

Quality systems by prototyping

M. Blackman^a, M. Jeffreys^b

^a*BRC Partnership Ltd. Bank Chambers, 68a High Street,
Stevenage Old Town, Hertfordshire SG1 3DH UK*

^b*Jeffreys Systems PLC, Jeffreys House, 21 Normandy Street,
Alton, Hampshire GU34 1DD, UK*

1.0 INTRODUCTION

This paper describes a methodology which evolved in the 1970's and 1980's for the development of commercial data processing systems on interactive minicomputers. It was then named Interactive Staged Enhancement (ISE) but it should be seen as a variant of what is now called Evolutionary Prototyping. It worked successfully in a commercial software house environment where meeting cost and time deadlines was essential and where maintainability and modifiability of finished systems was crucial. A description of ISE was first published in Jeffreys [1].

The relatively low cost and speed of prototyping methods, as well as their ability to readily satisfy user needs, is well known, but they are widely criticised for lacking a respectable methodology and for being difficult to quality assure.

In this paper we describe the ISE methodology with special emphasis on quality features. In the original method quality monitoring was the responsibility of the software house manager and the principal user representative, in this enhanced description a separate quality team is proposed to augment this.

This description is inevitably only an outline. For ISE (or a variant) to achieve the recognition and respect of one of the current structured (classical) methodologies, a much more detailed description will be necessary.

386 Software Quality Management

2.0 PROTOTYPING

Prototyping is the name given to a wide variety of techniques used in software development but all characterised by the production of a working version of the software (or main parts of the software) before it has been decided exactly how the software is to work.

There are two main types of prototyping Throw-It-Away Prototyping and Evolutionary Prototyping.

In Throw-It-Away Prototyping, a working version of the system is produced by a small team of skilled programmers to show users and to capture their real requirements before the analysis and design proper is started. Clearly some analysis and design is needed (or the prototype will bear no relation to the users needs), but everything is thrown away, or put on one side, once the prototype is complete and the users express satisfaction that their needs are satisfied. The prototype serves only to clarify to the analyst the users' perceived requirements. Sometimes the prototype is kept to be used to test out changes requested by users during and after proper development. An early description of Throw-It-Away Prototyping is given in Boar [3].

Notice that Throw-It-Away Prototyping is not incompatible with classical software engineering methodologies discussed below, indeed many advocate it as a first step.

Evolutionary Prototyping differs from Throw-It-Away Prototyping only in so far as the prototype is the final system. When the classical analyst is about to start serious analysis, the evolutionary prototyper is involved in final implementation tests. Evolutionary Prototyping must clearly be cheaper than those classical software engineering methods which include Throw-It-Away Prototyping - but how can software quality be achieved or monitored? Questions such as security, performance under load and documentation are all largely irrelevant to the Throw-It-Away method, but are essential to a real live system. We propose one solution.

Notice that the term Prototyping in software engineering differs from its use in other areas of engineering, where a prototype is developed after the main engineering is completed. Here the prototype is used to test the engineering design and to allow for final refinements before production manufacturing.

Hekmatpour and Ince [2], in what is still one of the best reviews of software prototyping methods, propose a third class of prototyping they call Incremental Prototyping. We have excluded that from consideration since it strictly does not come under the definition of prototyping we have given in the first paragraph of this section.

3.0 CLASSICAL SOFTWARE ENGINEERING METHODS

Software engineering methodologies abound. The use of such methodology is currently considered essential for the successful development of quality software systems. They all involve the use of detailed rules (steps) for the analysis and design of the system before the writing of any actual production code. The strong support of the British Government's CCTA for one such methodology - SSADM - has reinforced their widespread acceptance.

