Software Engineering Practices that Count

A case study of the development of the ACT electronic voting and counting system: its successes and lessons learned

By Clive Boughton

The scene

Imagine being presented with the "opportunity" to undertake a software development project that:

- has an immutable deadline (seven months out),
- is most likely under-budgeted,
- possesses functionality that cannot be easily reduced,
- may have serious political and reputational consequences if it fails,
- must deliver a product that operates with a very high degree of integrity, and
- can be cancelled at any time by the customer.

Software Improvements faced this situation when the prime contractor (Linuxcare) of the successful joint bid for the Australian Capital Territory (ACT) Electoral Commission's *Electronic Voting and Vote Counting System* became an unfortunate victim of a take-over during contract negotiations.

The Software Improvements eVACS® Project Team then formed used common sense Software Engineering decisions and practices to turn a very risky project into an unmitigated success. The clear message I wish to convey to all other small, software development companies is:

> *Don't go with the status quo and forsake professional practices, especially when the going gets tough!*

The project in brief

No detailed description of eVACS® is provided here. Available from Elections ACT (the Customer) is information on the ACT Legislative Assembly election system[1] (http://www.elections.act.gov.au/Elecvote.html) together with specifics about the Hare-Clark proportional representation scheme that is used to count voters' preferences[2] (http://www.elections.act.gov.au/hare.html). A post-project report[3] (include website details and link) is available on the Software Improvements website.

The broad requirement for the eVACS® project was to construct a trial system on standard PCs that could both collect and count votes electronically with no less security, no impingement of voters' rights, and no less anonymity than is available with the current paper-based system. Internet solutions were not acceptable because of the possibility of voter coercion and system tampering.

The project was to start on March 1, 2001 and be completed by the beginning of September 2001 for full "customer" testing and preparation for pre-poll voting that began on September 30. Election day was October 20, 2001.

The Software Improvements Team was responsible for the construction of the software, whilst InTact (the ACT Government IT provider) provided the hardware.

The essence of the solution implemented by Software Improvements is available in the post-election report mentioned earlier.

Why do projects succeed or fail?

There is rarely just one specific reason for project success or failure. Also, what might work well for one project may not work at all for another. Although the Standish Group CHAOS report of 1996[4] (http://www.standishgroup.com/sample_research/chaos_1994_1.php) paints a sad picture of the software industry at the big end of town, it is interesting to take a look at the identified reasons for both success and failure of such software development projects. Success means within budget, schedule and all agreed requirements are implemented and accepted by the customer. The top ten reasons given for successful and failed projects include (in priority order):

Reasons for success	***Reasons for failure***
User involvement	*Lack of user input*
Management support	*Incomplete requirements & specifications*
Clear statement of requirements	*Changing requirements & specifications*
Proper planning	*Lack of executive support*
Realistic expectations	*Technology incompetence*
Smaller milestones	*Lack of resources*
Competent staff	*Unrealistic expectations*
Ownership	*Unclear objectives*
Clear vision and objectives	*Unrealistic time frames*
Hard working and focussed staff	*New technology*

Do these same reasons for success or failure apply to very small projects? I think they do. On thinking about the various decisions and actions that were taken by the Software Improvements Team on the eVACS® project, and the reasons for them, I can see definite parallels. Fortuitous? I don’t think so. Most of the reasons for success and failure are intuitive, or common-sense, and it doesn’t require rocket science to act to prevent a reason for failure becoming a reality. However, a software development company or team cannot act confidently if the overall software engineering knowledge base within the company or team does not extend beyond coding. Continuous learning is required.

What decisions or actions, lead to success?

When Linuxcare dropped out of the picture, negotiations started with Software Improvements, and so, the success-oriented decisions began to be made.

The following sections constitute a relatively brief synopsis of the various decisions and actions, and the reasons behind them. Each section (negotiation, system analysis, software analysis, software design, software coding and software testing) represents a project phase describing the main decisions and actions that affected the success of the project. All except the negotiation phase were determined from the planning that occurred during negotiations.

