This is the published version:


Available from Deakin Research Online:

http://hdl.handle.net/10536/DRO/DU:30004638

Reproduced with the kind permission of the copyright owner.

Knowledge Mediation in Software Quality Engineering

Bruce A. Philp
MetaSilicon
Sydney, Australia
bphilp@acm.org

Brian J. Garner
Professor of Computing
Deakin University
Geelong, VIC 3217 Australia

Abstract

The risk of failure of the software development process remains high despite many attempts to improve the quality of software engineering. Contemporary approaches to process assurance, such as the Capability Maturity Model (CMM) have not prevented systemic failures, nor have project management methodologies provided guarantees of software quality. This paper proposes an approach to software quality assurance based on Knowledge Mediated Concurrent Audit, which incorporates essential feedback processes. Through a tightly integrated approach to quality audit, programmers would be empowered to use any chosen methodology to advantage, supported by intelligent monitoring of the essential interactions which occur in the development process. An experimental application implementing some aspects of the proposal is described.

Keywords: knowledge mediated concurrent audit; software risk; software development process; XP

1 Introduction

Whatever methodology is used, any complex software development is dependent on one or more hierarchies of human intelligences. Human intelligence and memory can be enhanced at the level of the individual programmer by development software which supports him or her as a self-spectator, observing the act of code creation and offering integrated testing. Without wishing to encroach on the fields of team management, change management, or specialised testing, we suggest that these techniques can be used to advantage at all points in the hierarchy.

2 Risk in Software Development

The evolution of the programming environment over several decades is a story of successful problem-solving. Boundy, when writing before the widespread adoption of Object Oriented methods [3], classified programmers from "minimally competent" to "truly great"; characteristic of the latter was their ability to create program-writing tools, tailoring languages and methods to their own requirements. With the development of new programming tools, this ability is now commonplace. At the same time, however, the scope and complexity of programming tasks has increased, and quality risks remain unacceptably high. The former emphasis on efficient tracing, debugging tools, and risk minimisation by a carefully managed software development process, has necessarily been replaced by more mature process architecture [25] and emphasis on control of the programmer's entire development environment. The logical next step is to enhance this further with intelligent assistants that identify potential sources of error and assist the software developer in avoiding risks. By applying knowledge-mediated concurrent audit techniques to the development process, intelligent knowledge tools will expand the "brainspace" for developers, supporting and supplementing their expertise in whatever methodology they favour. In this way concurrent audit is expected to reduce risk in software development and support a quality assurance process.

Efforts to improve the software development process range from strict, formal codes, for example CMM [26, 4] to the romantic (and very successful) heroism of the "Open Source revolution"[7]. Team co-ordination is always at risk, and dependencies between risk factors make the overall risk assessment problematic. Methodologies are reluctantly embraced for a variety of reasons [24]. The larger the project, the greater the management and communication/co-ordination difficulties likely to be encountered, with an attendant risk of catastrophe [5, 15, 16].

An economic way of enforcing best practice is unlikely to be found [22, 30]. Professional programmers are willing, however, to be helped, reminded, warned, and empowered [31, 9]. They have historically embraced models and tools that expand their practical knowledge, skills and productivity, and standards acknowledge this: in the words of L Gray [17]."In many modern standards, the only truly mandatory
activity is tailoring the standard to your particular needs”. Rather than new forms of testing that do not empower the programmer, the need is for a new way of bringing quality assurance into the pre-emptive programming task aptly termed “anti-bugging” by Cooper [6]. It is not our purpose to encroach on the field of development team management, but to show the benefits that managers and programming teams may derive from knowledge-mediated concurrent audit.

