International Journal of Information, Control and Computer Sciences
ISSN: 2517-9942
Vol:2, No:3, 2008

A Pattern Language for Software Debugging

Mehdi Amoui, Mohammad Zarafshan, and Caro Lucas

***Abstract*—**In spite of all advancement in software testing, debugging remains a labor-intensive, manual, time consuming, and error prone process. A candidate solution to enhance debugging process is to fuse it with testing process. To achieve this integration, a possible solution may be categorizing common software tests and errors followed by the effort on fixing the errors through general solutions for each test/error pair. Our approach to address this issue is based on Christopher Alexander's pattern and pattern language concepts. The patterns in this language are grouped into three major sections and connect the three concepts of test, error, and debug. These patterns and their hierarchical relationship shape a pattern language that introduces a solution to solve software errors in a known testing context.

Finally, we will introduce our developed framework ADE as a sample implementation to support a pattern of proposed language, which aims to automate the whole process of evolving software design via evolutionary methods.

***Keywords*—**Coding Errors, Software debugging, Testing, Patterns, Pattern Language

I. Introduction

TESTING and debugging are two crucial parts of software development. Actually, in typical software projects, 50% of resources are dedicated to testing, and in safety-critical systems this ratio could be raised to 80% [1]. Alan Turing early in 50's wrote an article entitled "Checking out a Large Routine", introducing testing algorithms in a software development process [2], and since then various testing techniques are developed. In addition, several contributions are proposed to classify these techniques [6]. Although these classifications differ in an abstraction level of software as well as the aspects of software under test, they are introduced to provide a complete solution for a set of vital tests. However, because of the huge input domain space of complex systems, in most cases it is impossible to verify the correctness of software systems for their behavior and structure. As a result, many test automation techniques have been proposed to ease and reduce the cost of test process [7, 8]. Most of these techniques are based on search-based techniques whereas others use random test [9].

In spite of all advancement in software testing, debugging remains a labor-intensive, manual, time consuming, and error prone process. In fact, a report in 1997 claimed that many programmers still prefer the manual insertion of "print" statements as their debugging technique of choice [3]. In addition, an informal survey in the same year indicates that manual techniques, such as inserting print statements, manually executing a test case, or inserting breakpoints, accounted for 78% of real-world programmers' attempts to solve exceptionally difficult bugs [4, 5].

As testing and debugging are tightly coupled, a candidate solution to enhance debugging process is to fuse it with testing process. To achieve this integration, a possible solution may be categorizing common software tests and errors. This solution is followed by the effort on fixing the errors through looking up a table of solutions for each test/error pair. The categories themselves may be described using subcategories that are more detailed. Using this approach, we could form a tree of related solutions that are grouped based on their context and problem. Ultimately, this tree could able us to define patterns that cover each category and its subcategories.

Our approach to address claimed issue is based on Christopher Alexander's pattern and pattern language concept [10]. A pattern is a piece of literature that describes a problem and a general solution for the problem in a particular context [11]. Each pattern is a three-part rule, which expresses a relation between a certain context, a problem, and a solution. The pattern is, in short, a *solution* to a *problem* in a *context*, also includes the rules, which tell us when this pattern happens. Every pattern contains a description for each part of its elements, and an example that helps to illustrate the pattern [12]. Moreover, a pattern language is a collection of patterns that are related to each other by virtue of solving the same problems or parts of a solution to a larger, partitioned problem [10]. Therefore, our patterns and their hierarchical relationship shape a pattern language that introduces a solution to solve software errors in a known testing context.

This paper is organized as follows: In section II pattern language for software debugging is proposed. Section III introduces patterns of this language and finally section IV explains the advantage of this pattern language and its possible applications in further works.

II. Pattern Language

As test and debug are general terms used in various fields of engineering, the patterns of this language focus on a software system regardless of other miscellaneous systems. The

Manuscript received December 14, 2005.

M. Amoui is with the Control and Intelligent Processing Center of Excellence, Electrical and Computer Eng. Dept., University of Tehran, Tehran 11365-4563, IRAN (phone: +98-912-3161002; fax: +98-21-66485489; e-mail: mehdi.amoui@ece.ut.ac.ir).

