GUI Testing Assisted by Human Knowledge: Random vs. Functional

Weiran Yang¹, Zhenyu Chen¹,∗, Zebao Gao¹,², Yunxiao Zou¹, and Xiaoran Xu¹,³

¹State Key Laboratory for Novel Software Technology, Nanjing University, Nanjing 210093, China
²Department of Computer Science, University of Maryland, College Park, MD 20742, USA
³Department of Computer Science, William Marsh Rice University, Houston, TX 77005, USA

*Corresponding author: zychen@software.nju.edu.cn

Abstract

Software testing is a labor-intensive task in software development life-cycle. Human knowledge is useful in the practices of software testing, especially GUI testing. There are many strategies for GUI testing assisted by human knowledge, in which manual random testing and manual functional testing are two of widely used ones. In this paper, an empirical study is conducted to compare random testing and functional testing in order to provide guidelines for GUI testing. 234 participants were recruited to create thousands of random and functional test cases for open source GUI applications. Some of these test cases were selected with certain coverage criteria and then run on GUI applications to evaluate random testing and functional testing. We study three aspects on the two testing strategies: effectiveness, complementarity and impact of test case length. Some useful observations in the empirical study are: (1) Random testing is more effective in the early stage of testing on small applications and functional testing has more extensive applicability for testing large sized applications. (2) Random testing and functional testing exhibit some complementarity in our experiment. (3) Short test cases can reveal some faults more quickly and long test cases can reveal more faults lasting.

Keywords: Random Testing; Functional Testing; GUI Testing; Fault Detection Capability; Human Knowledge
1. Introduction

Software testing is a laborious and expensive task in software development life-cycle. In the past decades, software engineering research has put much emphasis on automation of different tasks, in order to reduce the cost of software development and maintenance. However, 100% automation is still a dream, or even an illusion, for current software testing in industry [1]. Human knowledge still plays a key role in the practices of software testing.

Software applications equipped with GUIs (Graphical User Interface) help promote ease-of-use. However, the other side of coin is that GUIs cause difficulties to software testing [2][3][4]. In current industrial practices, software testers generate test cases for GUI applications manually, although these test cases might be executed automatically [5]. Random testing may be the simplest testing strategy and is widely used. GUI events are triggered randomly on GUI applications to develop test cases. Manual random testing depends little on human knowledge, i.e. the intuitive understandings of GUIs. Researchers cast doubt on the effectiveness of random testing [6] [7] [8] because random testing neglects the knowledge of software specification and underlying software structure. Nevertheless, Arcuri et al. [9] surveyed and analyzed the properties of random testing, and found that random testing performed better than a group of other testing strategies.

Functional testing is one of the most widely used strategies of GUI testing [10]. It is often required that testers conducting functional testing must understand applications’ functions according to software specifications and even have some domain knowledge. Testers should list functional points for the software under test and design test cases to cover all these functional points. Functional testing is used to assure that all functions of applications are adequately tested. Obviously, functional testing requires more human interventions than random testing.

The cost-effectiveness of a testing strategy is important for software testing of which resources are limited in many cases. Which testing strategy is more cost-effective [11] between functional testing and random testing, is controversial, especially in the area of GUI testing. Therefore, we conduct an empirical study to compare random testing with functional testing. We are interested in the debate on which one is the better between these two testing strategies and want to reap a tangible experimental result rather than an in-
tuitive conclusion. We evaluate their effectiveness on some common metrics. The complementarity of these two testing strategies is also analyzed in our study. In addition, the test pool is divided into five parts according to the length of test cases, which is a key factor for the effectiveness of testing[12]. And then test cases in each length division are executed separately for comparison.

We study the following three questions in our experiments.

- **RQ1**: Which one is more effective: random or functional, with regard to the number of detected faults?

- **RQ2**: Are functional testing and random testing complementary to each other?

- **RQ3**: Does test case length play a key role in fault detection capability?

In order to conduct the experiment with sufficient data, 234 participants were recruited to create random test cases on two open source GUI applications. A month later, the functional testing specifications were provided to these participants for creating test cases to cover all functional points. As a result, thousands of test cases were collected. Aiming at ensuring the quality of this experiment, these test cases were inspected and selected prudently to meet experimental requirements. Then the selected test cases were sampled to run on GUI applications. The faults detected by test cases were used as the basic metric to evaluate the effectiveness, the complementarity, and other factors.

The major observations of this study are:

1. Random testing is more effective than functional testing in the early stage of testing. However, functional testing can detect more faults as the number of test cases increases. Overall, random testing is more effective on small applications and functional testing works better on large applications.

2. The different fault detection ratios of two strategies indicate the complementarity is a possibility. Our experimental results exhibit the complementarity of these two testing strategies on some faults.

3. In order to verify whether test case length affects the effectiveness significantly. We computed the 20th, 40th, 60th and 80th percentiles of...
test case length of all test cases in order to divide the test pool into five parts with regard to length. The results show that short test cases can reveal faults more quickly while long test cases can reveal more faults as time goes on.

