International Scholarly Research Network
ISRN Software Engineering
Volume 2012, Article ID 598150, 20 pages
doi:10.5402/2012/598150

*Research Article*

# Prioritizing Program Elements: A Pretesting Effort to Improve Software Quality

**Mitrabinda Ray and Durga Prasad Mohapatra**

*Department of CS&E, National Institute of Technology Rourkela, Orissa 769008, India*

Correspondence should be addressed to Mitrabinda Ray, mitrabindaray@yahoo.co.in

Received 28 September 2011; Accepted 27 October 2011

Academic Editors: K. Framling and G. Visaggio

Improving the efficiency of a testing process is a challenging task. Prior work has shown that often, a small number of bugs account for the majority of the reported software failures; and often, most bugs are found in a small portion of the source code of a program. First, prioritizing the code elements according to their criticality and then conducting testing, will promote to reveal the important bugs at the early phase of testing. Keeping it in view, we propose an efficient test effort prioritization method that give a chance to the tester to focus more on the parts of the source code that are highly influenced towards the system failures or in which, the failures have high impact on the system. We consider five important factors such as *influence towards system failures, average execution time, structural complexity, severity, and business value* associated with a component and estimates the criticality of the component within a system. We have experimentally proved that our proposed test effort prioritization approach is effective in revealing important bugs at the early phase of testing as it is linked to external measure of defect severity and business value, internal measure of frequency, complexity, and coupling.

## 1. Introduction

There are two major challenges faced by the testing team in a software industry. The first one is the size of possible test cases which is infinite and the second one is the time-box set for the testing. Sometimes, the testing time get squeezed due to delay in upstream task duration (design and coding). In this situation, testing effort prioritization helps the tester to do the best possible job within the limited test resources [1]. The tester gets the best possible chance to reveal the defects that are occurring frequently or have a negative impact on the user. Bugs in some parts of the source code may cause more severe and frequent failures compared to that in other parts. For example, if a method of a class produces crucial data that is used by many other classes, then a bug in this method will affect many other classes. A component with high-execution time tends more to failures [2, 3]. Upon analyzing the software failure history data from nine large IBM software products, Adams [4] demonstrated that a relatively small proportion of bugs in the software account for the vast majority of reported failures in the field. Boehm and Basili [5] have also proposed a *Pareto distribution* in which 80% of all defects within a software are found in 20% of the modules. Then, the question arises: how to identify the critical parts (a part is critical if it is responsible for frequent failures or serious type of failures.) of a program which is needed to be bug free?

Our aim is to identify the critical parts in the source code. A part of the source code is critical if the chance of failure is high in that part or the severity of failures in that part is high. For this, we prioritize the program elements within the source code for testing according to their criticality. A bug in a critical part may cause severe failure whereas the same bug in a low critical part may cause a minor or negligible failure. We consider two important criteria such as risk and business value associated with a component for estimating criticality. Risk of a component is determined by checking the likelihood of fault occurrence within a component (complexity) and the impact of failure (severity) of the component in the system. Similarly, a technical staff cannot guess which high-level functions are important to the customer. A customer also cannot estimate the cost and technical difficulties in implementing a specific high-level function. So, the adoption of value-based testing [6, 7], where the information from the market and the customer is gathered to gain the knowledge of the importance of high-level function, increases the return on investment (ROI) on testing. In our approach, we consider the business value associated with a component as a factor for computing criticality.

In an object-oriented program, a component may be a class which is the smallest executable unit or it may be elaborated as a collection of classes in a package. However, a component may be a collection of many other things. For example, a mixture of some source code templates with related documentation might be called as a component. After all, everything is a class in an object-oriented program, every component is a class too. In our approach, we take a class as a component and write class and component interchangeably throughout our paper.

We compute the criticality of a component based on the following factors.

(1) Average execution time of a component within a system.

(2) Influence_value of a component (influence metric) showing that the component is providing services to how many components directly or indirectly within a system.

(3) Structural Complexity of a component: response for a Class (RFC), and Weighted methods in a Class (WMC).

(4) All possible types of system failures for which the component is responsible and the severity associated with each failure.

(5) Business value associated with a component.

User's perception is a good indicator on the acceptance of a system. User's view on the reliability of a system is improved and almost cheaper, when faults which occur in the most frequently used parts of the software are almost removed [2, 3, 8, 9]. The reliability of a system is related to the probability that a fault leads to a failure that occurs during software execution [10]. It is because the data input supplied by the user decides which parts of the source code will be executed. A bug existing in the nonexecuted parts will not affect the output. So for a class, it is required to know how many other classes are requesting services from the class and also, how often these requests are executed at run time. The idea behind the consideration of average execution time for a class as a parameter is that when, a class is executed for longer time or more frequently compared to other classes during a typical run of the software, any existing bugs in the class are more likely to be executed during the run and will cause the frequent failure of the system.

However, the length of time a component is executed does not wholly determine the importance of the part in the perceived reliability of the system. It is possible that the results produced by a function which is executed only for a small duration is saved and extensively used by many other functions. Sometimes, a function whose produced results are widely used by many other functions, would have a very high impact on the reliability of the system even though it is itself getting executed only for a small duration. For this, we consider the interlink with other elements (an element may be as small as a statement and as elaborate as a class or couple of related classes (component).) through coupling. We have introduced a metric called *influence metric* [11, 12] for an element that shows the degree of influence of the element towards system failures. It is decided by checking how many elements within a system are directly or indirectly using the result that is produced by the element.

The idea of including structural complexity is to estimate the probability of presence of faults in a component. This can be determined through the existing metrics: response for a class (RFC) and weighted method per class (WMC) [13]. For evaluating the structural complexity, Chidamber and Kemerer [13] have proposed six metrics. We found that consideration of all six metrics at a time is complicated, time consuming, and also sometimes not useful for a particular purpose. At the same time, a single metric is also not sufficient for complexity estimation. At least the use of two or three CK metrics gives a proper estimation of potential problems [14]. For our purpose, we are using two CK metrics: RFC and WMC. RFC gives an idea about the longest sequence call of methods, and WMC provides the cyclomatic complexity of each method implemented in a class.

We are including severity of the failure of a component within a system as another factor for computing criticality. It is because, there are some components which have low complexity, but the failure of any one of that may have a catastrophic impact on the system. For example, a critical code my be called in case of an emergency, which happens infrequently, but the existence of bug in that part may cause severe failure. The impact of the failure may cause severe damage to the system (causality, loss of important data) or a huge financial loss. The severity factor is dependent on the nature of the application. It is basically assessed by domain analyst, who have knowledge on the environment in which the software will be used. The basic input for severity assessment is the costs of various failure modes. Detailed procedure of severity estimation is addressed in [15].

There is also a close relationship between testing and business value that comes from market or from customers [16]. Each use case of a system should not be treated with equal importance [17]. Boehm [17] has proposed a *value-based software engineering* (VBSE) practice that integrates the *value* coming from customers and market to the software engineering practices. Many times, it is decided based on who within the organization, the use case is important to. It is purely subjective rather than objective. We estimate the Value (business value) of a component by checking its interaction within various use cases. Based on the values of use cases, a component's value is estimated.

This work comes as a continuation of our previous work presented in [11, 12, 18]. In our previous work [12], we have only considered two internal factors of a component-Influence_value (influence of a component towards system failures) and average execution time for computing criticality. In this paper, we add the external factors: severity and business value and also include the internal factor: structural complexity, for criticality computation. The case studies in [19] show that the residual bugs location is strongly correlated with module size and complexity. Our approach is adding the structural complexity which is priorly used as a fault prediction method. Our approach does not only show the classes with high structural complexity within a program but also the likelihood of these classes to fail in the operational environment through the factor, *average execution time of a class*. Once the criticality of a component is estimated through our approach, more exhaustive testing has to be carried out to minimize bugs in high-critical components. This means we can cut down testing in less-critical components. The total test effort is distributed among various components according to their criticality. As a result, not only the postrelease failures will be minimized but also the severed types of postrelease failures will be also minimized within the available test budget.

Please note that we use the terms bug, fault, and defect interchangeably, when no confusion arises. An error done by a programmer results in a defect (fault, bug) in the software source code. The execution of a defect may cause one or more failures. As per the IEEE standard, failure is the inability of a system or component to perform its required functions within specified performance requirements. A failure in a system can be observed by the user externally. Failure is neither a coding fault in the development process nor a defect that put an element of the program or the system into an erroneous state that does not lead to a visible failure.

This paper is organized as follows. Section 2 gives a short discussion on ESDG (intermediate representation of an object-oriented program), slicing, and the *influence metric* on which our methodology is based. Section 3 discusses the proposed methodology for ranking components. In Section 4, experimental studies are conducted to test the effectiveness of our approach. Section 5 summarizes the related work. The paper is concluded in Section 6.

## 2. Background

The work in this paper is based on our earlier work on analysis of source code for test effort prioritization [12]. We have used influence metric [12] as one factor in our previous test effort prioritization methodology. The influence metric is derived from the intermediate representation of an object-oriented program called extended system dependence graph (ESDG) through forward slicing. In the following sections, we give a quick overview on ESDG, program slice, and influence metric. For more details, the reader is referred to [11, 12].

### 2.1. ESDG.

ESDG models the main program with all other methods. Each class in a given program is represented by a class dependence graph. Each method in a class dependence graph is represented by procedure dependence graph. Each method has method entry vertex that represent the entry in the method. The class dependence graph contains a class entry vertex that is connected with the method entry vertex

```
Class Task {
      puplic:
2   int add (int x,int y){
3   return (x + y);}
4   int incr (int i){
5   i=i+1;}
6   void fun (){
7   int sum=0;
8   int i=1;
9    while (i < 11){
10   sum=add (sum,i);
11   i = incr (i);}
12   cout<< "SUM="<<sum;
}
main () {
     Task ob;
1  ob.fun ();
     }
```

Algorithm 1: An object-oriented program.

of each method in the class by a special edge known as class member edge. To model parameter passing, the class dependence graph associates each method entry vertex with formal-in and formal-out vertices.

The class dependence graph uses a call vertex to represent a method call. At each call vertex, there are actual-in and actual-out vertices to match with the formal-in and formal-out vertices of the called method. If the actual-in vertices affect the actual-out vertices, then summary edges are added at the call-site, from actual-in vertices to actual-out vertices to represent the transitive dependencies. To represent inheritance, we construct representations for each new method defined by the derived class and reuse the representations of all other methods that are inherited from the base class. To represent the polymorphic method call, the ESDG uses a polymorphic vertex. A polymorphic vertex represents the dynamic choice among the possible destinations. The detailed procedure for constructing an ESDG is found in [20]. Each node can be a simple statement, a call statement, a class entry, or a method entry. An example of an object-oriented program with its ESDG is shown in Algorithm 1 and Figure 1, respectively.

