Software Development as a Collaborative Writing Project

Brian Bussell[1] and Stephen Taylor[2]

[1] Norwich Union Life, 60 Wellington Way, York YO90 1LZ
brian.bussell@norwich-union-life.co.uk
http://www.norwichunion.com
[2] British APL Association, 81 South Hill Park, London NW3 2SS
editor@vector.org.uk
http://www.vector.org.uk

Abstract. Software describes an imagined machine. To be software, the description must be executable, which means written so a computer can animate it. Non-executable descriptions (specifications, designs, &c.) are instrumental to this work; they are intermediate texts. We advance a model of software development as the collaborative writing of a series of descriptions. We propose the chief distinction of agile development to be the exclusion from this process of the human translation of intermediate texts. We distinguish supported and unsupported communication. We analyse the success of Extreme Programming in terms of avoiding unsupported communication and prioritising feedback from executable descriptions. We describe functional programming techniques to construct notations that allow programmers and users to collaborate writing executable system descriptions, collapsing distinctions between analysis, design, coding and testing. We describe a metric of code clarity, semantic density, which has been essential to the success of this work. We report the use of these techniques in the Pensions division of Britain's largest insurer, and its effect on the business.

1 Introduction

In this paper we advance a view of software development in which similarities to collaborative writing projects such as making movies or drafting legislation matter more than resemblances to civil engineering. This view is grounded in professional experience of writing software, in the mathematics of computer science, and in the philosophy of linguistics. It is contrary in general to the conventional model of software development, and in particular to what has become known as 'software engineering'.

From it we derive radical development practices and report their use at Norwich Union.

To establish common ground, we start with fundamentals.

2 Universal Turing Machines and Programs

The important characteristic of a computer is that it can be loosely thought of as a Universal Turing Machine (UTM). Without software a computer is useful only as a

P. Abrahamsson, M. Marchesi, and G. Succi (Eds.): XP 2006, LNCS 4044, pp. 21 – 31, 2006.

doorstop. Its real value is its ability to emulate the behaviour of other machines. The applications we run on computers are representations and emulations of the behaviour of machines.

Almost all machines emulated by computer applications are imaginary. While early word-processing applications emulated and extended the behaviour of real typewriters, modern word-processing applications now emulate machines that never have been built and never will be. Mechanical computing reached its limits with Babbage's Differential Engine; he never completed his more ambitious Analytical Engine. Our dreams exceed our abilities to press, cut and weld.

Babbage could not build his Analytical Engine with Victorian engineering and the funds he could raise; his completed machine remained a dream. But with programmable UTMs, we routinely emulate imaginary machines more complex than Babbage's.

The key is the program. A program describes the behaviour of a machine in terms that allow a UTM to animate the description. This is what programmers do – we write beforehand (*pro-gram*) executable descriptions of machine behaviour.

A UTM, like the magical character in a folk tale, grants wishes. As in the folk tales, accurately describing what you want turns out to be harder than it seems. Our dreams are light on detail and have unforeseen consequences. Software developers are the heirs of King Midas1[1].

3 Creative Writing and Translation

Here are two descriptions of a machine too complex for Babbage to have built.

```
mean=: +/ % #   NB. arithmetic mean of a list of
numbers
```

The first description (`+/ % #`) is executable, and written in the J programming language. The second description follows the NB. and is written in English. To a reader of both languages, the descriptions are equivalent; that is, each translates the other.

The first description can be animated with the help of a J language interpreter. A J interpreter executes C code, which becomes a new (and lengthier) description of the desired machine behaviour. A C compiler then composes an even lengthier description in a chip's instruction set. The chip is a UTM; animation can now begin.

Instructions to a chip are the final form of software. Now consider where a software development project starts; consider the following text.