The next section introduces the experimental design. Section 3 describes and analyzes the experimental results. Section 4 gives some guidance on software testing in industrial practices. Threats to validity of this study are also discussed in Section 4. Section 5 discusses related and future work, and Section 6 concludes this paper.

2. Experiment Design

This study is organized following a definite procedure shown in Fig. 1.

2.1. Experimental Procedure Overview

The whole experimental procedure is divided into the following 4 steps.

**Step 1-a. Create Test Cases:**

- **Random Testing:** In the very initial stage of this study, 234 participants were recruited to develop test cases randomly for testing applications under test (AUTs). No instruction or requirement is given on AUTs’ functionalities in this phase.
- **Functional Testing:** 30 days later, participants were instructed to learn the function sets and the business logic of the AUTs. Thus, provided with a definite functional testing requirement which offered clear descriptions of function sets of AUTs and guidance on the experiment clearly, participants developed test cases to meet the functional test requirements.

**Step 1-b. Create Oracle:** The oracle undertakes the responsibility for monitoring the runtime information of AUTs and generating test reports for each test case. Firstly, source code is scanned for reported faults and improperly handled exceptions and instrumented to log runtime information of each test case. After all test cases are executed, test reports are generated which indicate fault detection information of each failed execution. More detailed information of creating oracle is stated in Section 2.5.

**Step 2. Create Test Suites:** Thousands of test cases are generated in Step 1-a and Step 1-b based on random testing strategy and functional testing strategy, respectively. All test cases are inspected manually and those which can not be executed successfully are deleted. Then two test pools – functional test pool and random test pool are constituted by the remained test cases. Finally, various test suites were constructed according to different strategies and sampled from test pools. More detailed information is stated in Section 2.6.

**Step 3. Compare** All test cases in each test suite are executed, and logs of runtime information are studied to generate test reports to link each test case with detected faults during its execution. We collect these reports and make further analysis. Experimental results and advices on improving the effectiveness of testing come after evaluating a large amount of collected data.

### 2.2. Applications Under Test (AUTs)

Two GUI applications, Crossword Sage and OmegaT, are selected for this study. These GUI applications are free and open source on SourceForge. The applications, written in JAVA, were also used in some other studies, such as [13] [14].

---

Both of these two applications present unambiguous graphical user interfaces to users. Therefore, it is easy for testers to discern different function sets of AUTs predefined in the functional testing requirement.

Basic information of AUTs is shown in Table 1. Crossword Sage contains 80 GUI widgets, and OmegaT contains 519 GUI widgets. Hence, Crossword Sage can be taken as a representative of small GUI applications, and OmegaT is a larger one.

<table>
<thead>
<tr>
<th>Application</th>
<th>Version</th>
<th># Widgets</th>
<th># Faults</th>
<th># Test Cases</th>
<th># Detected Faults</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crossword Sage</td>
<td>0.3.3</td>
<td>80</td>
<td>14</td>
<td>738</td>
<td>1278</td>
</tr>
<tr>
<td>OmegaT</td>
<td>1.8.1.07</td>
<td>519</td>
<td>129</td>
<td>929</td>
<td>1476</td>
</tr>
</tbody>
</table>


2.3. Fault Information

AUTs with known faults are used as experimental subjects for evaluating the effectiveness of various testing strategies. Known faults are real faults which exist in the AUTs and can be detected by test cases and they are often utilized as an important metric for evaluating testing methods [31]. However, how to obtain known faults is a crucial issue in the experiment design.

Some faults can be detected and tracked in the testing phase before the software is released. And after software release phase, users may submit fault reports to help software evolution. These faults imply authentic problems of software systems and can be naturally regarded as known faults.

However, only the developer-found and user-reported faults insufficiently constitute known faults. For open source software, the faults detected by developers are usually fixed before the software is publicly available. And it is difficult to replay and locate every user-reported faults, for some fault reports lack clarity and are ambiguous. We read all fault reports in the fault tracking system on Sourceforge for all the experiment AUTs, namely Crossword Sage and OmegaT. According to these fault reports, only 14 original faults of Crossword Sage, among which 4 are failures and 10 are unexpected exceptions, are identified. These 4 failures are distinct user-discernible faults. However, no faults were found in OmegaT, although all submitted fault reports were read.

Noticeably, after an overall reading on these fault reports, we find that plenty of reported faults are exceptions revealed during execution. Some of
these exceptions affect user interfaces(UI) in an unexpected way, and some more dangerous ones even cause crashes.

The term “Exception” in software programs indicates that “an abnormal operation is executed”. Exceptions will be triggered when exceptional operations are executed. For instance, inputting a null value may reveal the NullPointerException in JAVA programs.

Generally, in JAVA programs, all errors and exceptions extend the java.lang.Throwable [15]. Exceptions can be classified into two categories: checked exceptions and unchecked exceptions. The previous ones must be thrown or caught definitely in JAVA programs but the unchecked ones are unconstrained by this rule [16]. Runtime exceptions that inherit from the java.lang.RuntimeException, may be triggered during the execution of applications. They are hard to be recognized by developers. Checked and unchecked exceptions can result in various faults in execution. Without given proper exception-handling mechanisms, exceptions will affect adversely the software’s robustness and practicality [17].

