Harnessing Taguchi Methods in Software Development

R K Gupta and Tapan P Bagchi
NDS InfoServ, Mumbai, India
bagchi@ndsinfo.com

Abstract

Many factors may influence the successful development of software (SW). Most practitioners, however, tend to use their own preferred methods—some ad hoc—for this task. This paper reviews recent attempts of using orthogonal array (OA) experiments to determine the effect of various influencing factors and then their optimum settings to reduce errors in the SW created. Genichi Taguchi popularized such an experimental approach—now known as the Taguchi Method—for hardware design. Indeed, this method has proved to be of high value in delivering superior quality products and manufacturing processes. This paper reviews the SW development scene and examines the potential utility of such a framework to improve SW development. Indications are that this approach can reduce bugs and errors in the SW thus developed, and also make the development and testing processes more effective and efficient.

Introduction

Software (SW) development today holds a prominent place in intellectual enterprise. The public at large often regards SW creation—a "high-tech" activity—to be synonymous with the highest level of sophistication ingrained in any job. Most, however, are unaware that the software creation process details, particularly their quality assurance aspects, still exist at a relatively primitive state—where hardware industry was perhaps 100 years ago. Most part of the SW creation process still depends heavily in *inspection*. By contrast, several HW manufacturers have touched Six Sigma through excellence in design and process control, modeling, optimization, and the use of advanced statistical techniques. SW failures and defects continue to be at a daunting level; methods are still sought to reduce/eliminate defects in the SW created and make defect prevention more effective.

Many challenges in SW creation are indeed unique and quite unlike widgets design and production. Generally, the reliability or performance of a piece of "quality" hardware gradually deteriorates. In SW, on the other hand, *all* its bugs and defects remain *built in* from day zero when the SW is declared ready for the store shelf. Few SW are guaranteed to be defect free by their creators—a characteristic attributed to the "inherent complexity" of the SW product.

In view of this and the critical role played by SW that fly airplanes, manage vast communication networks, produce investment advisory signals, or process utility bills, etc., the methodology or framework for creating a SW is currently under close scrutiny. We have now cast SW development into different life cycle frameworks and the scene is still evolving. Low customer satisfaction, high cost, a great deal of re-work on modules that do not integrate properly, low reliability and unacceptable development delays are too frequent. These have led to this inquiry of the practices that SW developers engage in. Models called Capability Maturity Models (CMMI) have been proposed to guide developers do their job better. SEI process capability models are such artifacts (Dymond 2002). It is now widely noted that error or defect density reduces as maturity goes up (Cote 2005). "Design-test-design", "build-and-fix", "Waterfall", rapid prototyping, incremental development, and extreme programming (XP), spiral model, and others have been proposed and tested—with the single goal of producing SW that reaches the highest level of user satisfaction, gets developed fast, and is not too expensive (Jayaswal and Patton 2006).

Extensive studies of "post mortem" work on SW failures, whether technical or commercial, indicate that SW bugs and errors are almost all created well *upstream*—during the design process. Of course there is the equivalence of "production" as in manufacturing in SW—the coding process—but a lot and indeed a LOT of things happen upstream, during requirements analysis, architecture or technical design, and functional design. Here all the guiding decisions are made, before programming actually begins. As with HW, it has been established that the majority of SW performance problems can be traced to inadequate effort put in at this initial "upstream" stage.

Decisions in setting the level or value or character of many of these factors—e.g. choice of the language, number of processors that will be engaged, memory and buffer use, complexity, etc.—one often banks on experience, for there still isn't a well-established theory available as to how to optimize such decisions proactively, before one taps the keyboard. Quality assurance (QA) in SW development, therefore, continues to be inspection-oriented—generally executed after the fact, to classify the SW to be OK, or defective, in need of rework, rewriting, or just scrapping.

HW folks also walked this path some 50 years ago. But they now use SPC, designed experiments, rapid prototyping, field testing, etc. as well-organized *proactive* and preventive QA strategies, which attempt to perfect the product's design (two-third quality problems being traceable to design defects) before mass production begins. Furthermore, since a SW is intrinsically much more complex than a typical hardware, not all its bugs can ever be found and fixed if we count on testing/inspection as the only QA process. The methods employed in SW creation must have large element of proactivity. That is the focus of this paper.

