Designing Software to Reduce Cost of Testing

Neelam Gupta

Department of Computer Science
University of Arizona
Tucson, AZ 85721
ngupta@cs.arizona.edu

Len Bass

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213
ljb@sei.cmu.edu

Abstract

Software testing is an important and expensive component of the software develoment life cycle. The testing community has always treated the design of the software to be tested as an input over which they have no control. In this paper, we propose a new approach to reduce the cost of integration testing by influencing the design of the system to be tested. We consider the simple pipe and filter architecture style and analyse its testability for integration testing. Our analysis shows that the size of test suite required for integration testing is a linear function of the number of modules in pipe and filter architecture style. In contrast, the size of test suite required for a general design, with arbitrary communication among its modules, is an exponential function of the number the modules in the design. This illustrates that the cost of the testing stage can be significantly reduced by appropriate selection of the architecture style during the design stage.

Keywords Software Architecture Style, Software Testing, Pipe and Filter

1 Introduction

Software testing is a critical element of software quality assurance. It is an expensive and time consuming component. It is not unusual for a software development organization to expend 40% of the total project effort on testing. In the extreme, the testing of safety-critical software such as flight control, nuclear reactor monitoring etc. can cost three to five times as much as all other software engineering steps combined [?]. Reducing the cost of testing clearly has large benefits. We propose to reduce the cost of testing by influencing the design of the system being tested.

The testing community has traditionally accepted a system design as a given over which they have no control. This has led to a number of test case generation techniques [?, ?, ?, ?, ?, ?] to deal with the potential complexity of systems. Detection of infeasible test cases and exponential growth in the number of test cases have been problems that have been left for the testing community to handle during test data generation stage without giving any consideration during design stage of software development. Other communities, however, have been concerned with affecting the design of systems in order to improve the system with respect to some attribute of interest. The growth of object oriented design and the use of encapsulation have been attempts to decrease the life cycle cost of systems by changing the design of systems. The reliability, security and performance communities have a number of techniques [?, ?, ?, ?, ?, ?, ?] for changing the design of a system in order to improve the respective quality attributes.

In this paper, we address the problem of designing software with the goal of reducing the number of coverage requirements for **integration testing** of the software. Our approach is to relate the testability of a system to its design. That is, we advocate choosing a design that simplifies the integration testing of a system by requiring a smaller test suite. In reality, of course, design is a more complicated problem than just optimizing for a particular attribute. Design is the process of making trade-offs among attributes and the designer needs to know the cost of making particular choices. So our approach is to develop techniques for determining the testing cost when particular design styles are considered. We measure the cost of testing in terms of the number of test cases that have to be covered for integration testing of a given design. This will enable the designer to determine the testing cost of making particular design choices and consider this cost as well as the cost of achieving certain levels of modifiability, performance, reliability, and security when making design decisions. The work we present here is the merger of ongoing work in both testing and software architecture analysis. We begin by reviewing the relevant work in each of these areas.

2 Software Architecture

A software architecture of a program is a structure comprising of software components, their externally visible properties, and the relationships among them [?]. The process of designing a particular system is the process of defining the software architecture that gets elaborated into an actual system. A key element to the study of software architecture is the discovery and analysis of architectural styles [?]. A software architecture style is a system level construct that has been observed many times in successful systems. It can also be thought of as a system level design pattern.

Architectural styles are important since they differentiate classes of designs by offering experimental evidence of how each class has been used along with qualitative reasoning to explain why each class has certain properties. A software architect can choose a style based on an understanding of the desired quality goals of the system under construction. Adoption of a software style for the design of a system acts as a set of constraints on the actions of the designer. This, in turn, enables the creation of techniques that are style specific, to analyze how suitable the style is for the achievement of particular attributes. A collection of Attribute Specific Architectural Styles (ABASs) for a variety of attributes is documented in [**?**, **?**]. We now discuss some attributes important for the testing phase of software development.