M. Zarafshan is with the University of Tehran, Tehran 11365-4563, IRAN (e-mail: mohammad.zarafshan@ece.ut.ac.ir).

C. Lucas is with the Control and Intelligent Processing Center of Excellence, Electrical and Computer Eng. Dept., University of Tehran, Tehran 11365-4563, IRAN (e-mail: lucas@ipm.ir).

patterns in our language are grouped and named into three major sections according to diverse set of test **contexts**. Each section starts with a brief summary that introduces the patterns described in the section. The patterns and their relationships in the language are depicted graphically in Figure 1, and summarized at the end of this paper in the Appendix as a quick reference. These sections are:

- Section *A*, **Software Patterns,** This section includes the two general patterns of behavioral and structural debugging. These two patterns are the parent of all patterns introduced in this pattern language. From another point of view, this section itself is the parent of the two other sections.
- Section *B*, **Program Patterns,** The patterns belonging to this section cover a run-time model of software called program. The tests in this section vary from non-functional to fully functional tests.
- Section *C*, **Code Patterns,** Patterns in this group deal with testing the source code of software under test. The code will be test for its structure, quality assurance, coding standards and compilation test. Optional test here may include model compatibility check, design pattern matching and code metrics.

Although program and code sections break down the **context** of software testing into two distinct partitions, there are quite a few test techniques, such as gray-box [27], which combines both behavior and structural information for the purpose of testing [9]. As a result, an additional program-code section may be added to this list.

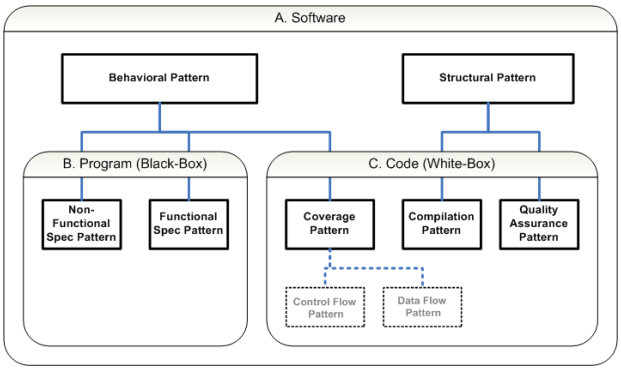

Fig. 1 Debugging Pattern Language

A. Conventions

This pattern language is written based on *Pattern Writing* pattern language developed by Meszaros and Doble [28]. Therefore, the patterns presented here are developed in reference to the solutions provided in *Pattern Writing* patterns. According to this pattern language, pattern names are indicated in *italics* and pattern terms are indicated in **bold**.

Moreover, all patterns in this language are developed regarding to a unique rule. According to this rule, the pattern **context** is the testing condition and the **problem** is a collection of errors generated after performing a test defined in **context.** Finally, the **solution** covers the debugging procedure in a specific software test. Figure 2 illustrate this rule.

B. How to Use These Patterns

This language is considered as a catalog of patterns. To find a solution for a particular coding error problem, one should refer to *'Patterns Quick Reference'*, and then check if any of the problems resemble the one we are trying to solve. Once a pattern of interest is determined, one could look at the **Context** and **Forces** parts for guidance on determining whether the pattern is applicable to the specific situation. We also developed an **Example** for each pattern but to get further appreciation of the nuances, the patterns can be improved by adding **Rationale**, **Resulting Context** and **Related Patterns** parts.

III. Patterns

A. Software Patterns

A.1 Pattern: Behavioral

Context: Perform one or more tests on software to ensure that it behaves exactly as it should, according to its specification or customer requirements.

Problem: Software does not behave as it should, due to unexpected error or undesired output.

Forces:

- To detect behavioral errors, an executable version of software is required; so the software should pass compilation tests.
- Software specification is required.
- Final user could be considered as a co-tester.
- Enterprise software systems are complex -up to 10^{20} states in a large system- that testers are not currently able to test software well enough to insure its correct operation [13].