Software is developed by humans for humans. It is not possible to avoid reliance on fallible human communication processes, on individual skill and creativity, and on the interpretation of appropriate reference models if these are available. This is an intractable underlying source of risk in the software development process. Typically, as soon as a system has been built and performs successfully, new features are specified, or a more ambitious system is attempted. The over-arching importance of change management during system evolution mandates ever greater weight being given to

- feedback in the Software Development process
- documentation
- on-going client/developer co-operation
- professional standards in work practices

While much progress has been made over the years, and many methodologies have been developed to capture and enhance best practice, risk management in software quality engineering requires the following to be explicit:

- awareness of individual strengths and weaknesses
- richer automated support for human memory over time

3 The Need for Intelligent Assistant Software

An individual programmer is empowered by intelligent software agents which contribute immediate feedback, adaptive learning and automated memory to the task. It is our hope that this will prove true at all levels of software creation, and that machine learning techniques will offer solutions that can scale up to large development projects. At the individual programmer level, the Concurrent Audit Tool (CAT, hence Felix) that we are developing internalises feedback and documentation by modifications to the programmer’s development environment. Ultimately, it is intended to model all human stakeholders and allow them an interactive view, tailored to their interests and privileges, but the initial version of the tool addresses the level of code creation. The proposed Knowledge Base, based on Conceptual Graphs [28], to be included within Felix, will permit reasoning about events which occur during the software development process, using contextual mappings to identify possible risks and to activate appropriate risk mediation strategies.

Blocks of code, the current programmer(s), and end users as represented by specifications and other involvement, are combined to form active contexts during system development. Each context also has a history, with associated elements of “documentation”, each source having its own contextual information. For example, human annotations such as to-do notes have an originator and an intended recipient, while hints and error messages generated in past compiles have programmer-at-the-time information.

Borland’s Delphi was chosen for the experimental platform, as it provides a mature Integrated Development Environment (IDE), including a ToolsAPI unit supporting customisation of the working environment. An active development community exists with a largely Open Source ethos—many “add-ins” to the IDE are available under a variety of licensing agreements. In addition, Kylix, a version for Linux, has been released, offering the prospect of integrating GNU tools, including Conceptual Graph applications. Lastly, Delphi is a commonly used commercial development platform. By releasing our tools as freeware, we hope to obtain valuable feedback from professional users.

The Delphi IDE has been developed in response to programmers’ needs for feedback and ready information; hence it is highly interactive, with many context-sensitive aids making large amounts of data available quickly. For example, as-you-type code completion lists all properties, methods, and events that are valid within the current scope for the class being typed. Passing the mouse over any identifier pops up details of its declaration, and a click can bring up the declaration code. When a programmer begins to type a function call, a parameter list will pop up—or lists, if the function has been overloaded. This feature illustrates the principle that software development risks change over time as tools are improved: the risk of errors caused by misuse of overloaded functions is now minimised. Our research investigates the scope for extending the Delphi IDE with an intelligent Concurrent Audit tool.

Integration of code editing tools, compiler and debugging of output in the programmer’s development environment has reduced risk by improved error prevention. Pitfalls of the past such as mistypes, memory errors, buffer overruns, and column- or row-major trouble with arrays have been fenced off, but new risks are evident with release of powerful new languages, such as java, and the move to application component integration. The research proposition may be stated thus: the IDE can be enriched by awareness of the human context, of design goals, and of changes over time, thus adding risk minimisation functionality.

One of the reasons for the development of the structured programming paradigm was that modularity could help pro-
grammers from having to keep too much in mind. The problem of programmers' mental overload justifies the need for an intelligent assistant:

"Bugs happen because people create them when they can’t remember the details needed to write correct programs. Why waste the user’s scarce biological cache memory when plentiful computer memory performs better?" [12]

Consideration of the "biological cache memory" raises another issue: programmers are all different. A truly intelligent IDE must be aware of the programmer and his/her experience, skills and limitations, and must be capable of enriching this process during the development, ideally by automated learning methods. We can distinguish at least two situations:

