

Additional Data from Software Inspections (Revised)

D R A F T E

C. C. Shelley

6 June 2005

Oxford Software Engineering
9 Spinners Court, 53 West End, Witney, Oxfordshire, England, OX28 1NH
shelley@osel.netkonect.co.uk

ABSTRACT

Software Inspections are a very effective software quality control, inspired by production engineering practice. Minor modifications to the inspection process are proposed to better fit it to the software design process, improve defect detection efficiency, and widen the scope of inspections to enable design excellence to be identified and evaluated.

1. Introduction

Software is complex and prone to error; good quality controls are essential. Among the best quality controls are formal technical reviews and inspections: software inspections can be very effective indeed. These originated in IBM, developed in IBM, by Michael Fagan, Ron Radice and other engineers, who were inspired by production engineering quality controls. The remarkable effectiveness of software inspections as defect filters is leading to their introduction to other engineering disciplines where requirements, design and other documents need to be of the highest quality.

2. Software Quality Control

While they take their inspiration from production engineering software inspections and formal technical reviews differ from production engineering quality controls in a number of important respects: Inspections are necessarily performed by people – intelligent agents – who evaluate the item being inspected using their intelligence and judgement. They are intelligently evaluating the product of an intelligent design process. While this is not unique – editors reviewing manuscripts perform a similar function – this is unusual in industrial quality control

where defect detection and measurement is, intentionally, and so far as possible, mechanical and with the minimum of judgement. Software inspections attempt to emulate this mechanical aspect of production engineering inspection by developing checklists and criteria for evaluating the inspected item. This gives a more consistent, inspection process and, with care, over time leads to more effective defect detection.

Another distinction is the nature of the defects themselves. Defects found in manufacturing – where the items inspected are produced to high tolerances (compared to software) with well defined measures – usually conform to well understood distributions. The defects found in software inspections and formal technical reviews can rarely be shown to conform to any recognized distributions. This is because the data are stratified and sparse, and the inspected items are design artefacts, not manufactured (replicated) items. Each software item inspected is essentially a ‘one off’. It will have similarities to other items, but is essentially unique.

3. Drivers for Software Quality

The primary driver for production engineering quality control is conformance to standard and the control of variation. Where defect measures or detection rates vary too much the manufacturing process is adjusted or revised to bring quality back within the required limits. (Defect measurement and analysis is one of the primary means of understanding and improving (revising) manufacturing processes.)

The control of software quality has a similar driver: reducing the number of defects to an acceptable level. But there is difficulty in measuring and controlling the quality of the reviewed items. Measures and analyses of software quality and

review effectiveness attempt to adapt the methods of production engineering but with limited success. The variability and diversity of the essentially unique reviewed items, and the design and review processes militate against measurably consistent quality control to anything like the precision achieved in production engineering. The necessary assumptions required for any analysis are often unfounded. This is not a failure of the inspection or review process or an indictment of the quality of software, it is simply due to software development being largely a design process.

This difficulty suggests that there is another driver for software quality – design excellence. This is not evaluated by defect levels or conformance to standards. An excellent design may have flaws or defects but never the less be considerably better than a lesser design that is nominally less defective. This is intuitively recognized in ‘design classics’, despite their limitations. This driver can show itself in reviews and inspections by the behaviour of inspectors that is actively discouraged by the inspection process itself: There is a marked tendency for inspectors to discuss the item being inspected; to consider fixes to defects found, or consider design alternatives or simply discuss points of interest. This activity is costly and reduces defect detection effectiveness and is discouraged. Some inspection processes allow a ‘third hour’ where distractions to the defect detection activity may be discussed but this is rarely effectively managed or exploited.

If design excellence really is an important aspect of software quality (and this can be debated) then the most effective means for evaluating software quality should provide some means for recognizing, evaluating and exploiting this excellence. At present it does not. It is not acceptable to spend time during an inspection discussing the attributes of the item, ad hoc, because this compromises the defect detecting activity; some other means is required.

Defective design is bad, design excellence is good, but an excellent design can contain defects, and a defect free design is not necessarily excellent. Something is missing. Defects indicate poor quality, what indicates good quality: what is the opposite of a defect? There is no antonym to defect but the concept is recognized: a ‘cool hack’, an ‘elegance’... are terms used by those involved in technical design - for exceptional micro-invention, those little events of unusually good design that set up ripples in the design aether. They are recognized by the reaction of peers; surprise and pleasure, or puzzlement and irritation, with perhaps a little envy too. This is what is happening in inspections; the inspectors, as intelligent agents, are recognizing excellence, or its lack. It is this that provokes the apparently unproductive debate. There is a desire to

discuss these events, to explore the design or propose better design solutions where they offer themselves.

