THE RELATIONSHIP BETWEEN USAGE OF SETUP FIELDS AND FIELD DEPENDENCY WITH DEAD FIELD DETECTION IN TEST CODE

ABDUS SATTER
BSSE 0401

A Thesis
Submitted to the Bachelor of Science in Software Engineering Program Office
of the Institute of Information Technology, University of Dhaka
in Partial Fulfillment of the
Requirements for the Degree

BACHELOR OF SCIENCE IN SOFTWARE ENGINEERING

Institute of Information Technology
University of Dhaka
DHAKA, BANGLADESH

THE RELATIONSHIP BETWEEN USAGE OF SETUP FIELDS AND FIELD DEPENDENCY WITH DEAD FIELD DETECTION IN TEST CODE

ABDUS SATTER

Approved:

Signature	*Date*
Supervisor: Dr. Kazi Muheymin-Us-Sakib	
Committee Member: Dr. Kazi Muheymin-Us-Sakib	
Committee Member: Dr. Md. Shariful Islam	
Committee Member: Alim Ul Gias	
Committee Member: Amit Seal Ami	

To *Rajia Sultana*, my mother
who has always been there for me and inspired me

Abstract

Dead fields are the unused setup fields in the test code which reduce the comprehensibility and maintainability property of the code. The reason of these fields' occurrences is that in most of the cases, developers initialize setup fields without considering the usage of those fields in the test methods. In order to increase the maintainability of the code, dead fields should be identified and removed which can be done by manually inspecting all the test code but it is not feasible when the project is large in size. However, existing automatic dead field detection techniques could not find dead fields correctly. So, this leads to the need of an automatic dead field detection technique by which dead fields will be detected correctly and removed to ensure the quality of the code.

In this research a technique named Dead Field Identifier (DFI) is proposed to find dead fields in the test code. The technique finds all the fields in the test class. Later, it detects the setup method and checks which fields are initialized in that method. Those fields are considered as setup fields. For dead field detection, the technique checks the usage of setup fields in the test methods. After that, it gathers all the used setup fields and identifies such unused setup fields on which used setup fields are dependent for the initialization. These fields are also marked as used setup fields. At last, unused setup fields are separated and marked as dead fields.

For the assessment of DFI, this technique as well as an existing dead field detection technique named TestHound (TH) is implemented in Java programming language. Another approach called Manual Inspection (MI) is also used in the experiment where MI involves manually reviewing the test code to detect dead fields and setup fields. In comparative analysis, it is found that DFI detects 14% more setup fields and 50.89% more dead fields than TH in two open source projects due to identifying setup fields, resolving field dependency, and finding usage of these fields properly. The results obtained using MI for these projects are also compared with DFI and for both approaches, the outcomes are alike.

Acknowledgments

I would like to thank Dr. Kazi Muheymin-Us-Sakib for his support and guidance during the thesis compilation. He has been relentless in his efforts to bring the best out of me.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Dead field is one of the common test smells found in test code. It is not a programming error or bug but it causes the attrition of test code quality by reducing test code maintainability and comprehensibility. While writing test code, it is required to define the configuration of the system under test which is commonly known as test fixture. One of the elements of test fixture is setup fields where these fields are initialized by the setup method in the test class. Here, setup method is a special type of method which invoked by the system before the execution of any test case and its intent is to prepare the test fixture first.

The purpose of initializing setup fields is that test methods will be able to use these fields without instantiating every time in each test case. However, if any setup field is found that is not used by any test method, this field is considered as dead field. It unnecessarily increase line of code, reduces maintainability and slows down software development. In order to detect this smell a technique named Dead Field Identifier (DFI) is proposed. The following sections explain the motivation for working in dead field detection, the research question which is deal through out this work, contribution in dead field identification and organization of this thesis.

1.1 Motivation

Dead fields are the initialized setup fields which are never used by any test method in the test code. There are many problems associated with the presence of dead fields in test code such as increasing size of the project by adding unnecessary code, making code hard to adapt any change in the code, creating misapprehension while refactoring production code and so on. All these problems are responsible to reduce maintainability of the code. An example case is described below which demonstrates how dead fields affect code maintainability.

```
public class Account {
       public void Login(LoginModel loginInfo){
              .........................
       }
       public void Logout(){
              ..........................
       }
}

public class GoogleAccount extends Account{
       public void Login(LoginModel loginInfo){
              .........................
       }
       public void Logout(){
              .........................
       }
}

public class FacebookAccount extends Account{
       public void Login(LoginModel loginInfo){
              ..........................
```

```
        }
        public void Logout(){
                ..........................
        }
}
public class AccountTest{
        private Account account;
        private GoogleAccount googleAccount;
        private FacebookAccount facebookAccount;
        @Before
        public void setUp() throws Exception {
                account = new Account();
                googleAccount = new GoogleAccount();
                facebookAccount = new FacebookAccount();
        }
        @Test
        public void testLogin() {
                boolean logged = account.login();
                assertEquals(added, true);
        }
}
```

In this case, there are three production classes in the following sample code which are *Account*, *GoogleAccount*, and *FacebookAccount*. A test class named *AccountTest* is also depicted in the sample code. *AccountTest* class just uses the *account* object but other two fields named *googleAccount* and *facebookAccount* are initialized in the setup method but these are not used in any test method. So these fields are dead fields. Now, if it is required to remove *GoogleAccount* and *FacebookAccount* from the production, it is essential to remove *googleAccount* and *facebookAccount* fields from the *AccountTest* class. If this issue is not considered while performing this change, the test class will not run as two unnecessary fields are initialized in the class. However, if these two dead fields are removed from the test class, there will be no problem to make this change because *AccountTest* class only depends on *Account* class. This is how dead fields create problems in refactoring or changing the code.

Another case is described below where dead fields decrease code maintainability by increasing the size of production code. In this case, the sample code is written following Test Driven Development (TDD) approach. In the following sample code, there is a test class named *ProductServiceTest* which contains five fields - *dummyProduct*, *dbConnector*, *dbContext*, *productRepo* and *productService*. All fields are initialized in the setup method so these fields are setup fields. There are two test methods in the class and these test methods use two fields which are *productService* and *dummyProduct*. So, *dbConnector*, *dbContext*, and *productRepo* are dead fields in this scenario.

```
public class ProductServiceTest {

        Product dummyProduct;
        DatabaseConnector dbConnector;
        DatabaseContext dbContext;
        ProductRepository productRepo;
        ProductService productService;

        @Before
        public void setUp() throws Exception {

                dummyProduct = new Product();
                dbConnector = new DatabaseConnector();
                dbContext = new DatabaseContext();
                productRepo = new ProductRepository(dbConnector, dbContext
                    );
                productService = new ProductService();

        }
        @Test
        public void testAddProduct() {
                boolean added = productService.addProduct(dummyProduct);
                assertEquals(added, true);
        }
        @Test
        public void testRemoveProduct() {
                boolean removed = productService.RemoveProduct(
                    dummyProduct);
                assertEquals(added, true);
        }
}
```

Now in TDD this class is written first and production code are derived from that class. If the dead fields are not removed from the test class, five classes will be generated which are Product, DatabaseConnector, DatabaseConetxt, ProductRepository, and ProductService. A sample code illustrating these classes is shown below.

```
public class ProductService {
        public boolean addProduct(Product dummyProduct) {
                ..........
        }
        public boolean RemoveProduct(Product dummyProduct) {
                ..........
        }
}



public class ProductRepository {
        public ProductRepository(DatabaseConnector dbConnector,
                        DatabaseContext dbContext) {
                .................
        }
        ...................................
}
public class Product {
        ..................................
}
public class DatabaseConnector {
        ..................................
}
public class DatabaseContext {
        .................................
}
```

As in the test class, test methods - *testAddProduct* and *testRemoveProduct* invoke two methods of *productService* object, so according to TDD, *ProductService* class holds two methods like *addProduct* and *removeProduct*. In the setup method, two fields *dbConnector* and *dbContext* are used for the instantiation of *productRepo*. So the constructor of *ProductRepository* class will take two parameters of type *DatabaseConnector* and *DatabaseConetxt*.

Here, the important observation is that many additional production codes have been generated from the test class *ProductServiceTest*. However, if the dead fields have been removed, only two classes will be derived from the test class instead of five, and these are Product and *ProductService*. Thus, the production code becomes more maintainable and comprehensible comparatively.

Martin Fowler coined code smell [1, 2] and later, van Deursen first introduced the concept of test smells in test code [3, 4]. Michael Grielar and van Deursen identified five new test smells including dead fields and developed a tool named TestHound[1] to identify those smells by analyzing test fixture [5]. The tool performs well in identifying those test smells but for dead field detection, manual code inspection is required to resolve field dependency and usage of setup fields among the test methods. TestLint, another automatic test smells identification tool, can deal with some test smells by finding the properties of those smells in the test code [6]. However, this tool cannot handle dead fields in the test code through test fixture and test method analysis because dead fields have not been considered here. Although Bart van Rompaey proposed a metrics based approach based on the unit test concept to identify eager test smell, the author did not address any metric to automatically detect dead fields [7, 8]. Bavota disclosed the distribution and impact of test smells in software maintenance but no approach was explained to automatically identify dead fields in his analysis [9].

1.2 Research Question

Resolving data dependency and finding usage of setup fields through analyzing the test code are required to locate dead fields in the test code. This will help to increase the understandability and maintainability of the test code by identifying and removing dead fields from the code. Developers can manually inspect and analyze the test code to find and remove dead fields from the test code. However, manually inspecting source code induces additional time and cost to software development and for large projects, it is not feasible at all. On the other hand, an

[1]http://www.swerl.tudelft.nl/twiki/pub/MichaelaGreiler/TestHound/TestHound

automatic process which will identify dead fields in test code by analyzing the code can mitigate the problem. However, existing automatic tools and techniques could not detect dead fields accurately in the test code. This ultimately leads to the following research question.

