DDU

Master's Thesis

Aaron Ang

DDU

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Aaron Ang
born in Amstelveen, The Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

PARC, a Xerox company
3333 Coyote Hill Road
Palo Alto, CA 94304
www.parc.com

DDU

Author: Aaron Ang
Student id: 4139194
Email: a.w.z.ang@student.tudelft.nl

Abstract

ABSTRACT

Thesis Committee:

Chair:	Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor:	Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
Company supervisor:	Prof. Dr. R. Maranhao, University of Lisbon

Preface

This is where you thank people for helping you etc.

Aaron Ang
Delft, The Netherlands
September 18, 2017

Contents

List of Figures

Chapter 1

Introduction

Software systems are complex and error-prone, likely to expose failures to the end user. When a failure occurs, the developer has to debug the system to eliminate the failure. This debugging process can be described in three phases [8]. In the first phase, the developer has to pinpoint the fault, also known as the root cause, in code that causes the failure. In the second phase, the developer has to develop an understanding of the root cause and its context. Finally, in the third phase, the developer has to implement a patch that corrects the behavior of the system. This process is time-consuming and can account for 30% to 90% of the software development cycle [12, 3, 4].

Traditionally, developers use four different approaches to debug a software system, namely program logging, assertions, breakpoints and profiling [15]. Program logging is the act of inserting *print* statements in the code to observe program state information during execution. Assertions are constraints that can be added to a program that have to evaluate to true during execution time. Breakpoints allow the developer to pause the software system during execution, and observe and modify variable values. Profiling is used to perform runtime analysis and collect metrics on, for example, execution speed and memory usage. These techniques provide an intuitive approach to localize the root cause of a failure, but, as one might expect, are less effective in the massive size and scale of software systems today.

Therefore, in the last decades a lot of research has been performed on improving and developing *advanced* fault localization techniques [15] such that they are applicable to the software systems of today. Specifically, a prominent fault localization technique is spectrum-based fault localization (SBFL). SBFL techniques pinpoint faults in code based on execution information of a program, also known as a program spectrum [11]. It does this by outputting a list of suspicious components, for example statements or methods, ranked by their suspiciousness. Intuitively, if a statement is executed primarily during failed executions, then this statement might be assigned a higher suspiciousness score. Similarly, if a statement is executed primarily during successful executions, then this statement might be assigned a lower suspiciousness score.

While SBFL techniques are promising for debugging purposes, these techniques are dependent on the quality of a test suite. Currently, test suites are optimized with respect to adequacy measurements that focus on error detection, e.g. branch coverage, line coverage. However, Perez *et al.* [10] show evidence that optimizing a test suite with respect to DDU — a metric to quantify the test suite's diagnosability — improves the diagnostic performance of SBFL by 34% compared to a test suite optimized with respect to branch coverage. The goal of DDU is to capture diagnosability and to serve as a complementary metric to code coverage for developers to use to improve the test suite's diagnosability.

1.1 Problem Definition

Currently, when the DDU is computed for a given test suite, its value is in the domain $[0,1]$, where 0 suggests that the test suite's diagnosability is low, and 1 suggests that the test suite's diagnosability is high. The problem with this value is that the developer does not know how to extend or update the test suite given a DDU value. For example, when the test suite's DDU is equal to 0.1, the developer does not know how to write tests that improve the DDU. In other words, time spent on software debugging cannot be reduced using DDU because its practical implications are unclear to the developer.

1.2 Goal

Although DDU is currently not usable in practice, Perez *et al.* [10] have shown that optimizing a test suite with respect to DDU can yield a 34% gain in diagnostic performance using SBFL. In addition, having a test suite with a high diagnosability could possibly reduce the time spent debugging because the fault is easier to find manually. Therefore, the goal of this thesis is to find ways to make DDU usable in practice. In other words, we explore possibilities to convey DDU to the developer such that the developer knows what kind of tests to write to improve the system's diagnosability. To be able to

1.3 Structure of Report

The structure of this report is as follows. TODO: Update once all chapters are done.

Chapter 2

Background

In this chapter, we discuss topics that are relevant to understanding the following chapters. First, we discuss spectrum-based reasoning, which is used in the experiments to compute the diagnostic performance. Second, we discuss the metric used to evaluate the diagnostic performance of spectrum-based fault localization (SBFL) techniques. Then, we explain the definition of diagnosability and diagnosability assessment metrics.

2.1 Spectrum-Based Reasoning (SBR)

Spectrum-based reasoning is a spectrum-based fault localization technique that leverages a Bayesian reasoning framework to diagnose fault candidates that could potentially be the root cause for a given software failure [2]. In SBR, we define a finite set $\mathcal{C} = \langle c_1, c_2, \ldots, c_M \rangle$ of M system components, and the finite set $\mathcal{T} = \langle t_1, t_2, \ldots, t_N \rangle$ of N system transactions, such as test executions. The outcomes of all system transactions are defined as an error vector $e = \langle e_1, e_2, \ldots, e_N \rangle$, where $e_i = 1$ indicates that transaction t_i has failed and $e_i = 0$ otherwise. To keep track of which system components were executed during which system transactions, we construct a $N \times M$ activity matrix $\mathcal{A}$, where $\mathcal{A}_{ij} = 1$ indicates that component c_j was hit during transaction t_i. The pair $(\mathcal{A}, e)$ is also known as a program spectrum, which was first coined by Reps *et al.* [11].

SBR distinguishes itself from SBFL techniques by leveraging a reasoning framework. More specifically, the diagnostic report is generated by reasoning about the program spectrum instead of using a so-called similarity coefficient. The two main phases of SBR are candidate generation and candidate ranking:

1. In the candidate generation phase, a set $\mathcal{D} = \langle d_1, d_2, \ldots, d_k \rangle$ is constructed using a minimal hitting set (MHS) algorithm to cover all failing transactions, where each candidate d_i is a subset of $\mathcal{C}$. An MHS algorithm is used to prevent generation of of a possibly exponential number of diagnostic candidates [1].

2. In the candidate ranking phase, the fault probability for each candidate d_i is computed using the Naive Bayes rule [1]:

$$P(d_i|(\mathcal{A},e)) = P(d_i) \cdot \prod_{j \in 1..N} \frac{P((\mathcal{A}_j, e_j)|d_i)}{P(\mathcal{A}_j)} \tag{2.1}$$

$P(d_i)$ is the prior probability, i.e. the probability that d_i is faulty without any evidence. $P(\mathcal{A}_j)$ is a normalizing term that is identical for all candidates. $P((\mathcal{A}_j, e_j)|d_i)$ changes the prior probability with every new observation from the program spectrum. This term can be computed using maximum likelihood estimation.

The final output of SBR is a diagnostic report, which is a list of components, e.g. statements or methods, ranked by their suspiciousness. In the experiments of this thesis, we will make use of *Barinel* [1], which implements the spectrum-based reasoning technique to perform software fault localization.

2.2 Evaluation of Diagnosis

Presently, cost of diagnosis C_d and wasted effort [1, 16, 5, 13, 10] are the most prevalent evaluation metrics for SFL techniques. In essence, C_d computes the number of components that have to be inspected before the actual fault is investigated in the diagnostic report. When $C_d = 0$, it indicates that the actual fault is ranked first in the diagnostic report and, therefore, no effort is wasted investigating diagnosed components that are non-faulty.

