Automatically Identifying Dead Fields in Test Code by Resolving Method Call and Field Dependency

Abdus Satter*, Nadia Nahar† and Kazi Sakib‡
Institute of Information Technology, University of Dhaka, Dhaka, Bangladesh
Email: *bit0401@iit.du.ac.bd, ‡nadia@iit.du.ac.bd, §sakib@iit.du.ac.bd

***Abstract*—Dead fields are the unused setup fields in the test code which reduce the comprehensibility and maintainability of a software system. A test class contains dead fields when developers initialize setup fields without analyzing the usage of fields properly. Manually identifying dead fields to remove from the code is a time consuming and error-prone task. In this paper, a technique named Dead Field Detector (DFD) has been proposed to detect dead fields automatically. The technique constructs Call Graph (CG) and Data Dependence Graph (DDG) from test code to find method invocation and field dependency relationships, respectively. It identifies the fields initialized in the setup method and its invoked methods from CG. It finds setup fields by collecting the initialized fields and their dependent fields from DG. To determine the usage of the setup fields, it checks the bodies of the test methods and their invoked methods obtained from CG. All the unused setup fields are separated from the used fields and considered as dead fields. In order to evaluate DFD, an open source project named eGit was used. The result analysis shows that DFD has identified 14.03% more setup fields and 60.98% more dead fields than an existing technique named TestHound for eGit.**

***Index Terms*—Test Smell, Software Maintenance, Dead Field**

I. Introduction

A dead field is an initialized setup field in the test code which has never been used by any test method [1]. A test class may contain one or more dead fields when developers declare and initialize setup fields without considering their usages in the test methods. A dead field is not a programming error or bug, and it is considered as a test smell. Like other smells, it reduces the test code maintainability and comprehensibility. It adds unnecessary code in the test class and creates misunderstanding among the developers. When a new developer starts working in a test class, she has to go through the whole code to understand the purpose of each declared field. Since dead fields have no usage, analyzing these fields induces unnecessary time and effort. Dead field also adds unnecessary code to the production class because production code is written from the test code in Test Driven Development (TDD) approach [2].

Manually scrutinizing the test code to find dead fields is a time consuming and error-prone task. However, several challenges are associated to the automatic detection of dead fields. The first challenge is to find initialized setup fields, because a setup field may be initialized by a setup method directly or indirectly through invoking other methods. The second challenge is to find the usage of setup fields since a test method may directly use a setup field, or invoke a non-test method that uses the field. For instance, a test method, m_1 invokes a method, m_2, and m_2 invokes another method m_3. A setup field, f may not be used by m_1. However, f may be used by m_3. The third challenge is to resolve field dependency to find the usage of a field. A field, f_x may not be used by any test method directly or indirectly. It may be used to initialize another setup field used by another test method.

Martin Fowler first introduced the term - code smell and showed how code smells affect software maintainability [3]. Later, van Deursen introduced the concept of test smell in test code [4]. Bavota conducted an empirical study on the impact and distribution of test smells in test code maintenance [5]. The study showed that test smells occurred quite frequently in test code, and these had a negative impact on the software maintenance. Greilar et al. proposed five new test smells including dead field, and developed a tool named TestHound to identify the smells automatically [1]. However, the tool cannot identify setup fields correctly due to not resolving field dependency among the fields and method call dependency in a test class. Stefan et al. proposed a rule-based tool named TestLint to find test smells automatically [6]. The tool cannot detect dead field because no rule was defined for the identification. van Rompaey proposed a metric-based approach to identify eager test smell, but the author did not provide any metric to detect dead field automatically [7], [8].

In this paper, a technique named Dead Field Detector (DFD) has been proposed to find dead fields in the test code. Usually, a setup field may be initialized by the setup method or its invoked methods. So, DFD constructs a call graph to define the caller-calling relationships among the methods and identifies all the methods invoked by the setup method. A setup field may depend on other fields for its initialization. To identify those fields and resolve field dependency, a data dependency graph is generated. In order to identify setup fields, DFD parses the test code and finds the setup method. Later, it obtains all the methods invoked by the setup method from the call graph and parses their bodies to detect initialized fields. The data dependency graph is traversed to find the fields on which the identified setup fields are dependent. Since header fields are those fields that are initialized directly without invoking the setup method, these fields are also marked as setup fields. To identify the usage of the setup fields, the technique parses all test methods. A test method may invoke another method, so all the invoked methods of the test methods are also obtained from the call graph. Next, the body of each

identified method is checked to find which setup fields are used by that method. Those fields along with their dependent fields obtained from the data dependence graph are marked as used setup fields. At last, all the unused fields are separated from the setup field list and those are considered as dead fields.