How to develop a process to identify dead fields correctly and automatically by analyzing usage of setup fields and field dependency in test methods?

A technique is required which may take test code and provide a list of dead fields, list of all setup fields in test fixture and list of setup fields used in each test case automatically. This splits the research question into the following sub questions which are required to be answered.

1. How to find usage of setup fields in the test code?
    (a) All test methods are required to be identified which are under the same test fixture for getting fixture specific test methods.
    (b) For each test method, the body of the method is needed to be analyzed for obtaining a list of methods invoked directly or indirectly by the method.
    (c) For each method invoked by the test method, it is required to check whether the method has used any of the setup fields for generating list of used fields.
2. How to detect dead fields from the identified setup fields in the test code?
    (a) All the setup methods are required to be analyzed to get all the setup fields in the test fixture.
    (b) For each setup field, the list of used fields obtained from the previous sub-question needs to be searched entirely for checking whether the list

contains the field or not. This will ultimately provide a list of dead fields.

1.3 Contribution of This Research

In this research, a technique named Dead Field Identifier (DFI) is proposed to automatically and correctly detect dead fields by analyzing usage of setup fields and field dependency in test methods. Initially, all the fields in the test code are gathered by parsing the code. As header fields[2] are also considered as setup fields, so all the header fields are figured out from the identified fields in the test code. Later, setup method and all other methods invoked directly or indirectly by it are identified. The body of those methods are extracted to find all the setup fields in the code. To find usage of those fields, all the test methods and other methods invoked by those are obtained and fields which are used in those methods are detected. Usually, it is found in the code that a setup field which is used in at least one test method may depend on one or more other setup fields which are never been used by any test method. So, those fields are identified through analyzing field dependency among the setup fields and considered as used setup fields. At the end, all unused fields are separated from the setup field list and those are marked as dead fields.

In order to evaluate DFI, an experiment is performed where this technique and an existing approach named TestHound (TH)[5] are implemented using Java programming language. For the comparative analysis, two open source projects - eGit and EquationSolverTest are used as test beds in the experiment where *eGit* is larger in size than *EquationSolver*. The reason of choosing two different sized projects is to observe the accuracy and behavior of DFI in dead field detection. In the experiment, both tools are run on these projects and it is seen DFI detects

[2]Header fields are those fields which are initialized in the class header

14% more setup fields and 50.89% more dead fields than TH in total. More precisely, DFI identifies additional 12% setup fields and 49.51% dead fields in eGit in comparison with TH. On the other hand, for EquationSolverTest, DFI finds 82% more setup fields and 66.67% more dead fields than TH. Another approach named Manual Inspection (MI) is also applied in the experiment. MI involves manually reviewing the test code to find setup fields and dead fields in the test beds. The results obtained for this approach are similar to the outcomes of DFI.

1.4 Organization of the Thesis

This section provides an overview about the remaining chapters of this thesis. The chapters are organized as follows.

Chapter 2: Classification of smells, different types of code smells and test smells are exemplified in this chapter. Besides, various code metrics like Object Oriented Metrics and Source Code Metrics are also described here.

Chapter 3: This chapter focuses on the existing works in literature regarding the identification, reduction and impact of test smells.

Chapter 4: The proposed technique named DFI is explained in this chapter. All the devised algorithms for DFI along with complexity analysis of each algorithm are also shown here.

Chapter 5: This chapter provides a detailed explanation on the implementation of DFI and existing technique like TestHound. A comparative result analysis between DFI, TestHound and Manual Inspection (MI) is also discussed in this chapter

Chapter 6: It is the concluding chapter which contains a discussion about the proposed technique, important threats to validity and some future directions.

Chapter 2

Background Study

Smells in the code mean that the block of the code exhibits some symptoms which are responsible for reducing the comprehensibility and maintainability of the code. The presence of smells in the code does not represent bug or error in the program, even does not have any impact on the output of the software. However, smells forecast time consuming and expensive software development and harder software maintenance in the long run. This chapter outlines different types of smell and the impact of these smells during software development and maintenance.

2.1 Types of Smells

In modern software development, smells can be broadly categorized into two types-one is production code smell or simply code smell and another is test code smell or test smell. Both types of smells have been explained as follows.

2.1.1 Code smells

"Code Smells identify frequently occurring design problems in a way that is more specific or targeted than general design guidelines (like loosely coupled code or duplication-free code)."[10]—Joshua K

Code smell is an indicator that describes the reduction of code quality and violation of design principles. It is neither a programming error nor a bug; instead it is a hint for slow software development, difficult software maintenance and time consuming software change adaptation. Code smells are mostly found in production code where this kind of code is written for the development of the software. However, code smell is also applicable in test code as modern testing frameworks have been developed following XUnit frameworks [11].

Martin Fowler first introduced code smell in the literature [2]. Later, various code smells have been discovered and appropriate refactoring techniques have also been proposed by many researchers. A list of common code smells have been described in essence below.

- **Inappropriate Naming smell** means names given to variables, fields or methods are not clear or meaningful. For example, the code snippet given below contains a variable named *a* which is not a meaningful name in this context.

  ```
  public int a = 10;
  public int m(int a, int b){
     return a+b;
  }
  ```

- **Comment smell** occurs when comments are used to understand the intent of a code block which is hard to understand by inspecting lines of code. A code snippet having this smell is given below.

  ```
  void AddToList(string element){
      if (!m_readOnly){
          int newSize = m_size + 1;
          if (newSize > GetCapacity()){
              // grow the array
  ```

```
            m_capacity += INITIAL_CAPACITY;
            string[] elements2 = new string[m_capacity];
            for (int i = 0; i < m_size; i++)
                elements2[i] = m_elements[i];

            m_elements = elements2;
        }
        m_elements[m_size++] = element;
    }
}
```

- **Long Method smell** represents a method which is too long that it is difficult to understand the method. The following example demonstrates Long Method smell where *toStringHelper* is a long method.

```
private String toStringHelper(StringBuffer result)
{
    result.append("<");
    result.append(name);
    result.append(attributes.toString());
    result.append(">");
    if (!value.equals(""))
        result.append(value);
    Iterator it = children().iterator();
    while (it.hasNext())
    {
        TagNode node = (TagNode)it.next();
        node.toStringHelper(result);
    }
    result.append("</");
    result.append(name);
    result.append(">");
    return result.toString();
}
```

- **Long Parameter List** is seen in the code when a method takes too many parameters. In the example given below, *storeResult* method takes 11 parameters and thus, the method contains Long Parameter List smell.

```
public void storeResult(String firstName, String lastName, String
    registrationNum, String rollNumber, int marksInSubject1, int
    marksInSubject2, int marksInSubject3, int marksInSubject4, int
    marksInSubject5, int marksInSubject6, String obtainedGrade ){
   ........
}
```

- **Feature Envy smell** occurs when a method uses features of other classes rather than the class it belongs. For instance, the following code contains a class named *CapitalStrategy* and this class has a method called *capital.* Here, the method is using features of another class named *Loan* rather than using its own class's features. So, the class is suffering from Feature Envy smell.

```
Public class CapitalStrategy{
double capital(Loan loan){
    if (loan.getExpiry() == NO_DATE && loan.getMaturity() !=
        NO_DATE)
        return loan.getCommitmentAmount() * loan.duration() * loan.
            riskFactor();

    if (loan.getExpiry() != NO_DATE && loan.getMaturity() ==
        NO_DATE)
    {
        if (loan.getUnusedPercentage() != 1.0)
            return loan.getCommitmentAmount() * loan.
                getUnusedPercentage() * loan.duration() * loan.
                riskFactor();
        else
            return (loan.outstandingRiskAmount() * loan.duration() *
                 loan.riskFactor()) +
                (loan.unusedRiskAmount() * loan.duration() * loan.
                    unusedRiskFactor());
    }

    return 0.0;
  }
}
```

- **Dead Code smell** means code that is no longer used in a system or related system. An example is shown below where second and third constructor of *Loan* class have not been used in anywhere, in fact there is no need to declare these constructors according to the scope of the class. For this reason, these constructors are the examples of Dead Code smell.

```
public class Loan{

public Loan(double commitment, int riskRating, Date maturity, Date
    expiry) {
        this(commitment, 0.00, riskRating, maturity, expiry);
}

public Loan(double commitment, double outstanding, int
    customerRating, Date maturity, Date expiry){
        this(null, commitment, outstanding, customerRating, maturity
            , expiry);
}

public Loan(CapitalStrategy capitalStrategy, double commitment, int
     riskRating, Date maturity, Date expiry) {
        this(capitalStrategy, commitment, 0.00, riskRating, maturity
            , expiry);
}
 ...
}
```

- **Duplicate Code smell** represents a situation when same types of code are found in several places and similarity can be like exact copy of the code or identically not the same but logically equal. For example, in the following sample code, two methods *getStaticTemplate* and *getDynamicTemplate* are logically same.

```
public static MailTemplate getStaticTemplate(Language language){
        MailTemplate mailTemplate = null;
        if(language.equals(Languages.English)){
                mailTemplate = new EnglishLanguageTemplate();
        }else if(language.equals(Languages.French)){
                mailTemplate = new FrenchLanguageTemplate();
        }else if(language.equals(Languages.Chinese)){
                mailTemplate = new ChineseLanguageTemplate();
        }else{
                throw new Exception("Language Excepion");
        }
}
```

```
public static MailTemplate getDynamicTemplate(Language language,
    String content){
        MailTemplate mailTemplate = null;
        if(language.equals(Languages.English)){
                mailTemplate = new EnglishLanguageTemplate(content);
        }else if(language.equals(Languages.French)){
                mailTemplate = new FrenchLanguageTemplate(content);
        }else if(language.equals(Languages.Chinese)){
                mailTemplate = new ChineseLanguageTemplate(content);
        }else{
                throw new Exception("Language Excepion");
        }
}
```

- **Refused Bequest smell** exhibits when subclasses inherit code that they do not want. An example of Refused Bequest smell is shown in Figure 2.1. In this diagram, class *Rectangle* and *Triangle* do not require super class features like *Add*, *Draw* and *Remove* methods but due to extending the super class, these subclasses inherit all those methods.