1. The novice is learning new skills, is probably working under supervision on non-critical code, with generous deadlines, and consequences of failure are personal; and
2. The skilled programmer is practising from well assimilated, sub-conscious skills, possibly with a large scale view as part of a team, with tight deadlines, and greater risks. The expert is satisfied when the work flows; the novice tends to be satisfied when something merely works. Both novice and expert can suffer from "mental overload", but the meaning of this is different in each case: artificial intelligence (AI) techniques are required to support both over time. Successful commercial applications of Concurrent Audit for financial auditors [2] provides an analogous situation with respect to the demands on human memory. The skilled programmer participating in a large-scale software development is as far removed from a novice, as an auditor checking a complex, real-time financial system is, from a junior colleague working on closed books. Experiments with small scale applications as a source of programming quality issues are suspect if they do not challenge the "biological cache memory" of skilled programmers. On the other hand scalability can be enhanced by AI techniques, such as abductive inference, using an extensible Knowledge Base derived from monitoring cognitive designs and coding practices during the concurrent audit process. Risks which arise from overloaded memory can be addressed by concurrent audit techniques. In the software development context, these risks typically arise from forgotten quirks of syntax, requirements which the programmer simply forgets, requirements within the current context which might be overlooked, and from incorrect classification of work in progress. Code is incipient, stub, incomplete, checked, confident, template-generated, tested, etc., but it is not always correctly classified, especially in the case of third party modules. Experienced programmers, when writing fluently, tend to go through cycles, and if distracted in mid cycle, can neglect essential code by wrongly classifying a block of code. Traditional techniques to forestall this neglect such as to-do lists, remarks, and deliberate exceptions can be materially enhanced by intelligent assistant software.

Ontological classification of bugs has been proposed, not necessarily classified according to risk. Eisenstadt [10] collected programmers' "war stories" of their experiences with bugs, and approached the "taxonomy" question from various angles. Most of the bugs he studied belong to categories where memory inadequacy is a factor and a knowledge-based approach would help. Not all, however: for example, we will probably always be at risk of what he calls WYSIPIG ("What You See is Probably Illusory, Guv'nor") errors. The bugs that programmers are willing to recount, or to consider worth recording, are not necessarily the most risk-laden or time-consuming. Might less memorable bugs, not considered worthy of submission to the "war stories" collection, be even more amenable to knowledge-based vigilance? Even a "complete" list of bugs within a project only partly eliminates the subjective element of choice: Knuth [19] has maintained such a log from 1978 to the present time in the remarkable (and now bug-free!) T\TeX project, but earlier bugs recorded are more low-level than later ones. For example on 10th March 1978, he recorded "Fix bug: the test ‘id < 200’ was supposed to distinguish one-letter identifiers from longer (packed) ones, but negative values of id also pass this test". Compare the former low-level bug with: "Balance the parentheses shown on the terminal during normal runs" recorded on 23rd March 1990. Intelligent assistant software would not tire of writing detailed low-level logs.

A knowledge system that is aware of individual programmer risks can give customised warnings, tips, and reminders. It can learn about his/her bugs and vulnerabilities in various ways, including from stored hints, warnings and error messages from the compiler, the results of failed tests, and manual insertion. For example an intelligent assistant might record a programmer's particular "mental blocks" about library functions or elements of syntax and provide timely information. If a programmer has the habit of writing code that will not be caught by the compiler but will loop forever at run-time, this will merit a manual insertion in the knowledge base. Concurrent audit cannot replace the need for effective human supervision, but could add value to it by integrating supervision, testing and training. An intelligent log could be used, for example, to generate a personalised training session aware of weaknesses in programming style and knowledge deficiencies.