Solution: Examine the type of specification errors; divide them as functional or non-functional error. Refer to *functional spec pattern* in case of functional spec error and refer to *non-functional spec pattern* in case of non-functional spec error. For performing more precise test and debug procedure, Coverage Pattern is also suitable.

A.2 Pattern: Structural

Context: Software structure may be presented in various abstraction levels and notations. The two most common representations of software structure are a source code and a design model. Any test process that deal with these representations and the relation among them is considered as a structural test.

Problem: There is a structural error or a rule violation detected by a test procedure.

Forces:

- Source code should be accessible.
- Structural testing and verification in large and complex software, is highly dependant on test automation tools.
- Although each structural representation demonstrates an aspect of software, they all need to be coordinated.
- Reverse engineering could be used to check model compatibility.

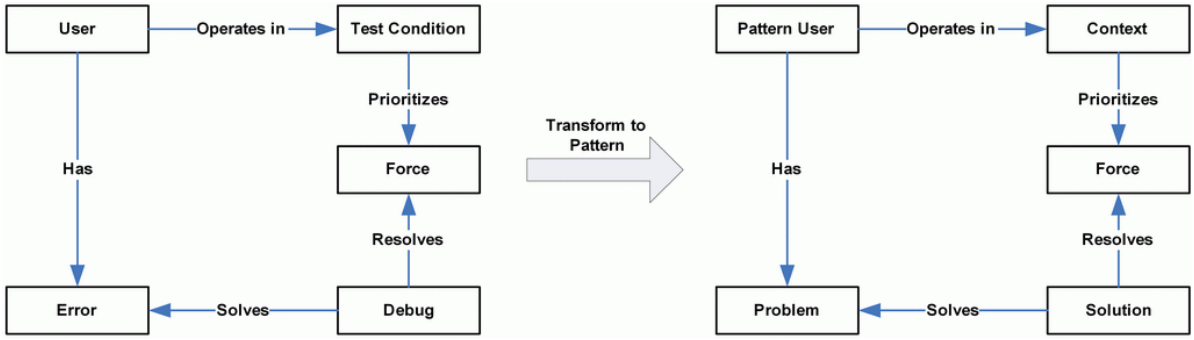

Fig. 2 Transformations to pattern form

Solution: Check the type of error and its level of abstraction. Then refer to *compilation pattern* in case of compile error and refer to *quality assurance pattern* in case of coding rules violation or bad metrics. In addition, to resolve incompatibility of code and model, CASE tools are helpful to detect discordant parts.

B. Black-Box (Program) Patterns

Black-box testing refers to the testing techniques that assume no knowledge of software internal structure or implementation. Usually this kind of test uses an executable program as a test input. The two most prominent black-box approaches are functional testing and non-functional testing. Moreover, user acceptance tests take place in this category, and may extensively assist the development team to find program errors. In particular, testing to ensure that the software correctly processes different parts of its input and output domains is an important black-box activity [14].

B1. Pattern: Functional Spec

Context: The ranges of tests that assume the program as a black-box are under consideration. The only important factor in functional testing is the relation between software input and its corresponding output.

Problem: According to software spec, there is at least one test case where the software output is not in correct relation with given input or there is no output at all. In case of simulating the input/output relation by oracle, the oracle's output and program being tested are not similar with a same input.

Forces:

- This relation needs to be the same as the input/output relation defined in software spec or by the software oracle.
- In some cases software spec may not be available.
- Software oracle may be developed by artificial neural networks [15].
- Intelligent and search based methods are useful for input domain reduction [9].
- Early Unit testing could help for early functional error detection as suggested in extreme programming and some other lightweight methodologies.
- Some functional errors are because of bad design model.

Solution: The list of functional errors should be created. The items in this list need to be sorted according to their severity, priority or customers requests. The Developer is responsible to fix functional errors due to bad implementation. The test and debug procedures have to repeated in iteration, until all functional errors fixed. A possible solution for functional error reduction at test time is early unit testing. In early testing developers forced to deliver those modules that pass all unit tests; so this will help error detection in large and complex software.