- **Large Class smell** defines a class that takes too many responsibilities. The following sample code demonstrates this smell where the class *DatabaseConnector* performs many operations like adding person, product, customer and so on. However, the actual responsibility of the class should be to connect with database and other methods should belong to a repository class.

```
public class DatabaseConnector {
        public void connectToDb() {
        ........................
```

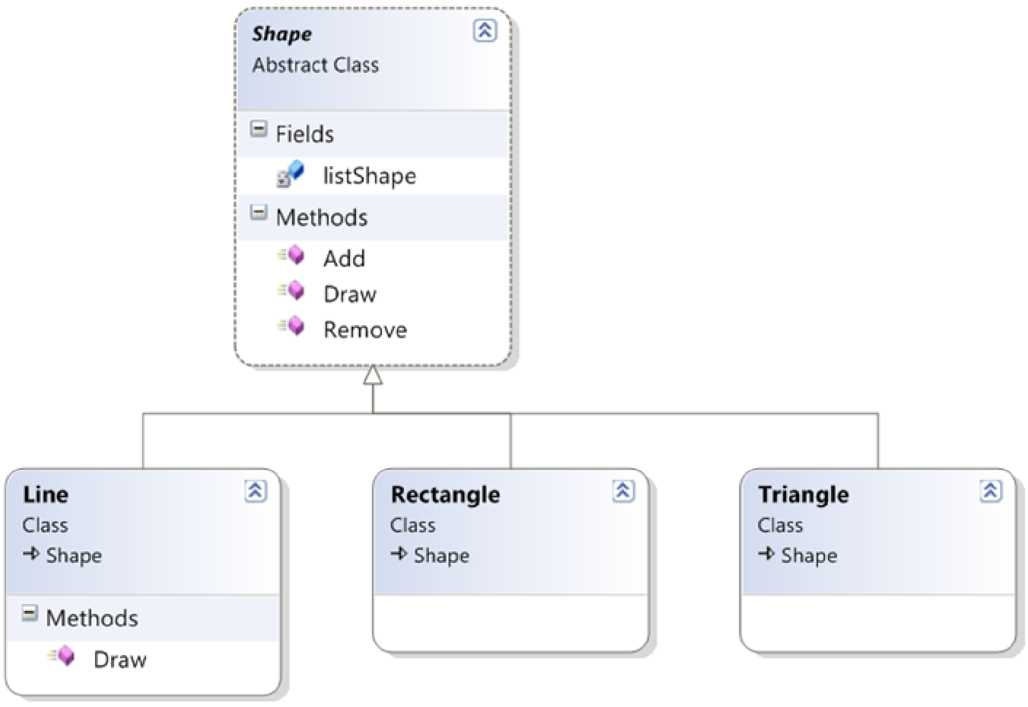

Figure 2.1: An example class diagram to represent Refused Bequest Smell

```
}
public void addPerson() {
........................
}
public void addCustomer() {
........................
}
public void addPruduct() {
.........................
}
public void deleteProduct() {
.........................
}
public void updateProduct() {
..........................
}
```

```
        .........................
}
```

- **Data Clumps** means software has data items that appear together and changing a single item of a group causes loosing of meaning of that data group. Here is an example of Data Clumps that handles order processing using a customer's credit card.

```
public bool SubmitCreditCardOrder(string firstName, string lastName
    , string zipcode, string streetAddress1, string streetAddress2,
     string city, string state, string country, string phoneNumber,
     string creditCardNumber, int expirationMonth, int
    expirationYear, decimal saleAmount){
//    submit order
}
```

- **Switch Statements** is a violation of polymorphism in Object Oriented Programming (OOP) as using switch case statements define something which acts different ways in different situations. In order to handle this scenario, OOP provides a feature named polymorphism which is constructing subclasses to handle each case defined in the switch-case block. Here is a code snippet which contains this smell.

```
public static int GetCarMaxRoomNumberNominal(string trainName,
    TrainCar car){
    if (!String.IsNullOrEmpty(trainName)){
        trainName = trainName.ToUpper();
        if (trainName.StartsWith("        ")){
            return 66;
        }
        else if (trainName.StartsWith("          ")){
            return 72;
        }
    }

    switch (car.Category){
        case TrainCarCategory.Lux:
        case TrainCarCategory.Soft:
            return car.TwoStorey ? 96 : 18;
        case TrainCarCategory.Sedentary:
            if (car.ServiceClass.Contains("1C")){
                return 42;
            }
            if (car.ServiceClass.Contains("3C")){
                return 117;
            }
            return 1;
        default:
            return 1;
    }
}
```

- **Temporary Field** is a field in the class which is not actually a property of the class rather it is used to store a value temporarily in some situations. In the following example, the field *temp* is a Temporary field.

```
public class Calculator {

        private int temp;

        public int add(int a, int b, int c){
                temp = a+b;
```

```
              return temp+c;
       }
}
```

- **Lazy Class** is a class that is not doing enough to carry its weight. The following example demonstrates Lazy class smell. In the following example, *LazyClazz* does not do anything by implementing *SomeInterface*, so its a Lazy Class.

```
public interface SomeInterface {
void methodOne();
void defaultMethod();
}
public abstract class LazyClazz implements SomeInterface {
public abstract void methodOne();
public void defaultMethod() {
       //do nothing
}
}
public class WorkerClazz extends LazyClazz {
public void methodOne() {
       // some actual code here
}
public void defaultMethod() {
       //some more actual code
}
}
```

- **Data Class smell** occurs when a class contains many attributes but has no or few logical operations. In the following example, class *Content* only has attributes or fields but does not have any method. So, this class is considered as Data Class.

```
public class Content {


        public int id;
        public String Name;
        public String Author;
        public String Details;
}
```

- **Speculative Generality** is a case, where developers write code which are not required for current context but may be needed in future. The following sample code contains this smell because there is no need to have *presentAddress* and *permanentAddress* fields according to the requirements of the class.

```
public class Person {
        public int Id;
        public String firstName;
        public String lastName;
        public String presentAddress;
        public String permanentAddress;
}
```

- **Message Chain smell** is a scenario where a class calls an object from another object, which then asks another and so on. A sample code having this smell is given below where it is required to follow a chain of object initialization in order to instantiate *mode* variable.

```
public class Calculator {


        public void setMode(){
                IMode mode = new ScientificMode(new AdvanceMode(new
                    DecimalMode(new Decimal64Mode(new BasicMode())))
                    ;
                ........
        }
        .........
}
```

- **Middle Man smell** means that a class acts as a router whose responsibility is to delegate tasks to other subsequent classes rather than performing these tasks. According to the following example, all the methods of the class *MiddleWare* just delegate given tasks rather than adding any logic or performing any activity.

```
public class MiddleWare {


        public int getOptionId(){
                return new Option.GetOptionId();
        }


        public String getParammeter(){
                return new Route.GetParameter();
        }


        public Node getNode(){
                return new NodeConst.GetNode();
        }
}
```

- **Inappropriate Intimacy** is a case where two classes are tightly coupled with each other. An example code having this smell is shown below where

CustomerRepository is tightly coupled with *Database* class.

```
class CustomerRepository{

    private readonly Database database;

    public CustomerRepository(Database database){
        this.database = database;
    }

    public void Add(string CustomerName){
        database.AddRow("Customer", CustomerName);
    }
}

class Database{
    public void AddRow(string Table, string Value){
    }
}
```

2.1.2 Test smells

Another type of code found in the Software Development Life Cycle (SDLC) is test code which is written to ensure that production code is performing correctly. Like production code, test code may also contain code smells but a special type of smell named test smell is also found in the test code. Test smells are responsible for the degradation of test code quality by reducing code maintainability and comprehensibility. The term test smell was first coined by van Deursen and he contributed in the literature by discovering 11 different test smells [12]. Later, various researches have been carried out for discovering new test smells, identification and refactoring mechanism of test smells. A list of significant test smells have been depicted as follows.

- **Mystery Guest smell** is a case where a test method depends on external resources for execution. For example, the following code snippet contains a test method named *testAddPerson* which depends on *person.txt* file.

```
@Test
public void testAddPerson() {
        personController.add(new DummyPerson());
        List<Person>newPersonList = new PersonUtility().getAllPerson
            ("C://data//person.txt");
        assertEquals(personList.size()+1, newPersonList.size());
}
```

- **Resource Optimism smell** means it is assumed to have a certain state or existence of external resources before running a test case. The previous sample code can be considered to have this smell because it assumes to have a file named *person.txt*.