[1] "Dionysus, who had been anxious on Silenus's account, sent to ask how Midas wished to be rewarded. He replied without hesitation: 'Pray grant that all I touch be turned into gold.' However, not only stones, flowers, and the furnishings of his house turned to gold but, when he sat down to table, so did the food he ate and the water he drank. Midas soon begged to be released from his wish, because he was fast dying of hunger and thirst; whereupon Dionysus, highly entertained, told him to visit the source of the river Pactolus, near Mount Tmolus, and there wash himself. He obeyed, and was at once freed from the golden touch, but the sands of the river Pactolus are bright with gold to this day." [1].

> Our competitors are beating us on delivery. Our process is too slow; we just can't get our goods out of the door fast enough. We need a new order-processing system.

It contains the following description of a machine: a new order-processing system. In its likely context – a senior-management conversation – it sufficiently describes the solution to a business problem. The description carries for that conversation the right three facts about the solution: its behaviour will be to process orders, it will involve a computer, and it will have to be acquired.

This description is not executable. It corresponds to many executable descriptions – to too many. Developing the system means writing an executable description of a machine that solves the business problem.

Note that this development process is not translation. The application code and a new order-processing system both describe the behaviour of an imaginary machine, but they are not translations of each other.

We propose that agile and conventional models of software development are most clearly distinguished by the inclusion or exclusion of human translation as a project activity.

Put another way, conventional software development projects measure their success by whether they have accurately translated a non-executable system description into an executable one – does the program match the specification? Agile projects ask only – have we solved the business problem?

Of course, no one attempts to write an executable translation of a new order-processing system. Instead, analysts meet sponsors and write specifications. Specialists read specifications and write design documents and data architectures. Analyst/programmers write program specifications. And programmers either write program documentation and translate it into code, or translate specifications into code and then back into documentation. Thus the conventional model. We know that human translation is included, because an ideal of the conventional model is to derive the code from the program specification alone, or from the documentation alone.

4 Two Great Lies of Software Development

When ye sup with the Devil, use a long spoon. (Trad.)

There are two Great Lies of software development. The first is *I can tell you what we need*. The second is *I can tell you what it will take to build it*. Both lies contain enough truth to nourish illusion.

A new order-processing system is the first of a series of descriptions. Each successive description expands its predecessor. The last and longest description is composed in a chip's instruction set; but the last several descriptions are all formal equivalents (translations), produced without human intervention by compilers and interpreters. Programmers make the last human contribution, by writing the first executable description. The point to keep in mind is that from a new order-processing system to chip instructions, is nothing but behavioural descriptions all the way down.

The speaker of *we need a new order-processing system* will not enlarge greatly on his description. He has competent staff to do that. Describing what you want (specifying requirements) is understood to entail much discussion and analysis, balancing of priorities, and consideration of foreseeable changes in the business. But the underlying assumption is that it can be done. In most fields of activity, inability to describe what you want is a reliable indicator of incompetence.

Let us entertain the contrary, and suppose the folk wisdom is right: imagining and describing accurately a complex machine 'from thin air' is at least greatly more difficult than it appears, and perhaps too difficult for practical purposes. Here is a thought experiment.

Call the speaker of *we need a new order-processing system* the system's *sponsor*. Suppose the sponsor actually does know precisely how the system should behave. He can give consistent answers to any question on the subject; he envisages its interfaces to the user and to other systems, and understands the important implications. The sponsor is ready to write a description of the imagined machine that requires only translation into source code. Call this description the *absolute specification*.

How is he to write it? Plain English will not do; its imprecision and elegant ambiguities disqualify it for the job. Formal notations are available for different parts of the work: for example, UML for user interfaces; functional decomposition diagrams. In principle, nothing prevents the sponsor learning these notations and writing the absolute specification. But mastering them requires years of work. The sponsor will not have done this; the premise is that he is a businessman.

5 Writing Software Without Programmers

Exceptions to this are important and instructive. In the 1980s spreadsheet applications removed programmers from an entire field of software development. Using spreadsheets is not considered programming, but Taylor recalls working on an Australian government tender in the early 1990s in which the spreadsheets developed in its support exceeded in complexity most software he had previously written.