Wasted effort (or effort) is the cost of diagnosis normalized by the number of diagnosed components in the diagnostic report. Both evaluation metrics assume *perfect bug understanding*, which has been pointed out by Parnin and Orso [9] as a non-realistic assumption. However, cost of diagnosis and effort serve as an objective evaluation metric that can be used for comparison and therefore will also be used in this study.

2.3 Diagnosability

Diagnosability is the property of faults to be easily and precisely located [14]. In other words, given that a fault exists in a software system, if the test suite's diagnosability is high and we would perform SFL using an automated debugging technique, then the faulty component would be ranked high in the diagnostic report, i.e. the wasted effort would be low. On the contrary, if the test suite's diagnosability is low and we would perform SFL using an automated debugging technique, then the faulty component would be ranked low in the diagnostic report, i.e. the wasted effort would be high.

2.3.1 Diagnosability Metric: Entropy

The optimal diagnosability is achieved by having an exhaustive test suite that would exercise any combination of software components. This way, any fault, whether it involves a single component or multiple components, can be diagnosed using an automated debugging technique with 100% accuracy. Perez *et al.* [10] find that Shannon's entropy accurately captures the test suite's exhaustiveness:

$$H(X) = -\sum_i P(x_i) \cdot log_2(P(x_i)) \tag{2.2}$$

However, an optimal entropy would require $2^M - 1$ tests, which from the practical perspective is impossible to achieve. First, not all activity patterns can be generated from tests due to ambiguity groups and software topology. For example, a basic block consisting of several statements; these statements will always be activated together. Second, systems of today can consist of millions of lines of code and would therefore require a non-realistic effort to generate the tests.

2.3.2 Diagnosability Metric: DDU

To elevate the problem with entropy, Perez *et al.* [10] propose a new diagnosability metric: DDU. DDU combines three diagnosability metrics that capture characteristics of the activity matrix, namely normalized density, diversity, and uniqueness.

Density

Prior work [6] has used density to assess the diagnosability of the activity matrix:

$$\rho = \frac{\sum_{i,j} \mathcal{A}_{ij}}{N \cdot M} \tag{2.3}$$

Gonzàles-Sanchez *et al.* [6] show by induction that the optimal density for SBR is obtained when $\rho = 0.5$. For DDU, Perez *et al.* [10] propose a normalized density ρ' where its optimal value is 1.0 instead of 0.5:

$$\rho' = 1 - |1 - 2\rho| \tag{2.4}$$

Note that an optimal value for normalized density can be obtained without improving the diagnosability, see Figure 2.1(a). For this reason, Perez *et al.* [10] propose two enhancements: diversity and uniqueness.

Diversity

The first enhancement to the normalized density is diversity, which ensures test diversity. Test diversity captures the diversity among test activity patterns. The test diversity is low when the test suite consists of many tests that have identical activity patterns. Conversely, the test diversity is high when the test suite consists of many

	c_1	c_2	c_3	c_4
t_1	1	1	0	0
t_2	1	1	0	0
t_3	1	1	0	0
t_4	1	1	0	0

(a) No test diversity. $\rho' = 1.0$, $\mathcal{G} = 0.0$

	c_1	c_2	c_3	c_4
t_1	1	1	0	0
t_2	0	0	1	1
t_3	1	1	1	0
t_4	0	0	0	1

(b) Test diversity. $\rho' = 1.0$, $\mathcal{G} = 1.0$.

	c_1	c_2	c_3	c_4
t_1	1	1	0	0
t_2	0	0	1	1
t_3	1	1	1	0
t_4	0	0	0	1

(c) Component ambiguity. $\rho' = 1.0$, $\mathcal{G} = 1.0$, $\mathcal{U} = 0.75$

	c_1	c_2	c_3	c_4
t_1	1	1	0	0
t_2	0	1	1	0
t_3	1	0	1	1
t_4	0	0	0	1

(d) No component ambiguity. $\rho' = 1.0$, $\mathcal{G} = 1.0$, $\mathcal{U} = 1.0$

Figure 2.1: The effect of diversity and uniqueness on diagnosability.

tests that have distinct activity patterns. From the diagnostic perspective, a high test diversity is desired such that the fault probabilities of more diagnostic candidates can be updated for a better fault diagnosis.

Perez *et al.* use the Gini-Simpson index $\mathcal{G}$ [7] to capture test diversity:

$$\mathcal{G} = 1 - \frac{\sum_{i \in 1..|G|} |g_i| \cdot (|g_i| - 1)}{N \cdot (N-1)} \tag{2.5}$$

where $G = \langle g_1, \ldots, g_k \rangle$ is the set of ambiguity groups, and N is the number of tests. As we can observe in Figure 2.1(b), when optimizing the activity matrix for test diversity, the shortcoming of normalized density alone, shown in Figure 2.1(a), is mitigated.

Uniqueness

The second enhancement to the normalized density is uniqueness, which controls for the number of ambiguity groups. An ambiguity group is a set of components that have identical activation patterns across the whole test suite, i.e. identical columns in the activity matrix. Component ambiguity is undesirable because it prevents SBR from updating the fault probabilities of the individual components in the ambiguity group, resulting in a less accurate diagnosis. If test suite's uniqueness is low, then many components in the activity matrix have identical activity patterns, i.e. components are involved in the same test cases. Conversely, if the test suite's uniqueness is high, then many components in the activity matrix have distinct activity patterns.

Given the set of component ambiguity groups $H = \langle h_1, \ldots, h_l \rangle$, then the test suite's uniqueness is computed as follows:

$$\mathcal{U} = \frac{|H|}{M} \tag{2.6}$$

We observe in Figure 2.1(c) that optimizing the test suite with respect to normalized density and diversity can still result in component ambiguity groups, namely $\langle c_1, c_2 \rangle$. When optimizing the test suite with respect to normalized density, diversity and uniqueness, we observe in Figure 2.1(d) that there are no identical test activity patterns and no ambiguity groups, which results in a better diagnosability.

Combined

Finally, we can combine normalized density, diversity, and uniqueness as follows because they all have the same domain with an optimal value of 1.0:

$$DDU = normalized\ density \cdot diversity \cdot uniqueness \tag{2.7}$$

If $DDU = 1$, then the test suite's diagnosability is high. Vice versa, if $DDU = 0$, then the test suite's diagnosability is low. Perez *et al.* [10] have shown in an experiment that optimizing a test suite with respect to DDU yields a 34% diagnostic performance compared to a test suite optimized for branch coverage.

Chapter 3

Research Questions

To explore possibilities to convey DDU to the developer with usability in mind, we define four research questions that are relevant to investigate. Note that the nature of this study will be exploratory because little research has been performed on software diagnosability. Moreover, DDU is a metric that has been proposed recently [10] and, therefore, to the best of our knowledge, no research has investigated this metric.

> **RQ1:** What kind of values do density, diversity, uniqueness, and DDU take on?

To make recommendations based on DDU, it is necessary to obtain a better understanding of DDU and its individual components: density, diversity, and uniqueness. Specifically, we are interested in what common values are for DDU and its individual components in software systems.

> **RQ2:** What is the relation between density, diversity, uniqueness, and DDU and diagnosability?

