Do Methodologies Matter?

Peter Brook

School of Information Technology and Electrotechnology
Otago Polytechnic
Dunedin, NZ
peterb@tekotago.ac.nz

ABSTRACT

This paper contrasts the classroom approach to designing a system that involves both hardware and software with an actual commercial product. The team found that human factors kept the group productive in spite of changing requirements, lose role definitions and complex networking. Violations of the normal restatement of the system development life cycle can be tolerated so long as there are stron, social mitigating factors. Such violations in the 1970's lead to the microcomputer revolution.

Keywords

Software engineering, methodology, team dynamics, complexity,

Given any hardware or software project, the academic community asserts that you should follow some methodology that will guide that project to a successful outcome. The methodology should be generally accepted, defensible and appropriate for the project.

The author was involved in a complex commercial project as well as teaching about aspects of IT including doing things correctly when confronted with a computing task. This allowed reflection on the two worlds and constructive criticism of some methodological precepts as well as confirmation of the veracity of various heuristic rules. This paper looks at some of our experiences and draws conclusions, now that the bulk of the project is finished, on how flexible one should be in applying such rules. The number of projects that have strictly adhered to some pure model seems to be very small (McConnell 1996). Even a simple principle such as sticking closely to the original requirements of a project is broken in 25% of cases found by Jones (1994).

The project was for a client in the food industry who wanted many quality meals available in a just in time fashion upon insertion of coins into a validator that opened one of forty doors on demand. All meals had to be timed for health regulation reasons, messages were to be delivered to LCDs and all transactions had to be recorded for future analysis. We constructed an RS485 network that talked to PCs and meal containers. A master processor box relayed information from the cook to each container which was, in turn, controlled by an AVR processor with various devices hanging off it. Any part of this project could be regarded as complex and the total network which included TCP/IP, RS232, (with both 12 and 5 volt variations), RS485 and ccTalk protocols, could be regarded as very complex as a whole (Figure 1).

Strong planning was required and an excellent team assembled. But in almost every way at some time during the project's one year life cycle, the standard SDLC sequence was violated and most of the rules taught to tertiary students about good methodology were broken, with varying consequences.

So what happens when the elements of the SDLC are violated? This is not a pure environment and there are real world examples abounding where teams did the right thing by textbook development strategies and still the project did not work well. Three major classics are the Denver airport baggage handling system, the Arianne rocket failure and the Incis fiasco of the New Zealand Police. We should not give up a methodology just because it doesn't always work, rather because there is a much truth and efficacy in it. Only in children's cartoons are methods that fail to work once abandoned forever. We use methodologies not because they are necessary or sufficient to adopt for project success but because their adherence brings in some discipline and rules of thumb whose utility has been inductively tested. So the violation of one of the SDLC principles may not lead to non-completion of the project and even be an advantage if supplanted with an equally sound principle, how ever casually inserted.

Figure 1: The author amidst a tangle of software and hardware that looks like it cries out for some disciplined methodology.

To take a specific example, we usually teach that a software/hardware task has to be completely specified before embarking on the development. Our minimal specification was soon completed but we gradually took over more and more of the "rest" of the IT part of the project until the new modules easily outnumbered the original sparse chain of boxes that constituted the starting backbone. The option to refuse to be a participant in feature creep may have lead to the cancellation of the project so for practical reasons, we absorbed the new requirements and moved on. Complexity of hardware modules increases as the square of their number and the addition of a few extra components imposes a burden on the designers that transcends their unit effort. So on the one hand the theory says: do not add new features after commencement. On the other hand we have a picture of a client saying: "we're paying for this new feature, please add it in".

The closest life cycle model to that which describes our recent project is the Evolutionary Delivery Model after Glib (1988) which provides some ability to change a product direction mid-course in response to a customer request. The risks are

(1) Wasted time in supplanted features

(2) Diminished project control

(3) Difficult scheduling and budgeting

(4) Feature creep

In the end the customer will get what he wants and prototyping became early versions of an emerging product.

