Why Making Software is so Difficult

Ryan Dawson

University of East Anglia
Norwich Research Park
Norwich, UK, NR4 7TJ
ryan.dawson@uea.ac.uk

Icon Solutions Limited
27 / 37 St George's Road
London, UK, SW19 4EU
www.iconsolutions.com

ABSTRACT
This paper develops a metaphor that expresses what is so difficult about software production. The purpose of the metaphor is to be applicable to real-world software production situations, especially on larger or more complex projects, where decisions are made in ways that neglect some of the aspects of software production that make it so difficult, especially that seeing how to solve a problem can be immensely difficult to do from the beginning and as a result one sometimes has to abandon a whole approach and start again. The use of the metaphor is justified with reference to related literature and by application to certain characteristic situations.

Categories and Subject Descriptors

C.5.0 [Computer System Implementation]: General

D.2.1 [Software Engineering]: General

K.6.1 [Management of Computing and Information Systems]: Project and People Management – *Systems analysis and design, systems development*

K.6.3 [Management of Computing and Information Systems]: Software Management – *Software process*

General Terms
Management, Design, Human Factors, Theory.

Keywords
Design, Agile, waterfall, lifecycle, estimation, development

1. INTRODUCTION
The purpose of this paper is to develop a metaphor that expresses what is so difficult about software production. The test of this metaphor is that it should be applicable to real-world software production situations where decisions are sometimes made in ways which overlook aspects of software production which make it so difficult –specifically that software problems are not all alike (though they often look alike), that seeing how to solve a problem can be immensely difficult to do from the beginning and as a result one sometimes has to abandon a whole approach and start again.

First I will reflect upon the role of metaphors in software production in order to suggest that metaphors could and should play a much more valuable role than is commonly thought. After considering how metaphors are commonly used in the software industry and its literature, I will argue that the software production process is too complex, divided and abstract to be easily grasped. What cannot be easily grasped is prone to be misunderstood and it will be argued that there are real-world situations where the software process is itself misunderstood in that decisions are sometimes made in a way which neglects certain subtle characteristics of software production. The characteristics which I take to be most overlooked are exactly the characteristics which make software production so difficult. This is not a coincidence – it is natural that we should be inclined to simplify something complex and abstract in situations where it is hard to see the problems in their entirety.

I will then develop a metaphor of software production as a kind of creative problem-solving. The purpose of developing the metaphor is to draw attention to these aspects and to argue that the metaphor could be used to fight the temptation to overlook them at key decision-points in projects. This will be illustrated with reference to example situations. The purpose of the metaphor is to work as a tool for reminding software workers that software projects can be difficult to envisage from the beginning, that work done on a project can sometimes have to be thrown out in a wholesale fashion and that the less attentive we are to these features then the more likely it is that wholesale revision will be necessary.

2. THE ROLES OF METAPHORS IN SOFTWARE PRODUCTION
One common place to find metaphors for software production is in relation to disputes about which methodologies are best for approaching software projects. Sometimes these debates are framed in terms of which methodology is a better fit for how the software production process works. In recent times much attention has been drawn to the idea that waterfall methodologies may embody a manufacturing/engineering metaphor, especially in regard to the strict separation of phases and the time and cost controls that these methodologies impose upon phases of software production. Here is a brief comment on this from Fowler:

> As regular readers of my work may know, I'm very suspicious of using metaphors of other professions to reason about software development. In particular, I believe the engineering metaphor has done our profession damage - in that it has encouraged the notion of separating design from construction. [1]

There is a suggestion here that maybe we shouldn't be appealing to metaphors of software production at all. There is certainly something right in Fowler's suspicion that metaphors can mislead. We appeal to metaphors when we want to see one thing by comparison with another and this can naturally lead to our seeing the one thing in terms of the other. Perhaps thinking of software in comparison with manufacturing can easily slip into thinking of software production *as* manufacturing. And yet McConnell, noting the abundance of metaphors for software production in the literature, voices a plausible view which is almost the opposite to that of Fowler:

> ...the person who uses metaphors to illuminate the software-development process will be perceived as someone who has a better understanding of programming and produces better code faster than people who don't use them. [2]

Fowler is surely right that metaphors can be dangerous if too much is read into them. But it would be taking Fowler's point too far to say that the software industry should try to do without metaphors entirely. It would be reasonable to do without metaphors if we were talking about something which was fully known and understood, but software production is complex and difficult to get a grip on. Metaphors can help illuminate particular aspects of the software production process for particular purposes. If we consider how metaphors are used in software

production then we can see ways how they can, as McConnell suggests, help software workers perform their roles more capably.

Metaphors for software production are perhaps most useful when in relation to broad questions that force us to consider many aspects of software production at once. For an illustration of how broad questions can be relevant to particular situations, consider this argument by John Reynolds:

> Some argue that one can manage software production without the ability to program. This belief seems to arise from the mistaken view that software production is a form of manufacturing. But manufacturing is the repeated construction of identical objects, while software production is the construction of unique objects, i.e., the entire process is a form of design. As such it is closer to the production of a newpaper [sic] — so that a software manager who cannot program is akin to a managing editor who cannot write. [4]

Reynolds' question is not merely abstract and it has a real practical pull to it – one can imagine that hiring discussions in some companies might really centre on a question about how much knowledge a software manager should have. The analogy that Reynolds presents – that of software production as like newspaper production – has advantages and disadvantages (as all analogies do). These advantages and disadvantages for a particular case will depend upon how a company structures its job roles. A serious disadvantage of the metaphor for many firms will be that the role of the newspaper editor has no single analogue in how they approach software production. An editor makes the decisions about which articles are going to be featured where in the issue and even makes decisions about the content of those articles. This is the sort of thing that a team lead would do in a software project and indeed we would expect this role to be filled by somebody who can program. But there are other things that an editor does too – presenting to the newspaper's owners and sponsors and managing budgets. These are things for which one does not need to be able to write and the software equivalent of these tasks is typically fulfilled by a project manager.

The sharp division of roles in software production is part of why broad questions about the process are likely to arise and to be difficult to deal with. Clearly, there are people working in software and in intimate connection with software who are only familiar with (in the sense of being able to work comfortably within) *part* of the software production process. Some project managers are not familiar with development, some testers are not familiar with design and many project sponsors and end-users that are not especially familiar with any part of the process. Metaphors can be useful when these people have to talk to one another with regards to points that concern parts of the process that bridge their respective areas (or even which concern how particular parts of the project relate to the project as a whole). For certain situations in software, understanding and explaining the relation of what one is doing to what somebody else is doing is a very important and a very difficult thing to do.

To give such an imaginary situation, let us say that an overrun takes place on a project and the cause of the overrun is that the developers were following a particular design and towards the end of the implementation they realize that what they have developed does not fit with the requirements. In this case we can imagine a project manager asking whether the developers have followed the design incorrectly or whether the design itself was wrong. Perhaps the project manager himself is being told by senior management to find out who is at fault. It might be tempting for senior management and perhaps for the project manager to think that there must be a clear answer but I would imagine most developers would say that sometimes there is no clear line. I think that most developers will share my intuition that a design does not always prescribe a single line possible result and also that sometimes one might try a variety of ways to follow an approach before having to acknowledge that one has to give up on the approach without necessarily being able to say that the approach cannot *somehow* (perhaps only with a inordinate amount of effort only with insight bordering on genius) be made to work.

For anyone who faces circumstances like this on a regular basis, it can be useful to have the right kinds of metaphors to fall back upon. Even if one were a developer experienced in all aspects of the software production process, it might be difficult in a situation like the one described to resist the temptation to spend long hours studying the design to try to work out if it can possibly work. One really might be taken in by management's insistence that there has to be an answer and this being taken in might not just be a matter of succumbing to pressure – it might be said to embody a misunderstanding about what producing software is like.