The important contribution of spreadsheets, where used, is to remove the element of human translation from software development. Microsoft has had some success in extending this with Visual Basic, making possible for users a good deal of tinkering with its products. Spreadsheet and Visual Basic users do not think of what they do as programming, nor are they encouraged to; Microsoft promotes Visual Basic as a 'productivity tool'.

We note here the bias in the usage of 'programming': what one person does for another, not what one does for oneself. A professional programmer programs machines for others to use.

Actuarial calculations provide solutions in the once stable but now fast-changing insurance business. These calculations routinely exceed the descriptive powers of spreadsheets. Actuaries have long written executable descriptions of their calculations. Bussell, an actuary, recalls writing them in APL in the 1980s. In the Pensions division he now directs, calculations are described either in APL, or by actuaries writing direct in Mathematica.

Instructive examples can also be found in the financial markets, another field in which the cost of delay has minimised or eliminated human translation in developing

software. Financial traders commonly either write their own software or seat programmers in the trading room.

> When I started in this business, every trader had a Visual Basic manual on his desk; now it's more likely to be a J2EE manual. [2]

In practice, competence in the business and in writing software coincide only where the business requires mathematical skills. In consequence, highly abstract executable notations such as APL, A+, J, K, Q, R and S flourish primarily among actuaries, financial traders and statisticians.

6 Notation as a Tool of Thought

In the thought experiment above we imagined our sponsor knew exactly how the machine should behave. From this we saw the lack of a suitable notation in which to write the absolute specification as a formidable, and probably an insuperable, obstacle.

But the premise itself is untenable. It supposes the sponsor has completely imagined the desired machine, which is to say that he has an absolute specification 'in his head'.

A strong body of opinion, associated with Chomsky and Fodor [3], maintains such a mental description entails mapping to some internally coded language. Even an Andersonian realist would suppose such a description handled by the brain in the same way that language is handled. We think and speak only what can be expressed in language.

> 7. Wovon man nicht sprechen kann, darüber muß man schweigen[2].

Wittgenstein's later argument [5] against the possibility of private language further restricts the scope of thought. We know languages only by sharing them. We think only what can be expressed in languages shared with others. In studying a subject one acquires new thoughts along with the vocabulary to express them. [6]

In his later argument, Wittgenstein also came to see that the relationship between language and its referents is inexact, slippery and ambiguous; mediated by what he called our language games. We can never find in our everyday languages the precision and completeness we aspire to in mathematical notation.

> Well here again that don't apply
> But I gotta use words when I talk to you.
> …
> I gotta use words when I talk to you.
> But if you understand or if you don't
> That's nothing to me and nothing to you
> We all gotta do what we gotta do
> *T.S. Eliot*/ "Fragment of an Agon"

[2] 7. Whereof one cannot speak, thereof one must be silent. [4].

7 Hubris and the Children of Dædalus

How can I tell you what I think,
till I've heard what I have to say?
Anon.

The first Great Lie of software development has two parts: *I know what I need* and *I can tell you what it is*. We have examined the difficulties of the second part while assuming the truth of the first. Now we will show what is more important: the first part is also false.

Whether writing novels, screenplays or software, we start with an incomplete vision. It is only in the process of elaboration that we understand the implications of our choices, and see, for example, that having *let everything I touch turn to gold* leads to death by starvation. Had King Midas been granted a revision of his wish, Version 2 would have included a switch to turn transmutation on or off.

Some software has to run correctly on first use – ask NASA. Such software has to be written without the benefit of feedback from use in the real world.

The legends of King Midas and Icarus [7] warn against hubris; that our dreams have unintended consequences. In pursuing our dreams, we need feedback from the world to understand their consequences. It was not the cautious craftsman Dædalus who fell to earth, but his son – carried away by his dreams.

When, sensitive to the appearance of incompetence, we subscribe to the first Great Lie and claim we know what is needed and that we can describe it, we set aside warnings from our culture's long tradition. We risk hubris rather than admit ignorance.

Even when we qualify the lie by allowing significant time to write and discuss the description, we ignore Wittgenstein's warnings about the slipperiness of language and the importance of grounding language games in reality. In thick, formal specification documents, thousands of pages of charts, tables and narratives, we detail the machines we shall conjure. So many details, so much dreaming – and no feedback.