- **General Fixture smell** means a case where a test case fixture is designed in such a way that instead of using the whole, the test methods only use a portion of it. In the following example, only two fields - *personList* and *personController* have been used by the test methods of *PersonControllerTest* class.

```
public class PersonControllerTest {

        List<Person>personList;
        PersonController personController;
        FileReader fileReader;
        @Before
        public void setUp() throws Exception {
                personList = new PersonUtility().getAllPerson("C://
                    data//person.txt");
                personController = new PersonController();
                fileReader = new FileReader();
        }

        @Test
        public void testAddPerson() {
                personController.add(new DummyPerson());
                List<Person>newPersonList = new PersonUtility().
                    getAllPerson("C://data//person.txt");
                assertEquals(personList.size()+1, newPersonList.size
                    ());
        }

        @Test
        public void testDeletePerson() {
                personList = new PersonUtility().getAllPerson("C://
                    data//person.txt");
                personController.delete(1);
                List<Person>newPersonList = new PersonUtility().
                    getAllPerson("C://data//person.txt");
                assertEquals(personList.size(), newPersonList.size()
                    +1);
        }

}
```

- **Eager Test smell** is a case where a test method checks several methods of the tested object. The code shown below has a test method named *testCalculator* which tests three methods of *Calculator* class like *add*, *convertToBin*, and *convertToDec*.

```
@Test
public void testCalculator(){
        String result = calculator.add(calculator.convertToBin(10),
           calculator.convertToBin(20));
        assserEquals(calculator.convertToDec(result), 30);
}
```

- **Assertion Roulette smell** is an indication of having several assertions with no explanation within the same test method. An example having this smell is shown below where the method *testAssertions* contains eight assertion statements.

```
@Test
public void testAssertions() {
        String str1 = new String("abc");
        String str2 = new String("abc");
        String str3 = null;
        String str4 = "abc";
        String str5 = "abc";
        int val1 = 5;
        int val2 = 6;
        String[] expectedArray = { "one", "two", "three" };
        String[] resultArray = { "one", "two", "three" };

        assertEquals(str1, str2);
        assertTrue(val1 < val2);
        assertFalse(val1 > val2);
        assertNotNull(str1);
        assertNull(str3);
        assertSame(str4, str5);
        assertNotSame(str1, str3);
        assertArrayEquals(expectedArray, resultArray);
}
```

- **Indirect Testing smell** is found when an object is tested indirectly through another object. The code segment given below is an example of Indirect Testing smell where *DbConnection* is tested though *ProductRepo*.

```
@Test
public void testDbConnection(){

        ProductRepo productRepo = new ProductRepo(new DbConnection()
            );
        bool added = productRepo.addProduct(new DummyProduct());
        assertTrue(added==true);
}
```

- **For Testers Only smell** occurs when a production class contains methods used only by the test methods. In the following sample code, *PrepareDummyAccountForTest* method is written only for the use of testing and it has no use in the production code.

```
public class AccountService {

        Account account;
        public void PrepareDummyAccountForTest(){
                account = new Account("userId","password");
                account.setToken("token");
        }
        public void logIn(){
                .......
        }

        public void logOut(){
                .......
        }

        public void signUp(User user){
                .......
        }
}

public class AccountServiceTest {

        Account account;
        AccountService accountService;

        @Before
        public void setUp() throws Exception {
                accountService = new AccountService();
                acccount = accountService().
                   PrepareDummyAccountForTest();
        }
        .......
}
```

- **Sensitive Equality smell** means the existence of the toString method in the assertion statements. A code snippet is shown below where *calcula-*

tor.Add(num1,num2).toString() contains *toString()* method in *assertEquals*.

```
@Test
public void testAdd(){
        Integer num1=10;
        Integer num2=20;
        assertEquals("30", calculator.Add(num1,num2).toString());
}
```

- **Dead Field smell** is a case where an initialized field in the test fixture has never been used by any test method. The sample code given below has an instantiated field named *connStr* which is not used by any test method. So, this field is considered as Dead Field.

```
public class RemoteConnectorTest {

        private ConnectionUtility connectionUtil;
        private ConnectionString connStr = new ConnectionString("
            localhost", "8080");

        @Before
        public void setUp() throws Exception {
                connectionUtil = new ConnectionUtility(new
                    ConnectionString(
                                "localhost", "8080"));
        }

        @Test
        public void testConnection() {
                connectionUtil.establishConnection();
                assertTrue(connectionUtil.connected == true);
        }
}
```

- **Test Maverick smell** is seen in the test code when a test fixture contains implicit setup but test method under the fixture is completely independent from the setup procedure. The sample code below has a field named *product* and a setUp method for preparing test fixture. However, it is seen in the test methods that no test methods use the fixture and this indicates the unnecessary use of setUp method for the fixture initialization.

```
public class ProductServiceTest {

        Product product;
        @Before
        public void setUp() throws Exception {
                product = new ProductFactory().getProduct();
        }

        @Test
        public void testProductTax() {
                int tax = new Product().calculateTax(1000,15);
                assertEquals(150,tax);
        }
        ......
}
```

2.2 Metrics for smell detection

You cannot control what you cannot measure [13]—DeMarco in 1982. So in order to identify smells and understand the impact of these on software maintenance, researchers have proposed various metrics like Line of Line of Code, McCabeś

Cyclomatic Complexity, Cohesion, and so on which are described as follows.

In order to detect code smells and test smells in code, different Source Code Metrics like Line of Code (LOC), Non-Commented Line of Code (NLOC) and Object Oriented Metrics like Cohesion, Coupling are commonly used. The reason is that most of the smellsćharacteristics depend on those metrics like large class smell and long method smell rely on LOC of the class and method respectively, inappropriate intimacy, test maverick and eager test depend on cohesion and coupling metrics. In subsection 2.2.1, different Source Code Metrics like LOC, McCabes Cyclomatic Complexity have been described in essence and later, various common Object Oriented Metrics such as Coupling, Cohesion, Weighted Methods Per Class, Depth of Inheritance Tree etc. have been explained (in subsection 2.2.2).

2.2.1 Source Code Metrics

Source code metrics assist to measure maintainability property of a software project. These metrics are directly calculated from the source of the project. Two commonly used source code metrics like Line of Code and McCabeś Cyclomatic Complexity are discusses as follows.

Line of Code (LOC) is the most commonly used source code metric which provides an initial idea about the size of the software. Fenton and Pfleeger figured out that some code lines are different like comments, blank lines etc. and they emphasized on the exact definition of LOC in terms of software maintenance [14]. Referring to the work by Grady and Caswell [15], Fenton and Pfleeged provided the most widely accepted definition of LOC. According to that definition, Line of Code is statements in the program except comments and blank lines. Sometimes this is said as Non-Commented Line of Code (NCLOC).

McCabes Cyclomatic Complexity was first introduced by Thomas McCabe which measures the number of independent execution paths in a computer program [16]. The formula for the calculation of cyclomatic complexity is shown as follows.

$$\begin{aligned} V(G) &= e - n + 2 \\ &\text{Where} \\ V(G) &= \text{cyclomatic complexity of the graph} G \\ E &= \text{number of edges} \\ N &= \text{number of nodes} \end{aligned} \tag{2.1}$$

2.2.2 Object Oriented Metrics

The main idea behind object oriented metrics is to calculate different object oriented maintainability properties like inter-class relationship, dependence among modules, polymorphism, intra-class relationship etc. Different popular object oriented metrics are described below.

Coupling and Cohesion in OOP were first presented by Stevens, Myers, and Constantine [17]. Coupling indicates the dependence between classes and/or objects. Chidamber and Kemerer proposed a coupling metric called Coupling between Object Classes (CBO) [18, 19] and they define it as follows: CBO for a class is a count of the number of other classes to which it is coupled On the other hand, cohesion describes the dependence among the attributes and methods inside a class. The best type of cohesion according to McConnell [20] is functional cohesion which means that a function or method performs one and only one task.

Chidamber and Kemerer proposed a metric suite for object-oriented design in the early nineties [18, 19]. The metric suit comprises five different metrics which are Weighted Methods Per Class, Depth of Inheritance Tree, Number of Children, Coupling Between Object Classes, Response for a Class, and Lack of Cohesion Methods. These metrics assist in measuring maintainability property of software developed based on Object Oriented Programming. In addition, this metric suit provides a clear idea about software architecture and helps to design the software in a better way.

Both source code metrics and object oriented metrics help to detect smells in the code by providing different information of the code like size of the code, inter-class dependency, relationship among classes etc. Smell identification involves finding characteristics of the corresponding smell in the code which are measured using those metrics. In this section, various metrics are explained which are widely used for smell detection in source code.

Chapter 3

Literature Review of Automatic Dead Field Identification

Dead field is one of the common test smells which reduces the quality of test code. The presence of dead field in the test code indicates incomplete or deprecated software development activities. This smell is a recent contribution in the literature. Several researches have been carried out so far for analyzing the impact of test smells in the test code. In addition, researchers proposed different techniques to identify and remove those smells from the code such static code analysis and dynamic code analysis. Some significant works related to this area are outlined as follows.

3.1 Evolution of Test Smell

The concept of code smell was first introduced by Fowler and Beck [2]. They have identified 22 different code smells like duplicate code, primitive obsession, switch statements, long method, long parameter list, god class, large class, and so on. These smells are commonly found in the code and such presence indicates the poor design of software, inexperience in coding and the risks of bugs or system failure in future. They have also proposed different refactoring techniques for removing these smells from the code such as method extraction, class extraction, pulling up attributes, removing duplicate code, dead code deletion and so on.

van Deursen et al. first described the concept of test smells [12, 21]. They defined test smells as trouble in the test code which cause reduction of test code maintainability in the long run. They identified a list of eleven different test smells such as Mystery Guest, Resource Optimism, Test Run War, General Fixture, Eager Test, Lazy Test, Assertion Roulette, Indirect Testing, For Testers Only, Sensitive Equality, and Test Code Duplication. They discussed about the characteristics of the smells and appropriate refactoring mechanism to remove those, but they did not provide any technique for automatically identifying dead field in the test code because this smell was not discovered at that time.

3.2 Impact and Distribution of Test Smells in Test Code

In order to understand the distribution of unit test smells and the impact of those smells on software maintenance, Gabriele Bavota et al. conducted an empirical analysis regarding this [9]. Two studies were carried out for the analysis where one was an exploratory study and another was a controlled experiment. The exploratory study was performed for the analysis of the distribution of test smells. On the other hand, the controlled experiment was carried out for analyzing the impact of test smells on the comprehension of test code during software maintenance. The exploratory study was conducted on 18 software systems where two of these were industrial and others were open source systems. In the study, they discovered that test smells were widely spread throughout the all systems. These smelly systems were given to twenty master students for the controlled experiment. During the evaluation of the study, they noticed that smells in these systems created problem among the students in comprehension of the code. In addition, they were facing difficulties while maintaining these systems. Although the authors of the paper provided an insight about the distribution and impact of test smells while managing test code, they did not provide any approach to automatically detect dead fields in the code. The reason is that they only analyzed the impact and distribution rather than detection of test smells.