Negotiation

Decision/action	*Reason(s)*
Construct a project management plan (PMP).	• *No proper plan existed.* • *Some objectives were vague.* • *Some expectations of Customer were unrealistic given budget and time.* • *Confirm project feasibility (or not).* • *Identify and treat risks.*
Construct a quality management plan (QMP).	• *eVACS had to operate with high integrity.*
Build a closer working relationship with customer.	• *Discover Customer's level of ownership.* • *Enhance the 'team'.*

The PMP and QMP together described the full scope of the system and software-related activities of the eVACS® project. The project was indeed feasible, even with a built in 6 week delay caused by extended negotiations of the contract. The project life cycle was a simple waterfall model the phases for which head up the following sections. The PMP included a basic software configuration management plan (SCMP) and a simple software change control plan (SCCP). The deliverables included, system and software requirements specifications, software design specifications, source code, acceptance tests and (of course) the fully installable system with instructions. The schedule for software delivery was based on an average estimate of 20,000 source lines of code to be developed in a 'C'-like language. All known risks were identified. Extra specific resources were required.

Now the Software Improvements Team and the Customer had something with which to gauge progress.

It was during this phase that Software Improvements came to know the Customer better and vice versa. A reference group, including politicians, previous electoral commissioners, typical users and a representative of the visually impaired, had been set up by the ACT Electoral Commission. That group posed questions and raised issues. The Commissioner could cancel the project at any time if there were too much negative press or political backlash. The Customer thus set the criteria for success. This input further helped to define the process for achieving a successful outcome.

System Analysis

Decision or action	*Reason(s)*
Construct an eVACS system specification (SS).	• *Requirements as stated in the Tender document had changed.* • *Hardware components proposed in the tender response were not finalised.* • *InTact needed to know all hardware specifications as soon as possible.* • *identify who had ownership of what.*

The SS represented an edited version of the relevant portions of the original tender document. It now contained clear system objectives, descriptions of all required hardware, and overall system functions that needed to be developed in software. It included the provision of a prototype for the voting and counting parts of the system.

At least now all parties involved in constructing eVACS® were clear on overall system requirements. Some ownership was beginning to occur.

Software Analysis

Decision or action	***Reason(s)***
Nail down software requirements.	• *Some requirements were vague.* • *Requirements didn't seem complete.* • *The customer had not sanctioned fully the proposals in the tender response.* • *Setting and confirming expectations.*
Construct an eVACS software specification in the form of a model.	• *Capture requirements more formally.* • *A model provides an easy way to identify the affects of proposed changes.*
Identify separation of concerns within the model.	• *Smaller project milestones.* • *Less time between deliveries.*
Provide a protype.	• *Show customer and users the nature of computer interfaces.* • *Obtain user involvement.*

It was no surprise to hear the Customer express great pleasure and relief when approached by Software Improvements to discuss the requirements. A full scenario analysis was performed by identifying required *actions* of the system as produced by expected stimuli (*events*). The Customer had no trouble at all with the concept. In addition to identifying *event/action* pairs, system *state* at the time of the event, and any *necessary action data* were also identified. Stick figures and named ovals were not used. Sorry Ivar!

The scenario document had begun with analyses of the tender and tender response, producing a 9-page table containing 41 requirements. After two days with the Customer this document expanded into a 39-page table containing 79 requirements.

The benefits were obvious. The software functionality was now fully scoped and therefore a better estimate of required effort could be made. Acceptance tests could now easily be constructed. The Customer now realised the extent of the work being undertaken by the Software Improvements Team and thus some prioritisation of requirements was quite acceptable.

The requirements were modelled using a real-time structured analysis approach, in line with eVACS® being a very event and process driven system.