We assert what Wittgenstein would predict: little effective communication is possible about purely imaginary machines.

8 Raiders of the Inarticulate

Because one has only learnt to get the better of words
For the thing one no longer has to say, or the way in which
One is no longer disposed to say it. And so each venture
Is a new beginning, a raid on the inarticulate
T.S. Eliot/ *"East Coker"*

Extreme Programming (XP) shares this fear of hubris [8] and addresses it briskly. XP teams begin by building the smallest system of any possible value [9] and use that with the sponsor to conjure, explore and revise his dreams. Incremental development treats knowledge as scarce and communication as uncertain.

Communication within XP projects reflects scepticism about its value divorced from a running system. The sponsor's representative sits with the programmers at all times, to resolve immediately questions about the business values of different

behaviours. These questions can always be related to the behaviour of the running system, for XP projects always have a running system. Small changes in behaviour are delivered frequently, so their consequences can be explored piecemeal. Incremental developers know that dreams turn easily into nightmares.

Everything is referred to the final human text, that is, the source code. There are no intermediate texts for humans to translate into executable notation, and against which to measure accuracy of the translation. Agile projects do not ask *did we implement the specification accurately?* but *have we solved the business problem?* Such intermediate texts as are written are destroyed after use; the running system is the starting point for any discussion of change. All changes are changes to the running system.

In similar vein, for software quality and cross-training, XP programmers collaborate upon the source code (pair programming) rather than talk or write about it.

Call communication, spoken or written, *unsupported* in the absence of its object. Talk about a system, in its absence, is unsupported. Talk about source code, in its absence, is unsupported.

Such talk is not cheap. Unsupported talk is unreliable and expensive.[3]

An engineer designing a machine minimises use of components that are unreliable or expensive. XP teams use much less communication than the conventional model. Nearly all of it is supported.

9 Shrinking the Circle

How can we use this view of software development to solve business problems faster?

The Pensions division of Norwich Union processes claims from a range of products inherited through mergers and takeovers, and supported upon a range of administrative systems. Processing the claims is relatively complex; when tackled with word-processor scripts, spreadsheets and mainframe terminal emulators, claims took about an hour to process. Clerks required six weeks training to begin handling the simpler claims of their department, and six months to be able to handle all of them. Cross-training took about three weeks, so departments were rarely able to help each other balance work loads, and backlogs were common.

Regulatory and legislative changes are now relatively frequent, as are changes to the administration of the business, and the organisation of the company. This describes an environment changing so rapidly that no conventional software development project to cut processing costs has ever been contemplated.

We have nonetheless been able to exploit insights gleaned from XP and apply them with a team initially of two programmers. The system went into production use after some months, and two years on, now processes as many pension claims as the company's core IT systems. It has halved the average processing time for claims, slashed training time to days, and eliminated backlogs. The system is now supported and developed by four programmers.

A key technique to the success of this has been programmers and users collaborating on writing the source code.

Not all of it, by any means. But key business rules, previously enshrined in user training and check lists, had appeared impractical to analyse in such an unstable environment. For example, a 5-page check list described how to determine whether a

[3] This is the subject of a forthcoming paper.

claim was liable to a certain penalty. An executable version of the rules was first worked out with a test battery of examples, then refactored for clarity into its present form, in which it is occasionally amended by the senior clerks and programmers together. The complete rules fit on an A4 page, and are appended.

Programmers have made this possible by constructing local, domain-specific executable notations. The vocabulary of these notations is drawn from the users' talk about the work. Our tiny team does not have an Onsite Customer; instead it works among its customers. These notations constitute the shared languages required by Wittgenstein and lauded by Whitehead.