### 2.2. Program Slice.

A program slice is a part of the code that contributes in computation of certain variables at a program point of interest. For a statement $s$ and variable $v$, the slice of a program $P$ with respect to the slicing criterion $\langle s, v\rangle$ includes only those statements of $P$ that are needed to *capture* the *behavior* of $v$ at $s$ [21]. Slicing can be static or dynamic. Static slicing technique uses static analysis to derive slicing. That is, the source code of the program is analyzed, and the slices are computed for all possible input values. No assumptions are made about the input values. Since the predicates may evaluate either to true or false for different values, conservative assumptions have to be made, which may lead to relatively large slices. So, a static slice may contain

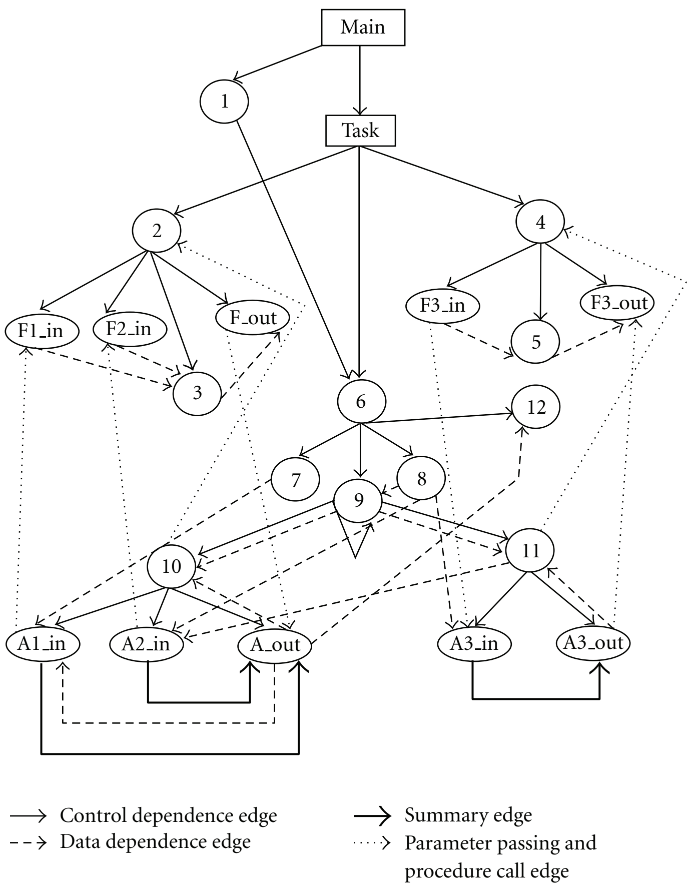


Figure 1: ESDG of Algorithm 1.

```
1 main()
2 {
3   int i, sum;
4     cin>> i;
5     sum   = 0;
6     while(i <= 10)
7     {
8       sum=sum+i;
9       ++ i;
10     }
11    cout<< sum;
12    cout<< i;
13    }
```

Algorithm 2: An example program.

statements that might not be executed during an actual run of a program, whereas dynamic slicing makes use of the information about a particular execution of a program. The execution of a program is monitored, and the dynamic slices are computed with respect to execution history. A dynamic slice with respect to a slicing criterion $\langle s, v\rangle$, for a particular execution, contains only those statements that actually affect the slicing criterion in the particular execution. Therefore, dynamic slices are usually smaller than static slices and are more useful in interactive applications such as program debugging and testing.

Consider the C++ example program given in Algorithm 2. The static slice with respect to the slicing criterion $\langle 11; \text{sum}\rangle$ is the set of statements $\{4, 5, 6, 8, 9\}$. Consider a particular execution of the program with the input value $i = 15$. The dynamic slice with respect to the slicing criterion $\langle 11; \text{sum}\rangle$ for the particular execution of the program is $\{5\}$.

Slices can be backward or forward. A *backward slice* contains all parts of the program that might directly or indirectly affect the slicing criterion but a *forward slice* with respect to a slicing criterion $\langle s, v\rangle$ contains all the parts of the program that might be affected by the variables in $v$ used or defined at the program points. A forward slice provides the answer to the question: “which statements will be affected by the slicing criterion?,” whereas a backward slice provides the answer to the question: “which statements affect the slicing criterion?” [22].

*2.3. Influence Metric.* In a program, the incorrect results computed by a called method may affect other calling methods. This is because the value returned by the called method may be used by a calling method, for taking any further action or for some computation, in which inaccuracy may lead to catastrophic consequence. The influence of a given method $M$, Influence_of_method ($M$), is the set on nodes of ESDG that are directly or indirectly using the result computed by $M$, and the set is called *the set of nodes* *influenced by M*. If the influence set of a method is high, then a bug in $M$ makes the chance of the overall system failure too high. First, we have represented the input program as an intermediate representation (ESDG). Then, we have applied our proposed algorithm in [12] on the ESDG to compute the influence of a method in a program. We have counted the number of nodes marked as influenced by a method $M$ in a program from the data dependent set of that method's formal parameter-out nodes.

The influence_value of a method $M$ is expressed as:

$$\text{influence\_value}\,(M) = \frac{\#\text{ nodes influenced in ESDG by } M}{\text{Total }\#\text{ nodes in ESDG}}. \quad (1)$$

*Influence of a Class.* The nodes in the set *influence_of_class* $(c)$ for the class $c$ is the union of all the sets influence_of_method $(m_i)$, where $m_i$ is the $i$th method of class $c$.

Influence_of_class $(c) = \bigcup_{i=1}^{k}$ Influence_of_method $(m_i)$, where $k$ is the number of methods in class $c$.

## 3. Testing Effort Prioritization Methodology

Our proposed methodology is defined by the following steps.

(1) Computing the influence_value of a component within a system (discussed in our previous work [12]).

(2) Estimating the average execution time of a component by executing the test data based on operational profile (Section 3.1).

(3) Computing the structural complexity for a component (Section 3.2).

(4) Analyzing the severity associated with a component through simulation runs (Section 3.3).

(5) Estimating the business value associated with a component (Section 3.4).

(6) Performing priority estimation and ranking components based on their priority value (Section 3.5).

*3.1. Average Execution Time of a Component within a System.* The average execution time of a component within a system is estimated based on the *operational profile* [2] designed for the system and a tool called *profiler* [23]. Profiler shows the average time in which different functions are executed. The average execution time for a class $c_i$ is defined as:

$$\text{ET}(c_i) = \sum_{j=1}^{\text{nos}} p_j * \left(\text{Time}(c_i)^j\right), \quad (2)$$

where nos is the total number of scenarios in a system under test, $p_j$ is the probability of $j$th scenario, and $\text{Time}(c_i)^j$ is the total activation time of $c_i$ in $j$th scenario. The probability of execution of a scenario in a system is determined from operation profile developed for a system, and the total activation time of a component within a scenario is obtained through profiler.

*3.2. Analyzing the Structural Complexity of a Component.* Our aim is to find the complexity associated with a component by analyzing the complexity of various services provided by the component. So for simplicity, we consider only two CK metrics (RFC and WMC), out of six metrics proposed in [13]. It is experimentally proved that a class with high RFC and high WMC is associated with fault-proneness [24]. Hence, these two chosen metrics (RFC and WMC) are used as inputs to derive the complexity for our purpose. Response for a class (RFC) contains a set of member functions directly or indirectly called by the class, whereas WMC is checking the complexity associated with all member functions of a class through cyclomatic complexity.

RFC metric measures the cardinality of a set of methods that can potentially be executed in response to a message received by an object of that class [13]. In RFC, the basic unit is *method*, which refers to the message passing concept in OO programming. The RFC value for a class $c$ is

$$|\text{RS}| = \sum_{i=1}^{M} R_i, \quad (3)$$

where RS, $M$, and $R_i$ represent the response set for the class $c$, number of methods in the class $c$ and the set of methods called by $i$th method. A high value of RFC indicates that the complexity of services provided by the class is increased and the understandability is decreased. When a larger number of methods are invoked from a class through messages, it complicates the testing and debugging process, and also it is difficult to change a class due to the potential for a ripple effect. As testing and maintenance are complicated, the chance of failure increases. We have derived $R_i$ through the intermediate representation of the source code (ESDG). We start traversing from each *method-entry* vertex of a class and traverse only the *call-edges* in a forward direction and generate a set of nodes called by each method of a class. We repeat this process for each method of a class and finally merge the sets to get *RS* for the class.

Luke [25] argued that there is really no way to know a software failure rate at any given point in time because the defects have not yet been discovered. According to his statement, the design complexity is positively linearly correlated to defect rate. Hence, the occurrence of software defects should be estimated based on McCabe's complexity value or Halstead's complexity measure [25]. Therefore, we are considering WMC metric that gives a rough estimation of total complexity associated with a class. WMC metric is correlated with defect rates [14]. It counts local methods and calculate the sum of the internal complexity of all local methods in a class [13]. The internal complexity of each method is decided through cyclomatic complexity. WMC value for a class $c$ is

$$\text{WMC} = \sum_{i=1}^{M} W_i, \quad (4)$$

where, $M$ and $W_i$ stand for number of methods in a class and cyclomatic complexity of $i$th method. It helps to evaluate the minimum number of test cases needed for each method.

Therefore, it is used as a guideline by test manager to estimate how much time and effort is required to develop and maintain a class.

We estimate the probability of faults in a class based on two parameters such as RFC and WMC. First, we assign threshold value to each metric as defined in [26]. For each parameter, we use only three weights: low (0.3), medium, (0.5), and high (1). The assignment of points to the three weights is a rough guideline. The following threshold values are assigned to the two parameters:

(1) Weighted methods per class (WMC): $\lessapprox 25$ preferred and $\lessapprox 40$ acceptable [26].

(2) Response for class (RFC): $\lessapprox 40$ preferred and $\lessapprox 50$ acceptable. It has been observed that very few classes with RFC over 50 exist within a system [26].

The complexity information for case studies, LMS and TAS, are shown in Tables 1 and 2, respectively. The classes that are within preferred (acceptable) limit are low (medium) in complexity and exceed the acceptable limits are high in complexity. Out of these two parameters, if one parameter is in low range and the other one is in medium range, we are accepting the whole complexity of the class as medium, the higher one between the two factors. The intent of computing the structural complexity of a component is to show the probability of existence of faults within the component, whereas the intent to compute the influence_value of a component is to show how many other components will be affected by the faulty behavior of the component.

*3.3. Severity Analysis.* Severity is a rating and is basically applied to the effect of a failure. It shows the seriousness/impact of the effects of a failure within a system. Severity of a failure within a system decides how badly a bug within a component affects the whole system. We have inserted some bugs in different components of a system and executed the system for some duration in the operational environment. We observed that similar types of bugs in different components cause failures with different severities. Hence, we use the severity factor of a component as a measure to the overall quality of the product. We consider that a component is critical, if the failure of the component causes severe effect on the whole system. In our proposed criticality evaluation method, our aim is to first reveal bugs from high critical component and, then reveal from a low critical component. If there is an urgency to release the system before time or the testing time is shortened due to some unavoidable circumstances then, the test manger will be sure that the bugs responsible for severe type of failures are revealed and fixed.