They provide a number of advantages:

1. They give a clear body of complex knowledge that can be taught and for which examinations can be set.
2. They provide assured employment for systems analysts who cannot, or can no longer, write compliant code.
3. They provide a systems development route map which can be monitored by management and quality control specialists.
4. They give novice developers a clear set of steps to follow to achieve successful systems development.
5. It is claimed that with the best methodologies, the development team can be changed mid-stream and the new team - if trained in the methodology - can rapidly restart development from where it was stopped.
6. A good methodology can help the systems analyst to understand the real IT needs of an organisation.
7. Because they are rule based and have sequential steps and usually involve elaborate graphical representations of the analysis, Computer Aided Systems Engineering tools have been developed to the financial benefit of their vendors.
8. They provide a systems development "best practice" which can be recommended by consultants and professional bodies.
9. They provide developers with a defence against the consequences of failure. If "best practice" has been used, failure must be due to circumstances outside the developers' control and for which they must be blameless.
10. The best methods have logical as well as physical design. They are therefore future proof against technological change (e.g. much larger "memory files" or distributed databases) and fashion change (e.g. post-relational databases).

388 Software Quality Management

They have one or two drawbacks:

- a. They are expensive and slow.
- b. They do not guarantee success - though their proponents argue that they maximise the chances of success.
- c. Changes in user requirements during or after their use are often difficult or cumbersome to cope with.

4.0 THE ISE METHOD

4.1 The Method in Outline

Three key people are needed:

- a. The User Project Leader (UPL) who is the ultimate project manager and the representative of the user community. He is in a very real way the customer and is responsible for ensuring that the users get satisfaction.
- b. The Analyst/Programmer (AP) who will design, code and implement the system.
- c. The Quality Specialist (QS) who is technically the equal of the AP but is responsible not just for monitoring progress but also for ensuring that all those aspects which are not visible to the UPL in the short term (e.g. security, performance, maintainability, etc.) are provided.

The first step is for the AP and the UPL to produce an Outline System Specification. This is a brief document (two or three pages at most) which defines what the system is to do, at least to a level that a prototype can be produced. Since some sort of user requirement must have been recognised for an AP and UPL to have been appointed in the first place, this may be no more than a clearer rewrite of some earlier document.

The next step depends on the circumstances. In a green field situation, questions such as what hardware, operating system, DBMS, 4GL and overall future systems development strategy may need to be considered. We shall assume these questions have been broadly settled (as is usually the case these days). There is usually an installation or company wide policy on these matters.

Next the AP works out an outline system design and a development plan with dates for both showables (working functionality which can be demonstrated to the UPL and his user community) and deliverables (code which can be considered complete and usable). He also produces cost budgets. The AP's

cost and time budgets are agreed with the UPL (who also has his own contingency budget) and are copied to the QS.

Finally the development proper starts and as soon as possible (a week or so) the AP meets with the UPL and QS to show the first showable actually working.

These AP and UPL meetings occur very regularly with other members of the user community attending and contributing to the evolving software. The QS attends each of these meetings and both the QS and the UPL monitor progress against time and cost budgets. At every stage both the QS and the UPL are responsible for alerting IT management if development progress seems unsatisfactory.

The QS is additionally responsible for ensuring that aspects obscured from the UPL are being addressed (e.g. performance under load, access security, data security after hardware/software failure, maintenance documentation, interfaces with other systems, etc.).

All three team members know the agreed delivery targets and all are responsible for achieving them.

4.2 Selecting The Team

The biggest weakness of most complex human endeavours - and ISE is no exception - is that highly competent people are needed to ensure success.

4.2.1 The User Project Leader

The UPL must be an energetic and respected member of the user community. He should clearly know that the project's success is directly down to him. His responsibilities include:

1. Ensuring the system meets the users' perceived needs.
2. Encouraging users to evaluate the system, to contribute to its development and sometimes to accept somewhat less than they might desire.
3. Monitoring the progress of the AP against both time and cost budgets. He measures progress only by seeing working code.
4. Liaising with the QS.
5. If he becomes worried about progress he must escalate his worries to senior IT management.
6. Planning test data to test and stretch the evolving software.

390 Software Quality Management

7. Planning with the AP how the business will cope if (when) the new system crashes - disaster planning!

The UPL must be in every way a discerning customer, anxious to help his supplier (the AP) but determined to get fair value for money. The UPL will be at most a half time member of the team.

4.2.2 The Analyst Programmer

Ideally the AP will work alone (a Virtuoso Prototyper), but is often supported by a small team. He is expert in the hardware, operating system and software tools being used (DBMS, 4GL, etc.). This will not be the first project for which he has been AP.