> By relieving the brain of all unnecessary work, a good notation sets it free to concentrate on more advanced problems, and in effect increases the mental power of the race. [...] By the aid of symbolism, we can make transitions in reasoning almost mechanically, by the eye, which otherwise would call into play the higher faculties of the brain. [...] It is a profoundly erroneous truism, repeated by all copy-books and by eminent people when they are making speeches, that we should cultivate the habit of thinking of what we are doing. The precise opposite is the case. Civilisation advances by extending the number of important operations which we can perform without thinking about them. [10]

Constructing a notation is not the end of the task, but it brings the end near. Users and programmers can now converge quickly on and verify a common understanding. The notation enables them to avoid ambiguity; it is a "tool for thought" in the sense of Iverson's Turing Award lecture [11]. Because the notation is executable (and interpreted), the running system animates the described behaviour in front of them.

Taylor describes a typical scenario with an expert user, S.

> S comes and sits beside me, facing the system running in my development environment. She has not written or spoken to me about the change she contemplates. She begins by getting the system to do something she wants changed: either crashing it, or displaying or printing something other than what she wants. She can now point at the behaviour she wants changed. I use my knowledge of the source to locate the rules controlling that behaviour, and we trace execution, jointly examining how those rules are expressed and applied. We then agree an amendment, which seems to express what she intends, and resume execution. We try other examples, explore implications and revise the rules until we're satisfied the revision seems to express what S contemplated. I save the revised version where she can test and explore it further.

Working this way is insanely productive, because the system's behaviour can be revised without interrupting the conversation. If changes to the code took only an hour, the process described above would cover days. As it is, analysis, design, implementation and alpha-testing collapse into periods sometimes measured in minutes. In the last two months of 2005 the team released over 200 changes to the production system.

10 Semantic Density

Our ability to work at this speed depends on programmers and users collaborating on the source code. This in turn depends upon the value of a metric we have dubbed *semantic density*. [12]

Semantic density is the proportion of tokens in the source code that are drawn from the users' *semantic domain*; that is, the vocabulary they use to discuss the work. It ranges between 0 and 1. We are able to achieve very high values because

- functional- and array-programming techniques, and judicious use of anonymous lambdas, enable us to avoid defining terms (such as counters) outside the users' semantic domains;
- languages derived from Iverson's notation support a version of Church's lambda calculus, enabling us to define entire local vocabularies in a few lines of code;
- users ignore APL's analphabetic glyphs, which have no impact on semantic density.

11 Conclusion

A software development project imagines and describes a desired process. This is difficult work, which has become no easier since King Midas tried it. The unsupported communications and intermediate texts of software engineering do little to help this work and a great deal to prolong it. Much XP practice can be characterised as eschewing intermediate texts in favour of high-bandwidth communication between sponsor and programmer, supported by feedback from a running system. Functional programming techniques extend this by allowing sponsor and programmer to collaborate on source code, and permit radical gains in productivity. These techniques are in use in the Pensions division of Britain's largest insurer.

Acknowledgements

The authors thank Paul Berry, Chris Burke, Gitte Christensen, Romilly Cocking, Roger Hui and Charlie Skelton for comment on a draft of this paper.

References

1. Graves, R. *The Greek Myths*, London, Folio Soc., 1996, p.263
2. Mark Sykes, Director, Global Markets Finance, Deutsche Bank, London addressing Kx Systems User Meeting; see report in *Vector*, Vol.21., No.2 http://www.vector.org.uk/archive/v212/kx212.htm
3. Fodor, J., The Modularity of Mind, MIT Press, Boston, 1983
4. Wittgenstein, L., *Logische-Philosophische Abhandlung*, Cambridge, 1922
5. Wittgenstein, L. *Philosophical Investigations: German Text, with a Revised English Translation* Blackwell, Oxford, 2002
6. Taylor, S. "The Experience of Being Understood: on requirements specification as a Wittgensteinian language game" http://www.5jt.com/articles.php?article=beingunderstood
7. Graves, R. ibid., p.291
8. Beck, K. *Extreme Programming Explained: Embrace Change* Addison-Wesley, Boston, 2000, p.165
9. Beck, K. ibid. p.131&ff.
10. Whitehead, A.N. *An Introduction to Mathematics*, p59, H. Holt & Co., London, 1911 http://www.headmap.org/unlearn/alfred/1.htm
11. Iverson, K.E. "Notation as a Tool of Thought" 1979 ACM Turing Award lecture, *Communications of the ACM*, Vol.23, No. 8; see http://elliscave.com/APL_J/tool.pdf
12. Taylor, S.J. "Pair Programming With The Users", *Vector*, Vol.22, No.1 http://www.vector.org.uk/archive/v221/sjt221.htm