3.3 Dynamic Code Analysis Technique to Detect Test Smells

A dynamic code analysis technique was proposed by Stefan Reichhart et al.[6] to detect two test smells like under-the-carpet failing assertion and badly used fixtures. A tool named TestLint was also developed to detect these smells automatically. In order to locate under-the-carpet test smell, the tool finds hidden failures by removing comments put around valid code and running the test code. For badly used fixtures test smell detection, the tool properly checks instrumentation of all test methods with the usage of instance variables and method in the test code. The tool could detect these smells by dynamically analyzing test code but it could not identify dead field in the code as no rule was defined and incorporated with the tool for the detection of this smell.

3.4 Static Code Analysis Techniques to Detect Test Smells

A metrics-based approach was proposed by Bart van Rompaey et al. [7, 22] for the detection of two test smells which were test fixture and eager test to increase the quality of test cases. To identify test fixture, they used several metrics like setup size, fixture size and fixture usage. Setup size is the combination of the number of method or attribute references to non-test object from the setup method of a test case and number of production type used in the test code. They also defined fixture size as number of fixture elements and production type in the fixture. For eager test identification, they used production type method invocation as metric which is the number of invocations to the methods in the production code from a test command. The proposed approach was tested using a UML modeling tool called ArgoUML[1] and the outcome of the technique is compared to the result of the manual inspection. The technique worked well in identifying test fixture and eager test smell. However, the metrics that were used to identify those smells are not adequate enough to detect dead field in the test code as its characteristics are different from those smells.

Stefan Reichhart et al. developed a tool named TestLint for assessing the quality of test code and finding test smells in the code[6]. This rule-based tool could identify static test smells such as Guarded Test, OverReferencing, Assertionless Test, Long Test, Overcommented Test and so on by parsing the source code, analyzing the source tree, detecting patterns and computing metrics on the test code. All the rules used to develop the tool were the characteristics of those smells [3, 11, 23] such as for assertionless test the tool checks whether a test case contains at least one valid assertion, the rule for detecting guarded test is detecting test cases which implemented conditional branches, anonymous test is detected by analyzing the test method signature and so on. However, the tool can not identify dead field in the test code because no metric was defined for the identification of this smell.

Manuel Breugelmans and Bart Van Rompaey presented a tool called TestQ[2] for exploring structural and maintenance characteristics of unit test suites [8]. It allows developers to visually explore test suites and quantify test smelliness.

[1]http://argouml.tigris.org/

[2]http://tsmells.googlecode.com/

Visualization facilities integrated in the tool assist developers to identify relevant test cases for further exploration of the test code and observe the structure of the test suites from birds eye view to in depth. The tool could identify twelve different test smells proposed by van Deursen [7]. For the detection, the tool uses a list of metrics defined by the authors such as number of invoked framework asserts for Assertionless, number of invoked description-less asserts for AssertionRoulette, number of invoked production methods for EagerTest, number of invocation and accesses for EmptyTest, invocation of production entities only in test code for ForTestersOnly, invocation of a standard set of I/O entities for MysteryGuest and so on. User can customize threshold values of the metrics that best fit for the exhibition of test smells in a particular context. However, the tool can not detect dead field in the test code because the authors did not define any metric or strategy for it.

A static analysis technique to identify test fixture related smells in the test code was presented by Michalela Greiler et al. [5]. Here they introduced five new test smells which are Test Maverick, Dead Fields, Lack of Cohesion of Test Methods, Obsecure In-Line Setup, Vague Header setup. According to their opinion, test maverick smell occurs when a test class has an implicit setup but it has at least one test method which is completely independent from the implicit setup procedures. They defined dead fields as setup fields that are never used by any test method in the test code. The smell lack of cohesion of test methods occurs if test methods are grouped together in one test class, but they are not cohesive. They considered inline setup as obscure inline setup if it contains too much setup functionality. Besides defining the characteristics of those smells, they developed a tool named TestHound in order to identify those smells. It takes the test code, all dependencies and an XML file of all test cases as input. After that, it analyzes the code, finds the smells and provides a report describing all identified test smells in the code. For the detection of test smells in test code, some metrics were proposed by the authors such as number of variables declared in a test method is greater than or equal 10 for obscure inline setup identification, at least one setup field that is never been used by any test method for dead field, and so on. The tool was assessed by running on three projects (eGit, HealthCare and Mylyn[3]) and it worked well in identifying those smells. However, it produced false positive results while detecting dead fields due to not being able to resolve field dependency and find usage of setup fields in the test code (for example 3 percent of the fields could not be mapped properly to field usage in eGit [5]). So, manual inspection was carried out to identify dead fields in the test code correctly.

3.5 Summary

Although dead field is a recently introduced test smell in the literature, some significant works have been performed in identification of test smells so far. Researchers explained the impact of test smells in test code maintenance and proposed different techniques to detect test smells like metrics based approach, rule based assessment, test fixture analysis and so on. Some of those could identify dead fields in the test code but the outcome is not accurate enough. Sometimes it is seen that those techniques provides false positive result which ultimately induces serious impact while managing the code. For that reason, test code is needed to be inspected manually for making sure the correctness of the result in dead field detection. So, automatically identifying dead fields in the code properly is still a problem in the literature.

[3]http://www.eclipse.org/mylyn/

Chapter 4

DFI: A Technique to Detect Dead Field Automatically

The intent of this research is to develop a technique named Dead Field Identifier (DFI) to detect dead fields in the test code for making the code more maintainable and comprehensible by removing those fields. For the identification, firstly, it is required to identify all the invoked methods for any method in the test code. In addition, all the setup fields are required to be obtained and usage of those fields are needed to be identified in the code which assist to detect dead fields automatically. So the technique for the identification comprises several steps like invoked method identification, setup field detection, finding usage of setup fields and dead field identification which are described in the following sections.

4.1 Invoked Method Identification

Usually, the first step to identify dead fields in the test code is to identify all the methods invoked directly or indirectly by any method in the code. This is required because fields in the test class may be initialized by any method invoked directly or indirectly by the setup methods. Even setup field(s) may not be used directly by a test method but may be used by other methods which are invoked by the test method directly or indirectly

In Algorithm 1, the procedure *GetAllInvokedMethod* takes a method as input and returns a list of all methods invoked directly or indirectly by that method. For this, first of all, a list is initialized to store all invoked methods and the body of the inputted method is parsed to identify all the methods invoked by it which are inserted into another list (Algorithm 1 Line 2-4). A loop is used to identify all the invoked methods for each method in the list by recursively calling *GetAllInvokedMethod*. For each iteration, corresponding method is also added into the list which is responsible for containing all invoked methods (Algorithm 1 Line 5-8).

Algorithm 1 Invoked Method Identification

```
Input: A method ($M$) for which all the methods invoked directly or indirectly
    by it will be identified
 1: procedure GetAllInvokedMethod($M$)
 2:     initialize an empty list $L$ to store invoked methods
 3:     add $M$ into $L$
 4:     get all invoked methods by parsing the method ($M$) body and add those
        into a list $N$
 5:     for each $m \in M$ do
 6:         $A \leftarrow$ GetAllInvokedMethod($m$)
 7:         Insert all items in $A$ into $L$
 8:     end for
 9:     return $L$
10: end procedure
```

4.2 Finding Setup Fields

Setup fields in the test code are those which are initialized in the implicit setup procedures or the class header. All the setup fields in the test code are required to be identified because such setup fields are considered as dead fields which have never been used by any test method in the test code.

Algorithm 2 describes a procedure *GetAllSetUpFields* which works on given test code and provides a list of all setup fields in the code. Initially two lists are initialized - one is to store all setup fields and another is to store all the fields by parsing the test code (Algorithm 2 Line 2-3). In the loop, all the header fields are identified from the list of fields and those are added to the setup field list as header field is also considered as setup field (Algorithm 2 Line 4-8).

Algorithm 2 Finding Setup Fields

```
Input: Test code $T$ for identifying all setup fields in $T$
 1: procedure GETALLSETUPFIELDS($T$)
 2:     initialize an empty list $S$ to store setup fields
 3:     identify all the fields in $T$ using parser and store those fields in the list $F$
 4:     for each $f \in F$ do
 5:         if $f$ is header field then
 6:             Add $f$ to $S$
 7:         end if
 8:     end for
 9:     find setup method $M$ by parsing $T$
10:     create an empty list $I$ to store method
11:     $I \leftarrow$ GETALLINVOKEDMETHOD($M$)
12:     add $M$ to $I$
13:     for each $m \in I$ do
14:         for each $f \in F$ do
15:             if $f \in S$ then
16:                 continue
17:             end if
18:             if $f$ is initialized in $m$ then
19:                 add $f$ to $S$
20:             end if
21:         end for
22:     end for
23:     return $S$
24: end procedure
```

After the completion of header field detection phase, the list S contains only the header fields in the test code. In Algorithm 2 Line 9 setup method of the target test class is identified by parsing the test code T and an empty list I is created to store all methods invoked by the setup method (Algorithm 2 line 10). Later, *GetAllInvoked* method is called to obtain all the methods invoked by the setup method and those methods are stored in I (Algorithm 2 Line 11). The outer *for* loop (Algorithm 2 Line 13) iterates on each method stored in I and for each identified method, the inner *for* loop (Algorithm 2 Line 14) checks two cases for each field in F - one is whether the field is already in the setup field list or not (Algorithm 2 Line 15), and another is whether it is initialized in this method or not (Algorithm 2 Line 18). If any such field is found that is initialized in the corresponding method, this field will be added to the list of setup fields. At last, the list of all identified setup fields is returned in Algorithm 2 Line 23.