Though, more testing focus should be given to the parts of the code which are executed frequently [2, 3, 27]; however, there is also a need for severity analysis for better quality of a system. There are some critical codes exist within a system, which are called in case of an emergency. Though these critical codes execute rarely, the existence of a bug with them may cause a severe failure. For example, let us consider a component which is providing exception handling of rare but critical conditions. In this case, the component is generally executed rarely. The influence of the component towards failure is low, and the structural complexity of the component is also low but, a bug in the component could cause catastrophic failure. Therefore, the severity of a component is included as an important factor for testing effort prioritization in our approach.

Table 1: Structural complexity of various entity classes within library management system.

| Class | WMC | RFC | Complexity |
|---|---|---|---|
| Borrower | 29 (Medium) | 42 (Medium) | Medium |
| Title | 18 (Low) | 39 (Low) | Low |
| Item | 29 (Medium) | 33 (Low) | Medium |
| Loan | 32 (Medium) | 52 (High) | High |
| Reserve | 15 (Low) | 30 (Low) | Low |

Table 2: Structural complexity of various entity classes within trading-house automation system.

| Class | WMC | RFC | Complexity |
|---|---|---|---|
| Productinfo. | 25 (Low) | 18 (Low) | Low |
| Category-Mgr | 28 (Medium) | 37 (Low) | Medium |
| Order handler | 33 (Medium) | 58 (High) | High |
| InventoryMgr | 27 (Medium) | 46 (Medium) | Medium |

We estimate the severity of a component within a system through *failure mode effect analysis* (FMEA) [28]. This is a bottom-up approach. FMEA is applied to a component to get the deficiency and hidden design defects. It focus on two points: (i) analyzing the potential failure modes of a component (how a component fails) and (ii) determining the effect of the failure modes on the system as a whole (consequences of failures). For a hardware component (electrical/mechanical), the failure modes are well known, but this case is not true for a software component. Nowadays, more and more system functions are implemented on software level, and hence, there is a need to apply FMEA methodology on software-based systems for determining the severity factor of a component. For a hardware component, the failure modes are weary, design flaws and unintentional environmental phenomena. Sometimes, the component manufacturing company discloses the possible failure modes and also the estimated frequencies of failures for their products. This is not possible in case of a software component. For a software component, the analyst decides the failure modes based on the design and development process. Software is not a physical entity, it is a logical construct. The analyst identifies the system level hazards both at the analysis, design, and implementation phase and translate it into software terms. Now, we discuss software FMEA.

*Software Failure Mode and Effect Analysis (SFMEA).* The detailed SFMEA focuses on the classes or modules. Several error conditions are checked in SFMEA. Table 3 shows

Table 3: Possible error conditions within a class/module.

| Error condition | Examples |
|---|---|
| Error in computation | |
| Wrong Algorithm | The module may carry out estimations wrongly due to faulty requirements or wrong coding of requirements |
| Calculation of underflow or overflow | The algorithm may produce in a divide by zero state |
| Error in data | |
| Unacceptable data | The module may accept out of range or wrong input data, no data, wrong data type or size, or premature data; produce wrong or no output data; or both |
| Input data trapped at some value | A sensor may read zero, one, or some other value |
| Bulky data rates | The module may not be able to handle a vast amounts of data or many input requests simultaneously |
| Error in logic | |
| Wrong or unpredicted commands | The module may receive improper data but continue to execute a process. It may be intended to do the proper thing under improper situation/state |
| Failed to issue a command | The module may not call a routine under certain circumstances |

various types of errors that may occur within a software module/class at the design or coding stage.

Ozarin [29] has discussed the advantages of performing SFMEA at various levels: (i) method-level analysis, (ii) class-level analysis, (iii) module-level analysis, and (iv) package-level analysis. According to him, SFMEA process is more accurate and effective at the method-level, which is the lowest level analysis. The authors of [30] have considered that a method within a software system is equivalent to a part of hardware system in which, there is a chance of failure under certain conditions. It is because, if a method within a class does not perform according to its predefined specification then, there is a chance of failure of the whole system under some conditions.

At the time of testing, the debugger analyzes the root cause of a bug and extracts the method within a class and more specifically the instruction within a method, which one is the source of bug. If any failure is occurred at the testing phase, then significant amount of searching is conducted to find exactly the faulty parts in the source code and more specifically the search is conducted to find the exact faulty instructions of a method.

As the source code is available in this stage, we conduct the operation level or method level SFMEA in this paper. During the execution of a scenario, a number of objects communicate through message passing. The message passing mechanism is implemented through method calls. A method within a class may or may not has formal parameters and may or may not has return value. To identify the severity of a class within a system, we have to identify the various types of failure modes within a method of a class, and also we have to estimate the severity of each failure mode by seeding some bugs, observing the failures and estimating the impact of failures. To estimate the severity level of a failure mode, we take the views of domain experts.

*Method-Level Failure Modes and Effect Analysis.* A method which is performing important tasks is generally viewed as an agent, which has to fulfill a contract to perform its operation. There may or may not be any formal parameter in a method, and a method may or may not return any value. A method maintains some preconditions and postconditions that explicitly state the agreement of a method for performing a task. A precondition is the entry condition to perform a task, and a postcondition is a condition that must be true after the completion of the task. Similarly, a class invariant states some constraints that must be true for its objects, at each instance of time during the life time of an object. A method job is divided into two parts: (i) constraint checking part and (ii) actual logic to perform a task. We assume that there is no time constraint, when a method is performing its task. In this paper, we have considered four failure modes of a method as defined in [30]. These are

(1) Precondition violation failure modes, $F_1$: there are two subfailure modes: (i) pre-condition is not satisfied but its corresponding exception is not raised, $F_{1.1}$ and (ii) pre-condition is satisfied but its corresponding exception is raised, $F_{1.2}$.

(2) Parametric failure modes, $F_2$: we check any failures regarding to formal parameters declared in a method. We only check the constraints on parameter values and do not consider the type of a parameter. If any parameter constraint is already stated in the precondition of the method, then we do not include that failure mode under the parametric failure mode. For any other parameter constraint, we check the response of the method. If any protection is not provided within the body of the method, then we consider the constraint itself as the failure mode. If any alarm is raised by the method in the form of exception, then we include two failure modes for two individual cases: (i) the constraint is false but its corresponding exception is not raised, $F_{2.1}$ and (ii) the constraint is true but its corresponding exception is raised, $F_{2.2}$.

(3) Method call or invoke failure modes, $F_3$: it consists of two subcases.

   (a) A Method *m1* invokes method *m2* of the same class or super class, then there is a possibility of the following failure modes in the list of failure modes of *m1*, and $F_{3.1}$:

      (i) *m1* invokes *m2* in the wrong order (when the invocation of *m2* is condition based), $F_{3.1.1}$.

      (ii) *m1* invokes *m2* by wrong parameters (when *m2* contains parameters. We consider only parameter's value not the type), $F_{3.1.2}$.

Table 4: SFMEA at method level for some components within the *withdraw* scenario.

| Triggered hazard | Component | Failure mode | Effect | Severity |
|---|---|---|---|---|
| A fault in dispensing cash | Cash dispenser | Cash dispenser is empty but not raising any exception | Money will be deducted from the account immediately, though the customer is not able to withdraw the said amount. As all transactions are maintained in the Log, the account will be updated by the banker later on | Critical |
| A fault in completing transaction | Withdrawal | The object of withdrawal component fails to create a new receipt | Receipt will not be printed | Minor |
| A fault in reading menu choice from the screen | Withdrawal | Failed to call the read menu choice method of an object of component customer console | Transaction cannot be performed | Marginal |

(b) A Method *m1* of class *A* invokes method *m2* of class *B* then, there is a possibility of the following failure modes in the list of failure modes of *m1*, and $F_{3.2}$:

(i) *m1* fails to invoke *m2* (because of the lack of instance of object of class B), $F_{3.2.1}$.

(ii) *m1* invokes *m2* in wrong order (when the invocation of *m2* is condition based), $F_{3.2.2}$.

(iii) *m1* invokes *m2* by wrong parameters (when *m2* contains parameters. We consider only parameter's value not the type), $F_{3.2.3}$.

(4) Postconditional failure modes, $F_4$.

We first obtain the FMEA of a component within a scenario and then, we assign severity through FMEA and hazard analysis. Severity of a component within a scenario shows how its failure affects the execution of the scenario. Domain experts play a vital role in hazard analysis and estimate the severity level of a component within a scenario. We rate the severity of a component within a scenario based on the worst effect of the failure of providing services by the component within that scenario.

As per the recommendation of [28], the severity is classified as:

(1) Catastrophic: a failure may cause death or total system loss.

(2) Critical: a failure may cause very serious effects (system may loose functionality, security concerns, etc.).

(3) Marginal: a failure may cause minor injury, minor property damage, minor system damage, and delay or minor loss of production, like loosing some data.

(4) Minor: defects that can cause small or negligible consequences for the system (e.g. displaying results in some different format).

We assign severity weight of 0.25, 0.50, 0.75, and 0.95 to minor, marginal, critical, and catastrophic severity classes, respectively. Researchers [31, 32] are accepting these selection of linear scale values for the severity classes. The damage may be classified into different classes as mentioned above or it may be quantified into money value, whatever the analyst feels better. For example, if a large volume data to be sent by mail are wrong, then the cost of remailing will be horrible.

Table 4 shows a part of SFMEA at the method level for some components within withdraw scenario of ATM system. The column *triggered hazard* shows the occurrence of failure when an event is triggered and some action is performed. The column *component* shows the component in which there is an occurrence of fault. In the table, failure mode is any one failure mode out of the four failure modes ($F_1$, $F_2$, $F_3$, and $F_4$) discussed above. The column *effect* shows the effect of the failure mode on the system. Severity is any one out of the four severities (catastrophic, critical, marginal, and minor) discussed above.

There can be more than one severity level for a component within a scenario. For example, the component *withdrawal* has two severity levels (minor and marginal) as shown in Table 4. We consider only the worst-case consequence of a failure as the severity level for the component. For the component withdrawal, we consider the marginal severity level within the *withdraw* scenario.

*3.4. Business Value Estimation for a Use Case.* We consider that the functionality of any system can be modeled through a set of use cases [33]. Each component of a system belongs to one or more use cases. That is, each use case is typically implemented by interaction among several components, and a component may participate in several use cases. Each use case has a main scenario and a number of alternative scenarios. We only consider the main scenario and assume that the business value (value) of a use case is same as the value of its main scenario. Let us consider the example of ATM system. The use cases could be *deposit*, *withdraw*, *inquiry balance,* and *transfer money*.

The following steps show a simple method adopted in various software industries for estimating the business value associated with a high-level function [16, 34].