3 Software Testing

The testing process consists of selecting a test adequacy criteria, generating test requirements for the selected criteria, generating test data that exercise the test requirements, monitoring the execution of the program on the test data and verifying the output produced by the program. The most commonly used strategy for software testing consists of *unit testing* that concentrates on each unit of software as implemented in the source code, *integration testing* that focusses on the design and the construction of software architecture, *validation testing* that validates the software requirements against the developed software and finally *system testing* that tests the developed software and other system elements as a whole [**?**]. In this paper, we focus on the testability of the system for integration testing.

We define the testability of a system in terms of cost of testing the system. The cost of testing a system is directly proportional to the size of the test suite which, in turn, is governed by the number of coverage requirements that must be exercised by the test data. The number of coverage requirements increase with the increase in number of interacting modules as well as the number of interactions among them. Besides, if the set of test requirements is large, there is higher likelyhood of having some requirements for which it is infeasible to generate test data. Since detecting infeasible test requirements is an undecidable problem in general, it would save significant effort if the software is designed in such a way that the allowed interaction between various components is kept to minimum. To reduce the cost of testing a system, we propose selecting a suitable design for the system that requires a smaller number of test cases to be covered. Our goal is to develop a collection of testability analyses related to specific architectural styles. We begin by analysing the testability of the pipe and filter style.

4 Pipe and Filter Style

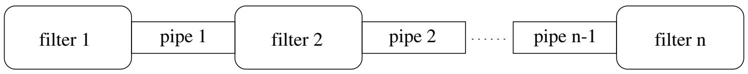

Figure 1: Pipe and Filter architecture style.

As shown in Figure **??**, in a pipe and filter style, data enters a filter from a single source, is transformed, and sent out through a single exit into a pipe. The pipe carries the data to the next filter in the design. The pipe and filter style supports system organization based on asynchronous computations connected by dataflow. Pipes and filters occur in a variety of systems. Systems based on signal processing such as image processing systems are pipe and filter systems. The case studies in [**?**] are all pipe and filter systems. Old fashioned compilers (lexical analysis, followed by syntactic analysis, followed by semantic analysis) were pipe and filters although more modern compilers utilize different styles. The computational elements scheduled by a cyclic executive can be thought of as a pipe and filter system where the output of the final filter is fed into the input of the initial filter. Pipes and filters are not a basis for all systems, but they are used in a substantial number of systems. Our goal in this paper is to provide a basis for analysis of testing cost for the systems describable by pipe and filter style. In our our future work, we will consider other styles for analysis of their testability. Thus, we exploit the restrictions imposed by the pipe and filter style (in particular, the limited interactions between filters) to carry out our analysis.

4.1 Modelling and Analysis

We use the number of coverage requirements for a given testing criteria (such exercising all interactions between every pair of modules) as a measure of the testability of an application for integration testing. A smaller number of the coverage requirements will result in a smaller test suite and hence will reduces the cost of the testing process.

In pipe and filter architecture style, each module in the design of the software is represented by a filter and the communication mechanism between a pair of modules is represented by a pipe. The pipe and filter architecture style enforces a simple communication protocol in which $filter_i$ can receive data only from $filter_{i-1}$ and send data to only $filter_{i+1}$. The communication between adjacent filters can be either using shared memory, message passing or procedure invocation. We assume that the $pipe_i$ simply provides a mechanism to transport data from $filter_i$ to $filter_{i+1}$.

We assume that the pipes and filters are correct i.e., each of the modules and their communication mechanisms have been unit tested with 100% reliability. We focus on the kinds of problems that can arise as a result of integration of all the modules and their communication mechanisms. We consider the total number of interactions (shared memory, messages or procedure invocations) to be tested between the filters as a measure of the testing cost of the style. In order to compare the testing cost of a design based on pipe and filter style and a design that allows arbitrary communication among the modules, we consider the example of four communicating processes in general shown in Figure 2. Let us assume there are at the most k interactions allowed in each direction between a pair of adjacent nodes.

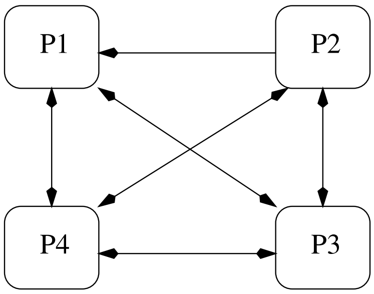

Figure 2: Communication paths among four communicating process in general.