4.3 Finding Usage of Setup Fields

After identifying all setup fields following the previous step, usage of all the setup fields are required to be found in the test code to identify all the used setup fields by each test method. Such identification will help to detect which setup fields are never been used by any test method in the test code.

Algorithm 3 Finding Usage of Setup Fields

```
Input: Test code $T$ for finding usage of setup fields in the test code
 1: procedure GetAllUsedSetUpField($T$)
 2:     initialize an empty list $U$ to store all used setup fields in $T$
 3:     initialize an empty list $M$ to store all test methods in $T$
 4:     identify all test methods by parsing $T$ and add those into $M$
 5:     $S \leftarrow$ GetAllSetUpFields($T$)
 6:     for each $m \in M$ do
 7:         $L \leftarrow$ GetAllInvokedMethod($m$)
 8:         add $m$ to $L$
 9:         for each $i \in L$ do
10:             Get the body of the method ($i$) and save it in $b$
11:             for each $f \in S$ do
12:                 if $f$ is used in $b$ and $f \notin U$ then
13:                     add $f$ to $U$
14:                 end if
15:             end for
16:         end for
17:     end for
18:     return U
19: end procedure
```

In Algorithm 3, all the test methods and all the setup fields in the test code are identified and stored in two different lists respectively (Algorithm 3 Line 3-5). For each identified test method, the procedure *GetAllInvokedMethod* is called to obtain all the methods invoked directly and indirectly by the method (Algorithm 3 Line 6-8). After that, the body of each invoked method and the test method are checked to identify which setup fields are used in the body and such fields are added to the used setup field list (Algorithm 3 Line 9-16). At last, the list of all used setup fields are returned by the procedure *GetAllUsedSetField* (Algorithm 3 Line 18).

4.4 Dead Field Detection

Section 4.3 provides all the setup fields that are used by at least one test method directly or indirectly. However, such setup fields can be found in the test code, which are not being used by any test method but some used setup fields may depend on those fields for initialization. So, those fields are not considered as dead fields. For finding all those fields, incorporating those with the list of fields obtained using section 4.3 and finally providing a list of all identified dead fields in the test code, Algorithm 4 is used for implementation.

To detect dead fields all setup fields and all the used setup fields are gathered (Algorithm 4 Line 2-3). A list is used to store all the setup fields which are not used by any test method (Algorithm 4 Line 4). The nested loops identify which setup fields of the list are never used for the initialization of any used setup field (Algorithm 4 Line 6-16). Here, the *for* loop defined in Algorithm 4 Line 6 iterates over the entire directly unused setup field list F. For each field f in F another loop (Algorithm 4 Line 8) is used to check whether any used setup field depends on f for its initialization or not. If it is found that f is not been used for the instantiation of at least one used setup field, f will be marked as Dead Field

Algorithm 4 Dead Fields Detection

```
Input: Test code $T$ to identify dead fields in the code
 1: procedure GetAllDeadField($T$)
 2:     $S \leftarrow$ GetAllSetUpFields($T$)
 3:     $U \leftarrow$ GetAllUsedSetUpField($T$)
 4:     $F \leftarrow S - U$
 5:     initialize a list $D$ to store dead fields
 6:     for each $f \in F$ do
 7:         $flag \leftarrow false$
 8:         for each $i \in U$ do
 9:             if $i$ depends on $f$ for initialization in the implicit setup then
10:                 $flag \leftarrow true$
11:             end if
12:         end for
13:         if $flag = false$ then
14:             add $f$ to $D$
15:         end if
16:     end for
17:     return D
18: end procedure
```

(Algorithm 4 Line 13). This field is then added to the list D which is used for storing all identified Dead fields. Lastly, the list of Dead Fields, D is returned in Algorithm 4 Line 17.

4.5 Complexity Analysis

Complexity analysis for each of the proposed algorithms stated above are described as follows.

Complexity Analysis for Invoked Method Identification

The overall complexity of this algorithm is $O(m)$ where m is the total number of invoked methods. However, the actual performance of the algorithm depends on the parsing technique of test code.

Complexity Analysis for Finding Setup Fields

This algorithm runs in $O(mn)$, where m is the number of invoked methods and n is number of fields in test code. Again the performance of the algorithm is influenced by the test code parsing algorithm's complexity.

Complexity Analysis for Finding Usage of Setup Fields

The complexity of Algorithm 3 is $O(pqr)$. Here,

p = number of test methods

q = number of invoked methods

r = number of setup fields

However, this algorithm depends on the parsing algorithm of test code and thus, parsing time needs to be considered to calculate its exact performance.

Complexity Analysis for Dead Field Detection

In this algorithm, the complexity of $GetAllSetUpFields$ and $GetAllUsedSetUpField$ are $O(pq)$ and $O(prs)$ respectively, where

p = number of invoked methods

q = number of fields in the test code

r = number of test methods

s = number of setup fields

m = number of unused setup fields

n = number of used setup fields

So, the complexity of the algorithm is $O(pq + prs + mn)$.

4.6 Summary

The proposed technique named DFI identifies dead fields in the test code by using four devised algorithms which are discussed in this chapter. The technique first identifies all the setup fields and header fields in the test code. Later, it finds the usage of each setup fields in the test methods and detects unused setup fields which are considered as dead fields. This chapter explains all the steps of DFI in dead field detection along with the complexity of the proposed technique.

Chapter 5

Implementation and Result Analysis

This chapter focuses on the evaluation of the approach in terms of accuracy in dead field detection. In the previous chapter, four algorithms have been devised in order to describe DFI where those algorithms involve invoked method identification, setup field detection, usage of setup field discovery and dead field recognition. For the evaluation, the technique has been implemented in Java programming language and two open source projects in different size have been used as experimental dataset. Besides, another tool named TestHound [5] is used for comparative analysis with DFI. At last, manual inspection is carried out to make sure the correctness of the result provided by DFI. In this chapter, a brief explanation regarding the implementation environment and dataset information for the experiment are provided and a comparative analysis is also explained in details.

5.1 Experimental Setup

The environmental setup and experimental data sets which are used for the experiment are described as follows.The environmental setup and experimental data sets which are used for the experiment are described as follows.

5.1.1 Environmental Setup

This section outlines the software tools required for the experimental analysis. For this analysis, DFI is developed using Java programming language. Although the tool works to identify dead fields in the test code written using Java, the approach proposed here is platform independent and only the facts extraction aspect is language specific. So, the technique can easily be implemented in any programming language. Some other tools are also used in the experiment and those are addressed as follows.

- **Juno**[1] An open source Integrated Development Environment (IDE) that facilitates developing software in Java programming language. The technique DFI is implemented using this IDE. Writing, building, managing, and running source code of DFI are done with the assistance of it.

- **Byte parser**[2]**:** It is an open source java library for parsing java byte code. In the experiment, this library is used to parse byte code of the experimental datasets. Different features of these datasets like method signature, class definition, attributes and methods of a class, invoked methodsśignature, and method body etc. are extracted using the library. It takes the byte code of a java project including all dependencies as input, parses the code by performing plain text search and provides information about those features as output.

- **Maven**[3]**:** Source code of datasets is build using Maven Apache build manager to produce byte code. These byte codes are used for test code feature

[1]https://eclipse.org/juno/
[2]https://github.com/rifatbit0401/ByteParser
[3]https://maven.apache.org/

extraction using Byte parser. Version apacahe maven 5.0.1 is used in the experiment for such conversion.

System Configuration

The experiment is performed in a single machine where the same datasets are run using TestHound and DFI. The configuration of that machine is outlined below.

- **Processor:** Intel(R) Core(TM) i5 -2430M CPU @2.40GHz
- **RAM:** 4GB
- **Operating System:** Windows 7 Ultimate
- **System Type:** 32-bit Operating System

5.1.2 Experimental Datasets

In order to evaluate the accuracy of DFI, two open source projects (depicted in Table 5.1) like EuqationSolverTest and eGit have been used as test beds in the experiment. These projects are different in sizes where EuqationSolverTest is comparatively smaller than eGit. The reason behind choosing these types of Datasets is to observe the correctness and behavior of DFI while detecting dead fields in different sized projects. Detailed information about these test beds are presented as follows.

EquationSolverTest[4]: A Java console based application which can solve different equations having various expressions like x+y, (x+y)/z, (x/2)+y etc.. It takes a single line as input where the line represents the expression of that equation to be solved and provides the result of the equation as output. This open source project has 800 lines of code and two packages. There are four test classes found in the project which contain 15 unit test cases. A Java unit testing framework

[4]https://github.com/rifatbit0401/EquationSolverTest

Table 5.1: Dataset Information for Dead Field Detection

Project Name	**Line of Code**	**Number of Test Class**
EquationSolverTest	800	4
eGit	130k	85

named JUnit have been used to write all these test cases.

eGit[5]: A popular eclipse plugin for managing source code of a project. This open source eclipse integrated version control system has been developed in Java programming language. It helps to perform various version control and source code management operations such as create repository, push code, pull code, commit changes in source code, and so on. The project consists of 130K lines of code and 12 test packages. It contains 85 test classes with an average of 10 test methods per class. All the test cases have written in JUnit version 4 testing framework.

Two different sized projects are used for the experiment to observe the behavior of the proposed technique. One of those is eGit which is large in size, and another is EquationSolverTest which is comparatively small. In the experiment firstly these test beds are run using DFI. After that TestHound (TH) is used for identifying dead fields in these projects. At last Manual Inspection (MI) is carried out to obtain actual dead fields and setup fields in the test code of the projects. Detailed result analysis for the projects is discussed as follows.