We would like to validate that DDU and diagnosability are positively correlated, i.e. the higher DDU, the better the diagnosability, and vice versa. This question is important to answer because DDU was proposed to quantify the diagnosability. Therefore, answering this question will function as a complementary study to Perez *et al.*'s work [10].

> **RQ3:** What is the relation between density, diversity, uniqueness, and DDU and test coverage?

The intention of test coverage is to optimize for error detection. Perez *et al.* propose DDU as a complementary metric to test coverage [10] because DDU is meant to capture the diagnosability and not error detection. However, if there is a strong correlation between DDU and test coverage, then DDU could possibly replace test coverage as a test adequacy metric, which is a use case that the authors have not thought of. Additionally, assuming that DDU is positively correlated with diagnosability and test coverage is representative for error detection, answering this question will give us a better understanding on the relation between diagnosability and error detection.

Chapter 4

DDU in Practice

NOTE: This chapter's content was written a while ago. I am now in the process of rewriting it, but it gives a general view of what is going to be in this chapter.

In this chapter, the goal is to obtain a better intuition on what common values are for density, diversity, uniqueness, and DDU by analyzing open source projects, hosted on GitHub. First, we take a look at the distributions of density, diversity, uniqueness, and DDU. Second, we give examples of components that have a low or high value for any of the metrics and analyze why these components have either a low or high value by investigating the component's test suite.

RQ1: What kind of values do density, diversity, uniqueness, and DDU take on?

4.1 Approach

To get a better understanding of how the values varies for normalized density, diversity, uniqueness, and DDU, we use `ddu-maven-plugin`[1], written by Perez, to instrument Java code and construct the activity matrix. Once we obtain the activity matrices, we analyze the data using multiple Python scripts[2]. With these two tools we collect data such as number of tests, number of components, density, normalized density, diversity, uniqueness, DDU, and the activity matrix.

Then, we analyze the collected data and show examples that illustrate how DDU and its individual terms vary as a consequence to particular kinds of tests. We are interested in what kinds of testing approaches result in a high or low DDU value.

Note that `ddu-maven-plugin` is able to instrument the code for three different granularity levels, namely statements, branches, and methods. By default `ddu-maven-plugin` uses the method level granularity. In this study, we make use of the same granularity used in the study performed by Perez *et al.* [10], namely branch granularity.

[1] `https://github.com/aperez/ddu-maven-plugin`
[2] `https://github.com/aaronang/ddu`

4.2 Selection

The selection of open source projects is done according to the following criteria.

- *The project must have an executable test suite.* To compute the DDU for a software project, we must construct an activity matrix, also known as program spectra. The program spectra is constructed by running the test suite and instrumenting the code such that we can keep track of what components are executed during a program execution.

- *The project must use Apache Maven, a software project management and comprehension tool.* The current tool that instruments the code to construct the activity matrix is implemented as a Maven plugin. Note that the current Maven plugin does not work for all Maven projects, and therefore only projects, that can be analyzed with this plugin, are used.

Based on these requirements, we choose the following open source projects.

- Commons CSV[3]: a library that provides a simple interface for reading and writing CSV files of various types.
- Commons Text[4]: a library focused on algorithms working on strings.
- Commons IO[5]: a library of utilities to assist with developing IO functionality.
- Guice[6]: a lightweight dependency injection framework for Java 6 and above.
- Jsoup[7]: an API for extracting and manipulating data, using the best of DOM, CSS, and jquery-like methods.

4.3 Normalized Density

In Figure 4.1, we show the distribution of normalized densities for all classes of the five open source projects mentioned before. The average equals to 0.5145. The peak for the interval [0, 0.1] is primarily caused by classes that are exercised by test cases that involve all components. 47 classes in the interval [0, 0.1] consist of only one method. A class with one method will always have a density of 1.0 and therefore a normalized density of 0.

The normalized density value is low when a class is tested by many tests that only cover a couple of methods. For example, `org.apache.commons.csv.CSVRecord` has 17 methods (excluding its constructors) and its spectra looks as follows. Note that not all transactions that hit `CSVRecord` methods are shown, but in the case of `CSVRecord`, all its transactions look similar; each transaction only hits a couple of components.

[3] `https://github.com/apache/commons-csv`
[4] `https://github.com/apache/commons-text`
[5] `https://github.com/apache/commons-io`
[6] `https://github.com/google/guice`
[7] `https://github.com/jhy/jsoup`

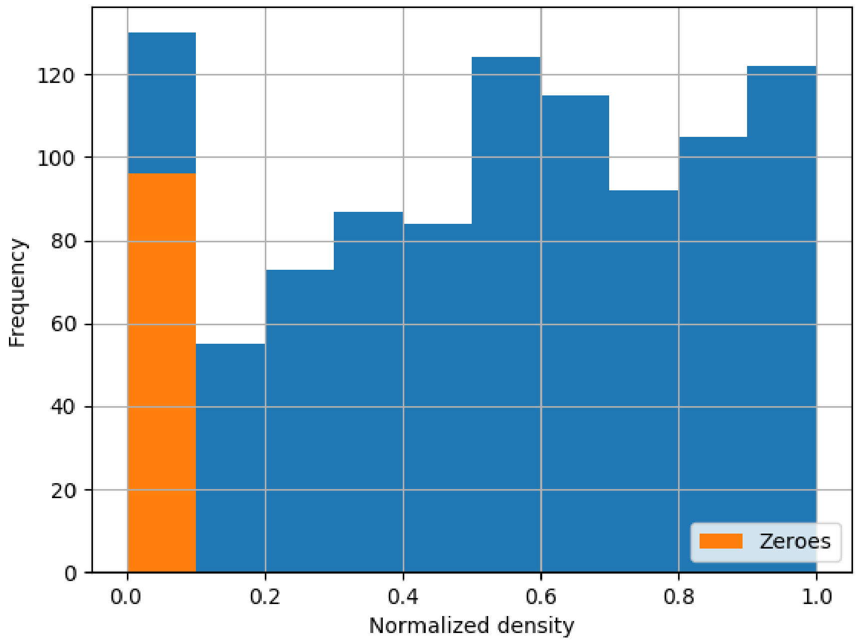


Figure 4.1: Normalized density distribution.

Table 4.1: Partial activity matrix of the `CSVRecord` class.

transaction	c_1								...								c_{17}
`CSVRecordTest#testGetStringNoHeader`	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
`CSVRecordTest#testGetUnmappedPositiveInt`	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
`CSVRecordTest#testGetUnmappedNegativeInt`	0	0	0	0	0	0	0	0	0	0	1	0	0	0	0	0	0
`CSVRecordTest#testGetUnmappedEnum`	1	0	0	0	0	0	0	0	0	0	0	0	0	1	0	0	0
`CSVRecordTest#testGetUnmappedName`	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

In the partial spectra of `CSVRecord` shown above, the columns are the methods of `CSVRecord`. So, for example, in the first row, we observe that the transaction `testGetStringNoHeader` only hits one method, indicated by a 1. The normalized density of `CSVRecord` is low because its test cases only cover a couple components and this causes the activity matrix to become sparse. Thus, the more components a class has, the greater the impact of a test, that hits a relative small number of components, has on the sparseness of the activity matrix.