Some studies proposed considerable problems which are the principle causes of improper exception-handling mechanisms [15][18][19]. The common problems on exception-handling are listed below:

- Problem 1: Exceptions are not handled.
- Problem 2: The exception-handling block is empty or without any useful codes, such as just logging it with some obscure words.
- Problem 3: The exception-handling block is designed for more than one distinct type of exceptions which can not be further derived.
- Problem 4: The exception-handling block is designed for a supper-exception aiming to handle all of its inherited exceptions.

All mentioned problems threaten the reliability of software applications. Although un-triggered exceptions can be harmless [20], once exceptions are triggered, it is exceedingly expensive. These exceptions, which are un-handled or handled but in an ill-suited way, distribute naturally as potential faults in the source code.

With regard to potential properties of exceptions, namely, the detectability and threats to programs, some researchers employed natural exceptions in applications as known faults which would be detected by test cases [21][22][23].
In this paper, natural exceptions with any problems listed above were employed as known faults to evaluate the fault detection capability of various testing strategies. We choose all improperly handled exceptions in OmegaT, concretely, 129 exceptions are found out. In the rest of this article, we also call these exceptions as known faults. Fig. 2 shows an overview of where the known faults, which will be checked by the oracle, come from.

2.4. Test Case Generation

As an empirical study, we want to get as much data as possible to evaluate random testing and functional testing. For the sake of collecting considerable test cases on these two testing strategies, 234 undergraduate students majoring in software engineering were recruited to work as testers. A popular automatic testing framework, HP QuickTest Professional (QTP) [24], was used in our study for creating test cases with two testing strategies. The test case generation is divided into two phases: manual random testing is conducted at first and the successive one is manual functional testing.

2.4.1. Random Testing (R.T.)

Firstly, participants were asked to create test cases on Crossword Sage and OmegaT with random testing strategy. It is intuitive to speculate that a novice user would interact with software interfaces more randomly. For the sake of bringing the random strategy into force, we required that all AUTs
should be entirely unfamiliar to these participants. That is, participants should be unacquainted with these applications before creating test cases. Moreover, it is specified in random testing requirement that GUI widgets of AUTs should be covered as many as possible.

Brief Testing Requirement to participants:

- You should keep your promise of obeying this testing requirement.
- Testing Strategy: Random testing
- A test case should be generated based on random strategy.
- Test suite should be constructed to cover widgets of AUTs as many as possible.
- Objective Apps: Crossword Sage, OmegaT

2.4.2. Functional Testing (F.T.)

Functional testing was conducted by the same participants worked in random testing a month later. A straightforward reason to keep the order of implementing these two testing strategies is that an experienced user is more familiar with functions of AUTs than a novice user. Thereby, test cases created by these participants with functional strategy reflect a sequence of business logic while test cases with random strategy distribute more uniformly in the input space.

Some teaching assistants (TAs), who have industrial experiences on software testing, defined functional points of each AUT before the creation of test cases. Here, one functional point is a basic task which is described in the functional requirements. Each functional point represents a part of AUTs’ functions, and will be covered by at least one test case. On the basis of previous random testing requirements, TAs designed definite and detailed functional testing requirements to guide all participants to divide all functions of each AUT into several function sets. Functions in the same function set may belong to the same functional module. For example, basic tasks ‘creating crossword puzzle’ and ‘editing crossword puzzle’ are marked as functional points of the AUT Crossword Sage. They should be categorized in the same function set because they are functions that are related to crossword manipulation. Exactly, functions of Crossword Sage and OmegaT were both divided into 5 function sets.

Brief Testing Requirement to participants:
• You should keep your promise of obeying this testing requirement.

• Testing Strategy: Functional testing

• A test case should be generated based on one of the predefined function sets of AUTs.

• For each functional point in the function sets, there is at least one test case which is created for testing it.

• Test cases should be named obeying the following rule: the first two letters of the title of a test case must be the corresponding set number; e.g. If you have created a test case for function set 01, then the test case should be named as 01***.

• Objective Apps: Crossword Sage (5 Function Sets); OmegaT (5 Function Sets)

TAs checked all test cases developed by participants, and wiped off the ones which could not be executed normally in QTP. Additionally, we made sure that, with serious inspection by TAs, the remained test cases were executable and all GUI widgets of the two AUTs were covered by their corresponding test pool. Table 1 shows the numbers of test cases created in random testing (R.T.) and functional testing (F.T.) on Crossword Sage and OmegaT.

2.5. Test Oracle

To evaluate the fault detection abilities of test cases in the test pool, it is indispensable to build test oracle for the AUTs. Test oracle information is used for indicating the execution results of test cases, i.e. whether a test case passes or fails on a specific version of software. We have introduced the constitution of known faults for each AUT in our experiment, thus the test oracle in our experiment should indicate whether a fault is detected by a specific test case from the test pools. For AUTs with known faults, test oracle information also shows which fault/faults are detected by a specific test case.