Example: Please refer to ref. [16] for early unit testing examples in extreme programming methodology.

B2. Pattern: Non-Functional Spec

Context: According to software spec, there may be one or more non-functional properties that the developed software should satisfy them. These non-functional properties imply on the final program and usually the complete and precise test of them needs a program running in a real and final environment. The major non-functional properties are those that defined for safety-critical and real-time software.

Problem: The software failed at one or more non-functional tests.

Forces:

- Applying the non-functional property to current code is costly or not possible.
- Non-functional test results strongly depend on a run-time environment and conditions.
- There is a need of overall software implementation view to fix non-functional errors.
- There is a limited tool support for automating the non-functional test and debug process.

Solution: Several transformation processes, such as normalization, program optimization, restructuring, etc., can extensively improve the non-functional properties. On the other hand, each non-functional property could be considered as an independent software aspect. These aspects are usually in contradiction. Therefore, system testers may enable or disable each aspect in a system to inspect overall system non-functional behavior. Finally, the tester is able to suggest the best usage of these properties.

Example: When building embedded systems from components, those components must interoperate, satisfy various dependencies [17], and meet non-functional requirements. The VEST toolkit can substantially improve the development, implementation and evaluation of these systems. The toolkit focuses on using language independent notions of

aspects to deal with non-functional properties, and is geared to distributed embedded system issues that include application domain specific code, middleware, the OS, prescriptive aspects, and the hardware platform [18].

C. White-Box (Code) Patterns

White-box testing (also known as glass-box or clear-box testing) is a testing methodology that explicitly makes use of the structure of a program. The goal is to increase the chances of finding errors in software by effectively increasing the density of errors. White-box testing schemes investigate program structures that are more likely to be problematic, and ensure that the entire program is tested [14, 19]. Some white-box approaches include dataflow testing, partition testing, symbolic execution, state-based testing, program slicing, and mutation testing [5] that most of them are various type of coverage testing.

C1. Pattern: Compilation

Context: Compilers as a tool, transform a source code written in a specific programming language, into an executable machine code. Although this transformation because of variety errors such as lexical, syntax and semantic may not be successful. In case of failure, instead of an executable program, compiler output would be a list of different compile errors.

Problem: In modern compilers, after an unsuccessful compilation, the compiler output would be a list of compile-time errors. Each item in this list has an error id, line number of occurred error and usually a short description of error.

Forces:

- Because each semantic error contains one or more lexical or syntax errors it would be hard to determine the correct source of error.
- Cascadeable errors usually causes incorrect error list.
- Some errors may need to fix by specifying necessary compile directives.

Solution: Most compilation errors are caused due to an incorrect use of the programming language. A clear understanding of syntax and semantic specification of the target programming language is required.

Example: Consider the following Java statements:

```
switch ((char) chosen) {
   cas e '1': {
      bw.write("Max Stat:\n + myMax.toString());
      bw.write("\naverage Stat:\n + myavg.toString());
      …
```

The report in Table 1 is generated by Eclipse platform. It shows the compilation error and its possible solution suggested by compiler.

TABLE I
SAMPLE JAVA COMPILER ERROR

Severity	Description	Resource	In Folder	Location	TimeStamp
2	Syntax error on tokens, they can be merge to form case	MyEightPuzzle.java	EightPuzzle	line 59	September 1, 2005 3:10:12 AM

C2. Pattern: Quality Assurance

Context: Quality Assurance is an ongoing comparison of the actual quality of a product with its expected quality. In the field of software development, software metrics are measured at various points in the development cycle, and utilized to guide testing and quality improvement efforts [20, 21]. These metrics are also used by program managers to track the status of a project; these metrics tend to be related to cost and schedule, rather than source code [22, 23].

Problem: Software fails on one or more quality assurance tests. The failure may because of coding standard violation, model compatibility errors, metric check errors or etc.

Forces:

- Fixing quality assurance violence shall preserve software behavior.

Solution: By defining a set of code refactorings, a developer would be able to change code structure in such a way that satisfies quality assurance tests. These refactoring sets could be structured in a way that enforces a design pattern of GOF in code to enhance code maintainability, readability and structure improvement [24].