This lack of an antonym to defect has not mattered until recently. Few people were involved in technical design work and the word was not needed; unlike ‘defects’ which have always affected those involved in the manufacture or use of manufactured items. But now, with software and other design work being undertaken and managed by very large numbers of people, the ability to recognize, evaluate, analyse, manage and communicate design excellence is needed. So a word is needed for these ‘undefects’, ‘pro-effects’ or ‘profects’¹. With the definition of this ‘new, good idea’ word comes the ability to count, and consequently, measure and manage excellence. Like defects, profects can be recognized recorded, counted and analysed. Unlike defects they can be reused and amplified.

4. Profects

Profects offer a pleasing symmetry to the evaluation of design quality – redressing the balance from defects alone - and should be welcomed by software engineers. At present an inspection or review identifies a list of defects for fixing, but fails to recognize the excellent aspects of the design. This can be difficult for software engineers who may have worked hard to produce an exceptional piece of work - albeit with some defects. It also deprives the software organization of valuable information about the quality of its products. This symmetry is not perfect. Profects have some distinctive characteristics: They have a half life. Profects are only profects because they are novel – they surprise, they are unexpected. After a while a profect becomes a piece of good design, or a pattern, or standard. Profects can be considered as pattern precursors (or antepatterns?). When the profect is accepted and widely adopted it ceases to be a profect. A design classic could be expected to have an exceptional number of profects, to be profect dense, or contain a few truly exceptional profects. A later imitation of the design will contain none. The essence of a profect is excellent, *recent* design activity resulting in an excellent *new* idea. The half life of a profect is determined by its value and take up. A powerful?, big?, profect (there is no term for describing the ‘unseverity’ of profects) may be rapidly adopted and rapidly graduate to a pattern or standard, so have a short half life. Alternatively it may take time to become accepted and the design element adopted, giving it a longer half life. Less significant profects will have short half lives.

¹ First reported use of ‘profect’ in this sense of Jennifer Reis in ‘Seven Habits of Highly Effective Hackers’ by Philip Johnson, 3 August 1999.

Each profect is necessarily unique, unlike defects that are mostly tediously similar, even if they are located in different parts of the documentation or code. Profects are also rarer than defects. Expect profect densities to be two or more orders of magnitude lower than defects.

Profects are treated differently to defects. Defects are identified in inspections or reviews and then removed, then defect data, counts, densities and so forth, may be analysed to provide information on product quality and software process quality. This is similar to the way in which defects are analysed in manufacturing processes (but unlike manufacturing, the analysis of software defects should use the more robust non parametric methods to look for patterns and outliers, rather the constrained parametric techniques use for well understood manufacturing process control).

Profects are identified in a similar manner to defects but are not removed. They are recorded to forestall unproductive discussion during the inspection and then examined to be better understood and perhaps exploited. The emphasis is on reusing them elsewhere and amplifying their effect: developing strategies for disseminating profects, injecting them into other designs, and triggering profect 'epidemics'.

The identification of defects improves product quality and reduces rework, delays and dissatisfaction, but is bounded by a lower limit of zero defects; the amplification of excellence has no limit.

5. Changes to the Inspection Process

To detect and reuse profects some minor changes to the software inspection process are needed: In the preparation for the inspection inspectors identify potential profects – recognized by their perceived value and the inspectors reaction to them (surprise, etc.). This detection cannot be guided or prescribed as well as defect detection, with the use of standards or checklists, because profects are intrinsically unique and unexpected, so are difficult to categorize, never the less an ability to recognize design excellence can be cultivated. In the inspection meeting potential profects are raised and logged just like potential defects. Unlike defects most profects do not withstand scrutiny. They will be recognized by others as nominal solutions, be revealed as defects or simply not make the grade. Those that do should be logged in a similar manner to defects.

To aid the detection of profects it is worth considering a 'cool hunter' role in software inspections. This 'value detector' should be capable of recognizing excellence or novelty. Care should be taken that this role does not, consciously or

unconsciously, quash profects. Excellent designers may not be the best cool hunters being unable or unwilling to recognize excellence arising from other sources.