He is responsible for:

- a. Designing, developing and implementing the system.
- b. Breaking the development down into a series of stages with showables and deliverables.
- c. Making and keeping to cost and time budgets or, with the agreement of the UPL, ensuring that budgets are changed.
- d. Persuading the user community (through the UPL) to accept reasonable compromises if necessary.
- e. For the implementation of hidden quality (access security, data security, performance, modifiability, maintenance documentation, easy maintainability, compatibility with other systems).
- f. Planning with the UPL how the business will cope when (if) the new system crashes - disaster planning!

4.2.3 The Quality Specialist

The QS has the primary role of monitoring progress and particularly of ensuring that factors hidden from the UPL are catered for.

He will not be a full time member of the team, but from time to time will spend protracted periods testing such things as performance under load and code understandability and modifiability.

4.3 The Outline System Specification

This is a document agreed between the UPL and the AP (and probably senior IT and user management) describing in a few paragraphs the functionality of the system. This is the same document which would need to be produced prior to the development of a Throw-It-Away Prototype.

4.4 The Outline System Design

Before the AP can produce his development plan he must have a vision of the system he is to produce. This is true for either type of prototyping. The great advantage of prototyping is that major design changes can occur during development, but before any development or even planning of any sort can occur the AP must have an overview, a conceptual model, of what he initially intends to produce.

This logically implies that some analysis and design process must occur at this stage, whatever prototyping method is used. By using skilled and experienced people the process is rapid and almost instinctive. Because the essence of prototyping is to allow for change, errors or omissions can be corrected later.

Since an analysis and design process must logically occur, we might call it Throw-It-Away-Structured-Systems-Analysis-and-Design-Method or Throw-It-Away-SSADM!

Implicit in the above argument is the notion that a classical structured analysis and design could occur at this stage. It would of course be slow and costly. For those organisation which could afford it (i.e. those currently committed to SSADM or one of the lookalikes), it would provide all the benefits of careful analysis and design before users really became involved, followed by the use of Evolutionary Prototyping (ISE) with all its benefits of user satisfaction and code modifiability. The only difficulty would be the IT cultural one of Throw-It-Away-SSADM.

4.5 Planning The Development

If the project contains a number of clearly separable elements, it is conceptually broken into those elements at this stage. For example, a comprehensive system for a manufacturer might consist of modules for:-

- order entry / finished goods stock control

- financial ledgers

- raw material purchasing

- production monitoring

392 Software Quality Management

Each one of these sub-systems might be given to a separate analyst programmer (or team) reporting to the AP project leader. Ideally, if time permits, one man does the lot, serially - it is invariably cheaper this way.

Each module is then broken down by the AP in consultation with the UPL into a series of target "showables" and "deliverables" with dates, i.e. dates on which for each key functional entity will be first demonstrated (a showable) and by when it will be considered to be completed (a deliverable).

For example, an order entry / finished goods stock control project might be broken down into major modules:

Order entry completed by

Picking slip and invoice production by

Finished goods entry to stock by

Inter-depot transfers by....

Each one of these modules would be further broken down to small function steps, for example order entry might comprise:

Customer name & address verification by

Credit checking

Discount application

etc.

Each larger module would have a deliverable date and every module a showable as well as a deliverable date.

These target dates can then be converted into a budget for programmer time (and hence cost).

4.6 Cost And Time Budgets

The AP estimates how much time it will take him to complete each deliverable. It is an almost impossible task since he knows that during development some things will prove much more difficult - or extensive changes will be demanded by the UPL - and others will be easier than he fears. He takes his pessimistic estimates and adds 30%. (Programmers always under-estimate, some workers have suggested adding 75%).

The UPL takes the AP's estimate and adds a further secret 30%. This is to allow the UPL to commission additional changes. Again others have suggested this figure should be 75%.

Notice that a budget is set for each step by the AP and is known by and agreed to by the UPL.

It is agreed between the AP and the UPL that the system will be completed within 70% of the (AP's) target budgets for time and cost! This clearly has implications on the way the AP computes his budgets. The remaining 30% will be reserved for final tuning, last minute changes and close AP support of the live program (and its users) during the first period of live running.