The success of our project in the face of feature add-on and intuitive, informal methodology was due to the development team's focus and integration. Good personal relationships have been sustained through some difficult times. It was difficult to say at times who was on the team and who was not as developers came and went even though the same few core people stayed on for the duration. In a real world project the team is not always a well defined entity. With us, people came and went and added expertise and advice in areas not necessarily associated with their initial domain of interest.

Boehm (1981) found that there was a four fold productivity increase possible in in the best teams compared to the worst. Productivity of our team is hard to quatify even though anecdotally members have noticed improvements under certain circumstances. Once a metric was imposed on productivity it would be hard to do anything about a low rate before the end of the project under some methodologies where the results at the end are what are counted. Again, anecdotally, team members asserted that each project was different and it was hard to extract permanent factors of high productivity independent of the nature of the project.

Our team members asserted that group cohesion, enhanced by social factors, made the work more enjoyable and focussed. Lakhanpal (1993) stated that, indeed, group cohesion contributed more to the success than the individual talents and experiences of the team members. Some characteristics of high performance teams have been asserted in the literature (Larson and LaFasto 1989) and may include:

(1) a shared, elevating vision or goal

(2) a high level of enjoyment

(3) a commitment to the team

(4) a sense of autonomy

(5) a sense of team identity

(6) mutual trust

(7) interdependence among team members

(8) effective communication

(9) a results driven structure

In an informal survey of our team we agreed that the first three items were the most important for us.

The trouble-shooting abilities of each team member was valued by the rest of the team. People who were working on their own aspect of the project tended to involve others most during the problem-solving times. The verbal analysis of problems, especially in hardware where intermittent faults can occur, is highly valued by the team members as are the articulate stating of hunches leading to solutions. Such attributes are seen by the group as being more important than to the adherence to an abstract methodology.

The size of the team changed. As the team numbers ebbed and flowed the activities of different members changed. Conversations had to be repeated with several members and formal meetings arranged. These were not always popular as some thought they would rather be engaged with the work at hand rather than talking about it. There are nCr two-person contacts in a team. This is manageable in a team of four with only six two-person conversations theoretically required. However, for ten people there are a maximum of 45 two person meetings making informal communications difficult. Communication complexity, like module creep complexity, is non-linear.

There were several other shibboleths of the systems world violated or only partially adhered to.

♦ Feature scrubbing was considered and enacted eclectically by both the designers and the clients.

♦ Tools, mostly assemblers, compilers, RAD programmes, terminal programs were picked by the individual members and not considered as a suite. Nor was there any attempt to drive the software towards any unified theme like open source.

♦ Group members had fuzzy roles and most could do parts of other people's jobs. Consensus drove most design decisions.

Many new hardware and software products have been produced by teams of semi-professionals with the knowledge of cutting edge software and new IC's. This is likely to become a popular development team compostion in the medium term for small projects. These teams will be breaking new ground using modern chips and generally emulating, in microprocessor hardware, the microcomputer software revolution of their fathers in the 70's. It is significant that the revolution that provided the Apple II and Visicalc was driven mostly by academics who adopted strange informal methodologies.

At the time of writing we can almost call the project a success. The fact that such a complex job could be done despite a strong methodological underpinning is due to the persistence and coherence of the design group.

REFERENCES

Boehm, Barry W. (1981) Software Engineering Economics. Englewood Cliffs, N.J.: Prentice Hall.

Glib, Tom. (1988) Principles of Software Engineering Management. Wokingham, England: Addison-Wesley, 1988. Chapter 7

Jones, Capers. (1994) Assessment and Control of Software rRisks. Englewood Cliffs, N.J.: Yourdon Press,1994. Chapter 9.

Lakhanpal, B (1993). Understanding the factors influencing the performance of software development groups: An exploratory group-level analysis. In formation and Software Technology, 35 (8) : 468-473

Larson, Carle E., and Frank M. J. LaFasto. (1989). Teamwork. What must go right; what can go wrong. Newbury Park. Calif.: Sage.

McConnell, Steve. (1996) Rapid Development: Taming Wild Software Schedules. Redmond, Wa.: Microsoft Press, 1996. p331