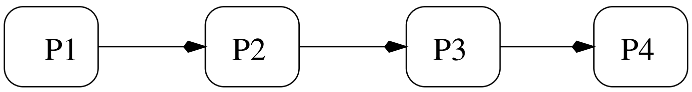

Figure 3: Communication paths among four filters in pipe and filter style.

There are four paths for communication between any pair of nodes in Figure 2. For example, there is one path (P1, P4) of length one, two paths (P1, P2, P4) and (P1, P3, P4) of length two and one path (P1, P2, P3, P4) of length three between the nodes $P1$ and $P4$. Therefore, to test the communication between $P1$ and $P4$, we need to generate test cases that exercise each possible interaction along all the four paths between $P1$ and $P4$. For the path (P1, P4), $2 * k$ interactions between the two nodes $P1$ and $P4$ need to be tested because there can be at the most k interactions in each direction. For the path (P1, P2, P4), $2 * 2 * k$ interactions need to be tested because at the most $2 * k$ interactions need to be tested between the nodes $P1$ and $P2$ and another $2 * k$ interactions need to be tested between the nodes $P2$ and $P4$. Similarly, $2 * 2 * k$ interactions need to be tested along the path (P1, P3, P4) and $3 * 2 * k$ interactions need to be tested along the path (P1, P2, P3, P4). Therefore, the total number of test cases needed to test the communication between $P1$ and $P4$ is at the most $2 * (k + 2k + 2k + 3k) = 16k$. In order to test communication between every pair of processes in Figure 2, we would need $6 * 16 * k = 96k$ test cases since there are 6 pairs of nodes possible.

Now let us consider the four nodes communicating using pipe and filter style as shown in Figure 3. In this design style, only adjacent nodes can communicate with each other and the communication is allowed only in one direction. If there are at the most k interactions between a pair of adjacent filters, only $3k$ interactions need to be exercised to test the communication between the nodes in pipe and filter style when all the four modules are integrated together.

In general, if there are n nodes communicating with each other in any arbitrary fashion and there are at the most k interactions allowed in each direction between two adjacent nodes, then the number of test cases needed to exercise each interaction between adjacent nodes on all the paths between a pair of nodes in the worst case is given by

$$2 * \binom{n}{2} * [\binom{n-2}{0} * k + \binom{n-2}{1} * 2 * k + \binom{n-2}{2} * 3 * k + \cdots + \binom{n-2}{n-2}(n-1) * k] \qquad \text{eq. 1}$$

where,

1. the multiplier 2 accounts for the interaction in both the directions,
2. the multiplier $\binom{n}{2}$ is the number of ways to choose a pair of nodes from n nodes, and
3. $\binom{n-2}{k}$ is the number of ways k nodes can be selected from the remaining $n - 2$ nodes, which is equal to the number of paths of length $k + 1$.

Writing the Binomial expansion of $(x+1)^{n-2}$, multiplying thoughout by x, differentiating throughout with respect to x, substituting x by 1 in the result obtained after differentiation, and using it to simply the equation 1, we obtain

$$\text{Number of Test Cases (general design)} = (k)(n^2)(n-1)(2^{n-3})$$

Therefore, in a general design with n modules, the number of test cases required for integration testing of communication between the modules is an exponential function of the number of modules.

However, the number of test cases required to exercise every interaction between adjacent filters, in an application designed with n filters with at the most k interactions between the adjacent filters, is given by:

$$\text{Number of Test Cases (pipe and filter)} = k * (n-1)$$

Therefore, the number of test cases required to test the interactions among the filters, during integration testing of a software designed using pipe and filter style, is a linear function of the number of filters.

Thus, if we define the testability of a design by the measure of the coverage required for integration testing, it is clear from the above discussion that a pipe and filter style is much more suitable for testability than a general design of communicating processes. Therefore, if a given application can be designed using pipe and filter style, then it will reduce the coverage requirements for integration testing of the application by a significant amount. Our analysis shows that it is worthwhile to consider the cost of testing while making tradeoffs with the design requirements of other attributes such as modifiability, reusability and security of software while choosing a particular architectural style in the design stage of the software.