As mentioned above, we collected some known faults in Crossword Sage and OmegaT, 14 and 129, respectively. We firstly located every known fault of AUTs, and marked the code points/segments which may trigger the fault. And then we instrumented the AUTs to get coverage information of each test case.
case. In this way, we know whether a test case has covered the code points of AUTs that may trigger a specific fault. As introduced above, known faults consist of two types: reported faults and improperly handled exceptions. It is non-trivial to build test oracle in our study because we need to handle these two types in different approaches.

For exceptions, we instrument the AUTs so that well-formatted logs can be output to console when improperly handled exceptions are triggered during the runtime. We build a exception monitor to track and analyze the console logs when a test case is executed. Then the monitor can extract all improperly handled exceptions triggered by this test case. As a result, we build a mapping between test case in the test pool and the improperly handled exceptions, and this map constitutes part of our test oracle.

Building the oracle for reported faults is more difficult because it is hard to build automatic monitor to judge the correctness of test cases’ outputs when reported faults are covered. In general, syndromes of the faults are violations against the expectation. But defining the expectation and detecting violation for every reported fault in an automatic way are rather difficult. We bring manual inspection into test oracle building. If a test case covers the code segments of AUTs corresponding to a reported fault, we then mark that test case as “suspicious” for the fault. Finally, testers will manually check whether the suspicious test cases really detect specific faults. The mapping between incorrect test cases and the reported faults detected by them are also merged into the test oracle.

The number of faults which were detected on Crossword Sage and OmegaT are exhibited in Table 1. A preliminary result is that functional testing detects more faults than random testing. Besides, a special phenomenon which should be noticed is that random testing and functional testing have detected different faults on OmegaT. Taking it into consideration, a question, whether random testing and functional testing are complementary, arises. We elaborate it in the research question 3 (RQ3).

2.6. Sampling

Randomness may be introduced into research by various non-deterministic factors and then influences related research results. So in some research, algorithms or experiments were run in multiple times and various methods of statistical analysis were used for adequately evaluating the testing strategies and obtaining reliable research results.
A systematic review of some research work in software engineering, conducted by A. Arcuri and L. Briand, shows that randomness is not taken into consideration properly in most of the previous research literature [9]. They pointed out although the randomized algorithm should be repeated at least 30 times as a rule of thumb, setting a repetition times n = 1000 or more is more preferable to get reliable results.

Following their suggestion, we sample 1000 test suites, for each testing strategy and then obtain statistical analysis of these samples.

In this study, with the purpose of comparing testing strategies reasonably, different sampling strategies are used on constructing test suites. The first sampling strategy used in this study is random sampling. Test suites of random testing strategy are sampled randomly from the random testing pool.

In functional testing, each AUT has several function sets, and at least one test case is created to cover one certain function set. Hence, we applied the second sampling strategy, named as proportional sampling, to select test cases that are evenly distributed in all function sets of each AUT.

Alg. 1 illustrates brief sampling procedure.

3. Result Analysis

3.1. Research Question 1

In the practice, the goal of software testing is to assure the quality of the software and eliminate the faults. Generally, one testing strategy is regarded as effective if it can find relatively considerable number of faults with relatively low cost. The fault detection capability is used to assess the effectiveness of testing strategies in many studies.

- **RQ1**: Which one is more effective: random or functional, with regard to the number of detected faults?

Generally, there are two assessment metrics that are often used to measure the cost of GUI testing: (a) the number of test cases and (b) the length of test cases. In general, a GUI test case consists of sequences of GUI events, which are invoked successively when the test case is executed. The length of one test case here is defined as the number of events involved in this test case. For more detailed studies, RQ1 is considered as two sub research questions: RQ1-a and RQ1-b.
Input:
- The required sample size of test suites based on one testing strategy, $S:S$;
- The required number of test cases in one test suite, $S:TS$;
- The test pool which test cases will be sampled from, $T:P$;
- The testing strategy, $T:Stgy$;

Output:
- Sets of sampled test suites, $SAMPLED$;

1: Create a new set $SAMPLED$ to store constructed test suites;
2: while $SizeOf(SAMPLED) < S:S$ do
3: Set $T:P'$ as a duplicate sample of $T:P$;
4: Create a new test suite $T:S$ for this sampling;
5: while $SizeOf(T:S) < S:TS$ do
6: if $T:Stgy$ is random testing then
7: Sample a test case, $tc$, from $T:P'$ by random sampling strategy, i.e. $tc \in T:P'$;
8: else if $T:Stgy$ is functional testing then
9: Sample a test case, $tc$, from $T:P'$ by proportional sampling strategy, i.e. $tc \in T:P'$;
10: end if
11: Put $tc$ into current test suite $T:S$;
12: Delete $tc$ from $T:P'$;
13: end while
14: Put $T:S$ into $SAMPLED$;
15: end while

Algorithm 1: SAMPLING

- **RQ1-a**: Which testing strategy can detect more faults, with the same number of test cases?

In this experiment, we collect data according to sampling strategies mentioned previously with the sample size 1,000. In *Crossword Sage* and *OmegaT*, we set the sizes of test suites as 300 for each sampling. For each sampling, we added 5 test cases at a step, and then we checked how many new faults could be detected as the size of a test suite grows. The results are demonstrated in different figures.