The model provided a clear picture of the operational scope of eVACS® as well as clear definitions of data input and output. Areas of re-use and also areas where functionality could be reduced without ill affect were easily identified. For example, there was obvious re-use of functionality between the voting part of the system and the paper vote data entry part of the system. Also, a clear reduction in functionality within the data entry part of the system was gained by reducing the size of paper vote batches from 500 down to 50.

Acceptance testing was enhanced with the model because data input and output ranges were well defined.

The modelling proceeded incrementally with separation of the system into four components: Election Setup, Voting, Data Entry, and Counting. Each of the component models was incorporated into separate software requirements specifications. The final combined set of four SRS documents consisted of 158 pages including, 34 data flow and/or state transition diagrams, written process specifications, an extensive data dictionary and a requirements traceability matrix.

The model provided the basis for a more accurate estimate of size and effort. The new estimate of size didn't vary much from the previous one, but the effort estimate was lower.

The SRS documentation provided all that was necessary to construct a design.

Software Design

Decision or action	***Reason(s)***
Construct architectural design of software.	• *Allow planning of coding effort.* • *Provide a visualisation of the structure of the software to all parties.* • *Focus the planning and implementation of integration and system testing.*
Provide detailed design specification.	• *Give hired programmers an easy way to understand what they are required to do.* • *Form basis for unit tests.*

The design came together relatively quickly, mainly because the team had worked closely together and captured a very similar visualisation and understanding of the system.

The identified reductions in functionality and areas for re-use (mentioned earlier) were included in the design so that programmers were not duplicating effort.

Each design module was specified using a standard format in pseudo-code with pre- and post-conditions described wherever possible. Thus a programmer who had to code up the module was presented with a contract for coding and testing.

The hired programmers (one of whom was very experienced) could not believe that they could be productive within 4 hours of joining the Software Improvements Team.

The architectural part of the software design document (SDD) contained 31 pages including 10 module specifications, whilst the detailed part of the document (SDDD) contained 306 pages including 47 diagrams and around 327 module specifications.

Software Coding

Decision or action	*Reason(s)*
Code according to SDD and SDDD specifications.	• *The design describes how the requirements should be implemented.* • *The design is also the plan for building the code.*
Perform code reviews and inspections.	• *Ensure consistent coding practices.* • *Sharing of knowledge and issues.*

The main coding and testing effort took seven weeks of elapsed time. Less than planned. Review meetings were held every other day and were intensive sessions to ensure that everyone was progressing, cooperating, and sharing problems and solutions. Each coder was aware of each other coder's productivity and so work was distributed accordingly.

The code was audited, both before and after the election, for possible insecurities and any hidden attempts to alter the outcome of the election. The auditor found no insecurities or hidden schemes but (before the election) did expose nine faults. Five were regarded as serious enough to possibly cause instabilities, but otherwise would not affect the outcome of the election result. Four faults were regarded as minor or cosmetic.

Toward the completion of the coding phase the Customer informed the Team that the facility for the sight-impaired 'had to be' included. The Team had not included the facility to date because there had not been enough time. However it became a political must. To cut a long story short, the Team and the Customer together implemented the sight-impaired facility. If there were no *ownership* on the part of both parties then the cooperative effort would not have occurred.

There were 21,810 non-blank, non-comment source lines of 'C' code in the final eVACS®. The source was distributed among 182 ".c" files together with 80 ".h" files.

Software Testing

Decision/action	*Reason(s)*
Write acceptance test cases & procedures for the customer in accordance with the SRS.	• *The tight schedule.* • *Ill-conceived tests from the customer.* • *Shared ownership and expectation.*

The well-structured design meant that unit and integration tests could be written, in accordance with the design, as the coding progressed. The Customer reviewed the acceptance tests (some of which were quite complex) and any errors were corrected.

The human side of the project

Most of the Software Improvements Team possessed information technology, computer science or software engineering qualifications together with several years experience. They all possessed strengths and weaknesses in their profession and these were acknowledged and accepted by all. The Team was truly a group of professionals and without their wise planning and decision making the project would almost certainly have failed.