The Critical Role of Design Excellence in SW Creation

Since SW creation is clearly all design and development with little scope of playing with "manufacturing" factors to improve quality, it is now recognized that everything that can be done to create bug-free SW must be done before coding begins (Jayaswal and Patton 2006). Even the right/wrong approach to coding (e.g. size of submodules, data structure, complexity, etc.) is decided during architectural work and it is best not left to the pure subjectivity of the developer. The focus therefore moves to improving and perfecting *SW design* methods. Vast bodies of knowledge have been created to address this under the umbrella of software engineering (Pressman 2005). Certain practices have been promoted, others discarded, and many innovated using the lessons learned. Unlike electronic devices engineering, however, there are few laws of physics or chemistry to guide here and one often has to rely on generating useful knowledge through expansive experience, or *empirically*—through the process of formally designed experiments. Such experiments form the bedrock of a vast body of knowledge that scientists and engineers have empirically compiled about the behavior of systems that involve (a) multiple control or influencing factors, and/or (b) multiple responses.

A highly regarded body of empirical studies falls under "design of experiments" (Montgomery 2001). It is applied to domains where there are no Newton's or Kirchhoff's laws to guide the designer. Here the designer can assemble the artifact from its components, but he/she cannot set the inherent parameters in it *optimally* (i.e. complete the design) using basic knowledge or well-known cause-effect relationships. An example would be correctly operating a wave soldering machine that is expected to deliver defect-free motherboards as a routine with the various conditions—bath temperature, pre-heat temperature, type of solder, fluxing methods, etc. etc. optimally controlled (Diepstraten 2005). There are no equations available here to help one do the job. SW development, it its entirety, is not much different from this *multiple*-control factor scenario. Whereas there exist several different methods to plan empirical studies involving multiple control factors, a method proposed by Taguchi (1986) is widely popular because of (a) its simplicity, and (b) its effectiveness in quickly optimizing the process. Taguchi's method utilizes an efficient experimental plan (known as Orthogonal

Arrays (OA)) to manipulate the control factors and to guide the running of the special experiments involved here.

Taguchi Experiments

The reader is referred to Phadke (1989) for details on the design of Taguchi experiments. A brief outline is given here. The motivation for conducting these experiments is two-fold: (1) Discover reliably which factor(s) have a strong influence on the quality of the response (for us defects created in the SW developed), and (2) Do it efficiently using the fewest number of experiments. Objective (1) is attained by using experimental plans that are based on sound principles of statistical theory, in particular, design of experiments (Montgomery 1990), while objective (2) is achieved by the use of orthogonal arrays. The method has been immensely useful to industries ranging from AT&T to Du Pont to Toyota. Dell used it to optimize advertising campaigns (Kowalick 2004). The basic steps in Taguchi Method are as follows (Sankar and Thampy 2002):

1. Brainstorm to define a *measurable* performance objective that is in need of improvement. Identify factors—qualitative or quantitative—*speculated* to be influencing this performance (this step requires familiarity with the domain of the process and is often facilitated by the use of the Cause-Effect or Fishbone diagram (Kanchana and Sarma 1999). No one is sure at this point whether a particular factor has *any*, a *strong*, or a *weak* influence of the performance targeted. Next, identify the different levels that a factor can be set at in the actual process.
2. Design or plan your experiments as guided by an appropriate orthogonal array (OA).
3. Assign factors to the OA columns.
4. Run the experiments. This is the step when the process being studied is actually run end to end (under conditions as specified in Step 3) and the response (performance of interest) is measured.
5. Analyze the results. This step determines the factor effects and the optimum level (setting) of each factor that when applied together other similarly identified settings leads to best performance.
6. Run the confirmation experiment. This step attempts to verify whether the "optimum" factor settings identified in Step 5 indeed lead to best or at least improved performance. This makes the results reproducible under normal operation.

The goal in these steps is to formulate the process improvement objective, identify the influencing factors, and determine their optimum levels which when combined produce best process results.

To illustrate the method we utilize the results of SW design experiments reported by Kanchana and Sarma (1999) who used the Taguchi methodology to explore the opportunities for SW quality enhancement in the performance of an airborne surveillance system's SW. The goal was to reduce the average numbers of errors in the SW module crafted. They investigated the effect of the number of requirements/module, Cyclomatic complexity and coupling—factors suspected to influence errors—using an orthogonal array.

Brainstorming led to a three-factor cause-effect diagram shown in Figure 1. For each factor, three possible levels could be chosen from while no one could tell *a priori* which of these 3×3×3 = 27 combinations would give the best (lowest average error) performance. Table 1 displays the possible settings for each of these factors. The L_9 OA (Phadke 1989) shown in Table 2 was chosen to guide the Taguchi experiments.