Older methodologies attempted to lock down "complete" requirements, while newer ones adopt an evolutionary approach, accepting change and claiming better results [13, 14]. A knowledge-based Intelligent Assistant, exporting logged data to Change Management software, could
help to manage change by anticipating, even encouraging, growing awareness of the complexity of the development task and of potential software solutions changes not just in specifications but also in assumptions, and agreements about them. Human factors play an unknown and elusive role in software quality engineering. While AI techniques cannot be expected to manage every conceivable cognitive style and misconception, they can enhance human memory effectiveness, and in effect, activate cumulative change management processes. Good intentions, client hopes, methodologies and interpersonal communication are all at risk of deteriorating over time, if not safeguarded by:

- Access to the knowledge required to sustain personal confidence (i.e. reinforcement processes); and
- Evidential facility for the continual assessment of changing work practices, as demanded by task complexity and scope, having regard to appropriate standards, reference models and control knowledge.

### 4 Choice of Techniques for Experimentation

Techniques compatible with our goals of risk reduction by concurrent audit, integration with the programmer's development environment, and awareness of human factors are to be found in the ultra-light methodology eXtreme Programming (XP) originating with Kent Beck[1].

**Pair Programming** is an XP technique that brings the benefits of code walk-through to an earlier point in the development process. It is significant in the context of anticipating, because programmers debugging their own code tend to use goal-driven reasoning to find bugs (either heuristic mapping or causal reasoning), whereas when debugging other programmers' code, they tend to use data-driven reasoning (typically hand simulation) [18, 10]. Pair Programming brings both points of view to the initial creation of code. Felix will support programming in pairs or groups by recording "smart" self-triggering messages, associated with contexts.

XP also brings testing and validation to an earlier point in the development process, by developing test units in parallel with code generation.

The idea is that, as you develop or change code, you develop appropriate verification tests at the same time, rather than postponing them to a later test phase. By keeping the tests up-to-date and by re-applying them at regular intervals, it becomes easier to produce reliable code, and to be confident that alterations... do not break existing code. Applications become self-testing. [29]

A formal metrics approach, which has resulted in many excellent initiatives, was considered [11]. However, our analysis of the crisis in software development has demonstrated the need for a theoretical model of risk management and accountability linked to the experiential requirement for human reinforcement. Knowledge-mediated concurrent audit, in relying on the exercise of human intelligence, appeared to be more promising than static, post-audit assessments. XP's Code Smells approach (see on-line discussion at http://c2.com/cgi/wiki?CodeSmells) substantiates this as a credible option. XP starts from the premise that experienced developers reached a state of "unconscious competence", in which they can very quickly get a "feel" for the quality of code if it is poor, they can "smell a rat". Aspects of code which should make XP programmers suspicious, and which an intelligent assistant can identify statistically and raise concerns about, include duplicated code, methods “too big”, classes with “too many” or “too few” instance variables, classes with “too much” or “too little” code, strikingly similar subclasses, a conceptually simple change that requires modification to code in many places, and parallel inheritance hierarchies. Thus, our second research proposition is that non-feasible determinism can be avoided in the synergy between Intelligent Assistant software and human intelligence [8, 27]. Large scale study would be needed to test this; but a third research proposition is more easily tested: that the extensibility of a Knowledge Base approach will facilitate continuous co-operation between software developers and clients. Close co-operation between developers and clients is recognized as a desirable quality control strategy in XP and other methodologies and it can be managed in parallel with testing, as tests are made available to clients. With a Knowledge Base approach, new user categories can be created as required, and given appropriate access to tests. The truth or otherwise of the research proposition will be seen in the quality and quantity of feedback from these users.

### 5 A Concurrent Audit Tool

Figure 1 shows Felix integrated into a Delphi IDE, with Chris Morris's Registry Example open [23]. Visible are a memo in which messages appear, and buttons which indicate methods that we have built, or plan to build.

The **Knowledge Assist** function refers to the knowledge base's profile of the individual programmer, giving a report on his/her history within the current context and wider. An **Annotate** function will allow a human user to create a smart self-triggering message, associated with the current context. The **Context Checker** gives details about the current method or unit compiler hints, warnings and errors, testing history, current programmer, pair programming history.
Other data relevant to the current code context might include:

- notes
- annotations
- expressions of concern (tracked over time, each with originator, date, comment)
- the original specification that it is meant to address
- to-do's for this block
- calls to online help from this context
- count of separate edits (too many changes might show trouble)

Another important function of the Context Checker will be to interface with change management software and assist in component re-engineering.