I intend to present a metaphor that is tailored to remind us of many of the subtleties of software production so that the metaphor can help us to see what goes wrong in this hypothetical circumstance. The metaphor should also be applicable to other kinds of circumstances where questions about the nature of software development arise (including questions about how to choose methodologies for particular projects). The point is that the software production process is various, complicated and abstract. Even if one is a very experienced software worker, it can be difficult to command a clear view of the process. When things are complicated and abstract, we sometimes want them to be simpler than they are and we can fail to respect their subtleties. In these circumstances a well-chosen metaphor can point us towards the subtleties of what we're trying to understand.

A metaphor will be needed which shows that design and development cannot easily be separated. Reynolds's newspaper metaphor is intended to reflect this – as he says "software production is the construction of unique objects, i.e., the entire process is a form of design." However, the metaphor is not a particularly close fit as we surely want to say that design and development are separated in some sense. Whilst there might be aspects of software production that make design and development difficult to separate in the way that the hypothetical project manager asks, this does not mean that there is no separation. The newspaper metaphor makes it hard to see how there could be any separation at all. We want an analogy that can fit closely enough that we can continue to come back to it and see different aspects of software production reflected.

Another flaw of the newspaper analogy points to a different aspect of software production which will need to be reflected in a good metaphor. The writing of a newspaper involves reporting on events in the world and this is clearly not what we do when we produce software. For this reason, newspapers are much less likely to need to undergo wholesale revision than software projects are. So long as a newspaper reflects and reports upon actual events then it cannot be entirely inadequate, even if it were badly-written. Software projects don't have a descriptive function to fall back upon and in this respect software projects can fail much more spectacularly than newspaper issues. Badly-written software is likely to require throwing out entirely.

Both of these aspects, the indefinite separation of design and development and the susceptibility to revision, are connected with the abstract character of software production. In order to find a metaphor which will reflect these aspects and enlighten us with regard to them, we need to consider a kind of activity which is abstract, is approached in stages and in the course of which work sometimes has to be thrown out in a wholesale fashion. Such an activity can be found in the field of creative problem-solving.

3. THE PROBLEM-SOLVING METAPHOR

There are many kinds of problems and puzzles, much as there are many kinds of software projects, and I don't mean to compare software projects to only one specific problem but rather to a whole class of

problems. There is a particular class of problems which are both creative and logical, a class which might be thought to fall under the heading of 'riddles' (though I do not think everything called a 'riddle' is necessarily a good fit). I hope to make it clearer how creative problem-solving of this kind can be seen as a metaphor for software production by considering how a particular story of puzzle-solving from the old Norse saga of Ragnar Lodbrok gives a surprisingly good analogy for software production.

As the story goes, Ragnar's men report to him that they have seen a very beautiful woman, Kraka. Ragnar's interest is aroused and he sends for her, but he decides to test her wits. He commands her to arrive neither dressed nor undressed, neither hungry nor full, and neither alone nor in company. Kraka is up to the challenge and arrives draped in a net and her long hair, biting an onion, and with only her dog as a companion. Ragnar is impressed and marries her.

Ragnar's request is not unlike the specification of a software project in at least one key respect. Ragnar does not entirely know what will satisfy his request until he sees it. He knows certain constraints that will need to be met but he will have no idea that the solution Kraka finds is a possible solution until he sees it.

To see the connection more clearly, let us imagine that Ragnar does not immediately marry Kraka. Instead he next specifies that Kraka should first prove her worth in an even more heroic exercise. He asks her to produce a system which will process the documentation approvals and rejections for which he currently employs a whole team. At the point of specification, Ragnar has little conception of how that task might be achieved without the need for a team. Much as with his previous riddle, Ragnar does not know it could be possible to process the documentations approval and rejections without a team. He needs to be shown the solution before he can know whether it does what he currently calls 'documentation approval and rejection.'