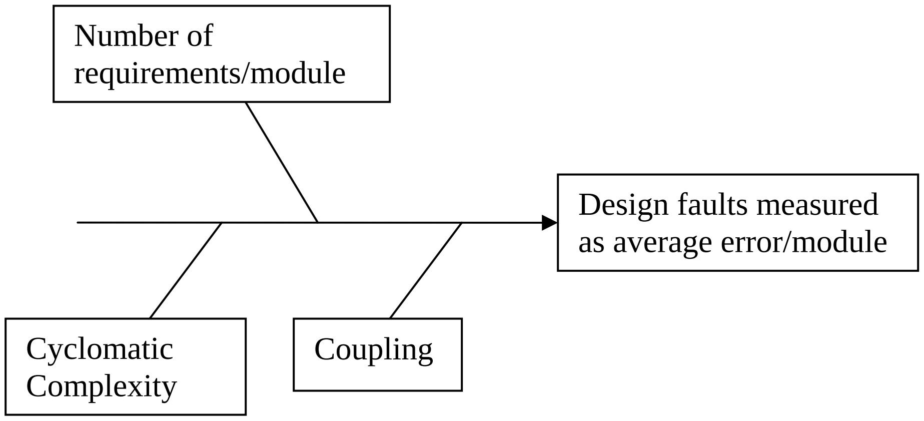
Number of requirements/module
Design faults measured as average error/module
Cyclomatic Complexity
Coupling

Figure 1: The Cause-Effect Diagram

Table 1: SW Design Factors and their Possible Settings

Factor Name	Label	Level 1	Level 2	Level 3
Number of Requirements/Module	"A"	1	2	> 2
Cyclomatic Complexity	"B"	< 5	5 – 10	> 10
Coupling	"C"	Data stamp and Control	Common and Content	Common and Content

Table 2: Experimental OA and Results

Experiment#	"A"	"B"	"C"	Response: Avg Error/Module
1	1	1	1	0.2
2	1	2	2	0.6
3	1	3	3	2.0
4	2	1	2	0.5
5	2	2	3	1.33
6	2	3	1	2.0
7	3	1	3	1.0
8	3	2	1	1.66
9	3	3	2	3.33

These tasks complete Steps 1 to 3 given above. Step 4 involved conducting the actual experiments, i.e., developing SW modules as per the factor settings prescribed by Table 2. The corresponding "response" results (average error/module) appear in Column 5 of Table 2. Our analysis to determine the factor effects on the SW creation process could now begin.

The use of the OA experimental framework makes the determination of factor effects quite straightforward (Phadke 1989, Taguchi, Chowdhury and Wu 2004). These are shown in Table 3, along with a graphical view of the results in Figure 2.

Table 3: Factor Effects Calculated

Factor	Avg Effect at Level 1	Avg Effect at Level 2	Avg Effect at Level 3
Number of Requirements/Module	1.00	1.28	2.00
Cyclomatic Complexity	0.57	1.20	2.44
Coupling	1.29	1.48	1.44