Fig. 3 shows a legible comparison between functional testing and random testing. It is not difficult to find out that functional testing can detect more faults than random testing finally. The efficiency of detecting faults in ran-
random testing rockets up in the early stage. But the increase is slow afterwards. For *Crossword Sage*, random testing can detect more faults than functional testing at the beginning. Subsequently, the number of faults detected in random testing remains unchanged in a long period. Meanwhile, the quantity of detected faults in functional testing keeps a comparative steady rate of increase. It is obvious that functional testing is more effective than random testing with the same quantity of test cases on *OmegaT* as showed in Fig. 3 (b).

![Comparison Graph](image)

Figure 3: Comparison on #TestCase, (a) Left: *Crossword Sage*, (b) Right: *OmegaT*

The results on *Crossword Sage* and *OmegaT* seem to be quite different. However, taking the sizes of AUTs into account, it is not hard to interpret what causes this result. Empirical researchers who want to settle questions about random testing [9] [25] have drawn a conclusion that random testing is more cost-effective. In this experiment, this conclusion holds only when the sizes of AUTs are within an uncertain range. To be specific, random testing, though used to be treated as a testing strategy of low effectiveness, [?] works excellently in small applications. While functional testing keeps the effectiveness even in large AUTs. In contrast to functional testing, the advantage of random testing is less obvious in large applications whose sizes are beyond an uncertain range.

For RQ1-a, with the same quantity of test cases, both random testing and functional testing perform well. With the increase of test cases, whereas, functional testing reaches its peaks (i.e., detect all faults that can be detected) earlier and works better in the long run.
A new question is proposed after getting the result of RQ1-a. The length of each test case in test pools is different. Admittedly, long test cases are more expensive than short ones when they are executed on AUTs. Hence, it seems much fairer to consider the length of test cases as the cost metric. The second sub research question RQ1-b is raised.

- **RQ1-b**: Which testing strategy can detect more faults, with the same length?

We reuse the previous experimental data (RQ1-a) as materials to study the RQ1-b. We calculate the lengths of test cases sampled from the previous experiment (RQ1-a). The length is set as an observable variable in this experiment.

Fig. 4 shows the experimental results. By assessing the quantity of detected faults with the same length of test cases, we learned that functional testing demonstrates a greater superiority in comparison with the first experiment.

![Figure 4: Comparison on Length, (a) Left: Crossword Sage, (b) Right: OmegaT](image)

For RQ1-b, comparing these two testing strategies under a fairer assessment than RQ1-a, functional testing shows a much better performance in detecting faults than random testing.

RQ1-a and RQ1-b were conducted with different assessment criteria, and similar results were shown. The experimental results illustrate that random testing is more cost-effective on small applications than functional testing, especially the initial stage of testing. Differing from results on small applications, random testing performs worse than functional testing on large
applications. One significant weakness of random testing is that it can not
detect all faults of AUTs.

3.2. Research Question 2

As mentioned in Section 2, faults which were detected in random test-
ing and functional testing are different in OmegaT after all test cases were
executed. If this phenomenon is not occasional, a hybrid or combination
of these two strategies is likely more effective than any single one. This
motivates us to study the complementarity of these two testing strategies.
Theoretically, the fault detection capability of two different testing strategies
is complementary if they tend to detect different faults instead of the same
faults. If the amount of total test cases is large enough, two test strategies
can be considered as complementary on detecting a definite fault when the
fault is detected by testing strategy A but hardly detected by testing strategy
B. Casting doubt on the complementary of random testing and functional
testing, we propose the following research question.

• RQ2: Are functional testing and random testing complementary to
each other?

We investigated all faults detected in functional testing or random testing
on AUTs. For each known fault, the numbers of test cases that have detected
it out of all test cases in functional testing or random testing are counted
precisely. Statistical results are demonstrated in Fig. 5. The FaultDetec-
tionTestCases(%) denotes the percentage of test cases which have detected
a specific fault in each testing strategy. The FaultID denotes the identifier
of each detected fault. Especially, in OmegaT, there are 23 known faults
which were detected among all 129 faults. We re-labeled the identifiers of
detected faults in practice for the sake of illustrating the complementarity
more clearly and comprehensibly.

It is clearly shown in the Fig. 5 that for a specific fault, the percentages
of test cases which detect the fault(FaultDetectionTestCases(%)) of random
testing and functional testing are different. The differences of fault detection
percentages denote different fault detection capabilities of the two strategies
for specific faults. By comparing fault detection percentages of random test-
ing and functional testing based on large amount of total test cases, there
does exist complementary between these two testing strategies on detecting
some faults, e.g. F5, F11 in Crossword Sage and F5, F7, F13, F17 in OmegaT
and so on.
Then we study these detected faults carefully. Some of these detected faults are unchecked exceptions and others are checked exceptions. The statistical data on these exceptions is shown in Table 2.

Considering the inherent differences between unchecked and checked exceptions, we also want to find out whether random testing or functional testing is better at detecting different types of exceptions and, further more, different types of faults. After checking all detected faults one by one, we found that although random testing and functional testing are complementary on detecting some faults, neither of them are found to be obviously more effective in detecting any certain type of faults.