In the partial spectra of `com.google.inject.internal.ConstructorInjectorStore`, a Guice class, shown above, we observe a high density; almost all methods of `ConstructorInjectorStore` are hit in every single test. Since the density is high: 0.8351, the normalized density is low: 0.3298.

So, ideally, to obtain a high value for the normalized density, we need a good balance between tests that cover many components and tests that cover a few. In

Table 4.2: Partial activity matrix of the `ConstructorInjectorStore` class.

transaction	c_1	...				c_6
`ImplicitBindingTest#testDefaultImplementation`	1	0	1	1	1	1
`ImplicitBindingTest#testProvidedByNonEmptyEnum`	1	0	1	1	1	1
`ImplicitBindingTest#testDefaultProvider`	1	0	1	1	1	1
`ImplicitBindingTest#testProvidedByEmptyEnum`	1	0	1	1	1	1
`ImplicitBindingTest#testImplicitJdkBindings`	1	0	1	1	1	1

Table 4.3: My caption

transaction	c_1	...		c_4
`StrLookupTest#testNoneLookup`	0	1	0	1
`StrLookupTest#testSystemPropertiesLookupReplacedProperties`	1	1	0	0
`StrLookupTest#testSystemPropertiesLookupUpdatedProperty`	1	1	0	0
`StrLookupTest#testMapLookup_nullMap`	0	1	1	0
`StrLookupTest#testMapLookup`	0	1	1	0

Commons Text, the `org.apache.commons.text.beta.StrLookup` class has a normalized density of 0.9512. In the partial spectra, shown below, we observe that each test case covers 50% of the components and therefore this particular class obtains a density close to 0.5 and a normalized density close 1.0.

In short, to obtain a good value for normalized density, we could suggest the developer to write tests that either cover many components, a few, or something in between based on the density. If the density is lower than '0.5', we could suggest the developer to write tests that cover many components. If the density is higher than '0.5', we could suggest the developer to write tests that cover a few components. Either way, suggesting the developer to write tests that cover a certain number of components is probably not practical.

4.4 Diversity

In Figure 4.2, the distribution of diversity of classes is shown. The average is 0.4813. The peak for the interval [0, 0.1] occurs for various reasons. The first reason is that there are classes with only one method and therefore every row is identical, resulting in a diversity of 0. The second reason is that for some classes there exist only one test case, and in the current Python script the diversity defaults to 0 when there is only one test case. The third reason is that some classes only have test cases that have identical activity patterns.

Intuitively, the diversity has a low value when the number of identical transactions, i.e. identical rows in the activity matrix, is high. Vice versa, the diversity is high when the number of identical transaction activity is low.

In the partial spectra of `org.jsoup.select.Evaluator` of Jsoup shown above, we observe that almost every transaction has an identical activity and therefore the diversity is low: 0.0901. Another reason for the low diversity of `org.jsoup.select.Evaluator` is that it is covered by 277 test cases, while there are only $2^4 = 16$ possible different

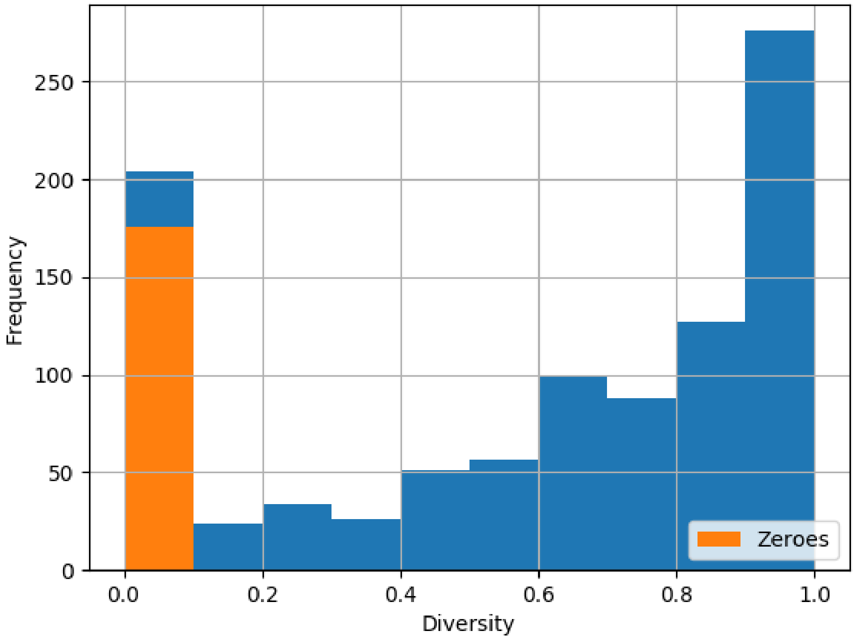


Figure 4.2: Diversity distribution

Table 4.4: My caption

transaction	c_1	…		c_4
`ElementTest#testCssPath`	1	0	1	0
`ElementTest#testSetHtmlTitle`	0	0	1	0
`ElementTest#moveByAppend`	0	0	1	0
`ElementTest#testClassUpdates`	0	0	1	0
`ElementTest#testAddNewElement`	0	0	1	0

test cases. After 16 unique test cases every additional test case will have a negative effect on the diversity because it will share similar activity with an existing test case.

In the activity matrix of `com.google.inject.spi.MembersInjectorLookup`, a Guice class, shown above, we observe that almost every transaction has a unique activity and therefore its diversity is high. Note that the diversity suffers when there are too many test cases, but does not suffer from a low number of test cases, i.e. a high diversity can be obtained with a couple unique system transactions.

An interesting case for diversity is parameterized testing, shown in the partial spectra above. The `LevenshteinDistance` spectra consists primarily of parameterized test and has a diversity of 0.3167. Although parameterized is a common practice to test different inputs for a unit, it has a negative effect on the diversity due to identical activity patterns.

Table 4.5: My caption

transaction	c_1				…					c_{10}
MembersInjectorTest#testMembersInjectorFromBinder	1	1	1	1	0	0	1	1	1	1
ElementApplyToTest#testGetMembersInjector	1	1	1	1	1	1	1	1	1	0
ElementApplyToTest#testElementInitialization	1	1	1	1	0	1	0	1	1	0
ElementsTest#testGetMembersInjector	1	1	1	0	1	0	1	1	1	0
ElementsTest#testElementInitialization	1	1	1	1	0	0	0	1	1	0
TypeListenerTest#testLookupsAtInjectorCreateTime	1	1	1	1	0	0	1	1	1	0
TypeListenerTest#testConstructedTypeListenerIs...	0	1	0	1	0	0	1	1	1	0
TypeListenerTest#testInjectMembersTypeListenerFails	0	1	0	0	0	0	1	1	1	0
NoopOverrideTest#testGetMembersInjector	1	1	1	1	1	1	1	1	1	0
NoopOverrideTest#testElementInitialization	1	1	1	1	0	1	0	1	1	0

Table 4.6: My caption

transaction	c_1		…		c_5
ParameterizedLevenshteinDistanceTest#test[0]	0	1	1	0	0
ParameterizedLevenshteinDistanceTest#test[1]	0	1	1	0	0
ParameterizedLevenshteinDistanceTest#test[2]	0	1	1	0	0
ParameterizedLevenshteinDistanceTest#test[3]	0	1	1	0	0
ParameterizedLevenshteinDistanceTest#test[4]	0	1	1	0	0
ParameterizedLevenshteinDistanceTest#test[5]	0	1	1	0	0
ParameterizedLevenshteinDistanceTest#test[6]	0	1	1	0	0
ParameterizedLevenshteinDistanceTest#test[7]	0	1	1	0	0
ParameterizedLevenshteinDistanceTest#test[8]	0	1	1	0	0