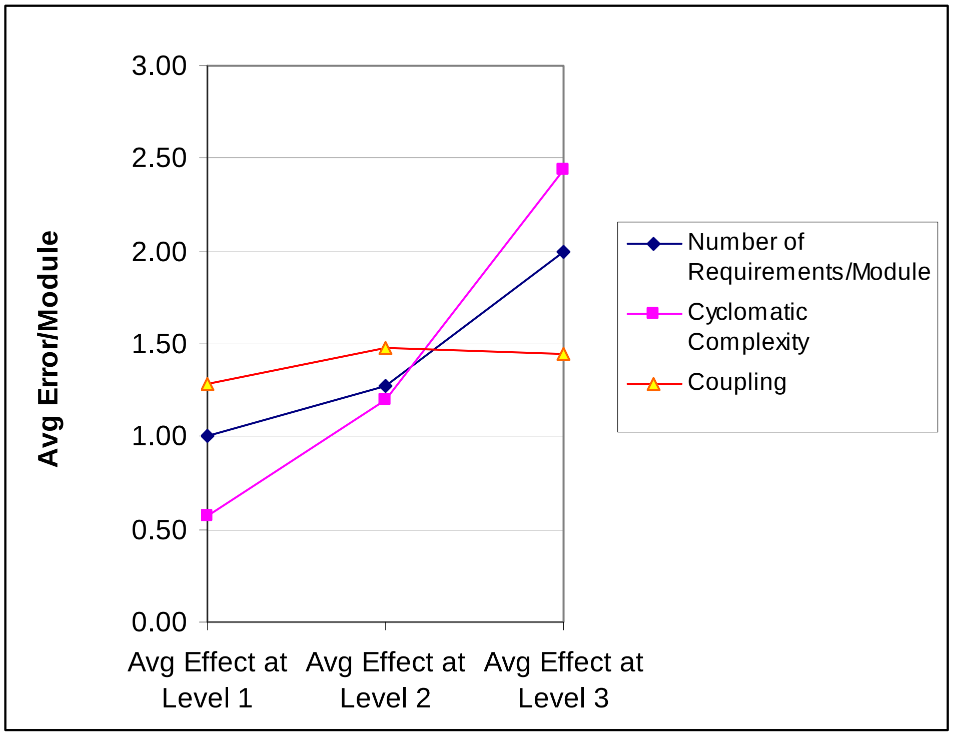

Figure 2: Graphical Representation of Factor Effects

Figure 2 makes the interpretation of factor effects or the influence of each of the three design factors on the resulting errors in the SW modules produced uncomplicated. It is evident that Cyclomatic Complexity dominates the scene (varying Complexity over the range shown in Table 1 has the largest impact on errors produced) whereas Coupling only mildly effects error production. In the overall SW creation process, this investigation suggests, keeping each factor at its "Level 1" should produce minimum errors. This last assertion now becomes the target of Step 6 (running the *confirmation* experiment). Kanchana and Sarma (1999) report that the confirmation experiment run this way indeed confirmed this assertion. (If this would not be true, one would suspect higher order interactions of factor effects to be active. In that case one would require full factorial or other methods rather than the simple L_9 OA as used here to help minimize errors in the SW being produced. See Bagchi 1993 or Montgomery 2001.)

These results closely parallel the experience hardware designers have in HW design or process optimization. Such examples are numerous. Examples in SW design domain are still rather few. However, some creative applications of OA experiments have been already made—particularly in *test case* generation.

OA in SW Test Case Generation

SW testing continues to be the predominant means to assure the quality of SW products delivered. Phadke (1997) pointed out that many SW faults escape conventional testing and indicated the use of exhaustive testing as infeasible in many situations. He suggested the use of orthogonal arrays to generate test cases. A recent white paper (Ponnusamy 2007) echoes this and it provides a good overview of the manner in which OAs may be used to plan SW testing effectively, rather than attempting to design test cases that cover all possible combinations of inputs that may be involved. While describing the verification suite development the author points out that typically test cases development consumes 60% of the testing time. In one illustration provided where the goal was to cover some feature of a RDBMS, 36 (as opposed to 117 conventionally planned) test cases were produced by an OA with two additional cases added for testing some special features. Being founded on sound statistical theory of design of experiments (Montgomery 2001), OAs greatly reduced the number of combinations while they maximized coverage.

In showing the benefits of using OAs the author reports that execution time for the existing suite was 8 hours whereas the OA covered it in 25 minutes. Also, while 117 existing cases checked 70% of the features, 36 cases of the OA covered 95% of these—a remarkable feat of efficiency that perhaps should not be overlooked. Noted also is the fact that the 36 cases involved 1367 LOC while the 117 cases used 28,122 LOC to test the same features. Shankar and Thampy (2002) describe how OAs may speed up configuration testing—a modern SW is expected to work in a variety of HW platforms, and different OS and support libraries. User interface testing is another area where effective test cases can be provided by OAs rather than one's attempt to test features exhaustively. Performance tuning is another area indicated. Lastly, OAs can also make regression testing efficient, for the method is simple yet thorough in its coverage. The authors provide two case studies, the first about formatting a flash memory card involving 10 factors with two levels each, which used the L_{16} OA (Phadke 1989), i.e. 16 test cases only while the conventional approach would use 44 test cases. The second case study was on the design of a GUI tool involving nine design factors with three levels each. The OA involved here was L_{27}. The authors note that conventional testing would leave out many important cases that got covered satisfactorily by the OA method.

Efficiency aside, appropriately designed OAs can uncover the suspected effect of *interaction* between factor effects, an aspect difficult to detect by randomly designed test cases. The theory here is quite rich and uses "linear graphs" in the best suited assignment of experimental factors to the OA columns (Bagchi 1993). Dalal and Mallows (1998) discuss the issues in factor coverage by OAs.

Performance Improvement by OA

Professor H N Mahabala (2007) has provided a cause-effect diagram as the starting point of SW system performance improvement or performance tuning. To the best of the authors' knowledge, such a study has not been conducted yet. Such a study is under way using actual HW/SW configurations at NDS InfoServ, Mumbai. In Figure 3 we show the factors involved in this study.