With the profects identified and logged they can now be analysed in more detail. This is in contrast to the traditional inspection third hour where any issues distracting from defect detection is deferred. Because profects can be named, tagged and managed inspectors can be confident that their insights identified during the inspection will not be lost and therefore need not attempt to discuss them during the inspection, or immediately afterwards. The simply act of logging profects for future examination defuses the distracting discussions that would otherwise arise.

During the inspection details of the profect are logged in an equivalent manner to defects. The defect log can be used for logging profects as well, with perhaps the defect severity field used to signal a profect. The value or 'unseverity' of the profect may be logged, terminology does not exist but a simple ordinal scale – 'cool' to 'hot' perhaps - will do. Notes describe why the profect is an exceptional solution, and give a brief statement of the problem it solves. If it is apparent that the solution can be generalized this should be noted too.

Classification of profects becomes possible as patterns emerge. The recognition of exceptional design implies that certain characteristics are evident. These fall into categories: simplifier, eliminator (of code), synthesizer, synergiser... etc.. The source of profect is implied by the artefact inspected, but in cases where the artefact has many authors consider identifying the originator of the profect².

Profect logs are used in a number of ways: Each profect having caught the attention of inspectors qualifies for further attention. Opportunities for reusing the profect, making it available or developing it further should be taken. Each instance of reuse providing a benefit and helping to transform the profect into a standard or pattern, reducing its half life and realizing its benefit. Profects counts can be used in a way analogous to defect counts. Profect levels can be set: systems or products

² This identification of the profect source may be problematic. It is well known that attempting to measure individuals causes problems. Profect sourcing is potentially valuable but could be prone to similar abuse: It is well known that the abilities of software engineers range over an order of magnitude or more. Profect counts may provide an indication of this variability. It is worthwhile discovering who are the best profect detectors too - the cool hunters. These individuals, able to recognize excellence, are also a significant asset.

required to capture or develop a new market can be expected to be innovative or perfect dense (perfects are a tool to manage cool). Tracking perfect levels during development may give an indication of this. Alternatively for safety critical systems innovation and design novelty is undesirable. A conservative design, reusing existing code, patterns and standards will be important where high confidence levels are required. Reviews and inspections should signal low perfect levels. Or a mix of innovative and conservative design work may be required – a necessarily innovative architecture may suggest using conservative low level design components and code – to rebalance risks.

Like defects perfects tend to have more impact early in the software development lifecycle.

The level of innovation required or expected may be used to select the most appropriate resources – assign innovative engineers (perfect sources) to the innovative products and the conservative engineers to the conservative products. ‘Innovative’ does not necessarily mean ‘best’, although this is often assumed.

Perfect levels can be analysed in concert with defect levels – the balance of design excellence and design failure proving a richer picture of software development progress and quality than defects alone.

6. Organizations

Perfect identification, reuse and analysis can be expected to be most effective only in organizations with particular characteristics: A degree of process sophistication is required. The customary and consistent performance of formal technical reviews or inspections tends to occur in process aware organizations, often with an engineering orientation. (Although this is now changing. The more sophisticated software organizations producing commercial products or delivering software services can also have this process orientation even if not engineering oriented.)

A measurement capability is also required. An effective defect tracking system is the minimum requirement, to enable perfects to be recorded. A meaningful data analysis capability is also highly desirable. If defect analysis is performed, and acted upon this provides a sound basis for perfect analysis too.

Finally the organization must have a genuine interest in design and innovation. Increasingly software development capability is seen as a commodity. There is little likelihood of perfects being valued in such an organization. Organizations that value the knowledge and skills of their people and actively manage intellectual assets will recognize the value of perfect analysis.

7. Benefits

The recognition and logging of perfects modifies the software inspection process to fit it into software development as an essentially design process, not a manufacturing process. Organizations that treat software development as design gain the following benefits:

- An immediate reduction of distractions in software reviews and inspections as matters of interest are captured for future analysis. With this understood there is no imperative to resolve these important matters in the inspection.
- Design quality is evaluated in a balanced manner – good and bad – not simply bad, in terms of defects.
- Design excellence is recognized, captured and made available for use elsewhere.
- Accelerated product and process improvement.
- Design capability can be recognized, evaluated, analysed and exploited more effectively.