Table 4.7: My caption

transaction	c_1					…						c_{12}
CopyUtilsTest#testCopy_byteArrayToWriterWithEncoding	0	0	1	0	0	1	0	0	0	0	1	0
CopyUtilsTest#copy_stringToOutputStream	0	0	0	0	0	0	0	0	0	0	1	1
CopyUtilsTest#testCopy_readerToOutputStream	0	0	0	0	0	0	0	0	1	0	1	0
CopyUtilsTest#copy_readerToWriter	0	0	0	0	0	0	0	0	0	0	1	0
CopyUtilsTest#copy_byteArrayToWriter	0	1	0	0	0	0	0	0	0	1	1	0
CopyUtilsTest#copy_byteArrayToOutputStream	0	0	0	0	1	0	0	0	0	0	0	0
CopyUtilsTest#copy_inputStreamToWriterWithEncoding	0	0	1	0	0	0	0	0	0	0	1	0
CopyUtilsTest#copy_inputStreamToWriter	0	1	0	0	0	0	0	0	0	0	1	0
CopyUtilsTest#testCopy_inputStreamToOutputStream	0	0	0	0	0	0	1	0	0	0	0	0
CopyUtilsTest#copy_stringToWriter	1	0	0	0	0	0	0	0	0	0	0	0
IOUtilsTestCase#testCopy_String_Writer	1	0	0	0	0	0	0	0	0	0	0	0
IOUtilsTestCase#testCopy_ByteArray_Writer	0	1	0	0	0	0	0	0	0	1	1	0
IOUtilsTestCase#testCopy_ByteArray_OutputStream	0	0	0	0	1	0	0	0	0	0	0	0
IOUtilsTestCase#testStringToOutputStream	0	0	0	0	0	0	0	0	0	0	1	1

4.5 Uniqueness

In the figure below, the distribution of uniqueness of classes are shown. The average is 0.7696.

The peak for the interval [0.9, 1.0] is partially caused by classes that only have one component; activity matrices that consist of one component always have a uniqueness of 1.0. More specifically, there are 130 classes that have a uniqueness of 1.0 and 47 out of the 130 only have one component.

The uniqueness of an activity matrix is high when the number of unique columns

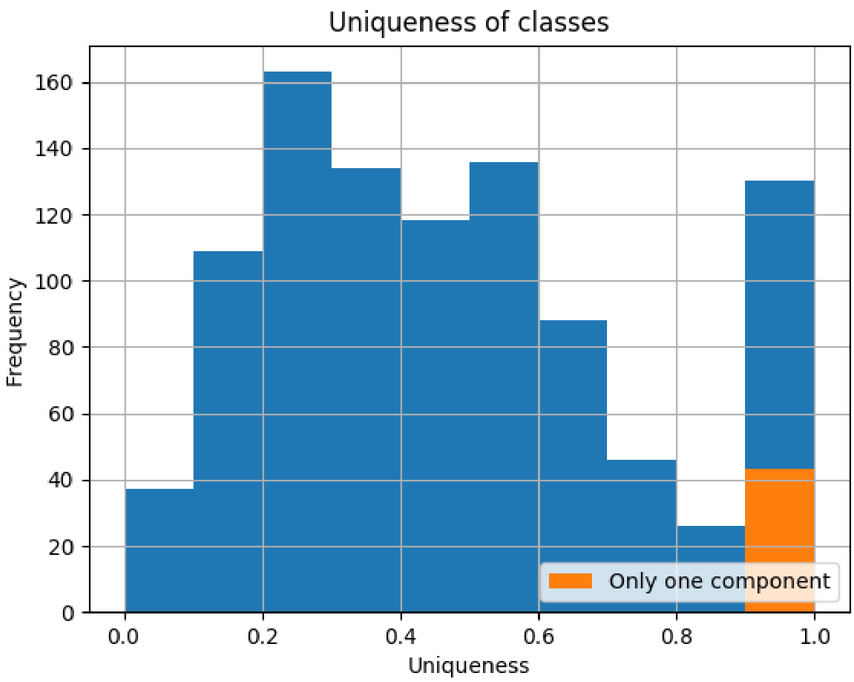


Figure 4.3: Distribution of uniqueness.

Table 4.8: My caption

transaction	c_1			...			c_7
`ReplacementsFinderTest#testReplacementsHandler[1]`	1	1	1	1	1	1	1
`StringsComparatorTest#testExecution`	1	1	1	1	1	1	1
`StringsComparatorTest#testLongestCommonSubsequence`	1	1	1	1	1	1	1
`StringsComparatorTest#testLength`	1	1	1	1	1	1	1

is high. For example, `org.apache.commons.io.CopyUtils` has a uniqueness of 0.9167, see spectra above. This class has a high uniqueness because its test suite mostly consists of test cases that only cover a couple components each.

In the spectra above of `org.apache.commons.text.beta.diff.StringsComparator`, we observe that every component is covered in every transaction and therefore the uniqueness is low: 0.1429. Although the matrix has no unique columns, its uniqueness is not 0. So, to improve the uniqueness of this matrix, we could write tests that cover a few components each, resulting in each column being unique.

4.6 DDU

In the figure below, the distribution of DDU of classes are shown. The average is 0.2264.

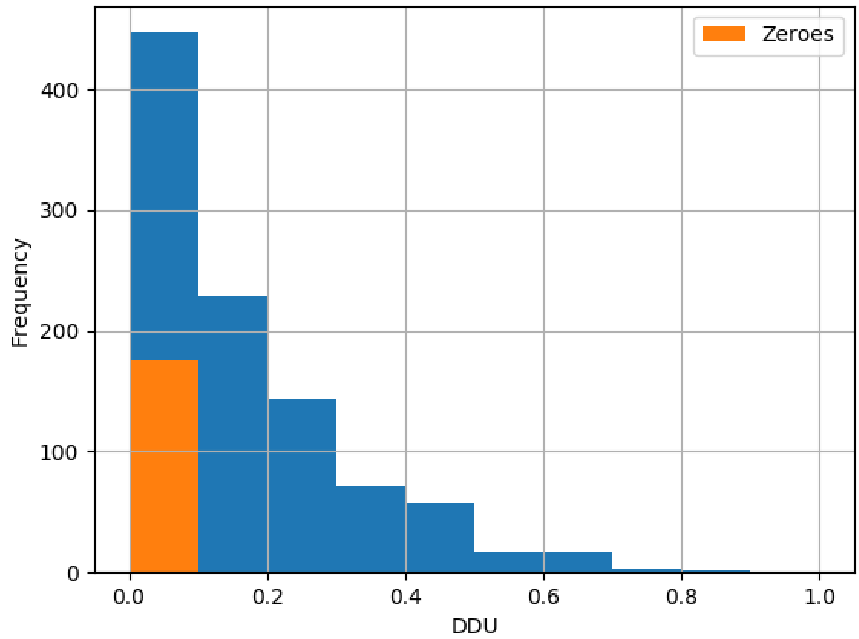


Figure 4.4: DDU distribution.