![Figure 5: Complementarity](image)

**Table 2: Detected Faults**

<table>
<thead>
<tr>
<th>Application</th>
<th>D.UncExc</th>
<th>D.CExc</th>
</tr>
</thead>
<tbody>
<tr>
<td>Crossword Sage</td>
<td>F1, F2, F3, F4, F5, F12, F13, F14</td>
<td>F2, F3, F4, F10, F15, F16, F17, F21</td>
</tr>
<tr>
<td>OmegaT</td>
<td></td>
<td>F1, F5, F6, F7, F8, F9, F11, F12, F14, F18, F19, F20, F22, F23</td>
</tr>
</tbody>
</table>

* D.UncExc denotes Detected Unchecked Exceptions. D.CExc denotes Detected Checked Exceptions.

For RQ2, although functional testing and random testing are not complementary on detecting all faults in our study, the complementarity of these two testing strategies is indeed shown.

### 3.3. Research Question 3

Test cases in a test pool vary in length. Intuitively, long test cases can detect more faults as a consequence of the fact that they include more oper-
Aiming at verifying the intuition, we propose research question 3.

- **RQ3**: Is test case length a key factor of fault detection capability?

![Figure 6: Short vs. Long on Crossword Sage](image)

In this experiment, we separated each test pool into five parts according to the length of each test case. Firstly, all test cases were ordered from the shortest one to the longest one. Sequentially, we extracted test cases, which were ordered in the first 1/5 part of each test pool, to form the Part1 here. Similarly, the other four parts were selected orderly. Hence, for each original test pool, we obtain five parts regarded as new test pools on which sampling will be applied separately.

We have described how to sample test cases from test pools in RQ1-a. Similar to sampling strategies applied earlier, random sampling and proportional sampling are also used to sample test cases from each new test pool. Here we set the sampling size as 1000 and test suite size as 250.

Fig. 6 and 7 illustrate the experimental results. In order to make the five curves on each AUT distinctive, two curves are depicted below the x-axis,
and we use overstriking curves to denote the upward trend of Part1 and Part5.

![Graph](image)

**Figure 7: Short vs. Long on OmegaT**

Specially, differing from other applications, some functions of GUI applications must be carried out following a specific operation sequence (sequence of GUI events). It is hence that some faults, in GUI applications, could only be exposed on condition that testers or users execute a special operation sequence. So, for a specific fault which could not be detected unless a specific long operation sequence is applied, short test cases may not find it out as their operations are insufficient to complete the specific operation sequence. Comparing with the relation between random and functional testing strategies discussed in RQ1, the case here is quite similar: short test cases might lack the capability of finding all faults in AUTs, and long test cases can do it but with considerable cost. Apparently, test cases in Part1 perform better than those in other parts in Crossword Sage. However, compared with small applications, the operation sequence is much longer in large applications. That is why the effectiveness of Past1 and Part2 is exactly similar in OmegaT and the benefit of short test cases becomes less distinct with the
increase of AUTs’ sizes.

For RQ3, we designed a comparison on the effectiveness of test cases with various lengths for investigating whether the length of test case plays a key role in fault detection capability. The answer for this research question is positive, indeed. In one test pool, short test cases are more cost-effective than long test cases, especially in small applications.

3.4. Statistical Test

We apply the t-test in this study to evaluate the difference between two samples of independent observations. The significant level $\alpha = 0.05$ for all rejections of hypotheses is set in these experiments because 0.05 is widely accepted in many subjects and applications. We define the null hypothesis ($H_0$) briefly here:

- $H_0 : A \leq B$. The fault detection capability of testing strategy A is not better than B.
- $H_1 : A > B$. The fault detection capability of testing strategy A is better than B.

Notice that the hypotheses defined above stand for a family of hypotheses. There is a separate hypothesis set (the set of $H_0$ and $H_1$) for each testing strategy on each experiment conducted on AUTs. We collect all sampling data for investigating RQ1 and RQ2 and calculate the area surrounded by the effectiveness curve and the x-axis for each test suite. Each area is collected as sampling object for the t-test.

For a practical application, we report other important testing values besides the p-value in Table 3 for further analysis and helping readers to make an alternative choice from the two compared strategies.

<table>
<thead>
<tr>
<th>RQ</th>
<th>Strategy A</th>
<th>Strategy B</th>
<th>Application</th>
<th>t statistic</th>
<th>p-value</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>RQ1-a</td>
<td>R.T.</td>
<td>F.T.</td>
<td>Crossword Sage</td>
<td>10.66</td>
<td>1.6E-25</td>
<td>Reject</td>
</tr>
<tr>
<td>RQ1-a</td>
<td>F.T.</td>
<td>R.T.</td>
<td>OmegaT</td>
<td>59.16</td>
<td>0</td>
<td>Reject</td>
</tr>
<tr>
<td>RQ1-b</td>
<td>R.T.</td>
<td>F.T.</td>
<td>Crossword Sage</td>
<td>1.63</td>
<td>0.051</td>
<td>NOT Reject</td>
</tr>
<tr>
<td>RQ1-b</td>
<td>F.T.</td>
<td>R.T.</td>
<td>OmegaT</td>
<td>83.32</td>
<td>0</td>
<td>Reject</td>
</tr>
<tr>
<td>RQ 3</td>
<td>S. in F.T.</td>
<td>L. in F.T.</td>
<td>Crossword Sage</td>
<td>98.24</td>
<td>0</td>
<td>Reject</td>
</tr>
<tr>
<td>RQ 3</td>
<td>S. in R.T.</td>
<td>L. in R.T.</td>
<td>OmegaT</td>
<td>114.15</td>
<td>0</td>
<td>Reject</td>
</tr>
<tr>
<td>RQ 3</td>
<td>S. in F.T.</td>
<td>L. in F.T.</td>
<td>Crossword Sage</td>
<td>72.68</td>
<td>0</td>
<td>Reject</td>
</tr>
<tr>
<td>RQ 3</td>
<td>S. in R.T.</td>
<td>L. in R.T.</td>
<td>OmegaT</td>
<td>77.14</td>
<td>0</td>
<td>Reject</td>
</tr>
</tbody>
</table>