Result Analysis for EquationSolverTest: For comparative analysis initially the project EquationSolverTest is run by TestHound. The project is also analyzed by DFI. In addition, manual inspection is also performed on the code. Table 5.2 summarizes the result produced by the tools and manual inspection. In the table, it is seen that there are four test classes. Comparative analysis for those classes are described below.

[5]http://www.eclipse.org/egit/

Table 5.2: Comparative Result Analysis for EquationSolverTest

Class Name	No. of Setup fields			No. of Dead Fields		
	TH	DFI	MI	TH	DFI	MI
SimulateEquationTest	0	5	5	0	4	4
ExpressionFormatterTest	1	4	4	1	3	3
ExpressionSimulationResultTest	2	7	7	2	2	2
OperationTest	0	1	1	0	0	0

For the test class SimulateEquationTest, testhound could not identify any setup field whereas DFI detects 5 setup fields as well as 4 dead fields from those. The outcome of DFI is equal to the result of Manual Inspection. The reason is that TestHound can not identify those setup fields which are initialized in the methods invoked by setup method but DFI considers all those methods and checks the initialization of setup fields.

In the test class ExpressionFormatterTest, there are 4 setup fields. Among those one is header field and others are initialized through indirect methods invocation by the setup method. Testhound detects the header field and considers it as dead field due to not being used by any test method but others are not taken into account because of the same reason as stated earlier. On the other hand, DFI identifies all those and recognizes as dead fields.

Both tools identify two dead fields correctly for the test class ExpressionSimulationResultTest. However, TestHound identifies 2 setup fields out of 7 because those two are header fields and rest 5 are initialized in the setup method which are not considered in it. On the other hand, DFI checks the setup method as well as header field, that is why it detects all setup fields.

There is a single header field in test class OperationTest and this field is used in all 4 test cases. As both tools can detect header fields and usage of setup fields in test cases so those tools provide the same result for the test class.

Result Analysis for eGit:

DFI and TH both are run on 10 modules of eGit. Besides MI is also performed on all the test classes of these modules. Detailed results for each module along with comparative analysis are explained as follows.

Comparative result analysis for org.eclipse.egit.core.test.op: According to the Table 5.3, there are 19 test classes in this package where 110 setup fields and 17 dead fields are identified by DFI. On the other hand, TH detects 97 setup fields and 8 dead fields in this package. Here, most of the test classes use the test fixture of GitTestCase but this class contains a header fields named testUtils which is not used by any test case of most the test classes extending it. As DFI detects all the header fields of super class and considers these as setup fields so the outcome of DFI is similar to MI. However, due to not considering header fields in the super class as setup fields, TH could not detect all the dead fields in the test code.

Comparative result analysis for org.eclipse.egit.core.test.rebase: This package contains a single test class named RebaseInteractivePlanTest which extends another class named GitTestCase. DFI and TH both could identify all the setup fields in RebaseInteractivePlanTest class as shown in Table 5.4. However, there is a header field named testUtils which is not considered as setup field by TH and thus it could not detect this dead field. As DFI takes all the header fields declared in both parent class and child class, so it finds one dead field and 7 setup fields which is equal to the result obtained by MI.

Comparative result analysis for org.eclipse.egit.core.test.internal.mapping: There is a single test class in this packaged which has a parent class named GitTestCase. In Table 5.5, the number of dead fields identified by DFI and TH is same. However, there is a single value difference between the number of setup fields identified by DFI and TH. The reason behind this is that DFI takes the whole test fixture of the super class whereas TH ignores initialized header fields in the super

Table 5.3: Comparative Result Analysis for org.eclipse.egit.core.test.op

Class Name	No. of Setup fields			No. of Dead Fields		
	TH	DFI	MI	TH	DFI	MI
AddOperationTest	5	5	5	0	0	0
BranchOperationTest	4	5	5	0	1	1
CloneOperationTest	4	4	4	0	0	0
CommitOperationTest	5	7	7	0	0	0
ConnectProviderOperationTest	2	3	3	0	1	1
CreatePatchOperationTest	5	6	6	0	1	1
DiscardChangesOperationTest	7	7	7	1	1	1
EditCommitOperationTest	5	6	6	0	1	1
ListRemoteOperationTest	6	6	6	0	0	0
MergeOperationTest	4	5	5	0	1	1
PushOperationTest	6	6	6	0	0	0
RebaseOperationTest	5	6	6	0	1	1
RemoveFromIndexOperationTest	6	7	7	2	2	2
ResetOperationTest	4	5	5	0	1	1
RewordCommitsOperationTest	4	5	5	0	1	1
SquashCommitsOperationTest	6	7	7	0	1	1
StashCreateOperationTest	4	5	5	0	0	0
TagOperationTest	5	5	5	2	2	2
TrackUntrackOperationTest	5	5	5	0	0	0

Table 5.4: Comparative Result Analysis for org.eclipse.egit.core.test.rebase

Class Name	No. of Setup fields			No. of Dead Fields		
	TH	DFI	MI	TH	DFI	MI
RebaseInteractivePlanTest	6	7	7	0	1	1

class.

Comparative result analysis for org.eclipse.egit.core.test: In Table 5.6,

Table 5.5: Comparative Result Analysis for org.eclipse.egit.core.test.internal.mapping

Class Name	No. of Setup fields			No. of Dead Fields		
	TH	DFI	MI	TH	DFI	MI
HistoryTest	6	7	7	5	5	5

comparative result for 13 test classes of this package is depicted where 53 setup fields are identified by TH and 64 setup fields as well as 9 dead fields are detected by DFI. The reason behind this difference is due to not considering header fields in super class issue as explained earlier. Besides, there are some classes like Eclipse-GitProgressTransformerTest, LinkedResourcesTest and these classes do not use super class fixture. For the EclipseGitProgressTransformerTest class, the results provided by TH, DFI and MI are the same but for LinkedResourcesTest, 4 dead fields are identified by DFI where TH could not find any dead fields. Here the reason is that DFI identifies all initialized fields in that class but TH ignores such field initialization in the class.

Comparative result analysis for org.eclipse.egit.core.synchronize.dto and org.eclipse.egit.core.storage: Results for org.eclipse.egit.core.synchronize.dto and org.eclipse.egit.core.storage are presented in Table 5.7 and Table 5.8. Both packages comprise a single test class each and those classes extend the same super class GitTestCase. In GitSynchronizeDataTest, TH identifies 3 setup fields and no dead fields but actually there are 4 setup fields and among these one is dead field which are identified by DFI. Again in GitBlobStorageTest, there is a single dead field in this class which is detected by DFI but TH fails to identify this. The reason of such resemblances among results is that DFI incorporates all the header fields and parent classs setup fields and recognize those as setup fields of the fixure but TH does not consider header fields in parent class.

Comparative Result Analysis for org.eclipse.egit.core.securestorage,

Table 5.6: Comparative Result Analysis for org.eclipse.egit.core.test

Class Name	No. of Setup fields			No. of Dead Fields		
	TH	DFI	MI	TH	DFI	MI
AdaptableFileTreeIteratorTest	4	5	5	0	1	1
CommitUtilTest	6	7	7	0	1	1
ContainerTreeIteratorResourceFilterTest	3	4	4	0	0	0
EclipseGitProgressTransformerTest	2	2	2	0	0	0
FileDeleteHookTest	5	6	6	0	1	1
GitProjectSetCapabilityTest	3	3	3	0	0	0
GitURITest	1	1	1	0	0	0
LinkedResourcesTest	11	12	12	0	4	4
ProjectReferenceTest	0	4	4	0	1	1
RepositoryCacheTest	5	6	6	0	0	0
RevUtilsTest	4	5	5	0	1	1
SubmoduleAndContainerTreeIteratorTest	9	9	9	0	0	0
UtilsTest	0	0	0	0	0	0

Table 5.7: Comparative Result Analysis for org.eclipse.egit.core.synchronize.dto

Class Name	No. of Setup fields			No. of Dead Fields		
	TH	DFI	MI	TH	DFI	MI
GitSynchronizeDataTest	3	4	4	0	1	1

Table 5.8: Comparative Result Analysis for org.eclipse.egit.core.storage

Class Name	No. of Setup fields			No. of Dead Fields		
	TH	DFI	MI	TH	DFI	MI
GitBlobStorageTest	4	4	4	0	1	1

org.eclipse.egit.core.internal.indexdiff, and org.eclipse.egit.core: Comparative results for these packages are shown in Table 5.9, Table 5.10, and Table 5.11 respectively where setup fields and dead fields detected by TH, DFI and MI are same. Each test class in these packages has its own fixture defined within the class that is there is no fixture dependency with any super class. Besides setup methods of these classes do not invoke any other method which indicates that all setup fields are initialized in the setup methods. As TH and DFI both could identify header fields and setup fields that initialized directly by setup method, so the results provided by these approaches are the same.

Table 5.9: Comparative Result Analysis for org.eclipse.egit.core.securestorage

Class Name	No. of Setup fields			No. of Dead Fields		
	TH	DFI	MI	TH	DFI	MI
EGitSecureStoreTest	2	2	2	0	0	0

Table 5.10: Comparative Result Analysis for org.eclipse.egit.core.internal.indexdiff

Class Name	No. of Setup fields			No. of Dead Fields		
	TH	DFI	MI	TH	DFI	MI
IndexDiffCacheTest	5	5	5	1	1	1
IndexDiffDataTest	0	0	0	0	0	0

Table 5.11: Comparative Result Analysis for org.eclipse.egit.core

Class Name	No. of Setup fields			No. of Dead Fields		
	TH	DFI	MI	TH	DFI	MI
GitMoveDeleteHookTest	4	4	4	0	0	0

Comparative result analysis for org.eclipse.egit.core.synchronize: In Table 5.12 it is seen that there are three test classes in this package which are GitResourceVariantTreeSubscriberTest, GitResourceVariantTreeTest, and ThreeWayDiffEntryTest. First two test classes extend GitTestCase class and the last one inherits LocalDiskRepositoryTestCase. Here the difference in the number of dead fields and setup fields identified by DFI and TH is due to not considering super classs header fields issue as explained earlier. However, DFI keeps this issue under consideration, identifies all setup fields and detects dead fields by analyzing usage of those fields.