The Audit function raises concerns for the wider current context, including triggering manual annotations, and metric "smells" identified by data mining.

The Test Audit function brings up Felix’s co-operation with DUnit [29, 23]. This is a framework of classes designed to support the XP approach to software testing, created by The DUnit group at SourceForge and released under the Mozilla Public Licence. We have integrated it with our system, so that Felix can initiate a DUnit testing session, receive reports of the tests returned from it, and inject test audit annotations into the code being tested. Programmers are expected to write in parallel testing units, as described above. The benefits of this methodology are obvious, especially where responsibility for writing the code and its testing unit can be taken by different programmers in cooperation. In order to integrate this methodology with the IDE, Felix injects a test audit annotation on the line immediately following the declaration. We assert that every method should be annotated in this way, even if only to say that testing is not required.

In Figure 2 Felix has called up DUnit, which is listing custom-built tests for code which is to plot a curve in a GUI. A test audit, initiated manually when required, checks declarations in the unit being inspected and reports on them. Methods can be classified (manually) as requiring zero or more tests, or the decision can be deferred. Where test(s) have been performed, the most recent result is recorded in the annotation. When the test status of methods is reported interactively, the current programmer can locate code which has failed tests, write tests for methods marked as requiring them, or reclassify them.

In Figures 3 and 4, output is checked visually for correctness against input parameters in two tests called by DUnit. Output is mathematically correct (Figure 3) but the user gives the method a "fail" on the second interactive test (Figure 4) because the curve is drawn outside the chart. Input parameters could be entered interactively or from file; testing does not have to be interactive. Such tests could be made available to clients for progress reporting and formal acceptance testing.

In Figure 5 the custom-built test unit has passed DUnit's test output to Felix, which has inserted an annotation into the unit code.
6 Significance of the Work

The extraordinary cost of failure in the software quality engineering process is well known. If the analogy with financial audit [2] holds good, concurrent audit techniques can significantly reduce these costs using our approach. The concurrent aspect addresses the need for dynamic feedback [20, 21]. The AI audit aspect expands the brainspace for human users, allowing them to apply human intelligence to code analysis.

Our experimentation has not extended beyond the single desktop, and doubts have been expressed about the scalability of XP. This research is creating a versatile tool, adopting some XP concepts but not an implementation of XP or any specific methodology; nevertheless its scalability depends on whether knowledge-mediated Intelligent Assistant techniques themselves offer the promise of scalability.

Given its add-in nature, we envisage that a project of any size using Delphi as its coding tool would be able to use this experiential process as a risk reduction method.

7 Preliminary Conclusions from the Exploratory Studies

This research demonstrates that known Concurrent Audit techniques can be integrated into the IDE, opening up the possibility of a knowledge base that can respond to the needs of individual programmers and their tasks, and of communications within the development team. It also points to the feasibility of tightly integrating testing with
the development process, and of quality progress communication with all stakeholders.

8 Further Work

Having experimented with extensions to the IDE and with the integration of messages and testing, the next step is to progress the evolution of the Knowledge Base from the first generation design. We would also like to introduce management and client views into the Knowledge Base; for example, to inspect progress, and browse problem areas. This will go towards supporting an evolutionary programming approach, in which clients and developers work together, and changing requirements are managed fluently. Other features for subsequent consideration include integration with the online help system, with an automated build of customised, programmer-specific Help files. It will also be desirable to integrate DUnit further by automatic generation of stub tests in the parallel testing unit.

Acknowledgements

The authors acknowledge the contribution of Lang Hames in research assistance, and the valuable observations and suggestions made by the referees.

References