(1) The relative benefit that each feature provides to the customer/business is estimated on a scale from 1 to 9, where 1 and 9 indicate the minimum benefit and the maximum possible benefit. The best people to

Table 5: Value (business importance) assignment.

| Relative weights | 2 | 1 | — | — |
|---|---|---|---|---|
| Use case | Benefit | Penality | Total value | Value% |
| Withdraw | 8 | 9 | 25 | 33 |
| Deposit | 7 | 5 | 19 | 24 |
| Transfer money | 9 | 5 | 23 | 30 |
| Inquiry balance | 9 | 9 | 27 | 13 |
| Sum | — | — | 94 | 100 |

judge these benefits are the domain expert and the customer representatives.

(2) We estimate the relative penalty that the customer or business would suffer by not including a feature. For this penalty, we also use a scale from 1 to 9, where 1 stands for no penalty and 9 shows a very serious problem.

(3) The sum of the relative benefit and penalty gives the total value. By default, benefit and penalty are weighted equally. The weights for these two factors can be changed. We have rated the benefit twice as heavily as the penalty ratings as defined in [6, 7].

The business values for various use cases of ATM system are shown in Table 5. We consider only the use cases that are used by the customer. *start up* and *shut down* use cases are not considered as they are the basic use cases to run the system.

Once, the values for all use cases of a system under test are estimated, the next job is to estimate the Values for various components of a system under test.

*3.4.1. Value Estimation for a Component.* The proposed methodology consists of the following steps.

(1) Constructing the Component Dependence Diagram (CDD) from the source code.

(2) Extracting slices of various scenarios from the CDD and use this for estimating Value (business value) for a component.

*Component Dependence Diagram.* A component dependence diagram (CDD) is a directed graph. Each node of a CDD corresponds to a component of the program. A component is a basic executable unit. In a procedure-oriented program, a component can be a function whereas it is as simple as a class in object-oriented program. The edges of the graph represent either control dependency or data dependency among the nodes. These dependencies are represented by directed arrows. We do not use different symbols to represent these two types of dependencies since our prioritization algorithm treats both the dependencies identically. If both data and control dependencies exist between two components, we draw only one arrow between the corresponding nodes. Figure 2 shows an example program and its CDD. In the example shown in Figure 2(a), the statement, *store (d)* in function $f2$ indicates saving of latest value of the variable $d$ to a memory location. Similarly, the statement *read (d)* in function $f3$ indicates reading of the value of variable $d$ that was last saved by function $f2$. This diagram is similar to a data-flow diagram (DFD) [35]. However, unlike a DFD which represents a program in a hierarchy of diagrams, a CDD represents a program in a single diagram. Also, a CDD captures control aspects unlike DFDs which represent data flows alone. Further, the nodes of a DFD may or may not correspond to the functions of a program [35].

We view a CDD as a simplified form of a *system dependence graph* (SDG) [36], but a CDD does not have as many types of edges as SDG [36]. Unlike SDG, a CDD does not represent the individual statements of a program because inclusion of individual statements make the graph looking complicated. In extended control-flow graph (ECFG) [37], nodes refer to methods in an object-oriented program whereas it refers to components in CDD. The aim of referring a node of CDD to a component instead of a method in an object-oriented program is to make the graph simple and easily understandable. We compare CDD with the *component dependence graph* (CDG) proposed by Yacoub et al. [38] where, nodes refer to components. They have adapted control flow graph principally to represent the dependency between two components and possible execution paths. Unlike our approach, Yacoub et al. have considered only control dependency between components. In their approach, the components are assumed to be independent (the existence of bug in one component is not responsible for the failure of another component). As we consider the data dependency between two components, a bug in one component may have an effect on other components.

The CDD generated in our approach is satisfying all the following constraints.

(i) No node is isolated.

(ii) All use cases put together cover all nodes.

(iii) No self-loops.

(iv) The node at which a use case starts execution is not control dependent on other nodes of the graph.

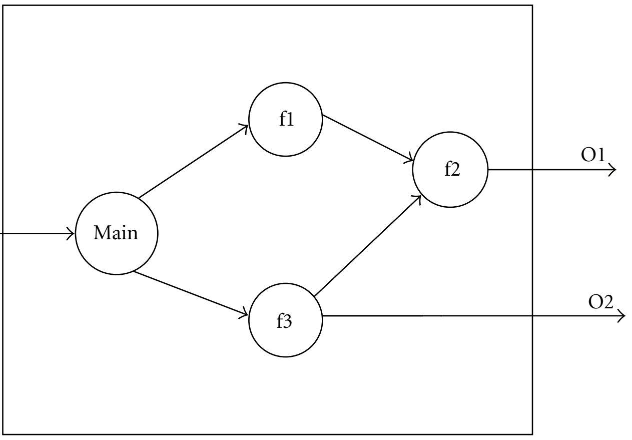
(v) The nodes tested by any one test case are a subset of nodes belonging to slice of a scenario.

Once the intermediate graph, CDD, is constructed, we use it to extract slices with respect to various scenarios for prioritizing components.

*Extracting Slices of CDD with Respect to Various Scenarios and Estimating the Value of a Component.* Each use case has one main scenario and a number of alternative scenarios. Let us consider a single use case say *withdraw* of ATM system. This use case has main scenario *correct transaction* and a number of alternative scenarios: *insufficient balance* and *requested amount exceeds maximum withdrawal limit per day*. For simplicity, we only consider the main scenario and do not consider the alternative scenarios. The value of a use case is same as the value of its main scenario.

```
int d[10];

main(){
int i;
i=readUserOption();
if(i==1) f1();
else        f3();

}
f1(){
        .
        .
        f2();
        }

f2(){
.
.
store(d);
print(o1);
}
f3(){
read(d);
 .
 .
print(o2);
}
```

(a) An example program



(b) Component dependence diagram of the example program

FIGURE 2: A program with its CDD.

We compute the slice $S_i$ of the CDD with respect to scenario $S_i$. The slice of a scenario, Slice $(S_i)$, contains the set of all components within a system that influence the execution of the scenario $S_i$. Thus, slice $(S_i)$ contains the set of components that are either get executed during the execution of scenario $S_i$ or the results of the components, saved in different variables, are used during the execution of $S_i$. Please note that we use the terms node of a CDD and a component interchangeably when no confusion arises.

*Value Estimation Scheme.* Once the business values for all scenarios of a system are determined, we estimate the business value Value $(C_i)$ for a component $C_i$ as follows.

$$\text{Value } (C_i) = \sum_{j=1}^{\text{nos}} q_j, \tag{5}$$

$$q_j = \text{Value}_j, \quad \text{if } C_i \in \text{slice } \left(S_j\right) \text{ else, } q_j = 0,$$

where nos is the number of scenarios within a system. Value$_j$ is the probability of $j$th scenario, and slice $(S_j)$ is the slice of the CDD with respect to $j$th scenario. The priority value of a component intuitively indicates the priority of its being used during an actual operation of the program. We now algorithmically present our business value estimation scheme for various components within a system.

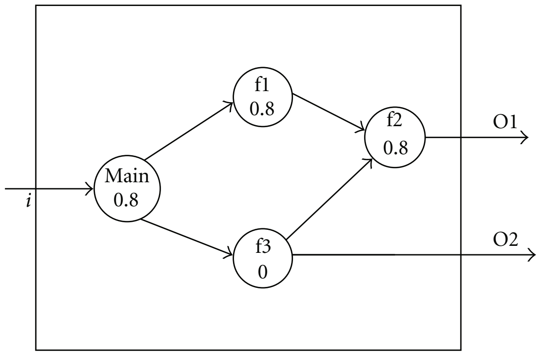
*Value Estimation Algorithm*

```
(1) Construct the CDD of the program.
(2) Determine the business value Value$_i$ for
    each scenario $S_i$ within a system.
(3) For each component $C_j$ of CDD let
    Value($C_j$)=$\emptyset$.
(4) For each scenario $S_i$ of the program:
    (a) Compute the slice of scenario $S_i$
        from the CDD.
    (b) for each component $C_j$ in slice($S_i$) let
        Value($C_j$) = Value($C_j$) + Value$_i$.
(5) Print the value computed for each
    component.
```

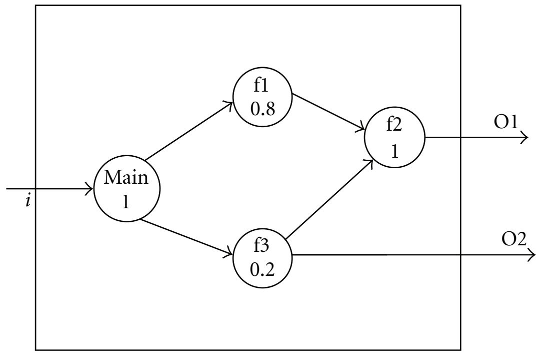
We now explain our value estimation method using a simple example. Let us assume that the program shown in Figure 2(a) has two use cases: $U_1$ and $U_2$, and each use case has only one scenario. Let the business values associated with scenarios $S_1$ and $S_2$ be 0.8 and 0.2, respectively. Figure 3(a) shows the computation of the slice, Slice$_1$, the slice of CDD with respect to $S_1$. Each component that is executed within the slice Slice$_1$ is getting marked by 0.8, which is the value of scenario $S_1$. Next, the slice of CDD with respect to scenario $S_2$, Slice$_2$, is computed. The values (business values) of various components after slicing the CDD with respect to both scenarios are obtained as shown in Figure 3(b). In the figure, the functions main and $f2$, each has value 1. This means that both the functions main and $f2$ are either getting executed or their results are used by both the scenarios $S_1$ and $S_2$. It may be noted that the business value of 1 for a function indicates that it is required for all scenarios of the system. Table 6 shows the business values associated with various components of ATM system.

*3.5. Priority Calculation.* For assigning priority, the commonly used method is to do a proper weight assignment and then calculate a weighted sum for a class [1]. We assign a relative weight for each chosen factor of a class. For each factor, we assign equal weight. The weight may vary depending on the nature of the system. An example of priority calculation for a component is shown in Table 7. The headings used for different columns of the table are Inf: influence_value, EEC: expected execution time, SC: structural complexity, Severity: impact of component failures, Val: business value, and TP: total priority of a component within a system. The priority value for a component is normalized by dividing the total priority value of a component with the sum of the total priorities of all components within a system.

There are a lot of technical, productive, and environmental complexity factors that exist within a component. For



(a) Priority values after slicing scenario $S_1$



(b) Priority values after slicing scenarios $S_1$ and $S_2$

Figure 3: Priority of each component after slicing $S_1$ and $S_2$.

simplicity, we are only considering five factors. Consideration of more factors makes the priority calculation method more accurate, but it will make the process more complicated and more confusing. In Table 7, we are assigning the weight of *1* to each priority factor. It may vary from application to application, and it is purely a subjective matter.

