphor, sometimes also referred to as the 'Software Factory', is probably the most common model of software delivery that I have come across in my experience. Organisations that have this as their dominant model give themselves away through their job titles, such as software engineers and solution or technical architects, and the way the way work is performed.

Software delivery in this model is based on the principles of

a manufacturing production line with a hierarchical system of control, detailed job specialisations and a focus on functional efficiency. The ideas behind this model originate in the industrial revolution and Frederick Winslow Taylor's theories of scientific management within manufacturing.

Scientific management concentrates on increasing efficiency, decreasing waste, and using empirical methods to decide what to do. By breaking the work up into specialist roles, the theory goes, the more of a product can be produced, defects minimised and costs reduced - giving the firm a distinct competitive advantage.

The idea of efficiency goes deep into the heart of this model. With work specialised and standardised, staff become like a cog in the gears of the organisations - they can easily be replaced at will by an identical, fungible part. Processes also becomes standardised using linear methods of control passing product development through separate phases of design, coding, quality assurance and, finally, delivery to the end customer.

The theory behind this may work for real engineering, but for software it is deeply flawed. Unlike typical engineering where the goal is to deliver identical parts over and over again (with zero variance and no defects) to be later reassembled into a complete item, the result of software development - the code - is only created once. Not only that, but for each software product or service you effectively have to create both the product and the factory needed to build it at the same time.

Organisations try to manage this major issue by standardising their software development methodology, the technologies used and the overall delivery processes to, in effect, create a 'super factory' to be used for everything. Compliance with these self-inflicted processes, rules and standards are achieved through governance teams who make sure all boxes have been ticked and reports filed in the right places.

For effective software delivery this model can - and indeed does - work, after all many of the products and services we enjoy today have been created in this way. The problems with it though are many: standardisation is not compatible with either flexibility or improvement. Through job specialisation no individual has a full knowledge of how the whole system works or how their efforts contribute to overall success. Motivation may suffer through the lack of a clear purpose, slow feedback and the repetitious nature of their work.

Specialisation also means your customer is probably an internal manager slightly higher in the pecking order than you. Unless you work on the front line of sales or marketing notions such as 'real' customers, costs, profits, quality and service can very easily be rationalised away as to be 'somebody else's problem' and not something for which your daily output of effort will have any measurable effect.

Systems thinking and theory of constraints tell us that optimising subsystems will lead to a suboptimal system overall. A focussed attention on efficiency at the depart-

ment level may well be causing problems elsewhere in the organisation.

Software engineering is the wrong metaphor for the vast majority of software development projects. By using an engineering metaphor we end up making mistakes because it hides the true nature of software development as a social and intellectual activity, not a mechanical one.

Software as a Craft

Software craftsmanship is a relatively new metaphor for software delivery that started around the year 2000 with the publication of two books: 'The Pragmatic Programmer' and, a year later, "Software Craftsmanship" by Pete McBreen. The main focus on this philosophy is that creating software is more of a craft than an engineering discipline. Furthermore, software craftsmanship draws the metaphor between modern software development and the apprenticeship model of medieval Europe - where developers progress in their experiential training by going from apprentice then "journeyman to master" - this last phrase also being the subtitle of the Pragmatic Programmer book.

The idea emphasises that something is missing from software development today: attention to quality, and attention to expertise.

In many companies, developers are told exactly what they must build, how they must build it, what tools they must

use, when they must get it done, and even what measures they must achieve to show that they have done their job properly. Most software developers simply accept that all of this is out of their control – it's someone else's responsibility to tell them how to work. This might be reasonable if software development was a predictable and repeatable task – like say, putting together flat-pack furniture using assembly instructions. But, as experienced developers know, there is nothing predictable about developing complex software systems. Every developer who writes code is making critical decisions that will determine what the product really does and the only design document the computer cares about is the source code.

As a result of this lack of respect for their skills, many of the best software developers find themselves aiming for non-development roles to further their careers. I've had several top-developer colleagues who could be mentoring younger, less experienced developers, discard their programming toolset for the world of budgets, targets and plans. This is clearly a massive waste of talent.

Craftsmanship aims to put the longer-term needs of software back into the process rather than leaving it as an afterthought or for other developers to worry about when the code you created today becomes a liability tomorrow. It acknowledges the need for mentors to bring novice developers up to standard by creating a viable technology career path within technology.

Software craftsmanship is an attempt to reverse the decline

in the quality of software development and developer training. It makes the point that learning software development is not the same as being taught how to program and that apprenticeship is more effective than training to learn a craft since it is more about learning than it is about teaching. The apprenticeship model deliberately avoids the "learned helplessness" of the traditional schooling model by making the apprentice an integral part of the software development team.

Software as Architecture

Many organisations I've worked with use this metaphor, and it is commonly used in conjunction with the engineering metaphor described earlier. In this model software is seen as a construction that needs designing, building and fitting, after which the intended residents can finally move in.

Software architects generally work in a small team of senior developers with a similar set of values, or more commonly alone. They see themselves as the design authority or accomplished solution design hero - the architect - creating the blueprints for other software teams to follow.