It is important that in both cases Kraka has to take Ragnar into a kind of unknown territory. Ragnar is not sure of what things he would call 'neither dressed nor undressed.' Similarly, Ragnar is not sure what he will call 'documentation approval and rejection' without a team performing the approvals/rejections and Kraka has to understand these expressions well enough to come up with something that fits well enough that Ragnar will call it a solution.

The basic uncertainty that goes along with software projects, then, is that we do not fully understand what the solution to our problems will be until we have a completed solution. If all goes well then one sees the solution take clearer and clearer shape as one goes through the project. This is the key difference between software projects and manufacturing projects – with a manufacturing project one can form a relatively clear picture very early on of the product to be produced and then it can be precisely drawn up and physically constructed. In a manufacturing project one has the possibility of forming a clear picture early on because the product to be produced is a physical thing. A software product is abstract and the solutions to software problems are conceptual. (Software programs may be physically instantiated but this does not take away from their conceptual nature.)

I do not claim that the model of software development as problem-solving is a perfect analogy in all respects. Most software projects are much larger and more complex than what we normally call problem-solving exercises. Furthermore, software projects sometimes change direction mid-course in a way that we would not expect problem-solving exercises to do (as though Ragnar were to change his mind about his riddle in response to changing business conditions). But I do wish to show that it is a useful analogy for certain purposes. The analogy not only reflects that software projects can be subject to wholesale revision but also gives us a perspective on why this is - namely, that we do not fully understand what the solution to our problems will be until we have a completed solution. The analogy can also point to and illuminate the feature of software production that design and development can be difficult to sharply separate. More than this, I will argue that the analogy can be seen to be enlightening with regard to the role of many of the key software lifecycle terms and that seeing this will cast further light on the ways in which software production can be subject to wholesale revision.

4. THE SOFTWARE LIFECYCLE AND PROBLEM-SOLVING STAGES

The point that design and development cannot be sharply separated is often made with regard to the inadequacy of a manufacturing metaphor (see the Fowler remark cited before). I think that this point can be extended to many of the key terms for phases of the software lifecycle:

> Software requirements are not like requirements for a physical object or structure.
>
> A software design is not like a design for a physical object or structure.
>
> Implementing or building a software product is not like building a physical structure, nor like following a design for a physical structure.
>
> Software testing is not like testing a bridge.

The problem-solving metaphor will be enlightening with regard to the phases of the software lifecycle if it can show us what is wrong with looking at those phases in terms of a manufacturing analogy. Seeing this will help to make the value of the metaphor clearer and point the way to how it can be applied to real situations.

4.1 Software Requirement as Problem-Probing

If one is drawing up requirements for a physical product or structure, those requirements are by and large much easier to understand from the beginning than is the case with a software project. For example, if one wants to build a bridge then it is fairly clear from the beginning what it means to say that it needs to take a load of X many cars each weighing so much. Software requirements are much more likely to be hard to understand and sometimes software requirements can look like they are perfectly clear but turn out to be unclear in the light of later developments.

Let us go back to Kraka and Ragnar and imagine that Kraka arrives accompanied by a rabbit instead of a dog. Ragnar might say that he will not accept this as a solution – he thinks that arriving with a rabbit does not count as being accompanied at all. If this seems implausible or unfair then perhaps imagine that Kraka arrives with a goldfish instead. It is plausible that we can find circumstances where Kraka and Ragnar will take different views of what counts as being 'accompanied.' But it is Ragnar's view that counts since he set the problem.

A parallel of the dispute between Kraka and Ragnar takes place on software projects with developers sometimes taking a different view of what fits the problem to what end-users will find acceptable. Kraka could feasibly have sought to forestall such an issue by probing the problem further from the beginning – she might have done an equivalent of requirements analysis. We can imagine Kraka putting a business analyst hat on and asking questions like 'Do you mean accompanied by a human being? What about a pet?' But even after much requirements analysis these types of ambiguities can remain. Even if it does become clear that Ragnar would accept a pet, it may not be made clear (not even to him until he sees it) that he would not accept a goldfish.