5 Conclusions and Future Work

In this paper, we have proposed that the cost of testing a software application can be reduced by inluencing the selection of a suitable architectural style during the design stage of the sofware. We analysed the cost of testing an application with pipe and filter style and compared it with the cost of testing an application with arbitrary communication among its components. We showed that the cost of testing increases exponentially with the number of interacting components in a general design, whereas it increases only linearly with the number of components in the pipe and filter style. This makes pipe and filter style particularly suited for testability. In our future work, we plan to extend this work to additional architectural styles. Each architectural style can be considered as a set of constraints imposed on the patterns of communication among the components of the style. Exploiting these constraints in the analysis of coverage requirements means that we should be able to calculate reduced costs of testing for systems not explainable by the pipe and filter style.

References

[1] Bass Len, Clements Paul, Kazman Rick "Software Architecture in Practice" SEI Series in Software Engineering, Addison Wesley, 1998

[2] L.A. Clarke, "A System to Generate Test Data and Symbolically Execute Programs," *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 3, pages 215-222, September 1976.

[3] M.J. Gallagher and V.L. Narsimhan, "ADTEST: A Test Data Generation Suite for Ada Software Systems," *IEEE Transactions on Software Engineering*, Vol. 23, No. 8, pages 473-484, August 1997.

[4] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, Grady Booch "Design Patterns: Elements of Reusable Object-Oriented Software Addison Wesley, 1995.

[5] A. Gotlieb, B. Botella, and M. Rueher, "Automatic Test Data Generation using Constraint Solving Techniques," *International Symposium on Software Testing and Analysis*, 1998.

[6] C. Hofmeister, R.Nord, P.Soni "Applied Software Architecture" Reading MA, Addison Wesley, 1999.

[7] Mark Klein, Rick Kazman, "Attribute Based Architectural Styles", CMU/SEI-99-TR-022, Technical Report, Software Engineering Institute, Pittsburgh, Pa

[8] M. Klein, R. Kazman, L. Bass, J. Carriere, M. Barbacci, H. Lipson , "Attribute Based Architectural Styles", Proceedings of the First Working IFIP Conference on Software Architecture, San Antonio, Tx, Feb 1999, pp 225-243 Kluwer Publishing

[9] B. Korel, "Automated Software Test Data Generation," *IEEE Transactions on Software Engineering*, Vol. 16, No. 8, pages 870-879, August 1990.

[10] Neelam Gupta, Aditya P. Mathur, and Mary Lou Soffa, "Automated Test Data Generation using An Iterative Relaxation Method" *ACM SIGSOFT Sixth International Symposium on Foundations of Software Engine ering(FSE-6),* pages 231-244, Orlando, Florida, November 1998.

[11] R.A. DeMillo and A.J. Offutt, "Constraint-based Automatic Test Data Generation," *IEEE Transactions on Software Engineering*, Vol. 17, No. 9, pages 900-910, September 1991.

[12] R.S. Pressman, "Software Engineering: A Practitioner's Approach." Fifth Edition, 1998, page 595.

[13] M.Shaw, P.Clements, "A Field Guide to Boxology: Preliminary Classification of Architectural Styles for Software Systems" *Proceedings of COMPSAC*, Washington, D.C, August 1997.

[14] L. Sha, R. Rajkumar, M. Gagliardi, "A Software Architecture for Dependable and Evolvable Industrial Computing Systems" *CMU/SEI-95-TR-005*, Pittsburgh, PA, Software Engineering Institute, 1996.

[15] C.U. Smith, "Performance Engineering of Software Systems," *The SEI Series in Software Engineering,* Reading, MA, Addison-Wesley, 1990.

[16] C.U. Smith and L.G. Williams, "Software Performance Engineering: A Case Study Including Performance Comparison with Design Alternatives," *IEEE Transactions on Software Engineering,* 19(7), pages 720-741, 1993.

[17] U.S.Deparment of Defense, "Technical Architecture Framework for Information Management (TAFIM)," Vols. 1-8, Version 2.0. DISA Center for Architecture (10701 Parkridge Blvd., Reston, VA 22091-4398), June 30, 1994.