## 4. Experimental Studies

We have applied our proposed test effort prioritization method on three case studies implemented in JAVA, library management system (LMS), trading-house automation system (TAS), and automatic teller machine (ATM). The aim of the experiment is to check the effectiveness of our proposed prioritization-based testing method over the commonly used approach. The design part of these case studies is well explained in [35]. These are neither very small nor very large, but of moderate size. We present a brief summary of these case studies in Table 8 so that the size of each can be well understood. In this table, *object points* shown in Column 4 are estimated based on how many individual *screens* are displayed, how many *reports* are produced, and number of *3GL modules* developed in the system [39, 40]. The value of #Cl shown in Column 3 is from user classes. System classes are not considered here. For better understanding of the above case studies, the use case diagrams of the case studies are shown in Figure 4.

Table 6: Business values associated with various components of ATM system.

| Component | Value |
|---|---|
| Session | 0.93 |
| Withdrawal | 0.77 |
| Deposit | 0.69 |
| Transfer | 0.39 |
| Inquiry | 0.58 |
| Card reader | 1 |
| Envelope accepter | 0.69 |
| Customer console | 1 |
| Log | 0.86 |

Table 7: Priority calculation.

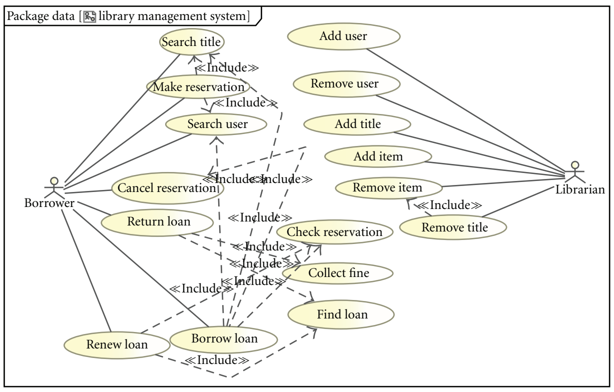
| Inf | EEC | SC | Severity | Val | TP |
|---|---|---|---|---|---|
| 0.67 | 0.3 | 0.3 | 0.75 | 0.36 | 2.38 |

First, we applied our proposed *influence* algorithm [12] to get the influence of classes of each case study. For this, we have developed a tool using ANTLR in the ECLIPSE framework to get the intermediate graph (ESDG) of the program. Average execution time of a class in a case study was decided by executing a case study 100 times (100 test cases were selected randomly based on operational profile data), collecting total execution time of a class at each run by the help of *Profiler* [23], and then calculating the average of total execution time. We calculated WMC for a class manually through the source code. To get the RFC, we have developed an algorithm that traverse the ESDG and generated RFC for a class in a case study. The severity of a class in a case study was decided by seeding faults in a chosen class and observing the type of system failures by executing the system in the operational environment. The group of students those have done the requirement analysis and design was assigned to check the severity of a class in a system. Domain experts were also involved for taking a proper decision on failure types. Finally, we have calculated priority value for each class as shown in Table 7 and generated a list of critical classes based on their priority value, in each case study. Once the critical classes in a system have been decided, our aim was to validate our result. For this, we have conducted experiments to check how the faults in these classes are affecting the reliability of the system. The experiment is described below.

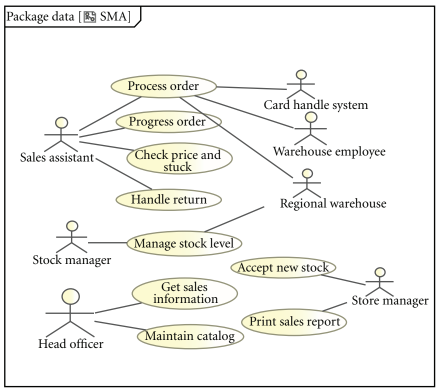
We have used fault seeding for evaluating the effectiveness of our proposed approach. It has been shown that fault seeding is an effective practice for measuring the testing method efficiency [41]. We have used some mutation operators to seed bugs randomly. The fault density is considered as a constant equal to 0.05 for each case study. This means that in a case study consisting of 1000 lines, 50 bugs were inserted. The seeded bugs are *class mutation operators* [42] and *interface mutation operators* [43, 44]. The class mutation operators are targeted at object-oriented specific features which java provides such as class declaration and references, single inheritance, interface, information

Table 8: Brief summary of our case studies.

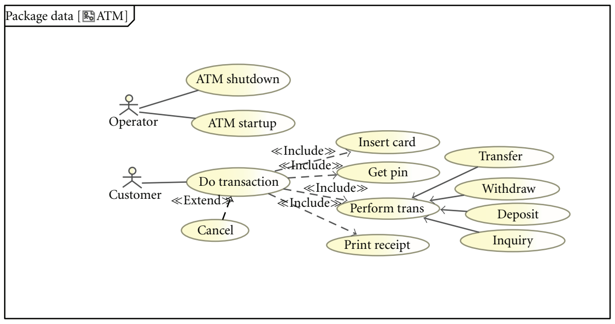
| System | LOCs | Use Cases number | Scenarios number | Classes number | Object-points number |
|---|---|---|---|---|---|
| LMS | 2486 | 16 | 56 | 18 | 153 |
| SMA | 1137 | 09 | 23 | 10 | 31 |
| ATM | 4217 | 12 | 30 | 22 | 82 |



(a) Use case diagram for LMS



(b) Use case diagram for SMA



(c) Use case diagram for ATM

Figure 4: Use case diagrams of the case studies.

hiding, and polymorphism and also provide coverage criteria with regard to these features. In this paper, the four classes mutation operators considered to simulate faults are

(i) CRT (compatible reference type) operator-type replacement: this operator replaces a reference type with all the compatible types (the name of other classes and interfaces) found from a cluster. There is a chance of subtle type errors by this mutant.

(ii) CON (constructor) operator-initial states and object replacement: a java class usually provides one or more constructors to capture the different ways of creating objects (constructor overloading). This operator replaces a constructor with other overloaded constructors. Some times, constructor of subclass may be replaced by constructor of super class by this operator. Object initialization error is related to this operator which happens frequently.

(iii) OVM (overriding method) operator-method replacement: this operator generates a mutant by deactivating the overriding method so that a reference to the overriding method actually goes to the overridden method. Actually an overriding method in a subclass has different functionality to the overridden method in a super class. So there is a chance of some semantic errors by this operator.

(iv) AMC (access modifier changes) access mode replacement: this operator replaces a certain Java access mode with three other alternatives such as private, protected and public. For example, a field declared with a protected access mode can be muted to private and public.

There is a number of interactions among components in an object-oriented application. Therefore, there are more opportunities for integration/interface faults. Delamaro et al. [43] have proposed interface mutation (IM) with the aim to test thoroughly the interactions among various units. Suppose, there are three functions $f1$, $f2$, and $f3$ within a system and to test the connection between $f1$ and $f3$, we insert mutants inside the component $f3$. In this case, these mutants may be identified through the test cases that execute calls to $f3$ from $f2$. As a result, the connection between $f1$–$f3$ is not tested. For this, there is a need to consider the proper place from where a function is called. Keeping this in view, we have carefully considered some *interface mutation operators* from the mutant set proposed in [43]. The IM is as follows.

(1) Applying mutants within the called function: the mutants considered under this category are *direct variable replacement operator*, *indirect variable operator,* and *return statement operators*.

(2) Applying mutants inside the calling function: it is applied to the call arguments. The mutants considered under this category are *unary operator insertion* and *function call deletion*. The last operator is a *missing transition*. It is not applied to the argument but to the whole function call. In a connection $f1$-$f2$, it deletes the call to the function $f2$. At the time of implementing the mutant inside an expression, special care is taken to replace it by an appropriate value if the deleted function is returning any value.

First, the testing time for each case study was decided based on the number of classes, complexity of each class, and number of object points [40] in the case studies. Then, we made two copies of the source code of each case study and applied two different testing methods. In the *first testing method*, the components are prioritized based on their structural complexity [13], whereas in the *second testing method*, the components are prioritized based on our proposed prioritization approach. The *first testing method* was applied to the *first copy,* and the *second testing method* was applied to the *second copy* of each case study.

Two separate groups were set for testing the two different copies of a case study. It is assumed that the competency level of both the groups are nearly equal. Same testing time was allocated for each copy of a case study. At this point, we emphasize the fact that our aim is not to achieve complete fault-coverage with this much test resources, but to check the efficiency of our proposed testing method. The number of test cases designed for a component at the unit level is decided based on their priority values (based on the complexity of a class in first copy and based on our proposed ranking method in the second copy). As a class with high *influence_value* provides more services to others (due to high influence_value), a single bug in the class may cause more interface bugs, which we cannot detect at the unit level. Interface bugs can be detected at the integration level to assure that the classes have communicated correctly. So, at the integration level, we have applied coupling-based testing techniques (client-server) [45]. In this approach [45] when, a class (client) is calling another classes (server) first, some method sequences of the client class are considered. These method sequences are subset of the set of method sequences decided at unit testing. Then, for each method sequence, the method sequences of the called class (server) are decided. At a time, one server class is considered for each client class. For one client method sequence, there can be number of server method sequences. In this level, the testing will be more effective if the method sequences of the client class will be more complete. As we have tested thoroughly the classes with high-priority value at the unit level in the second copy of a case study, we have considered the coupling-based integration testing [45] to cover all the possible interface faults of critical classes. We have taken the help of a coverage analysis testing tool *JaBUTi* [46] for getting the coverage analysis of test cases at the unit level for a case study. The example of the coverage report by two test cases at the unit level through JaBUTi is shown in Figure 5.

At the unit and integration level, though testing time is the same for the two copies of a case study, but the test set is different as the priority level of a component is different in different testing methods. After the completion of integration testing, we checked the mutation score $S$ of the test set generated for each copy of a case study.

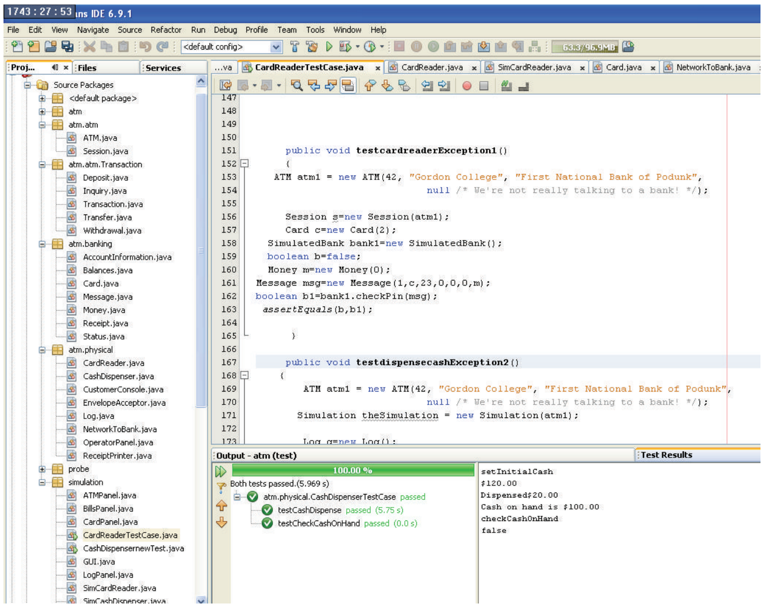
The mutation score for a test set $T$ is as follows:

$$\text{Mutation Score } (S, T) = \frac{\text{Total number of dead mutants}}{\text{Total number of mutants seeded} - \text{Total number of equival entmutants}}. \quad (6)$$

Table 9 shows the mutation score of generated test sets by two different testing methods.