Appendix

A fragment of source code maintained jointly by programmers and users.

```
[0]    WillMvrApply dic
[1]
[2]    applies doesNotApply refer←1 0 ¯1                    ⍝ range of answers
[3]
[4]    ⍝ LOCAL VOCABULARY ------------------------------------------------------------
[5]    asIDN←{dft 3⍴⍵.IDNToDate ⍺ ⍺⍺ ⍵.DateToIDN dtt ⍵}
[6]    daysAfter←+⍀asIDN
[7]    daysBefore←-⍀¯asIDN
[8]    oneOf←{(⊂⍺~⍬⍴TC)∊⍵}
[9]
[10]   all←∧/ ⋄ not←~ ⋄ before←< ⋄ after←> ⋄ no←~∘(∨/) ⋄ and←∧ ⋄ or←∨ ⋄ any←∨/
[11]
[12]   ⍝ BUSINESS RULES START HERE ---------------------------------------------------
[13]   isDeferred←OrigNRD≠Nrd
[14]   isBefore911←OrigNRD before 20040911
[15]   termOver5←OrigNRD≥5 yrsAfter StartDate
[16]   termUnder5←not termOver5
[17]   deferredBy5←Nrd≥5 yrsAfter OrigNRD
[18]   ThreePolAnniversariesBefore←StartDate∘{2 yrsBefore ⍺ aoob4 ⍵-⊃=/10000|⍺ ⍵}
[19]   oneMonthBefore←¯1∘DATE∆ADDMTHS
[20]
[21]   :If not InUWP
[22]       answer←doesNotApply
[23]   :ElseIf termUnder5 and RetirementAge≥75
[24]       answer←refer
[25]
[26]   :ElseIf RetirementAge≥75
[27]       answer←doesNotApply
[28]   :ElseIf ExitDate after oneMonthBefore 75 yrsAfter Dob
[29]       answer←doesNotApply
[30]
[31]   :ElseIf Assurer='CU'
[32]       answer←any MVRs>0
[33]
[34]   :ElseIf IsVista
[35]       answer←ExitDate before ThreePolAnniversariesBefore OrigNRD
[36]
[37]   :Else
[38]
[39]       exception←(Prepare≡'Quote')and(ExitDate=Nrd)and deferredBy5
[40]       exception(and)←isBefore911 or IsMvrFreePoint
[41]       :If isDeferred and not exception
[42]           answer←applies
[43]
[44]       :ElseIf (Prepare≡'Payment')and ExitDate before Nrd
[45]           answer←applies
[46]
[47]       :ElseIf ExitDate before 183 daysBefore OrigNRD
[48]           answer←applies
[49]       :ElseIf (Prepare≡'Quote')and ExitDate≠Nrd
[50]           answer←applies
[51]       :Else
[52]           useRule←deferredBy5 and(isBefore911 or(termOver5 and IsMvrFreePoint))
[53]           useRule(or)←termOver5 and not isDeferred
[54]           :If not useRule
[55]               answer←applies
[56]           :Else ⍝ SPTP rule
[57]               recentSPTP←PAYTS.EFFECTDATE after 5 yrsBefore OrigNRD
[58]
[59]               :If (all recentSPTP)and not RegPmtFlag
[60]                   answer←applies
[61]               :ElseIf no recentSPTP
[62]                   answer←doesNotApply
[63]               :Else
[64]                   answer←refer
[65]               :EndIf
[66]           :EndIf
[67]       :EndIf
[68]
[69]   :EndIf
[70]
[71]   MvrApplies←answer
```