Example: Fixing audit violations: Table 2 illustrates sample audit test error that is detected by Together case tool and its possible solution [16]. Also in the next section will propose a sample implementation of this pattern.

TABLE II
SAMPLE QUALITY ASSURANCE ERROR

Problem	Package AirlinerPD;
	import java.util.*;
	import java.util.ArrayList;
	public class Flight {
	public void makeReservation(String name, int tKing) {
	…

Rationale	The audit selected above complains that the import statement is too broad: **import java.util.*;**
Solution	We had inserted that import statement into our code so that the **Flight** class could have instance variables of type **Date**. **import java.util.Date;**(replacing **import java.util.*;**)

C3. Pattern: Coverage Errors

Context: White-box testing is a method to test the internal structure of the written software. To use this method it would be necessary to test the source code of the developed software. The goal of this method is to guarantee that all written statements of the under-test software has been fully investigated, so there would be no unexpected behavior due to untraced branches in code which is named coverage testing. Either structural coverage criteria can be categorized as control or data flow based with various demand at different levels [11].

Problem: After performing a white-box test there are different coverage errors in control flow or data flow structure

of the software. These errors are categorized based on the level of coverage test.

Forces:

- Coverage test will uncover faults of superfluous implementation [25].
- White-box testing could not expose all behavior errors.
- No automation tool is available to debug coverage errors.
- Intelligent and search based methods are useful for code coverage [9].

Solution: These kinds of errors should be solved manually by programmers [9]. Reviewing a list of revealed coverage errors and exploring the code is needed. Then after, the implementation should be approved to handle uncovered parts of the code. Please pay attention that this solution depends on the level of demand that specified in **context**.

Example: Consider the Java statement presented in Fig. 3.

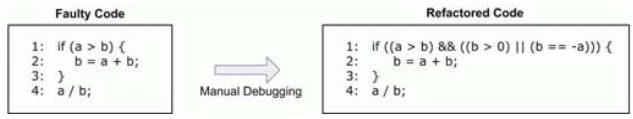

Fig. 3 Code coverage sample

Regardless of the value of a or b, executing line 1 at least counts it towards the coverage measurement. If a=10 and b=9, this test case will cause lines 1-4 to be executed successfully, yielding 100% coverage. However, further testing with the case where a = 10 and b = -10, values that will cause line 4 to fail, will never be considered.

The problem arises in programming languages that uses short-circuit operators. For example in Java, lines 1-3 will be executed as long as a > b and b > 0. They will not be executed if a <= b Therefore, the expression b = -a will never be invoked, so the tester will never know that the expression should instead be b != a until values like a = 10 and b = -10 appear [26].

In addition, there are many other kinds of coverage at different coverage testing levels. Similar examples may illustrate these techniques.

IV. Pattern Implementation

In this section, we will propose a sample implementation of *Quality Assurance* pattern. To do so, we introduce a search-based evolutionary method to find the best sequence of valid high-level design pattern transformations to improve software reusability while trying to preserve other aspects of software quality. Our developed framework, Automatic Design Enhancer (ADE), designed to automate the whole process of evolving software design via genetic algorithm, consists of three major subsystems including *Transformer engine*, *Design Metrics Evaluator engine*, and *Genetic Algorithm engine*. Fig. 4 illustrates the block diagram of this framework.