4.2 Software Design as Solution-Sketching

A design for a software product need not be like a design for a physical object. Once one has the design for a physical object, all of the questions about how the object will be structured are taken to be answered. This is not so with software design – a software design will typically only answer certain kinds of questions and leave others open.

Certain kinds of logical problems are typically approached by mapping out the relevant cases and outlining the approach to be taken for each case before pressing ahead. Certain problems in mathematics have this complex character. Mathematicians sometimes talk of a 'proof-sketch' as opposed to a full proof, with the proof-sketch only giving the key steps and perhaps some non-mathematical notes (suggestions made in natural language rather than in the formalization) on how the steps might be linked up mathematically. We might think of a software design as rather like a proof-sketch. It gives an initial picture of the various strands of the problems and the ways to tackle each strand.

Many tools have been developed to try to standardize approaches to software design and one might certainly characterize the design stage of a software project by the types of artifacts produced. But this does not defeat the basic point that these artifacts are used more like a proof-sketch than like a design for a physical object. Fowler makes this point:

> For many, this is the role of design notations such as the UML. If we can make all the significant decisions using the UML, we can build a construction plan and then hand these designs off to coders as a construction activity.
>
> But here lies the crucial question. Can you get a design that is capable of turning the coding into a predictable construction activity? ... Even skilled designers, such as I consider myself to be, are often surprised when we turn such a design into software. [5]

4.3 Implementation as Fleshing Out a Sketch

Implementing a software design is not like constructing a physical structure from a plan and set of instructions. Once one has the instructions and plan for a physical structure then all of the conceptual questions about how to build the structure are already settled. Software implementation requires putting together sequences of machine-executable statements so that the goals of the project can be realised. The design cannot possibly say what the right statements will be (since then the design would have already implemented the project). The software design can only point to approaches that developers are expected to be familiar with from their experience in the industry and knowledge of the relevant programming language. This is rather like fleshing out a proof-sketch into a full proof in that it requires exercise of creativity throughout the process.

We might say that software designs are typically less specific than proof-sketches in that it is far more likely that a developer will find a design to be unworkable than it is that a mathematician should find a proof-sketch to be flawed (since the latter are subjected to greater standards of rigour). Nonetheless both proof-sketches and designs do sometimes turn out to be flawed.

To carry the analogy further, sometimes it is possible to show that a certain proof-sketch cannot lead to the intended result. Mathematical arguments can sometimes be developed to show that a certain line of thinking cannot lead to the intended result – perhaps there was a contradiction in the proof-sketch that had been overlooked. Likewise, sometimes one can produce a good argument to show that a design will not work. But in either case it might be difficult to flesh out the sketch and equally difficult to show that the sketch cannot work.

This is why the developers in the hypothetical scenario presented earlier in this article might well be foolish to try to determine whether their inability to produce the intended solution was because of a flaw in the design or due to their own inadequacies. Sometimes it is very difficult to tell.

4.4 Software Testing as Solution-Checking

Testing software is not like testing physical objects. When testing a bridge, one knows what it's supposed to do. It was already planned out from the early stages of the project as to what load it should be able to take and what weather conditions it should withstand etc. One can easily conceive of how to test the bridge before one sees it. In order to conceive of how to fully test a software product, one needs to have a fairly detailed picture of how it will work. (This is not to say that some testing could not be conceived early on. It is just to say that it is much more difficult to see how all testing could be done from the early stages of a software project than from the early stages of a manufacturing or construction project.) Software testing is better seen as a way of gauging the satisfaction of the solution against the initial requirements. It is like ticking off each of the parts of a solution against the original specification of a problem. For Ragnar's original riddle, we might imagine Kraka considering her solution and saying to herself something like 'neither dressed nor undressed – check; nether hungry nor full – check; neither alone nor in company – check.'