S. denotes the Shortest test cases, namely Part1. L. denotes the Longest test cases, namely Part5.
Obviously, all null hypotheses ($H_0$) are rejected because the $p$-values are less than 0.05 apart from RQ1-b on Crossword Sage ($p$-value=0.051). Through investigating RQ1-a on Crossword Sage, the hypothesis that functional testing is better than or equivalent to random testing is rejected, and thus we conclude that random testing performs better in this experiment. However, in the other parts of RQ1, or RQ1-b, the same null hypothesis is accepted. The main reason is functional testing overshadows random testing although the latter performs better in the early period. From these statistical results, we can conclude that random testing is a good testing strategy on small applications. Advantages of random testing can be recognized when thinking out of the box. Randomness is a fundamentally important character of random testing. Randomness may mean “without logic” or “out of order”, but this character owns strong similarities with some behaviors of users, especially unprofessional users. In addition, randomness may help reveal unexpected problems that are very difficult to find out in regular ways. Our empirical studies show that random testing performs well on small applications. The shortcoming of random testing is its lack of capability of testing large applications with complex business logic. In such conditions, functional testing, which is more businesslike, can help detect more faults.

In all experiments for RQ3, the results of statistical test are rejected. We can conclude that functional testing and random testing with the shortest test cases are more effective than their counterpart strategies. The length of test cases can affect their fault detection capability. Meanwhile, it is apparent that shorter test cases are more cost-effective than the longer counterparts.

4. Discussion

4.1. Practical Guidelines

In this study, experiments on comparing different software testing strategies are conducted and related experimental results are analyzed statistically. Based on previous experiments and analysis, we draw some guidelines for reference. (1) If testing is bounded by limited time and is conducted on small applications, random testing is a good choice. (2) If thorough testing is required on software and there are relatively sufficient testing resources, functional testing is a much better choice compared with random testing.

Although short test cases could not detect all kinds of faults in AUTs, they can detect most of faults with a handful of operational events. So if testers want to find out as many faults as possible with limited resource,
preferentially, short test cases are suggested to be applied to testing. Additionally, a high cost-effective testing strategy is popular and valuable in practice. Hence, we suggested that creating short test case is worthy. Specially, testing large applications without time constraints, it is better to create test cases with medium length for testing AUTs more sufficiently and detecting more faults. Whatever, developing test cases with too long sequence of GUI events is not recommended on account of its low cost-effectiveness.

4.2. Threats to Validity

All test cases used in this study are conducted by 234 undergraduate students. The quality of test cases may be lower than the industrial ones. In order to reduce this threat [27], TAs inspected and checked all these test cases, and then only qualified ones are adopted in our experiment. Furthermore, a series of preparing work were done to conduct a fair comparison.

Only two GUI applications were used in our study. The two GUI applications are also used in the previous studies [5][26]. Additionally, preparing lots of test cases takes a large amount of time. Time is insufficient to implement our experiment on more applications. We do not choose the latest version of AUT because a sufficient number of faults are crucial for our evaluation and the earlier versions contain more natural faults.

In order to compare the effectiveness of random testing and functional testing, we identified 14 original faults in Crossword Sage, and 129 exceptions handled improperly in OmegaT. There is a threat to validity here as exceptions are not a conventional type of faults of software. However, exceptions handled improperly share similar features with the real faults. That is, both these exceptions and real faults may be exposed by unexpected GUI operations and, once revealed, they may result enormous threats to applications. Besides, compared to seeded faults, employing these exceptions in study is a better choice because that fault seeding may increase the threats to validity of experiments for two reasons [28]. (1) Seeded faults may not represent the real faults. (2) Some researchers are used to seed single fault to build a single buggy version of AUT, but this does not represent the reality: there are usually multiple faults in a specific version of software.

In other works [21][22][23][29], exceptions were used as known faults to assess effectiveness of testing strategies as well. We believe that exceptions are good representatives of faults in GUI testing. We concentrated on checking whether test cases have detected known faults in AUTs. Theoretically,
test cases may detect unknown faults, which are not identified in studies, besides those known faults. Unknown faults which are triggered but not been analyzed in this study threaten the validity of our experimental results. However, we have monitored the runtime information of AUTs through recording console logs. And since no unknown exceptions are triggered by test cases, the threat to validity is reduced.