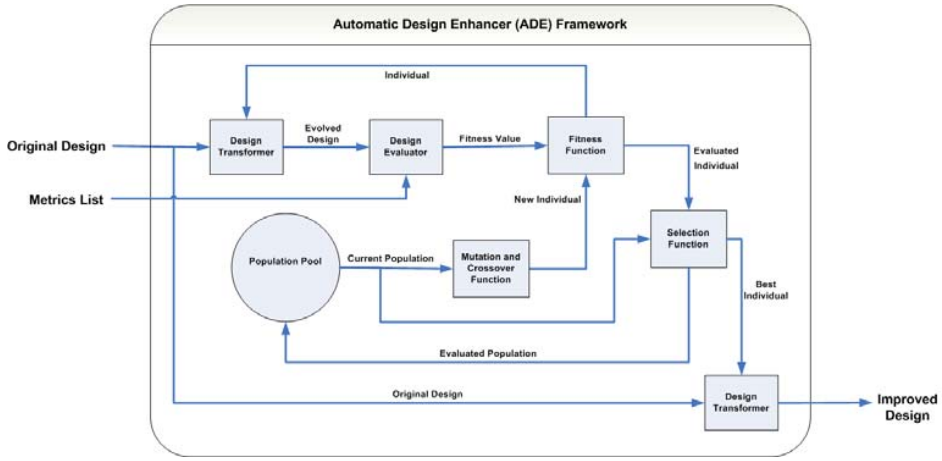

Fig. 4 Automatic Design Enhancer (ADE) Framework Schema

V. Conclusion

A debugging pattern language proposed in this paper, connects the three concept of test, error, and debug which provides a catalog of solutions for particular coding error problems. This language attempts to formalize the debugging strategies followed by software developers, and maintainers. However, due to the size of our language that is rather small, it may be impossible to prove the complitude of the language at this stage.

In contrast, it is still easy to extend the language by adding new patterns to each section. For instance, extension of *coverage* pattern to *control flow*, and *data flow* patterns is illustrated in figure 1.

The most important difference among these new patterns and their parents would be the abstraction level of their corresponding elements, which cover more specific scopes. Users of the language can add new patterns according to their own needs, and they are not obliged to use the current language.

As an improvement, to step forward to automate debugging process, it is possible to write the **solutions** in a pseudo code format, which makes the implementation of **solutions** easier, and helps the user dividing each **solution** into set of more simple actions that are easier to implement.

We believe it is possible to build a framework that allows users to employ debugging patterns in evolving faulty software by Refactoring, for example, we have propose an ADE framework to address this issue for *Quality Assurance Pattern*. The result of such tools will be a tool for debugging the errors found during test process and producing more extensible and reusable software testing/debugging mechanism.