5. APPLYING THE METAPHOR

5.1 The Problem-Solving Metaphor and Choosing Methodologies

The purpose of working the problem-solving metaphor out in such detail and relating it so specifically to software production has been to show that the metaphor can point to many subtle aspects of the software production process. Because of this the metaphor has the potential to be useful for particular situations where the nature of the software process is called into question. One such situation arises when determining which methodology to employ for a new project. Given what has been said about separation of phases, it should be clear that the problem-solving metaphor has something to offer when it comes to such decisions.

For many companies the question about which methodology to employ for a particular project is likely to centre on whether the methodology chosen should be agile or waterfall. It is often thought that the decision can be made based upon how subject the requirements are to change. Agile criticisms of waterfall are often put in terms of how 'defined' or otherwise the software production process is expected to be. Many might read this passage from Schwaber and take it to be making just this point about the possibility for revision in project requirements:

> In a nutshell, there are two major approaches to controlling any process. The 'defined' process control model requires that every piece of work be completely understood. Given a well-defined set of inputs, the same outputs are generated every time. A defined process can be started and allowed to run until completion, with the same results every time. Tunde [Babatunde 'Tunde' Ogunnaike, co-author of [5]] said the methodologies that I showed him attempted to use the defined model, but none of the processes or tasks were defined in enough detail to provide repeatability and predictability. Tunde said my business was an intellectually intensive business that required too much thinking and creativity to be a good candidate for the defined approach. He theorized that my industry's application of the defined methodologies must have resulted in a lot of surprises, loss of control, and incomplete or just wrong products. He was particularly amused that the tasks were linked together with dependencies, as though they could predictably start and finish just like a well defined industrial process. [6]

One could even imagine a project manager and a team lead at a company with this text in front of them, trying to decide whether to use agile for a particular project. We can imagine the project manager saying that the project requirements have been fleshed out clearly and that the client has promised no changes. We might also imagine the team lead accepting this point and still insisting that agile would be a better fit for the particular project. If the team lead wanted to make this point, he would do well to point to Schwaber's remark that software production can require "too much thinking and creativity to be a good candidate for the defined approach."

This is certainly not to say that agile methodologies are always preferable to waterfall methodologies. The point is rather that agile is preferable in circumstances where revision is likely to be encountered. These circumstances are not just circumstances when requirements change – they are circumstances when the problems by the project are hard problems. It is likely that revisions are going to be necessary whenever a problem is too difficult or complex to see from the start an approach to solving the problem that is clear and detailed enough that it can be used to give estimates that will control the costs and timelines for the project (or even just the next phase of the project).

The problem-solving analogy can be a useful reminder, in this imaginary case to the team lead, that hard problems tend to lead to revisions. Most team leads in are aware of this already but it can be difficult to be clear on the point and there persists a popular understanding that agile is preferable to waterfall when requirements are subject to change. This may be because requirements-change can sometimes be a symptom of the problem being subtle. Consider the example given previously with the types of pet that Ragnar might count as accompaniment. Requirements-change might be an indicator of seeing the problem better. It can be very tempting to look back on projects and think that they were only hard because the requirements kept changing, when in fact the requirements may have been changing because the problems were hard.

The problem-solving analogy brings out the important and subtle feature of software projects that one can rarely be entirely sure that a chosen approach will work until it has been seen to work. For an unfamiliar and/or difficult problem, there can be a strong temptation to assimilate the problem to a familiar one and so follow an approach which does not suit the problem. The analogy suggests that the hardest thing about software is getting a clear view of what the problems are without assimilating them to something more familiar or simpler. This comes out most clearly in the process of estimation.

5.2 The Problem-Solving Metaphor and Software Project Estimation

It is not easy to see a problem for what it is and develop, right from the start of a project, a clear view of what will be involved in approaching the problem and seeing the approach through. And yet this is the task that one faces in estimating for a software project. Sometimes it can be difficult to explain why this is such a difficult task and this is because sometimes questions are raised in regard to estimation from a perspective which neglects the subtle aspects of software production that the problem-solving metaphor is meant to bring out.