From Table 9, it is observed that $\mathrm{MS}_T$ and $\mathrm{MS}_P$ are nearly equal. In LMS case study, mutation score by first testing method is more, whereas in TAS and ATM case studies, the mutation score by second testing method (our proposed prioritization-based testing method) is more. Hence, we cannot argue that our proposed method is better in finding mutants, but we found that our proposed testing method is also equally competent in finding mutants. We have claimed that though our proposed method is not able to expose more bugs as compared to the first testing method (test efforts decided based on complexity), but we are sure that our method is able to expose important bugs which are responsible for frequent failures and severe failures as we are considering both internal (average execution time, influence_value, and also structural complexity) and external factors (severity and business value) for testing effort prioritization. We have conducted experiments to prove this.

After resolving the detected bugs, we found that some residual bugs are existing in both copies of the case studies. A few bugs were detected towards the end of testing, which



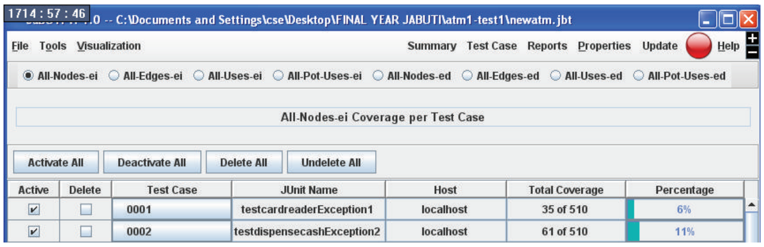
```java
    public void testcardreaderException1()
    {
  ATM atm1 = new ATM(42, "Gordon College", "First National Bank of Podunk",
                       null /* We're not really talking to a bank! */);

    Session s=new Session(atm1);
    Card c=new Card(2);
 SimulatedBank bank1=new SimulatedBank();
 boolean b=false;
 Money m=new Money(0);
Message msg=new Message(1,c,23,0,0,0,m);
boolean b1=bank1.checkPin(msg);
 assertEquals(b,b1);

     }

    public void testdispensecashException2()
    {
      ATM atm1 = new ATM(42, "Gordon College", "First National Bank of Podunk",
                       null /* We're not really talking to a bank! */);
     Simulation theSimulation = new Simulation(atm1);

      Log g=new Log();
```

Output - atm (test)

```
100.00 %
Both tests passed.(5.969 s)
atm.physical.CashDispenserTestCase passed
    testCashDispense passed (5.75 s)
    testCheckCashOnHand passed (0.0 s)
```

```
setInitialCash
$120.00
Dispensed$20.00
Cash on hand is $100.00
checkCashOnHand
false
```

(a) Test cases executed for the component Cash dispenser of ATM



All-Nodes-ei Coverage per Test Case

| Active | Delete | Test Case | JUnit Name | Host | Total Coverage | Percentage |
|---|---|---|---|---|---|---|
| ✓ | | 0001 | testcardreaderException1 | localhost | 35 of 510 | 6% |
| ✓ | | 0002 | testdispensecashException2 | localhost | 61 of 510 | 11% |

(b) Coverage shown by JaBUTi test tool

Figure 5: Test execution details of component Cash dispenser in ATM.

Table 9: Mutation score by two testing methods.

| Test | TC# | Mu# | EMu# | $MS_T$ | $MS_P$ |
|---|---|---|---|---|---|
| LMS | 112 | 22 | 2 | 0.89 | 0.82 |
| TAS | 73 | 17 | 0 | 0.74 | 0.77 |
| ATM | 211 | 31 | 7 | 0.8 | 0.89 |

TC#: number of test cases, Mu#: number of mutants, EMu#: number of equivalent mutants, $MS_T$: mutation score by first testing method (components are prioritized based on complexity), and $MS_P$: mutation score by second testing method (components are prioritized based on our proposed approach).

could not be fixed due to the shortage of testing time. At this point, we again emphasize the fact that our aim is not to achieve complete fault-coverage with a minimal test suite size. We have fixed a test budget for each case study before the testing phase and our aim is to ensure the efficiency of both testing methods within the available test budget. Therefore, after the completion of testing phase, we observed the effect of those residual bugs in both copies of each case study by invoking random services. For this, new system level test cases were randomly generated based on operational profile [2] for observing the behavior of the system at postrelease stage. At this point, we did not fix any detected bug. Analytical comparison of the two testing methods was done by running the same input set on the tested copies (one copy was tested by the traditional approach where the components are prioritized according to their structural complexity, and the other copy was tested by our proposed approach (discussed in Section 3) of each case study.

*4.1. Result Analysis.* The results of our simulation studies are summarized in Table 10. The headings used for the different columns of the table are listed below.

Test# is number of test cases in the test set.

$\text{Fail}_t$# denotes the number of failures observed in the first copy (testing efforts were assigned to various components based on the structural complexity).

$\text{Fail}_p$# is the number of failures observed in the second copy (testing efforts were assigned to various components based on our proposed testing effort prioritization technique).

*CS* is case study.

$F_{\text{Ca}}$, $F_{\text{Cr}}$, $F_{\text{Ma}}$, and $F_M$ represent the number of catastrophic, critical, marginal, and minor failures.

From Table 10, it is observed that the postrelease fail-ures are less in the second copy of a case study, to which our

Table 10: Failure observation at the time of release.

| *CS* | Test# | $\text{Fail}_t\#$ | | | | $\text{Fail}_P\#$ | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | $F_{\text{Ca}}$ | $F_{\text{Cr}}$ | $F_{\text{Ma}}$ | $F_M$ | $F_{\text{Ca}}$ | $F_{\text{Cr}}$ | $F_{\text{Ma}}$ | $F_M$ |
| LMS | 50 | 0 | **2** | 3 | 0 | 0 | 0 | 2 | 1 |
| | 100 | 0 | **5** | 3 | 4 | 0 | 0 | 3 | 3 |
| | 150 | **1** | **6** | 3 | 5 | 0 | **2** | 3 | 4 |
| TAS | 50 | 0 | **1** | 1 | 4 | 0 | 0 | 1 | 3 |
| | 100 | **1** | **1** | 1 | 4 | 0 | 0 | 2 | 4 |
| | 150 | **1** | **2** | 1 | 4 | 0 | **1** | 2 | 4 |
| ATM | 50 | 0 | **2** | 3 | 6 | 0 | 0 | 2 | 2 |
| | 100 | **0** | **2** | 4 | 6 | 0 | 0 | 4 | 2 |
| | 150 | **0** | **2** | 4 | 4 | 0 | **0** | 4 | 4 |

method is applied. Not only the number of failures is less, but also catastrophic and critical failures are rarely observed. Only some minor failures are observed in the copy tested by our approach ($\text{Fail}_P\#$) such as displaying results in some different format. This type of failures has very less effect on the system and also on customer. Some highly severed failures are observed in the first copy of each case study. It is because some critical bugs were detected towards the end of test cycle, which were not fixed due to shortage of testing time, whereas these critical bugs were detected at the early stage of test cycle in the second copy of a case study to which our approach was applied.

We are explaining a critical failure that is observed in the first copy of the case study "LMS", as shown in Table 10. As per the business rule of "LMS", a borrower can issue only one book, but we have violated this constraint by seeding a fault. As the first testing method gives importance to a class based on only the structural complexity not the value, the seeded bug (missing transition) could not be recovered due to a constraint in testing time. As a result, it allowed the same borrower to issue another book, who has already issued a book.

Through a detailed analysis of the results of both testing methods (testing effort prioritization based on complexity, testing effort prioritization based on our proposed method), we conclude that our proposed test effort prioritization helps to minimize the post-release failures of a system and also helps in minimizing the catastrophic and critical types of failures at the operational environment. As a result of this, user's perception on overall reliability of the system is improved. The efficiency of our proposed method will be increased if we run the software for long duration by taking more number of test cases based on operational profile.

In the discussed three case studies, we observed that the performance rate is drastically increased by our method, when the system is executed for a long time.

*4.2. Threats to Validity of Results.* In order to justify the validity of the results of our experimental studies, we got the following list of threats.

(i) Biased test set design and influencing results.

(ii) Seeding-biased errors in both the copies of each case study.

(iii) Testing only for selected failures and loosing generality of results.

(iv) Competency level of two testing groups are not same.

*Measures Taken to Overcome the Threats.* In order to overcome the above mentioned threats and validate the results for most common and real life cases, we have taken the following corrective measures.

(i) At the testing phase, same test efforts was allocated for both testing methods.

(ii) We have taken care that the seeded bugs match with commonly occurring bugs. For this, we have inserted some class mutation operators to seed bugs. Using mutation operators, we can ensure that a wide variety of faults are systematically inserted in a random fashion. While traditional mutation operators are restricted to a unit level, class mutation operators [42] have impact on cluster level.

(iii) We have taken in consideration the kind of failures which include almost all variety of bugs.

(iv) Each testing group contains five members. The competency level of each tester was tested based on questionnaires and formed two groups. Both groups have same competency level.

## 5. Related Work

Researchers have proposed a variety of prioritization-based testing techniques in order to do the best possible job with limited resources. These techniques make the testing process more effective and cheaper. An effective test could find more number of defects or more important defects in the same amount of time. As testing is expensive and time consuming, the test manager has to choose among alternatives, not use them all. A lot of work has been done on *test case prioritization,* but work on *source code prioritization* has scarcely been reported. It is a research area on improving testing before test case generation. First, we discuss the work on

Table 11: Comparison of existing work on code prioritization with our work.

| Comparison criteria | Li's work [47] | This work |
|---|---|---|
| Aim of code prioritization | Quickly improve code coverage (control-flow-based coverage) | Quickly improve user's perception on the reliability of the system |
| Priority level | Line of code or block | Method or class |
| Analysis point | Structural | Structural and behavioral |
| Factors for prioritization | Number of lines of code covered | Influence, execution time, structural complexity, severity, and business value |
| Intermediate graph used | Control flow graph | ESDG (both data and control flow) |
| Fault criticality | Equal importance to the discovery of each fault | Faults in high-priority areas are more critical and hence are given more importance. |
| Effectiveness | Finding more numbers of bugs | Finding bugs that have more contribution to unreliability |

*source code prioritization* and compare it with our proposed work. Then, we discuss the work on fault-proneness of a component which discuss the methods to identify faulty components within a system. Finally, we discuss on another related research area on pretesting efforts, *usage-based testing*, with the objective to improve the reliability of a system. Lastly, we present a review on the reported work on some *test case prioritization* techniques.

*5.1. Code Prioritization.* The basic aim of this technique is to focus on testing efforts before the generation of test cases. It helps to prioritize the testing efforts based on test objectives. With a prioritized testing effort and focused test architecture, test cases are created and executed.