Table 5.12: Comparative Result Analysis for org.eclipse.egit.core.synchronize

Class Name	No. of Setup fields			No. of Dead Fields		
	TH	DFI	MI	TH	DFI	MI
GitResourceVariantTreeSubscriberTest	7	8	8	0	1	1
GitResourceVariantTreeTest	2	3	3	0	1	1
ThreeWayDiffEntryTest	6	8	8	5	7	7

5.2 Summary

DFI and TestHound, both can identify dead fields in the test code. However, TestHound could not detect dead fields correctly due to not hadling some cases properly like setup fields intialization in a method invoked by setup method, field dependency among setup fields, and usage of setup fields by test methods indirectly. On the other hand, DFI can appropriately deal with those and as a result it detects dead fields correctly in the test code.

Chapter 6

Discussion and Conclusion

The presence of dead fields in the test code reduces the manageability and comprehensibility of the code. In this research, an automatic dead field detection technique named DFI is proposed which finds dead fields by identifying setup method and invoked methods, detecting all setup fields, resolving field dependency and finding usage of these setup fields. While performing experiment and comparative result analysis, it is seen that DFI performs 49% better than existing technique like TestHound [5] and produces similar result to Manual Inspection (MI). This chapter describes DFI in essence with its achievement in dead field detection. Moreover, several threats to validity and future direction of this work are also discussed in this chapter.

6.1 DFI: The proposed dead field detection technique

DFI comprises four algorithms which are proposed in this work and these algorithms are *Invoked Method Identification*, *Finding Setup Fields*, *Finding Usage of Setup Fields*, and *Dead Fields Detection*. The algorithm *Invoked Method Identification* helps to find all the methods invoked by a test method or setup method directly and indirectly in the test code. The second proposed algorithm *Finding Setup Fields* figures out all the setup fields and header fields in both child test class and parent test class if inheritance is present in the test code. Algorithm *Finding Usage of Setup Fields* detects the usage of each setup field in test methods. Algorithm *Dead Fields Detection* performs field dependency resolution among setup fields and dead field separation from the setup fields.

By incorporating these four algorithms, DFI first identifies all the fields in the test code. Setup fields are identified from those by analyzing the initialization of those fields in the setup method and its invoked methods. At last, usage of those fields in test methods and dependency relationship among those fields are resolved to figure out dead fields in the code.

6.2 Discussion of The Results

In order to check the accuracy of DFI, an experiment is performed where two open source projects (EquationSolverTest and eGit) are used. For the experimentation, DFI and an existing technique TestHound are implemented in Java programming language. Later, both techniques are run on the test beds and Manual Inspection (MI) is also performed on these projects.

In the comparative analysis, it is seen that DFI identifies 580 setup fields in eGit and 17 setup fields in EquationSolverTest. This technique also detects 103 dead fields in eGit and 9 dead fields in EquationSolver test. On the other hand, TestHound detects 510 setup fields and 52 dead fields in eGit, and 3 setup fields as well as 3 dead fields in EquationSolverTest. Here, DFI detects 12% more setup fields and 50.89% more dead fields than TestHound in eGit due to identifying all setup fields and header fields in super class, and resolving field dependency among setup fields. For EquationSolverTest, DFI identifies 82% more setup fields and 66.67% more dead fields by finding usage of setup fields in test methods. All the results found by DFI are compared to MI and both techniques provide the same setup fields and dead fields.

6.3 Threats to Validity

Although DFI performs better than other existing technique like TestHound for the experimental setup described in this research, there are some notable dynamics, that may also be considered. These issues are discussed below.

- DFI is implemented in Java programming language only but the behavior of this technique is not confirmed if it is developed in other platforms or languages like C#, Visual Basic, C++, and so on.
- DFI provides better results than existing technique for eGit and Equation-SolverTest. However, if the test beds are changed or new projects are introduced for the experiment, the validity of DFI may have been changed.
- In this research, Manual Inspection (MI) involves scrutinizing test codes manually. There is no standard process to check the validity of those obtained result. However, several times the results are checked in this research.
- DFI depends on third party source code parsing library in order to extract source code features. In this research, ByteParser[1] is used for obtaining different test code features like method signature, method body, fields of a class and so on. However, the accuracy of this parser is beyond the scope this research.

[1]https://github.com/rifatbit0401/ByteParser

6.4 Future work

This research contributes in the literature by devising a technique named DFI which paves the way for more improvement and extension. This technique can be incorporated with other existing test smell detection tools or frameworks like TestLint, TestHound etc. to identify dead fields along with other test smells in the test code. The scope of DFI is to identify dead fields in the test code automatically but this technique can be further extended to support automatic dead field deletion as refactoring mechanism of this smell.

In this research, DFI is implemented in Java programming language which works on only those test code written in Java. However, the technique is language independent and thus, it can be developed in other platforms or languages like C#, C++, VB and so on in order to support dead field detection in these languages. The technique can also be associated with different automatic unit test case generators like SSTF, AutoTest and so on. Such integration will help to generate dead fields free test scripts and these scripts will be more maintainable and comprehensible.

For the evaluation of DFI with respect to other dead field detection technique, only two open source projects are used. However, it could be better if the experiment is performed on some industrial projects as well. So there is a plan to add some industrial projects as well as more open source projects to the test beds and observe the behavior of DFI.

Bibliography

[1] M. Fowler, *Refactoring: improving the design of existing code.* Pearson Education India, 1999.

[2] M. Fowler, "Refactoring: Improving the design of existing code," in *Proceedings of the 11th European Conference on Object-Oriented Programming. Jyväskylä, Finland*, 1997.

[3] A. van Deursen, L. Moonen, A. van den Bergh, and G. Kok, *Refactoring test code.* CWI, 2001.

[4] A. Van Deursen and L. Moonen, "The video store revisited–thoughts on refactoring and testing," in *Proceedings of the 3rd International Conference on eXtreme Programming and Flexible Processes in Software Engineering*, pp. 71–76, Citeseer, 2002.

[5] M. Greiler, A. van Deursen, and M.-A. Storey, "Automated detection of test fixture strategies and smells," in *Proceedings of the Sixth International Conference on Software Testing, Verification and Validation (ICST), 2013 IEEE*, pp. 322–331, IEEE, 2013.

[6] S. Reichhart, T. Gîrba, and S. Ducasse, "Rule-based assessment of test quality.," *Journal of Object Technology*, vol. 6, no. 9, pp. 231–251, 2007.

[7] B. Van Rompaey, B. Du Bois, S. Demeyer, and M. Rieger, "On the detection of test smells: A metrics-based approach for general fixture and eager test," *Software Engineering, IEEE Transactions on*, vol. 33, no. 12, pp. 800–817, 2007.

[8] M. Breugelmans and B. Van Rompaey, "Testq: Exploring structural and maintenance characteristics of unit test suites," in *WASDeTT-1: 1st International Workshop on Advanced Software Development Tools and Techniques*, 2008.

[9] G. Bavota, A. Qusef, R. Oliveto, A. De Lucia, and D. Binkley, "An empirical analysis of the distribution of unit test smells and their impact on software maintenance," in *Proceedings of the 28th IEEE International Conference on Software Maintenance (ICSM), 2012*, pp. 56–65, IEEE, 2012.

[10] J. Kerievsky, *Refactoring to patterns.* Pearson Deutschland GmbH, 2005.

[11] G. Meszaros, *xUnit test patterns: Refactoring test code.* Pearson Education, 2007.

[12] A. v. Deursen, L. Moonen, A. v. d. Bergh, and G. Kok, "Refactoring test code," in *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes (XP2001)*, pp. 92–95, University of Cagliari, 2001.

[13] T. DeMarco, *Controlling software projects: Management, measurement, and estimates.* Prentice Hall PTR, 1986.

[14] N. Fenton and S. Pfleeger, "Software metrics-a rigorous & practical approach, international thomson computer press," tech. rep., ISBN 1-85032-275-9, 1996.

[15] R. B. Grady and D. L. Caswell, "Software metrics: establishing a company-wide program," 1987.

[16] T. J. McCabe, "A complexity measure," *Software Engineering, IEEE Transactions on*, no. 4, pp. 308–320, 1976.

[17] W. P. Stevens, G. J. Myers, and L. L. Constantine, "Structured design," *IBM Systems Journal*, vol. 13, no. 2, pp. 115–139, 1974.

[18] S. R. Chidamber and C. F. Kemerer, *Towards a metrics suite for object oriented design*, vol. 26. ACM, 1991.

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

[20] A. Thigpen, R. Silver, J. Guileyardo, M. L. Casey, J. McConnell, and D. Russell, "Tissue distribution and ontogeny of steroid 5 alpha-reductase isozyme expression.," *Journal of Clinical Investigation*, vol. 92, no. 2, p. 903, 1993.

[21] A. v. Deursen, L. Moonen, A. v. d. Bergh, and G. Kok, "Refactoring test code," in *Extreme Programming Perspectives* (G. Succi, M. Marchesi, D. Wells, and L. Williams, eds.), pp. 141–152, Addison-Wesley, 2002.

[22] B. Van Rompaey, B. Du Bois, and S. Demeyer, "Characterizing the relative significance of a test smell," in *Proceedings of the 22nd IEEE International Conference on Software Maintenance, 2006. ICSM'06.*, pp. 391–400, IEEE, 2006.

[23] G. Meszaros, S. M. Smith, and J. Andrea, "The test automation manifesto," in *Extreme Programming and Agile Methods-XP/Agile Universe 2003*, pp. 73–81, Springer, 2003.