Imagine that a software consultancy has the chance to win a large contract with an important new client and needs to estimate the project. The work relates to an area of business in which they have never worked before and they will need to work with tools which the development team have never worked with before. The development lead has been asked to provide a fixed-price quote and only has a week to do so as the client has stipulated a very short deadline. Further, the client has suggested that a lot more work could be available if the quote is reasonable and the project is delivered successfully.

Given the way that I describe this scenario, the risks of working in a new area with new tools are quite clear. Let us imagine that the development lead is also very aware of this and finds himself struggling to come up with any sensible estimate, unable to fully comprehend what it is that the business wants an automated solution for and equally unable to see how long it will take his team to even become proficient in the new tools, let alone how to bring them to bear on the problem. Let's say that he tries to explain this to management and management simply fails to see his predicament. Some of the sorts of things management might say, for example, could be:

> Is this not just a document system? We've done document systems before, haven't we?
>
> Surely you can estimate based upon the document provided – why not just run it through a mechanical estimation technique?
>
> How long do the courses for these tools take? Surely we can put the guys through a course and then they should know all they need to know or else we should get new developers.

This is of course a difficult situation for the team lead but what makes the situation difficult is not just its politics. Part of what makes it difficult is management is approaching the matter from a perspective which questions (and embodies simplifying assumptions about) what is even involved in producing software. Whilst this particular situation might be thought to be rather extreme and unusual, the sorts of questions being asked are not unprecedented and giving enlightening answers to these questions is not an easy thing to do.

The suggestion of this article is that the team lead will find the questions easier to answer, and his management will find his answers easier to understand, if software production is viewed as a kind of complicated exercise in problem-solving. Then it should be easier to see that whilst various problems may have certain features in common, all problems are not alike (indeed, all document systems are not alike) and the subtle differences can sometimes be very important. Whilst mechanical estimation methods have their advantages, they can easily miss the relevant subtleties and are no substitute for attempting to think a difficult problem through. Whilst everybody working in software has skills and experience that they can bring to a problem, sometimes the difficult thing about a problem is seeing the respects in which it is not like what one has done before (knowing a programming language, for instance, does not guarantee being able to do everything that can be done in that language any more than speaking English guarantees being able to write prize-winning poetry).

6. CONCLUSION

As I have emphasized, no analogy can on its own resolve the problems that come up in software development. Analogies always have to be related to particular circumstances in order to be useful. Nonetheless, issues to do arise in software production where we might be inclined to make simplifying assumptions about the problems at hand and how they relate to software production as a whole. Sometimes it can be very tempting (particularly for developers) to assimilate complex or unfamiliar software problems to simpler or known ones. Sometimes it can also be tempting (particularly for project managers) to assimilate software production as a whole to a more 'defined' and controllable process like manufacturing. The problem-solving metaphor that I have presented is intended to point towards the features of software production that are most subtle and for which simplification is both most likely and hardest to spot. The examples of using the metaphor to remind us of these subtle features were chosen because they represent cases where large decisions are made that involve seeing features of a project that span the software production process. The test of the metaphor lies in how well it can serve this function across a wider range of situations.

7. REFERENCES

[1] Fowler, M. 2004. http://martinfowler.com/bliki/MetaphoricQuestioning.html.

[2] McConnell, S. 2004. *Code Complete, Second Edition.* Microsoft Press Redmond. 5. DOI= http://dl.acm.org/citation.cfm?id=1096143.

[3] Reynolds, J. 2008. Some thoughts on teaching programming and programming languages. SIGPLAN_Notices 43, 11 (November 2008), 108. DOI= http://dx.doi.org/10.1145/1480828.1480852.

[4] Fowler, M. 2005. http://martinfowler.com/articles/newMethodology.html.

[5] Oggunnaike, B. and Ray, W. 1994. *Process Dynamics, Modelling and Control*. Oxford University Press.

[6] Schwaber, K. and Beedle, M. 2001. *Agile Software Development with SCRUM*. Prentice Hall. 24-25.