A true test of the professional nature of the Team was had with several very unfortunate and one tragic episode during the course of the project. The project lead's life-partner was hospitalised for several weeks with an unknown liver disorder and the diagnosis was fatalistic. The wife of another major member of the team fell in a car park and splintered both elbows, making it virtually impossible for her to hold and feed their young child of 5 months. The elderly mother of the quality assurance manager died and he had to go to the UK to make funeral arrangements and arrange for his elderly father to be put into proper care. Another Software Improvements employee with three young children lost her husband in tragic circumstances. My own mother died of cancer on September 11. Through all this the Team remained a team.

Epilogue

The elapsed time of the project was 135 working days containing an overall effort of 334 person days. The hardest-worked Software Improvements employee averaged 45 hours per week during the project period. Contracted people were typically hired for a maximum number of hours within a defined period and very rarely did more than an 8 hour day.

The effort distribution over the project was as follows:

- 11% - prototype.
- 27% - requirements (scenario analysis, modelling and design).
- 38% - coding, unit, integration and system testing.
- 24% - acceptance testing and deployment.

The implemented eVACS® did what it was required to do at the October 2001 ACT Legislative Assembly elections. By the time the full count had been completed the ACT Electoral Commission had proved that counting by hand was too inaccurate, especially for close-run results – as occurred in this election.

No voter who used the system complained, even when the barcodes used to start and finish a voting session sometimes had to be swiped many times before the barcode reader accepted the input. Voters often went out of their way to vote electronically and then often had to line up in long queues.

eVACS® went slightly over-budget, but Software Improvements does have the IP.

Since project completion the ACT Electoral Commission has promoted both eVACS® and Software Improvements several times. The Commission has also contracted Software Improvements to undertake modifications to eVACS®. The planning and documentation processes followed during the development of eVACS® were necessary inputs to the enhancements and have been continued for the enhancements.

Conclusion

How you address the common reasons for success and failure (as reported by the Standish Group) seems very important. It's not appropriate to be aware of these various reasons and then use ill thought-out actions to try and support success or counteract failure. Software development companies and teams need to possess software engineering knowledge and capability. If the team cannot put a plan together, and/or doesn't understand how to clarify or elicit requirements, and/or doesn't know how to construct models or designs, and/or doesn't know how to plan and construct tests, and/or doesn't know how to manage a project then they are not going to be able to make reasonable decisions that positively promote success.

The ACT Electoral Commission has received a well defined, well documented, tested, and maintainable system. I'd hate to think what might have been produced if the Software Improvements Team had undertaken the usual panic approach of "code, code, code for there isn't enough time to do that other stuff".

In addition to the Standish Group of listed reasons for success, I believe that the following are also very important:

- *Make sure that the customer's objectives are always being met.*
- *Form a close professional, relationship with the customer.*
- *Make sure the team is a team.*
- *Construct clear specifications at every phase of the project.*

References

1. ACT election system: http://www.elections.act.gov.au/Elecvote.html
2. Hare-Clark counting system: http://www.elections.act.gov.au/hare.html
3. Post-project report: (include correct website details and link).
4. Standish report: http://www.standishgroup.com/sample_research/chaos_1994_1.php

Note

This article is an updated version of a paper published in the December 2002 edition of "*software*" the journal of Software Engineering Australia, and republished with modifications by the Australian Computer Society in the *Best of Information Age*, June 2004.

Author

Clive Boughton studied Physics at RMIT and later gained his PhD in experimental physics from the Australian National University. He has been involved in industrial and defence applications of software engineering over the last 15 years. He has been providing software engineering training and consulting for over 10 years, and more recently introduced the Bachelor degree and then the Master degree in Software Engineering at the Australian National University. Clive Boughton is a founding Director of Software Improvements Pty Ltd.