Li [47] has proposed a priority calculation method that prioritizes and highlights the important parts of the source code based on *dominator analysis*, that need to be tested first to quickly improve the code coverage. Before test construction, Li's method decides which line of code will be tested first to quickly improve code coverage. According to his approach, first the intermediate representation of the source code, known as control flow graph (CFG) is constructed. Then, a node (a node is a basic block in the source code.) of the CFG is prioritized based on measuring quantitatively how many lines of code are covered by testing that node. A weight is calculated for each node considering only the coverage information. It does not take into account, for instance, the complexity or the criticality of a given part of the program. A test case covering the highest weight node will increase the coverage faster (the tester, based on his/her experience, may desire to cover first a node with a lower weight but that has higher complexity or criticality.). There are two kinds of code coverage such as control-flow based and data flow based. Li's work focuses on control flow coverage. Code coverage helps the developers and vendors to indicate the confidence level in the readiness of their software, but the limitation is that it gives equal importance to the discovery of each fault. So, no information is gained on how much it affects the reliability of a system by detecting and eliminating a failure during testing process, as different failures have different contribution to the reliability of a system. Though Li's work [47] is more effective at finding bugs, but it oftenly spends more resources by uncovering many failures having occurrence rate very much negligible during actual operation. As a result of this, some times test efforts are wasted without appreciably improving the reliability of the software. Therefore, this code prioritization technique could not always detect the bugs, which are responsible for frequent failures, when the system is executed for some duration by providing some random inputs. Unlike Li's work [47] on code prioritization, our code prioritization strategy analyzes failure-prone parts in the source code and able to focus first on detecting the faults that cause the most frequent failures. The failure-prone parts are identified based on influence metric and operational profile. We compare our work with the existing work on code prioritization [47] according to the following six criteria given in Table 11.

Li et al. [48] presented a methodology for code coverage-based path selection and test data generation, based on Li's previous work [47]. They [48] proposed a path selection technique that considers the program priority and call relationships among class methods to identify a set of paths through the code, which has high-priority code unit. Then, constraint analysis method is used to find object attributes and method parameter values for generating tests to traverse through the selected sequence of paths. It helps to automatically generate tests to cover high priority points and minimize the cost of unit testing.

We have proposed a few testing effort prioritization approach [11, 12, 18] based on code analysis and also on UML models. In that work, we have proposed a code prioritization method based on only internal parameters. In that approach, we have assumed that the effect of each failure is equal, which is not true. In this paper, the external parameters: severity and business value are added for prioritizing components.

*5.1.1. Fault-Proneness-Based Testing.* The work on fault-proneness identifies faulty components in a system, and testing effort prioritization is done accordingly. It estimates the probability of the presence of faults in a component, which helps to take valuable decisions on testing. A lot of research work have been done to identify the fault-prone components in a system [49–52], which are very relevant to our work. Different authors have focused on different characteristics associated with a component for counting faults.

Eaddy et al. [49] have experimentally proved that concern-oriented metrics (a concern is anything a stake holder may want to consider as a conceptual unit, including features, nonfunctional requirements, and design idioms.) are more appropriate predictors of software quality than structural complexity measures, and there is a strong relationship between scattering and defects. Ostrand et al. [50] have proposed a novel approach to identify the faulty files in the next release. For prioritizing testing efforts, their approach considers the factors that are obtained from the modification requests and the version control system. These factors are (i) the file size, (ii) whether the file was new to the system, (iii) fault status in previous release (whether the file contained faults in earlier releases, and if so, how many), (iv) number of changes made, and (v) programming language used for implementation. For some initial releases, the models were customized based on the above observed factors. Based on the experimental results, the authors conclude that their methodology can be implemented in the real world without extensive statistical expertise or modeling effort. Ostrand et al. [52] proposed a negative binomial regression model. The binomial model is used to predict the expected number of faults in each file of the next release of a system. The predictions are based on the code of the file in the current release, fault, and modification history of the file from previous releases. El Emam et al. found that a class having high-export coupling value is more fault-prone [51]. A complex program might contain more faults compared to a simple program [53]. As the factor complexity is the most important defect generator, the *complexity metric* is used as a parameter for testing [54, 55].

Some researchers have proposed prediction of faulty components from design metric at the architectural level. Researchers [14, 54] relate the structural complexity metric (CK metric suite [13]) to fault-proneness. From these papers, it is observed that the estimated defect density (fault-proneness) through static analysis and the prerelease defect density computed by testing is strongly correlated. El Emam et al. [56] have experimentally proved that inheritance and external coupling metrics are strongly associated with fault-proneness. Unlike these papers, our aim is not to investigate the characteristics of various components to check which components have high fault densities. Our aim is to make the testing process more effective by finding more and more important defects and improving the reliability of a system without increasing the testing budget.

*5.2. Usage-Based Testing.* The first step in usage-based testing is to develop a usage model that describes the anticipated behavior (usage) of the system under test. Usage Model helps to improve the user's perception on the system's reliability. It is also designed before the construction of test cases. It basically represents the events and transition between events in the system, where events can be user input or environmental input. Its main purpose is to describe the possible behaviors of the user and to quantify the actual usage in terms of probabilities for different user's behavior. Though unlike our approach, no prior knowledge of the program is necessary for usage-based testing. The authors found that a lot of research work have been done on usage-based testing at the abstract level based on operational profile [2], which focus on detecting faults that cause the most frequent failures. By prioritizing our tests based on usage probabilities, it ensures that the failures that will occur most frequently in operational use will be found early in the test cycle, while keeping testing effort to a minimum. In terms of mean time between failure (MTBF), Cobb and Mills [3] say that "It is shown based on a study of a number of projects that usage testing improves the perceived reliability during operation 21 times greater than that using coverage testing." However, these works have concentrated on selection of test cases based on black-box approach compared to our white-box approach. As a result, it ignores the structural (syntactic and semantic) relationships that exist among the elements of source code, which is vital for reliability assessment. Sometimes the infrequently executed code of a complex module that is a source of failure could not be tested by this method. Cheung [10] has proposed a user-oriented software reliability model, which measures the quality of service that a program provides to a user. His Markov reliability model uses a program flow graph to represent the structure of the system. The flow graph structure is obtained by analyzing the code. It uses the functional modules as the basic components whose reliabilities can be independently measured. It uses branching and function-calling characteristics among the modules as the user profile so that they can be easily measured in the operational environment. Similar structural models have been proposed by Littlewood [57] and Booth [58] to analyze the failure rates of a program. Lo et al. [59] have proposed a structural model for estimating the reliability of component-based programs where the software components are heterogeneous, and the transfer of control between components follows a discrete-time Markov process. However, in all the related techniques discussed above, the data dependency among the components has been ignored. As a result, the tester finds it very hard to describe the interdependency of software failures in detail.

*5.3. Test Case Prioritization.* There has been a large number of work [60–63] reported on prioritizing test cases. A meaningful prioritization of test cases can enhance the effectiveness of testing within the same testing effort. The authors of the papers [60–62] have proposed test case prioritization techniques to reduce the cost of regression testing based on total requirement and additional requirement coverage. Their basic aim is to improve a test suite's fault detection rate. Elbaum et al. [62] have added two major attributes such as test cost and fault severity to each element of the test suit. They have experimentally validated that the test suites executed based on prioritization technique always outperform unprioritized test suites in terms of fault detection rate. The authors of the paper [63] have also proposed a test case prioritization technique for regression testing using relevant slicing. These discussed prioritization techniques are used to improve testing by ordering the test cases in a test suite for testing.

Though an ordering of test cases helps to find more number of defects at the early phase of testing, it requires to run all test cases of a large test suite at the time of regression testing, which is impossible. To solve this problem, Sapna and Mohanty [64] have used clustering to select a subset of scenarios for testing. Unlike these discussed test case prioritization techniques, our aim is not to increase the rate of fault detection at the time of regression testing, which is generally applicable during maintenance phase. Our approach identifies the critical parts of the source code at the development phase and detects more faults from the critical parts to minimize the postfailure rate without increasing the testing effort.

Bryce et al. [65] have proposed various prioritization criteria for prioritizing test cases. These are parameter-value interaction, coverage-based, count-based and frequency-based. They have applied them to specifically some stand-alone GUI and Web-based applications and found that the fault detection rate is increasing over random ordering of test cases.

## 6. Conclusion

We have proposed a test effort prioritization approach to rank components at the code level for testing. The degree of thoroughness with which an element is tested is made proportional to its priority value. This helps to detect more faults from the critical parts of the source code. We have considered five important factors of a component: *influence_value* (influence towards system failures), *average execution time*, *structural complexity*, *severity,* and *business value* for computing the criticality of the component and prioritizing the components according to their criticality. Our test effort prioritization method guides the tester to detect the important bugs at the early phase of testing that are responsible for frequent or severe failures. As a result, the user's perception on the reliability of the system is improved within the available test budget. Our approach helps to increase the test efficiency as it is linked to the measure for both internal factors (influence_value and structural complexity) and external factors (severity and business value) of a program element.

We have validated our claim experimentally by comparing our proposed approach with a related schema in which the components are prioritized according to their complexity. Our experimental results show that the rate of postrelease failures are minimized, and also the severe failures are minimized in our approach, when the system is executed for some duration at the operational environment. Analysis of the source code of a software in this way before test case generation helps software management team to locate the elements, those are critical but getting less attention in terms of testing.

## References

[1] S. Naik and P. Tripathy, *Software Testing and Quality Assurance: Theory and Practice*, Willey, 2008.

[2] J. D. Musa, "Operational profiles in software-reliability engineering," *IEEE Software Magazine*, vol. 10, no. 2, pp. 14–32, 1993.

[3] R. H. Cobb and H. D. Mills, "Engineering software under statistical quality control," *IEEE Software*, vol. 7, no. 6, pp. 45–54, 1990.

[4] E. N. Adams, "Optimizing preventive service of software products," *IBM Journal of Research and Development*, vol. 28, no. 1, pp. 2–14, 1984.

[5] B. Boehm and V. R. Basili, "Software defect reduction top 10 list," *Computer*, vol. 34, no. 1, pp. 135–137, 2001.

[6] B. Boehm and L. G. Huang, "Value-based software engineering: a case study," *Computer*, vol. 36, no. 3, pp. 33–41, 2003.

[7] Q. Li, M. Li, Y. Yang et al., "Bridge the gap between software test process and business value: a case study," in *Proceedings of the International Conference on Software Process: Trustworthy Software Development Processes (ICSP '09)*, 2009.

[8] I. Sommerville, *Software Engineering*, Pearson, 5th edition, 1995.

[9] J. D. Musa, *Software Reliability Engineering: More Reliable Software Faster and Cheaper*, AuthorHouse, 2004.

[10] R. C. Cheung, "A user-oriented software reliability model," *IEEE Transactions on Software Engineering*, vol. 6, no. 2, pp. 118–125, 1980.