Table 4.9: My caption

transaction	c_1	c_2	c_3
`BrokenOutputStreamTest#testClose`	0	0	1
`BrokenOutputStreamTest#testFlush`	0	1	0
`BrokenOutputStreamTest#testWrite`	1	0	0
`TaggedOutputStreamTest#testBrokenStream`	1	1	1

Out of all the classes, there is only one class that has a DDU of 1.0, namely `org.apache.commons.io.output.BrokenOutputStream`, see spectra below.

Multiplying the individual terms might not be the right approach.

Based on the uniqueness observations, it seems that uniqueness at least requires the developer to write tests that cover a few components.

Chapter 5

DDU vs. Diagnosability

In prior work Perez *et al.* [10] show that optimizing test suite generation with respect to DDU results in better fault diagnosis. Therefore, in this chapter, we perform experiments to verify the correlation between DDU and diagnosability, that is, answering RQ2. We do this by discussing the experimental setup followed by the experimental results.

> **RQ2:** What is the relation between density, diversity, uniqueness, and DDU and diagnosability?

5.1 Experimental Setup

First, we have to define the subjects of interest that will be used during the experiments. For the experiments, we use the open source projects Commons Codec, Commons Compress, and Commons Math, which were also used in prior work by Alex *et al.* [10]. In addition, we include the open source projects Guice and Jsoup due to their popularity on GitHub; both roughly have 5000 stars.

To test the correlation between DDU and diagnosability, we generate 10 artificial multiple components faults of cardinality 2 for each class that has at least 8 components, i.e. method branches. For each generated fault set, we construct an activity matrix. We determine for each test that exercises the faulty components whether it is failing according to an *oracle quality probability* of 0.75, which was also used in prior work [10]. Then, for each generated activity matrix, we use *STACCATO* to generate fault candidates and *BARINEL* to generate a diagnosis report. Based on the diagnosis report we compute the wasted effort which is a measurement for diagnosability. To account for randomness of generating fault sets, we repeat this process 10 times. Note that we do not generate single component faults because in this case the optimal matrix for diagnosability is a diagonal activity matrix, i.e. each component each tested individually by a unit test. Additionally, *STACCATO* can sometimes take hours or days to generate fault candidates. Hence, we discard classes when gener-

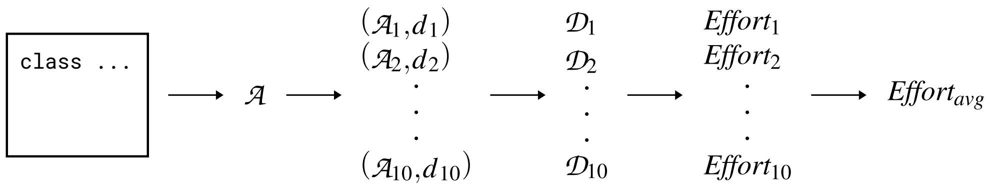


Figure 5.1: An activity matrix $\mathcal{A}$ is generated for a particular class. Then, 10 fault candidates of cardinality 2 are generated with a corresponding activity matrix $\mathcal{A}_k$. For each generated matrix, we perform fault diagnosis with *BARINEL* resulting in diagnostic report $\mathcal{D}_k$ and compute the wasted effort. Finally, we compute the average wasted effort. This process is repeated 10 times.

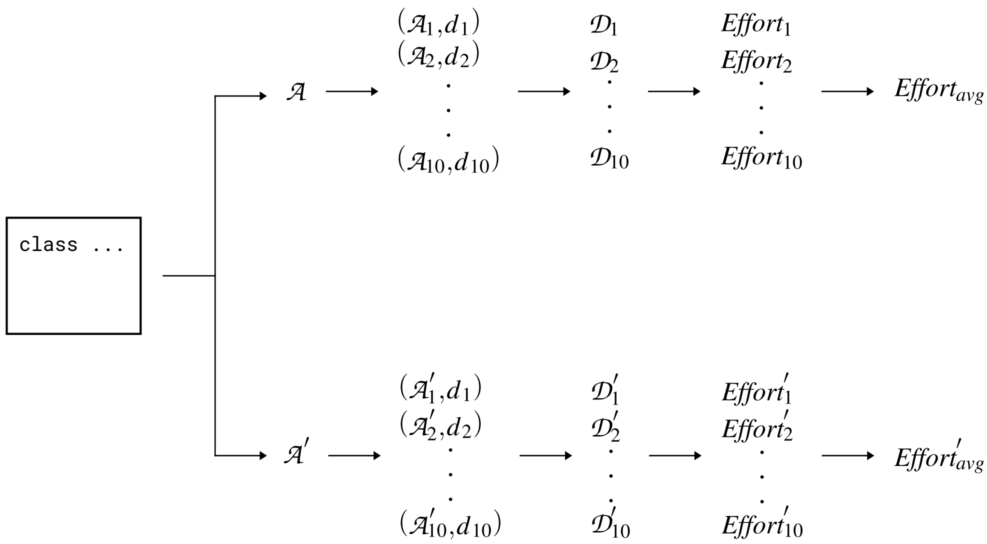


Figure 5.2: Two activity matrices $\mathcal{A}$ and $\mathcal{A}'$ are generated for a particular class based on two different test suites. We generate 10 fault candidates of cardinality 2 and accordingly generate 10 activity matrices. Then, we use *BARINEL* to perform fault diagnosis and compute the wasted effort.

ating fault candidates takes longer than 10 seconds; this resulted in 16 classes being discarded.

In the construction of the activity matrix we use the branch granularity, that is, every component of the activity matrix represents a method branch; this granularity is also used by Perez *et al.* [10]. To construct the activity matrix of a class we use Perez' DDU Maven plugin[1] using the `basicblock` granularity, which represents branch granularity. The steps after the obtaining the activity matrix in Figure 5.1 are performed using Python scripts[2].

This experiment is different from Perez *et al.*'s work because we do not improve

[1] `https://github.com/aperez/ddu-maven-plugin`
[2] `https://github.com/aaronang/ddu`

the DDU of a fixed system. Specifically, in Perez *et al.*'s study, the authors improved the DDU of a fixed system under test by generating new test cases using *EvoSuite*. However, in this experiment, we compute the DDU for each class and measure for each class its diagnosability using the aforementioned approach. Essentially, the difference is that we do not improve the DDU of a fixed system but we are simply measuring the DDU.

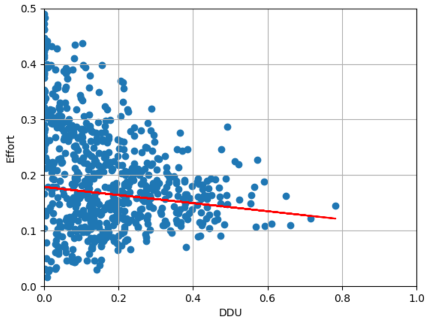
For this reason, we perform another experiment where we generate two test suites for each class with at least 8 components and at least 10 unit tests. In addition, we perform the same experiment but for all classes with at least 8 components. We generate two test suites by using all test cases and 50% of the test cases. For both test suites we compute the DDU and randomly generate 10 multiple components faults of cardinality 2 to compute the wasted effort. Similar to previous experiment we perform this process 10 times to account for randomness of generating fault sets. The intuition behind this experiment is when we improve the DDU of a fixed system, its diagnosability should improve too. The setup of this experiment is illustrated in Figure 5.2.