In this model, the main value comes from the experienced architect having a complete understanding of the problem domain, in him creating the software model that needs to be implemented and then passing it down the line for other teams to implement. Here, software development itself is

viewed as a relatively unskilled profession as the code packages have been auto-created by modelling tools, the package structure has already been specified and the classes and interfaces already defined. Software developers simply have to 'fill in the blanks' and we're done.

Unlike the craftsman, the architect is not thinking about sharing responsibility with others or mentoring younger players - he's not even part of the development team being conspicuous by his absence when it comes to delivery or integration. The architect is viewed as issuing technical edicts to the developers, while having little or no accountability for specific projects himself. Instead the perception may be that he's solely interested in the glory and kudos of being at the top of the pyramid; in creating the tools, platforms and frameworks for others to use throughout the enterprise.

I've seen many organisations stuck with unsuitable or dying code frameworks created by their incumbent software architects that never really solved the problems they were meant to solve and instead created untold unintended consequences that served to hold the company back.

Ripping these frameworks out for more suitable software models is a big job that takes a fair amount of courage for any manager to approve and a lof of resources to accomplish. Furthermore, with the architecture model in place the 'not invented here' syndrome is a common dysfunction leading to one poor framework be replaced by another equally unsuitable one that has been sold to the purseholders as the next big thing.

Software architects face many challenges, not the least of which is that among developers, architects are often seen as ineffectual and disconnected impediments to meeting project deadlines. By sitting in their ivory towers architects get little respect. What they do get is resentment and resistance.

If there were only architects, we would be stuck in Utopia. If there were only developers, we could end up with a mess because of a lack of cohesion and direction. Architecture is clearly needed, however software development is a social activity not an individual one. The 'designer as architect' metaphor encourages ego before leadership. Good, robust software architectures are created by a team in the trenches together, not by distant decree.

Software as Gardening!

The main problem with the engineering and architecture metaphors above is that they assume software, and the foundation on which it sits, is a solid, unmoving platform, that there is a 'best' solution to be found and the work itself is rational, simple and controllable.

The 'software as gardening' metaphor rejects these ideas for one where the environment is always changing, the seasons come and go, where bugs and predators are everywhere and weeds, if not removed, take over from the flowers and lawn.

As a gardener you have an idea on what makes a 'great' garden: the layers within the underlying landscape, the properties of the soil, the arrangement of flower beds and trees and the colours and textures that work together.

You'll probably have a good idea of what your garden should look like a week into the future. You might even have a rough idea of the shape you expect it to be in a year from now. But you have no idea of where each branch, leaf, stem and flower will be.

You'll know you cannot make your garden grow faster by hiring more gardeners or throwing more fertiliser or water at the plants, and that every garden is different because the environment it is in - the soil, climate and local wildlife - is different. Even gardens that are within throwing distance of each other can have wildly different soil - what the winemakers of France call *terroir*.

With a garden, there's a constant assumption of maintenance. Everybody says, I want a low maintenance garden, but the reality is a garden is something that you're always interacting with to improve it or even just keep it looking the same.

Your garden will grow weeds, and the longer you leave it alone the more weeds and other problems will emerge. Similarly with software, you will need constant pruning and weeding to keep the codebase clean and functional even, and this is important, if you leave it well alone. Even if you don't make any changes to your code at all it will still decay as the surrounding environment - the

language, framework, libraries or the underlying platform will all continuously be updated in the background. Come the point when you need to make even the most minor change you may find all the things your code relies on are no longer current, are unsupported or discontinued. This gradual buildup of technical debt will need to be repaid in full in one go whereas a continuous tending to your software garden would have prevented this altogether.

In gardening you do need to plan. Your plan details the size of your plot, how the soil needs to be prepared and the types and species of plant you're going to install. You know to put the big plants in the back and short ones in the front and that the garden , as well as being pretty to look at, also has a functional purpose that it must fulfil.

You've got a great plan - a complete design- but what happens when you plant the bulbs and the seed? The garden doesn't quite come up the way you drew the picture. This plant gets a lot bigger than you thought it would. You've got to prune it. You've got to split it. You've got to move plants around the garden. This big plant in the back died. You've got to dig it up and throw it into the compost pile. These colors ended up not looking like they did on the package. They don't look good next to each other. You've got to transplant this one over to the other side of the garden. So it is with software.

Only an experienced gardener really knows another good gardener when they see one. Someone who has merely managed gardening projects will have no idea what they

should be looking for and have the added problem that they are unaware of this fact - second order ignorance if you will. So if you are not a gardener, but need to recruit good gardeners, then you'll need to quickly find an experienced gardener you trust to vet your candidates.

Software, like gardening, is a process of continuos learning in context. Planning is important, but of more value are the ability to inspect and adapt as your assumptions are proved false and previous knowledge shown to be flawed or inappropriate in this context.

The gardening metaphor, although at first glance unusual, is an appropriate one for those experienced in software development.

Conclusion

The metaphors we use for software development and varied and many. If you do not carefully chose and communicate the metaphor you wish to be dominant - an intentional metaphor - then an unintentional model - your 'metaphor in use' - will become dominant and direct people unconsciously in their behaviours and actions.

Remember that, like the blind men feeling the elephant, all metaphors are maps of the territory and not the territory itself. Also remember that some metaphors are more appropriate than others for the task of developing valuable, dependable and adaptable software and that with each choice

come tradeoffs in shared understanding and assumptions on how work should be done.