[11] M. Ray and D. P. Mohapatra, "Reliability improvement based on prioritization of source code," in *6th International Conference on Distributed Computing and Internet Technology (ICDCIT '10)*, T. Janowski and H. Mohanty, Eds., vol. 5966 of *Lecture Notes in Computer Science*, pp. 212–223, Springer, 2010.

[12] M. Ray, K. Lal Kumawat, and D. P. Mohapatra, "Source code prioritization using forward slicing for exposing critical elements in a program," *Journal of Computer Science and Technology*, vol. 26, no. 2, pp. 314–327, 2011.

[13] S. R. Chidamber and C. F. Kemerer, "Metrics suite for object oriented design," *IEEE Transactions on Software Engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[14] L. C. Briand, J. Wüst, J. W. Daly, and D. Victor Porter, "Exploring the relationships between design measures and software quality in object-oriented systems," *Journal of Systems and Software*, vol. 51, no. 3, pp. 245–273, 2000.

[15] A. Hassan, K. Goseva-Popstojanova, and H. Ammar, "UML based severity analysis methodology," in *Proceedings of the Annual Reliability and Maintainability Symposium (RAMS '05)*, pp. 158–164, Alexandria, Va, USA, January 2005.

[16] R. Ramler, S. Biffl, and P. Grnbacher, *Value-Based Management of Software Testing*, Springer, Berlin, Germany, 2005.

[17] B. Boehm, "Value-based software engineering: reinventing," *ACM SIGSOFT Software Engineering Notes*, vol. 28, no. 2, pp. 3–10, 2003.

[18] M. Ray and D. P. Mohapatra, "A scheme to prioritize classes at the early stage for improving observable reliability," in *the 3rd India Software Engineering Conference (ISEC '10)*, pp. 69–72, February 2010.

[19] N. Nagappan, T. Ball, and B. Murphy, "Using historical in-process and product metrics for early estimation of software failures," in *Proceedings of the 17th International Symposium on Software Reliability Engineering*, pp. 62–74, 2006.

[20] L. Larsen and M. J. Harrold, "Slicing object-oriented software," in *Proceedings of the 18th International Conference on Software Engineering (ICSE '96:)*, pp. 495–505, 1996.

[21] M. Weiser, "Program slicing," *IEEE Transactions on Software Engineering*, vol. SE-10, no. 4, pp. 352–357, 1984.

[22] Y. N. Srikant and P. Shankar, Eds., *The Compiler Design Handbook: Optimizations and Machine Code Generation*, CRC Press, 2002.

[23] Profiler, http://www.sics.se/man2html/gprof.1.html.

[24] M. H. Tang, M. H. Kao, and M. H. Chen, "Empirical study on object-oriented metrics," in *Proceedings of the 6th International Software Metrics Symposium*, pp. 242–249, November 1999.

[25] S. R. Luke, "Failure mode, effects and criticality analysis (fmeca) for software," in *5th Fleet Maintenance Symposium*, pp. 731–735, Virginia Beach, Va, USA, October 1995.

[26] L. H. Rosenberg, R. Stapko, and A. Gallo, "Risk-based object oriented testing," in *Proceedings of the 24th Annual Software Engineering Workshop*, NASA, Software Engineering Laboratory, 1999.

[27] J. A. Whittaker and M. G. Thomason, "Markov chain model for statistical software testing," *IEEE Transactions on Software Engineering*, vol. 20, no. 10, pp. 812–824, 1994.

[28] D. Department, "Procedures for performing a failure mode, effects, and criticality analysis," US MIL STD 1629A/Notice 2, November 1984.

[29] N. Ozarin, "Failure modes and effects analysis during design of computer software," in *Proceedings of the Annual Reliability and Maintainability Symposium*, pp. 201–206, January 2004.

[30] P. Vyas and R. K. Mlittal, "Operation level safety analysis for object oriented software design using sfmea," in *WEE International Advance Computing Conference*, Patiala, India, March 2009.

[31] S. M. Yacoub and H. H. Ammar, "A methodology for architecture-level reliability risk analysis," *IEEE Transactions on Software Engineering*, vol. 28, no. 6, pp. 529–547, 2002.

[32] K. Goseva-Popstojanova, A. Hassan, A. Guedem et al., "Architectural-level risk analysis using UML," *IEEE Transactions on Software Engineering*, vol. 29, no. 10, pp. 946–959, 2003.

[33] I. Jacobson and M. Christerson, *Object-Oriented Software Engineering: A Use Case Driven Approach*, Addison-Wesley, 1992.

[34] L. Huang and B. Boehm, "How much software quality investment is enough: a value-based approach," *IEEE Software*, vol. 23, no. 5, pp. 88–95, 2006.

[35] R. Mall, *Fundamentals of Software Engineering*, Prentice Hall of India, 3rd edition, 2009.

[36] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," in *Proceedings of the ACM SIGPLAN Conference on Programming Language design and Implementation (PLDI '88)*, pp. 35–46, 1988.

[37] S. Bhattacharya and A. Kanjilal, "Code based analysis for object-oriented systems," *Journal of Computer Science and Technology*, vol. 21, no. 6, pp. 965–972, 2006.

[38] S. Yacoub, B. Cukic, and H. H. Ammar, "A scenario-based reliability analysis approach for component-based software," *IEEE Transactions on Reliability*, vol. 53, no. 4, pp. 465–480, 2004.

[39] B. Boehm, B. Clark, E. Horowitz, C. Westland, R. Madachy, and R. Selby, "Cost models for future software life cycle processes: Cocomo 2.0," 1995.

[40] O. Point, 2008, http://sunset.usc.edu/csse/research/COCOMOII/cocomo_main.html.

[41] J. H. Andrews, L. C. Briand, and Y. Labiche, "Is mutation an appropriate tool for testing experiments?" in *the 27th International Conference on Software Engineering (ICSE '05)*, pp. 402–411, May 2005.

[42] S. K. John, J. A. Clark, and J. A. Mcdermid, "Class mutation: mutation testing for object-oriented programs," in *Proceedings of the Net.ObjectDays Conference on Object-Oriented Software Systems*, pp. 9–12, 2000.

[43] M. E. Delamaro, J. C. Maldonado, A. Pasquini, and A. P. Mathur, "Interface mutation test adequacy criterion: an empirical evaluation," *Empirical Software Engineering*, vol. 6, no. 2, pp. 111–142, 2001.

[44] R. Binder, *Testing Object-Oriented Systems—Models, Patterns and Tools*, Addison-Wesley, 2000.

[45] L. C. Briand, Y. Labiche, and Y. Wang, "A comprehensive and systematic methodology for client-server class integration testing," in *Proceedings of the 14th International Symposium on Software Reliability Engineering (ISSRE '03)*, p. 14, Washington, DC, USA, 2003.

[46] A. M. R. Vincenzi, W. E. Wong, M. E. Delamaro, and J. C. Maldonado, "Jabuti java bytecode understanding and testing: users guide," Carlos,S. and SP, Brazil, March 2003.

[47] J. J. Li, "Prioritize code for testing to improve code coverage of complex software," in *Proceedings of the 16th IEEE International Symposium on Software Reliability Engineering (ISSRE '05)*, pp. 75–84, 2005.

[48] J. J. Li, D. Weiss, and H. Yee, "Code-coverage guided prioritized test generation," *Information and Software Technology*, vol. 48, no. 12, pp. 1187–1198, 2006.

[49] M. Eaddy, T. Zimmermann, K. D. Sherwood et al., "Do crosscutting concerns cause defects?" *IEEE Transactions on Software Engineering*, vol. 34, no. 4, pp. 497–515, 2008.

[50] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Automating algorithms for the identification of fault-prone files," in *Proceedings of the International Symposium on Software Testing and Analysis (ISSTA '07)*, pp. 219–227, July 2007.

[51] K. El Emam, W. Melo, and J. C. Machado, "The prediction of faulty classes using object-oriented design metrics," *Journal of Systems and Software*, vol. 56, no. 1, pp. 63–75, 2001.

[52] T. J. Ostrand, E. J. Weyuker, and R. M. Bell, "Predicting the location and number of faults in large software systems," *IEEE Transactions on Software Engineering*, vol. 31, no. 4, pp. 340–355, 2005.

[53] J. C. Munson and T. M. Khoshgoftaar, "The detection of fault-prone programs," *IEEE Transactions on Software Engineering*, vol. 18, no. 5, pp. 423–433, 1992.

[54] R. Subramanyam and M. S. Krishnan, "Empirical analysis of CK metrics for object-oriented design complexity: implications for software defects," *IEEE Transactions on Software Engineering*, vol. 29, no. 4, pp. 297–310, 2003.

[55] M. R. Lyu, "Software reliability engineering: Aroadmap," in *Future of Software Engineering (FOSE '07)*, pp. 153–170, 2007.

[56] K. El Emam, W. Melo, and J. C. Machado, "The prediction of faulty classes using object-oriented design metrics," *Journal of Systems and Software*, vol. 56, no. 1, pp. 63–75, 2001.

[57] B. Littlewood, "A reliability model for systems with markov structure," *Journal of the Royal Statistical Society. Series C*, vol. 24, no. 2, pp. 172–177, 1975.

[58] T. L. Booth, "Performance optimization of software systems processing information sequences modeled by probabilistic languages," *IEEE Transactions on Software Engineering*, vol. SE-5, no. 1, pp. 31–44, 1979.

[59] J. H. Lo, S. Y. Kuo, M. R. Lyu, and C. Y. Huang, "Optimal resource allocation and reliability analysis for component-based software applications," in *the 26th Annual International Computer Software and Applications Conference (COMPSAC '02)*, pp. 7–12, August 2002.

[60] G. Rothermel, R. H. Untcn, C. Chu, and M. J. Harrold, "Prioritizing test cases for regression testing," *IEEE Transactions on Software Engineering*, vol. 27, no. 10, pp. 929–948, 2001.

[61] S. Elbaum, A. G. Malishevsky, and G. Rothermel, "Test case prioritization: a family of empirical studies," *IEEE Transactions on Software Engineering*, vol. 28, no. 2, pp. 159–182, 2002.

[62] S. Elbaum, A. Malishevsky, and G. Rothermel, "Incorporating varying test costs and fault severities into test case prioritization," in *Proceedings of the 23rd International Conference on Software Engineering (ICSE '01)*, pp. 329–338, 2001.

[63] D. Jeffrey and N. Gupta, "Experiments with test case prioritization using relevant slices," *Journal of Systems and Software*, vol. 81, no. 2, pp. 196–221, 2008.

[64] P. G. Sapna and H. Mohanty, "Clustering test cases to achieve effective test selection," in *Proceedings of the 1st Amrita ACM-W Celebration of Women in Computing in India (A2CWiC '10)*, September 2010.

[65] R. C. Bryce, S. Sampath, and A. M. Memon, "Developing a single model and test prioritization strategies for event-driven software," *IEEE Transactions on Software Engineering*, vol. 37, no. 1, pp. 48–64, 2011.