5.2 Experimental Results

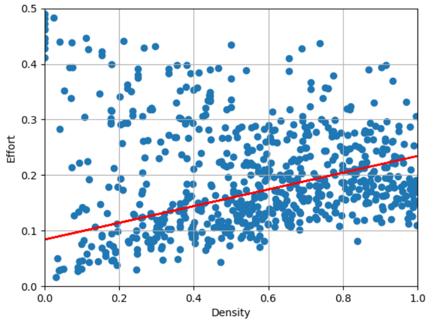
In the first experiment, we measure for each class the density, diversity, uniqueness, DDU, and diagnosability. The results of this experiment are shown in Figure 5.3. Note that in Figure 5.3 the population comprises all classes of all projects. Each datapoint in Figure 5.3 represents a class for which 100 fault candidates are generated in (potentially overlapping) sets of 10 fault candidates as described in Figure 5.1. We observe the that there is a positive correlation between density and effort, a negative correlation between diversity and effort, a negative correlation between uniqueness and effort, and a statistically non-significant weak correlation between DDU and effort.

To investigate the relations between these metrics in more detail, we display the correlation values per project in Table 5.1. For three projects we can say with 95% confidence that density is correlated with effort. However, the Pearson correlation for Commons Compress is negative while the Pearson correlation values for Commons Math and Commons Codec are positive. Hence, the results show no strong evidence that density is correlated with effort. Regarding diversity and uniqueness, we observe in Table 5.1 and Figure 5.3 that both metrics have a weak negative correlation with effort, and that the correlation values in 4 out of 5 projects are statistically significant. Regarding DDU, for two projects the results show statistical significance that DDU is negatively correlated to effort. However, for three projects there is no evidence that DDU is correlated to effort. Therefore, there is no strong evidence that DDU is negatively correlated to effort.

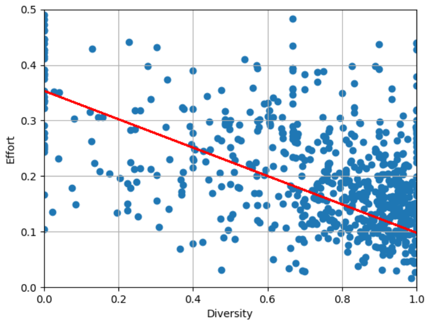
In the second experiment, we generate two test suites for a given class: a test suite with 50% of the test cases enabled, and a test suite with 100% of the test cases enabled. This naturally results in two test suites with two different DDU values. For



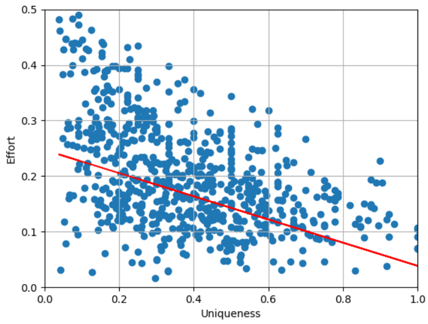
(a) DDU, $r = -0.046$, $p = 0.224$.



(b) Density, $r = 0.183$, $p < 0.001$.



(c) Diversity, $r = -0.326$, $p < 0.001$.

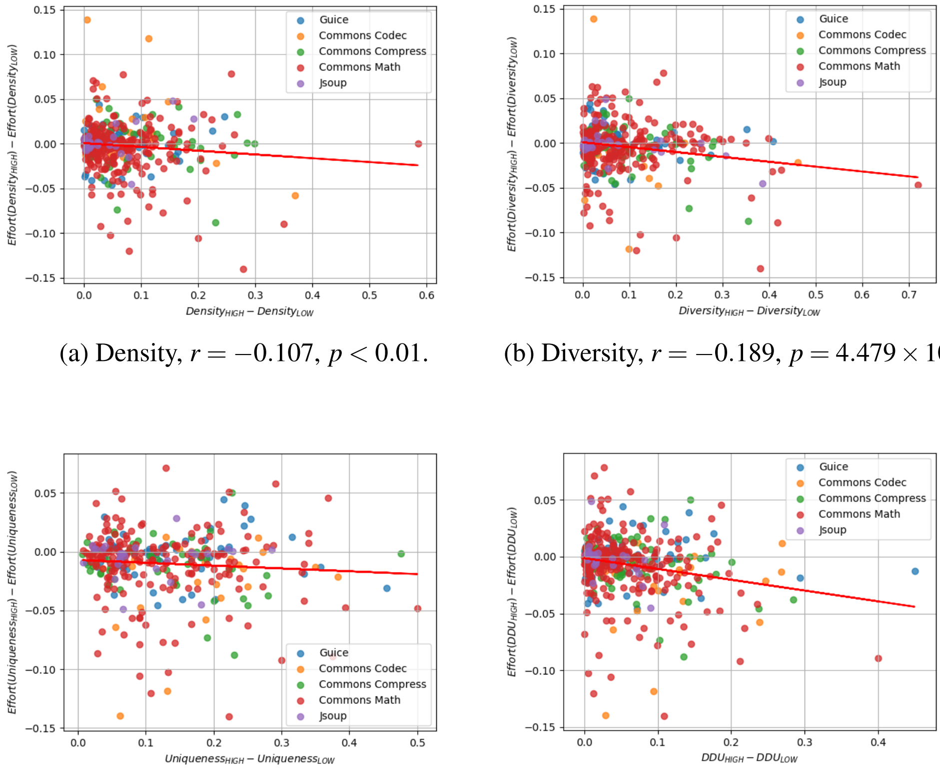


(d) Uniqueness, $r = -0.120$, $p < 0.001$.

Figure 5.3: Scatterplot of density, diversity, uniqueness, and DDU against effort.

Table 5.1: Correlation between density, diversity, uniqueness, DDU, and effort.

Subject	Size	Pearson correlation / Correlation p-value			
		Density	Diversity	Uniqueness	DDU
Commons Codec	34	0.63	-0.33	-0.65	-0.23
		5.880×10^{-5}	0.057	**3.713×10^{-5}**	0.197
Commons Compress	104	-0.22	-0.45	-0.40	-0.37
		0.027	**1.348×10^{-6}**	**2.083×10^{-5}**	**1.121×10^{-4}**
Commons Math	420	0.20	-0.36	-0.19	-0.03
		4.982×10^{-5}	**1.782×10^{-14}**	**7.553×10^{-5}**	0.572
Guice	94	0.01	-0.31	-0.29	-0.22
		0.935	**0.002**	**0.005**	**0.031**
Jsoup	37	0.29	-0.37	0.16	0.20
		0.085	**0.024**	0.337	0.229



(a) Density, $r = -0.107$, $p < 0.01$.

(b) Diversity, $r = -0.189$, $p = 4.479 \times 10^{-5}$.

(c) Uniqueness, $r = -0.079$, $p = 0.137$.

(d) DDU, $r = -0.223$, $p = 9.044 \times 10^{-7}$.