The AP's budget is communicated to the QS but the UPL's secret contingency is kept secret to him and his management.

4.7 The Actual Development

As soon as possible after the budgets have been agreed the AP produces a demonstrable skeleton system. This will typically consist of menu screens plus one or more data entry screens. These days the skeleton can be produced in hours (or at most days).

Regular progress meetings occur - at least fortnightly - at which progress will be demonstrated with working code, and the system will be critically examined by the UPL (and perhaps other users) and the QS, and improvements or difficulties discussed. It is of course imperative that the program is designed to be changed easily. The AP soon learns how to do this - or is rapidly found out. The examination of cost and time budgets is mandatory at these meetings.

The inevitable changes to the program suggested at these meetings have cost and time implications, and the lot now falls to the AP to point out that certain changes will increase costs or extend time deadlines.

Some AP's find this the most difficult part of the method - it is because it runs against the method's whole ethos as seen by the AP, viz. to satisfy the user. The UPL must balance the value of the enhancements against delay or costs. The great advantage of ISE is that since development can continue after the system is useable, some desirable enhancements can be delayed until "phase 2" and next year's budget!

The UPL and the QS are again the key to the success of the progress meeting. It is they who monitor that progress is being made on schedule and that the program is fulfilling user needs. If progress is slow or they suspect the AP is finding things too difficult - the UPL or QS can detect this (the demonstrations are disappointing) - they must report this to IT management before it is too late.

394 Software Quality Management

4.8 User Involvement

The UPL is a user representative. It is his responsibility to ensure that the "users" i.e. the people in the organisation who will praise or curse the finished system, are more likely to praise than curse. Attendance at progress meetings by other members of the user organisation - including some quite junior ones, is often very sensible. It has been our experience that rules for data validation are often better understood by clerks than their manager. A clerical officer who has a hand in designing the data entry screen most often used by himself and his colleagues, is more likely to defend the system when it goes live, than someone who is handed a carefully thought out screen designed by a genius.

The more the users can feel part of the development and the more they feel significant contributors to its success or failure, the more likely it is that they will make it succeed.

4.9 Live Testing

The UPL and AP may agree on test data and perhaps even parallel running, though our experience is that programs fail on real data not test data, and true parallel running is almost impossible. Vast amounts of effort can be spent in tracking down discrepancies only to find that they are due not to a bug but to a failure to achieve true parallelism. A better approach is to have a series of spot checks on key paths through the program, coupled with involving all users in bug hunting. The key thing is that users must expect bugs - always. They must never become complacent.

In the early days of live running, users should feel part of the debugging team and should be praised for finding errors. It must be clearly explained that bugs are a natural, nay healthy, part of computer programs. That users should find them, demonstrates the user's intellectual strengths and business efficiency. At all costs avoid the situation where the users are told "the program is now completed". It will only be followed by "Oh no, not another bug - don't those computer people ever get anything right?"

5.0 DISASTER PLANNING

At every stage a careful consideration of what will happen if a major disaster occurs, must be considered. Programs - particularly new programs - can corrupt files and render the system un-useable. All possible scenarios must be considered and a fall-back position agreed. The user organisation must continue to function at all costs. The UPL is responsible for this, fully supported by the AP. All user management must be aware that disasters may occur and what the proposed strategy in each case will be.

6.0 THE USER PROJECT LEADER'S ROLE

The UPL has such an important influence in making the project a success that it will be fruitful to set out below a summary of his responsibilities, they include:

- a. Ensuring that the project is on time and to budget - it is made clear that if at any stage he has worries about progress he must escalate his concern within the IT department (or software house). It is of course the responsibility of the IT (or software house) management to make sure the UPL understands this, and to impose their own checks as well. The QS has a similar parallel responsibility.
- b. Controlling the project by:
 - i. Insisting that cost and time budgets are examined at every progress meeting.
 - ii. Limiting the number of changes made to each aspect, to ensure that the project does not bog down in one area. Further changes will be inevitable, but by saying to other users, "Let's settle with what we've got for the moment and improve it next year in phase 2", he will ensure that a useable sub-system is completed on time.
 - iii. Insisting that at every progress meeting, he sees the agreed enhancements from the previous meeting actually working. This is vital. His check on progress must consist of comparing what has been promised with what he can see working!
- c. Ensuring that the system will satisfy user needs. He does this by:
 - i. Expressing requests for changes at each progress meeting.
 - ii. Encouraging contact between the development team and the ultimate users (especially data entry staff) - perhaps by inviting them from time to time to progress meetings and involving them in debugging.
 - iii. Training other users and "selling" the system in his organisation. Targeting on getting one key system in and running to the delight of the relevant department is one way of making everyone else clamour jealously to be the next beneficiary of his wonderful software!



396 Software Quality Management

7.0 THE QUALITY SPECIALIST'S ROLE

The QS is there both to support the commitment of the AP and UPL and to be the "customer" when it comes to things hidden from the UPL. We discuss below aspects of the development which are the primary responsibility of the AP but might be easily overlooked by the UPL.

7.1 Documentation

Documentation is like snuff, the less needed for the desired effect, the better. All documentation must fulfil a purpose. Documentation must never be produced "because we always have one of those".

The most crucial documentation is the maintenance documentation. The words and pictures which will supplement the compilable code and will enable those who follow the AP to comprehend the programs.

7.1.1 Program Documentation

This section might have been alternatively titled System Documentation. It is that which with the source code describes the system in its entirety for posterity. The closer it is to the source the better. The best way is to have it embedded in the source as comments.

The only way to check that it is adequate is by peer review. The QS, or an appropriately competent member of his staff, reads the source code and documentation. If he cannot understand exactly how the program works - this documentation needs to be improved.

Certain overview documentation may also be needed to explain links between modules and overall system structure. Again the closer this is to the actual source the better.

In some systems we have developed, a switch in the executable code will display at run-time the source file names and directory paths as well as load times and dates. This has been an invaluable aid to maintenance years later.

Separate documentation on paper should be avoided - it just gets lost or out of date.

The QS is responsible for ensuring that this maintenance documentation is to a high standard.

7.1.2 User Documentation

As much user documentation as possible should be on-line and available to a user at the touch of a button. This should be carefully designed and tested by

users. The QS needs to ensure this happens. The AP is often too close to the program to realise what novice users might find confusing.

Other user documentation can also be produced. Detailed user manuals for organisations which measure documentation quality by its weight. Though a reference manual for a system which provides enormous functionality at every prompt - like a word processor - is clearly desirable.

Management overview documentation - extolling the virtues of the system and explaining its place in the company's IT strategy as well as introductory texts for novice terminal users may be produced at the UPL's discretion.

7.2 Code Modifiability

What makes a program modifiable? Is it structured program design or object oriented programming or high modularity? For a program to be easy to change, it must be easy to understand, and it must be easy to change just one aspect of its functionality with no risk of side effects.

What ever technique the AP chooses, the QS must satisfy himself that it works. The progress meetings during development will give the QS some indication of this. Examination of the source may also be useful.

7.3 Performance Under Load

The only real way to test performance under load is to do it! Getting 100 data entry clerks to key in test data over a two hour period may not be feasible. The QS will have sufficient skills to be aware of those aspects of the AP's system that could cause response or load problems. Actions which imply a search of the entire database may work well in the development environment when data is minuscule. The QS must be constantly on the lookout for such design flaws and prompt the AP to address them.

7.4 Access Security

That the QS is aware of the need for access security is usually sufficient for it to become an important design criterion. The users' needs and company policy will reinforce this.

7.5 Data Security

Most modern database management systems come with a recovery system (where database transactions are logged separately). In some situations a terminal transaction logging system is also necessary - so that transactions are logged even before they become database transactions. The QS is responsible for ensuring the AP explains these factors to the UPL and that the UPL understands them and takes a reasonable decision.



398 Software Quality Management

7.6 Training

Clearly some user training will be needed. In those situations where the system replaces an older system (perhaps a manual one) users will know what the system is supposed to do, but not quite how it will do it. In these circumstances data entry staff can become proficient quite quickly. Managers who may need to query the database may take longer to train.