REFERENCES

[1] Brooks, F.P., Jr., "The Mythical Man-Month: Essays on Software Engineering," Anniversary Edition, Reading, MA: Addison-Wesley Pub., 1995.
[2] Miller, E.F., Jr., "Program testing: guest editor's introduction," IEEE Computer, vol. 11 no. 4, April 1978, pp. 10-12.
[3] Lieberman, H., "The debugging scandal and what to do about it," Communications of the ACM, vol. 40 no. 4, April 1997, pp. 26-29.
[4] Eisenstadt, M., "My hairiest bug war stories," Communications of the ACM, vol. 40 no. 4, April 1997, pp. 30-37.
[5] Dick S. H., "Computational Intelligence in Software Quality Assurance," PhD Thesis, Department of Computer Science and Engineering, College of Engineering, University of South Florida, 2002.
[6] Grindal, M.; Offutt, J.; Andler, SF., "Combination testing strategies: a survey," Software Testing, Verification and Reliability, vol 15 no. 3, 2005, pp. 167–199.
[7] Weyuker, E. J.; Weiss, S. N.; and Hamlet, D., "Comparison of program testing strategies," In Proceedings Fourth Symposium on Software Testing, Analysis, and Verification, ACM Press, Oct. 1991, pp. 1-10.
[8] Tonella, P., "Evolutionary testing of classes," in Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis, Boston, MA, July 2004, pp. 119–128.
[9] McMinn, P., "Search-based software test data generation: A survey," Journal of Software Testing, Verification and Reliability, vol. 14 no. 2, June 2004, pp. 105–156.
[10] Christopher Alexander et al., "A Pattern Language," Oxford University Press, New York, 1977.
[11] Coplien, J., "Software Patterns," SIGS books, 1996.
[12] Christopher Alexander, "The Timeless Way of Building," Oxford University Press, New York, 1979.
[13] Friedman, M.A.; Voas, J.M., "Software Assessment: Reliability, Safety, Testability," New York: John Wiley & Sons, Inc., 1995.
[14] Peters, J.F.; Pedrycz, W., "Software Engineering: An Engineering Approach," New York: John Wiley & Sons, 2000.
[15] Saraph, P.; Kandel, A., "Test Case Generation and Reduction by Automated Input-Output Analysis," IEEE, 2003
[16] TogetherSoft, TogetherSoft Home Page, http://www.borland.com/together/index.html
[17] Gray, J.; Bapty, T.; Neema, S.; and Tuck, J., "Handling Crosscutting Constraints In Domain Specific," Modeling, CACM, Vol. 44 no. 10, 2001.
[18] Stankovic, J. A.; Zhu, R.; Poornalingam, R.; Lu, C.; Yu, Z.; Humphrey, M.; and Ellis, B., "Vest: an aspect-based composition tool for real-time systems," in Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium, 2003, pp. 58–69.
[19] Duran, J.W.; Wiorkowski, J.J., "Quantifying software validity by sampling," IEEE Transactions on Reliability, vol. 29 no. 2, June 1980, pp. 141-144.
[20] De Almeida, M.A.; Lounis, H.; Melo, W.L., "An investigation on the use of machine learned models for estimating software correctability," International Journal of Software Engineering and Knowledge Engineering, vol. 9 no. 5, 1999, pp. 565-593.
[21] Ebert, C.; Baisch, E., "Knowledge-based techniques for software quality management," in W. Pedrycz, W.; Peters, J.F., Eds., Computational Intelligence in Software Engineering, River Edge, NJ: World Scientific, 1998, pp. 295-320.
[22] Sedigh-Ali, S.; Ghafoor, A.; Paul, R.A., "Software engineering metrics for COTS-based systems," IEEE Computer, vol. 35 no. 5, May 2001, pp. 44-50.
[23] Brown, N., "Industrial-strength management strategies," IEEE Software, vol. 13 no. 4, July 1996, pp. 94-103.
[24] Tokuda, L., "Evolving Object-Oriented Designs with Refactorings," Ph.D. thesis, University of Texas at Austin, 1999.
[25] McMinn, P., "Improving Evolutionary Testing in the Presence of State Behavior," Ph.D thesis, University of Sheffield, 2002.
[26] Agustin, J. M., "Improving software quality through extreme coverage with JBlanket," M.S. Thesis CSDL-02-06, Department of Information and Computer Sciences, University of Hawaii, Honolulu, 2003.
[27] B. Korel, A. M. Al-Yami, "Assertion-oriented automated test data generation," In Proceedings of the 18th International Conference on Software Engineering (ICSE), 1996, pp. 71-80
[28] G. Meszaros, J. Doble, "A Pattern Language for Pattern Writing,",

APPENDIX (PATTERNS QUICK REFERENCE)

Problem	Solution	Pattern Name
Behavioral error	• Divide them as functional or non-functional error. • Refer to *functional spec pattern* in case of functional error • Refer to *non-functional spec pattern* in case of non-functional error	*Behavioral*
Structural error	• Refer to *compilation pattern* in case of compile error • Refer to *quality assurance pattern* in case of coding rules violation.	*Structural*
Functional test error	• Fix all functional errors due to bad implementation. • Detect faulty design models with designer's assistantship. • Perform early unit testing for functional error reduction	*Functional Spec*
Non-functional test error	• Consider each non-functional property as an independent software aspect. • Enable or disable each aspect in a system to inspect the system behavior. • Suggest the best usage of non-functional properties to developers.	*Non-Functional Spec*
Compilation error	• Developer is responsible to fix all compilation errors according to the each error's id and its correspondent line of occurrence.	*Compilation*
Quality assurance, standard or design violation	• Define a set of refactorings or use predefined set. • Refactor code structure in such a way that satisfies quality assurance tests according to refactoring set.	*Quality Assurance*
Code coverage error	• These kinds of errors should be solved manually by programmers [9]. • Reviewing a list of revealed coverage errors and exploring the code is needed. • Then after, the implementation should be approved to handle uncovered parts of the code.	*Coverage*