Figure 5.4: Scatterplot of delta density, delta diversity, delta uniqueness, and delta DDU against delta effort.

each class, we compare the effort of a test suite with a lower DDU value with a test suite with a higher DDU value. Identical approach is used for density, diversity, and uniqueness. Note that we exclude classes where the two test suites do not result in a metric difference. The results of this experiment are shown in Figure 5.4 and Table 5.2. In Figure 5.4, we observe that an increase in any metric — density, diversity, uniqueness, DDU — results in a lower required effort to diagnose mistakes. Although the results are statistically significant, see Figure 5.4, the correlations are weak.

Revisiting the second research question:

> **RQ2:** What is the relation between density, diversity, uniqueness, and DDU and diagnosability?

A: In the first experiment, two out of the five projects show that DDU is negatively correlated with diagnosability. In the second experiment, two out of the five

Table 5.2: Correlation between delta density, delta diversity, delta uniqueness, delta DDU, and delta effort.

Subject	Size / Pearson correlation / Correlation p-value			
	Density	Diversity	Uniqueness	DDU
Commons Codec	29 -0.277 0.145	24 -0.229 0.281	26 0.230 0.256	29 0.011 0.950
Commons Compress	74 -0.098 0.401	73 -0.372 **0.001**	55 -0.217 0.111	74 -0.297 **0.010**
Commons Math	251 -0.142 **0.023**	256 -0.162 **0.009**	186 -0.109 0.135	262 -0.280 $\mathbf{3.855 \times 10^{-6}}$
Guice	78 0.103 0.368	78 0.001 0.995	57 0.047 0.728	78 -0.006 0.952
Jsoup	29 0.452 **0.013**	29 -0.465 **0.011**	27 0.012 0.951	29 -0.273 0.150

projects show that improving the DDU value of a class' test suite will improve the diagnosability. Therefore, based on these two experiments, there is no evidence that DDU is strongly correlated to diagnosability. However, in the second experiment, we observe that an increase in DDU is weakly correlated to an decrease in effort and, thus, weakly correlated to diagnosability. There is also no strong evidence that density, diversity, and uniqueness are correlated to diagnosability.

Chapter 6

DDU vs. Error Detection

In this chapter, we discuss

Table 6.1: Correlation between density, diversity, uniqueness, DDU, and branch coverage.

Subject	Size	Pearson correlation / Correlation p-value			
		Density	Diversity	Uniqueness	DDU
Commons Codec	35	0.01 0.909	0.66 $\mathbf{1.27 \times 10^{-5}}$	0.66 **0.001**	0.54 $\mathbf{7.095 \times 10^{-4}}$
Commons Compress	108	0.21 **0.024**	0.42 $\mathbf{3.627 \times 10^{-6}}$	0.38 $\mathbf{4.736 \times 10^{-5}}$	0.37 $\mathbf{5.142 \times 10^{-5}}$
Commons Math	433	-0.04 0.307	0.18 $\mathbf{9.635 \times 10^{-5}}$	0.16 $\mathbf{7.426 \times 10^{-4}}$	0.16 $\mathbf{3.848 \times 10^{-4}}$
Guice	96	0.04 0.672	0.07 0.445	0.34 $\mathbf{4.969 \times 10^{-4}}$	0.25 **0.011**
Jsoup	38	0.36 **0.023**	-0.31 0.057	0.23 0.163	0.29 0.070

Table 6.2: Correlation between density, diversity, uniqueness, DDU, and error detection.

Subject	Size	Pearson correlation / Correlation p-value			
		Density	Diversity	Uniqueness	DDU
Commons Codec	35	0.64	0.15	-0.17	0.24
		$\mathbf{2.973 \times 10^{-5}}$	0.366	0.313	0.163
Commons Compress	108	0.25	0.18	0.15	0.27
		0.008	0.052	0.109	**0.004**
Commons Math	433	0.17	-0.18	-0.145	0.06
		$\mathbf{1.835 \times 10^{-4}}$	$\mathbf{1.281 \times 10^{-4}}$	**0.002**	0.146
Guice	96	0.28	0.020	0.12	0.27
		0.004	0.842	0.207	**0.006**
Jsoup	38	0.59	-0.26	-0.09	0.40
		$\mathbf{8.175 \times 10^{-5}}$	0.104	0.560	**0.010**

Chapter 7

Conclusion

Bibliography

[1] Rui Abreu, Peter Zoeteweij, and Arjan J. C. van Gemund. Spectrum-based multiple fault localization. In *Proceedings of the 2009 IEEE/ACM International Conference on Automated Software Engineering*, ASE '09, pages 88–99, Washington, DC, USA, 2009. IEEE Computer Society.

[2] Rui Abreu, Peter Zoeteweij, and Arjan JC Van Gemund. Spectrum-based multiple fault localization. In *Automated Software Engineering, 2009. ASE'09. 24th IEEE/ACM International Conference on*, pages 88–99. IEEE, 2009.

[3] Boris Beizer. *Software testing techniques*. Dreamtech Press, 2003.

[4] Tom Britton, Lisa Jeng, Graham Carver, Paul Cheak, and Tomer Katzenellenbogen. Reversible debugging software. *Judge Bus. School, Univ. Cambridge, Cambridge, UK, Tech. Rep*, 2013.

[5] J. Campos, R. Abreu, G. Fraser, and M. d'Amorim. Entropy-based test generation for improved fault localization. In *2013 28th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 257–267, Nov 2013.

[6] A. Gonzalez-Sanchez, H. G. Gross, and A. J. C. van Gemund. Modeling the diagnostic efficiency of regression test suites. In *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops*, pages 634–643, March 2011.

[7] Lou Jost. Entropy and diversity. *Oikos*, 113(2):363–375, 2006.

[8] Chris Parnin and Alessandro Orso. Are automated debugging techniques actually helping programmers? In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, pages 199–209. ACM, 2011.

[9] Chris Parnin and Alessandro Orso. Are automated debugging techniques actually helping programmers? In *Proceedings of the 2011 International Sympo-*

sium on Software Testing and Analysis, ISSTA '11, pages 199–209, New York, NY, USA, 2011. ACM.

[10] Alexandre Perez, Rui Abreu, and Arie van Deursen. A test-suite diagnosability metric for spectrum-based fault localization approaches. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, pages 654–664, 2017.

[11] Thomas Reps, Thomas Ball, Manuvir Das, and James Larus. The use of program profiling for software maintenance with applications to the year 2000 problem. In *Software EngineeringESEC/FSE'97*, pages 432–449. Springer, 1997.

[12] John Robbins. *Debugging Applications for Microsoft .NET and Microsoft Windows*. Microsoft Press, 2003.

[13] Friedrich Steimann, Marcus Frenkel, and Rui Abreu. Threats to the validity and value of empirical assessments of the accuracy of coverage-based fault locators. In *Proceedings of the 2013 International Symposium on Software Testing and Analysis*, ISSTA 2013, pages 314–324, New York, NY, USA, 2013. ACM.

[14] Y. Le Traon, F. Ouabdesselam, and C. Robach. Software diagnosability. In *Proceedings Ninth International Symposium on Software Reliability Engineering (Cat. No.98TB100257)*, pages 257–266, Nov 1998.

[15] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, Aug 2016.

[16] W. E. Wong, R. Gao, Y. Li, R. Abreu, and F. Wotawa. A survey on software fault localization. *IEEE Transactions on Software Engineering*, 42(8):707–740, Aug 2016.