The system cannot be considered implemented until most of its users feel competent.

7.8 Implementation

By this we mean going live! The QS is responsible for ensuring that adequate testing has been done, that essential data is pre-loaded and that a disaster plan is in place. The AP will be in continuous attendance during the early days of live running.

7.9 Interfaces With Other Systems

Typically the AP and QS will be aware of the need of the system to interface with other systems. The UPL may not care. The QS is responsible for seeing that the AP gives appropriate weight to this matter.

8.0 SUBSEQUENT DEVELOPMENT

Everyone should expect further changes to be requested post live running. The only system that no one wants to improve, is the system no one uses! The users will want to change a good program because:

- a. They can see ways of making their jobs easier.
- b. They can see ways of making their jobs more effective.
- c. The business changes.
- d. They find irritating restrictions or clumsiness with minor aspects.
- e. The program is a fundamental part of their job. As their job develops they want the program to develop.

Provided the program has been developed by ISE further changes are just a matter of re-starting the development process.

9.0 THE ADVANTAGES OF ISE

1. There is a higher probability of user acceptance of the finished product.
2. The AP is forced to design his programs so that they are highly modular and hence capable of easy change. This imposes a strict programming discipline and makes for good programmers. The use of 4GL's is clearly a great benefit here, though efficiency may be sacrificed in some 4GL's for the sake of programming ease.
3. Inefficiencies in the program are likely to be discovered during progress meetings or through the activities of the QS and remedial action (by writing the code differently) taken immediately.
4. The AP, since he is forced to make his programs work for every progress meeting, is at all times debugging those changes he has just made. He not only has an easy time of debugging, but he is far less likely to get lost in his programs. (Getting lost in a program is, we believe, a more frequent event than is generally realised. It is one of the causes of inordinate development times and lingering bugs.) The frequent feedback the AP gets both from the user and from the computer (because his programs are almost always working) is highly motivating.
5. If the AP does get into difficulties, it is very apparent at progress meetings and remedial action can be taken at an early stage.
6. The AP is responsible for all aspects of the program's design and implementation. This can give him great professional fulfilment and leads to him becoming an even better analyst/programmer.
7. Further enhancements to a "finished" program are often no more than re-starting the development process. Though programs can become obsolete when the original design or business purpose are inappropriate, the fact that high modularity is forced (or progress meetings become very painful) means that programs have a much longer life. It has been our experience that systems are more often than not re-written for somewhat dubious reasons; a new fashionable language has become available or technology changes and no one wants to feel left behind.
8. The process of program development is cheaper. The classical approach can be very expensive in both time and money. It produces copious documents which describe the system to both the user and the programmer. It is nearly as cheap (it can be cheaper) to get an AP to describe the system in a programming language (3GL or 4GL) and show it to the user by demonstrating it to him. Since each chapter is



400 Software Quality Management

"read over to the user" as it is written (at each progress meeting) any mistakes are quickly noticed, easy improvements from the user are quickly incorporated and when the "analysis documentation" is completed the system is finished!

9. Cost and time control by the user (the ultimate funder with all techniques) is greatly improved. Since the program progresses by a series of staged enhancements to a working system, not only can the team get a good feel for progress towards the first really useable version, the user can also stop development at an adequate but not perfect sub-system or sub-system aspect. Remember that at each stage the program works. Stopping development leaves working code!

10.0 CONCLUSION

ISE a variant of Evolutionary Prototyping we believe offers the prospect of high quality and rapid software development.

REFERENCES

1. Jeffreys, M.E. 'Application Development via Evolutionary Prototyping with Cost and Time Controls.' Chapter 11, *Fourth-Generation Systems* ed. Holloway, S. pp. 150 - 161, Chapman and Hall, London and New York, 1990
2. Hekmatpour, S. and Ince, D.C. *Rapid Software Prototyping*, Mathematics Faculty, Open University, Walton Hall, Milton Keynes, MK7 6AA, UK, 1986
3. Boar, B.H. *Application Prototyping* John Wiley & Sons, New York, Chichester, Brisbane, Toronto and Singapore, 1984