5. Related Work

Random testing is widely used in industry and studied by researchers. Richard [32] did research on random testing and discussed the importance of random testing. When investigating whether random testing is effective and useful, comparison between random testing and other testing strategies such as functional testing attracts the attention of researchers. Duran et al. [25] presented an empirical study on random testing. They discovered that random testing is more cost-effective than partition testing in terms of cost-per-fault-found. Arcuri et al. [9] surveyed and analyzed the properties of random testing and found that random testing performed better than lots of partition testing techniques, functional testing included, on small software. However, the limitation of the effectiveness of random testing on large applications was not discussed in their studies. In our study, comparison on random testing and functional testing was conducted on both small and large applications. Results derived from RQ1-a and RQ1-b show that the performance of random testing may disappoint people who believe it is more effective although random testing performs better on small applications.

As existing software testing strategies are gradually advanced, many assessment criteria to evaluate the effectiveness of testing strategies have been proposed and used in certain areas of software testing. Among them, the fault detection capability is reliable and widely used in comparing the effectiveness of different testing strategies. Basili et al. [26] have used the assessment criterion to compare such testing strategies as code reading, functional testing, and the statement coverage testing. Frankl et al. [33][34] also used the fault detection capability to measure different testing techniques. At present, highlighting effectiveness is regarded as the most general way in experimental work of evaluating testing strategies, acknowledged by Gupta et al. [30]. Concluding the studies above, we also choose the fault detection capability as the assessment criterion on comparison of random testing and functional testing on GUI applications.
6. Conclusions and Future Work

Software testing is a labor-intensive work. It is necessary to provide a practice guide to software testing assisted by human knowledge, such as random or functional. Doubts on the superiority of random testing and functional testing drive us to conduct the study of this paper. Aiming at collecting a vast mass of test cases, we convened 234 participants majoring in software engineering to create test cases on two GUI applications: *Crossword Sage* and *OmegaT*, according to random testing strategy and functional testing strategy. The experiment was designed in a systematic way to care non-deterministic factors, such that statistical test could be applied on the collected data.

This study contains four sub-experiments (RQ1-a, RQ1-b, RQ2 and RQ3) relating to comparison. RQ1-a and RQ1-b focus on investigating the effectiveness of random testing and functional testing, in regard to different cost metrics. The observation is that random testing could reveal faults more quickly and functional testing could reveal more faults in total. While in large applications, functional testing works better than random testing, with faster fault detection and a greater number of totally detected faults. In addition, the experiment RQ2 exhibits the complementarity between random testing and functional testing. In RQ3, we investigate if the length of test case may play a key role in detecting faults. Experimental comparisons illustrate that test cases’ length does affect their fault detection capability. If testing resource is limited, attention should be paid more on short test cases rather than longer test cases for that short test cases are more cost-effective than the longer ones, especially on small applications. We evaluate each comparison between two test approaches in the *t-test*. The differences of the above comparisons are significant in statistics.

Based on this study, some work is worthy being conducted in the future. In this study, experiments are conducted for comparing the effectiveness of random testing and functional testing on GUI applications. With limited resources, we just implement experiments on 2 GUI applications. If provided with sufficient time, we will conduct these experiments on more GUI applications to collect more experimental data. Testers regard attributes of testing strategies as keys in adopting one specific testing strategy into practice. In this study, we take the attributes of AUTs into consideration. We carry out random testing and functional testing on AUTs with different size. We concluded that the size of AUTs should be taken as a factor into consideration.
when testers are choosing testing strategies. However, there still are some other attributes, such as language and utility, which have not been analyzed in this study. We will investigate these valuable attributes of AUTs in our future work.

Random testing and functional testing have their different advantages. The complementarity of random testing and functional testing is discussed in this study and these two testing strategies show a complementarity on fault detection. In the future, we will focus on how can we combine functional testing and random testing to conduct a more cost-effective testing. We will also devote ourselves to figure out whether the combination of these two testing strategies can detect some specific classes of faults. Besides, investigating the strategy of combining long test cases and short test cases to improve the effectiveness of software testing is also an interesting research subject.

Following the guideline of A. Arcuri and L. Briand[9], we sampled 1000 test suites from each test pool and make statistical analysis on the fault detection capability of these test suites in order to get more reliable results on comparing different testing strategies. For further study, we will discuss whether it is necessary to sample more test suites and whether it can make experimental results more reliable.

For QTP, there is usually a pause time after executing every operation to make sure the normality of the test case replaying. In that condition, the time cost of executing a test case mainly depends on the length of the case. In some important testing scenarios, such as in regression test, the execution time is a factor of more importance. Besides, it is difficult to record the time cost of creating test cases precisely. So in this study, we simply assume the time costs of creating test cases are also in proportion to the lengths of them and the difference on average cost of developing test cases in random testing and functional testing was not taken into account. We will deepen our study in the future by taking all cost into consideration.

**Acknowledgments**

The work described in this article was partially supported by the National Natural Science Foundation of China (61003024, 61170067, 61373013).
References