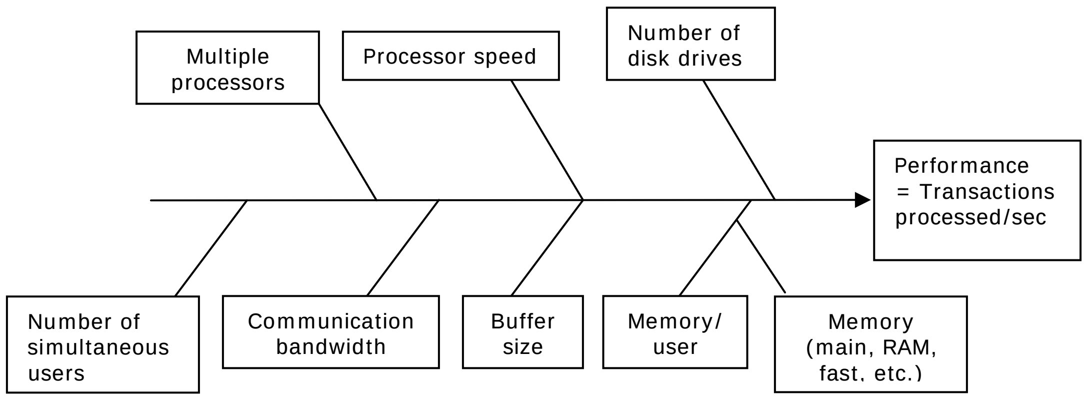

Figure 3: Factors possibly influencing System Performance

Concluding Remarks

OAs provide a remarkably *easy-to-understand and apply* framework for SW defect reduction. It has been lamented that the SW development world is stuck at inspection whereas our HW colleagues have marched well into the domain of Six Sigma. Till we internalize superior process methods to proactively control the possibility of error creation, we should maximize the use of the knowledge and means that are already available to help SW designers, SW architects, and coders who make critical design and related decisions on ad hoc basis for want of more powerful approaches.

This paper illustrates one such approach—the Taguchi Method of experimentally determining the "right" answers here. Indeed the macro factors thus identified and optimally set can reduce a great deal of hunting, test runs and delays in product delivery. Even on the smart coverage of the testing domain, evidence now exists that methods such as the use of Taguchi's OAs can greatly reduce the effort expended.

References

Bagchi, Tapan P (1993). *Taguchi Methods Explained – A Practical Guide to Robust Design*, Prentice-Hall (India)

Bagchi, Tapan P (1999). *Mutiobjective Scheduling using Genetic Algorithms*, Kluwer

Cote, M-A, W Suryn and E Georgiadou (2005). Software Quality Model Requirements for Software Quality Engineering, *Software Quality Professional*, 6(3), pp. 4-17.

Dalal, Siddhartha R and Colin L Mallows (1998). Factor-Covering Designs for Testing Software, Technometrics, Vol 40, 3, pp. 234-243.

Diepstraten G (2005). Analyzing Lead-free Soldering Defects in Wave Soldering using Taguchi Methods, http://www.zetech.co.za/

Dymond, KM (2002). A Guide to the CMM, *Process Transition International*, pp. 1-4.

Jayaswal, Bijay K and Peter C Patton (2006). *Design for Trustworthy Software: Tools, Techniques, and Methodology of Developing Robust Software*, Pearson Education.

Kanchana, B and VVS Sarma (1999). Software Quality Enhancement through Software Process Optimization using Taguchi Methods, IEEE

Kowalick, D (2004). Dell Employee Purchase Program (EPP) Email Advertising Project: Taguchi Optimization Project, http://www.AdEvaluator.com/adtech

Mahabala, H N (2007). *Personal communications*

Montgomery, D C (2001). *Design and Analysis of Experiments*, 5th ed., Wiley.

Ponnusamy, S (2007). Orthogonal Array: Applicability in Software Product Testing, *White Paper*, Wipro Technologies.

Phadke, M S (1989). *Quality Engineering using Robust Design*, Prentice-Hall.

Phadke, M S (1997). Planning Efficient Software Tests, http://www.stsc.hill.af.mil/crosstalk/1997/10

Sankar Unni and Deepa Thampy (2002). Applying Taguchi Methods in Software Product Engineering, *SEPG Conference Proceedings*.

Taguchi, G (1986). *Introduction to Quality Engineering*, APO.

Taguchi G, S Chowdhury and Y Wu (2004). *Taguchi's Quality Engineering Handbook*, Jossey-Bass.