In order to check the accuracy of DFD, an experimental analysis was performed on an open source project named eGit. For the experimentation, DFD was implemented in Java, and an existing technique named TestHound [1] was used. Later, both techniques were run on the test beds and Manual Inspection (MI) was also performed on these projects. The comparative result analysis shows that DFD has detected 14.03% more setup fields and 60.98% more dead fields than TestHound in eGit due to identifying all the setup fields and header fields in the super classes, resolving field dependency among setup fields, and analyzing method call dependency. All the results found by DFD have been compared to MI and both techniques have provided the same setup fields and dead fields.

II. Motivational Example

There are several problems associated to the presence of dead fields in test code such as increasing size of the project by adding unnecessary code, introducing code smells, making code hard to adapt any change in the code, creating misapprehension while refactoring production code and so on [9]. All these problems are responsible to reduce maintainability of the code. An example case is described in Fig. 1 which demonstrates how dead fields affect code maintainability.

```
public class Account{
      public boolean login(LoginModel loginInfo){....}}
public class GoogleAccount extends Account{
      public boolean login(LoginModel loginInfo){....}}
public class FacebookAccount extends Account{
      public boolean login(LoginModel loginInfo){....}}
public class AccountTest{
      Account account; GoogleAccount googleAccount; FacebookAccount facebookAccount;
      @Before
      public void setUp() throws Exception{
            account = new Account(); googleAccount = new GoogleAccount();
      facebookAccount = new FacebookAccount();}
      @Test
      public void testLogin(){
            assertEquals(account.login(new LoginModel("uname","pwd")),true);}}
```

Fig. 1: Code Snippet Depicting the Impact of Dead Field on Software Maintenance

In Fig. 1, there are three production classes 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. While performing this change, if this issue is not considered, 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.

Another case is described in Fig. 2 and Fig. 3 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 Fig. 2, there is a test class named *ProductServiceTest* which contains five fields - *dummyProduct*, *dbConnector*, *dbContext*, *productRepo* and *productService*. All the 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();
  productRepo = new ProductRepository(dbConnector,dbContext);
  productService = new ProductService();}
 @Test
 public void testAddProduct(){assertEquals(productService.addProduct(dummyProduct),true);}
 @Test
 public void testRemoveProduct(){assertEquals(productService.removeProduct(dummyProduct),true);}}
```

Fig. 2: Sample Test Code Containing Dead Fields

Now, in TDD, the test class, *ProductServiceTest* 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, DatabaseContext, ProductRepository* and *ProductService*. A sample code illustrating these classes is shown in Fig. 3.

```
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{....}
```

Fig. 3: Production Code Written from Fig. 2

Since in *ProductServiceTest*, test methods - *testAddProduct* and *testRemoveProduct* invoke two methods of *productService* object (as shown in Fig. 2), according to TDD, *ProductService* class holds two methods - *addProduct* and *removeProduct*. In the setup method, two fields *dbConnector* and *dbContext* are used for the instantiation of *productRepo* (as shown in Fig. 2). So, the constructor of *ProductRepository* class will take two parameters of type *DatabaseConnector* and *DatabaseContext* (as depicted in Fig. 3).

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*. The production code will then become more maintainable and comprehensible.

III. Related Work

Dead field is one of the common test smells which reduces the quality of test code. Its presence indicates the incomplete

or poorly written test code that affects software maintenance. Although the smell has been recently discovered, several researches have been carried out to understand the impact of test smells. Few automatic techniques have been proposed to detect test smells in the test code. Most significant works in this domain have been explained as follows.

van Deursen et al. coined the term test smell and defined it as symptoms of poor test code quality [10]. They proposed eleven different test smells such as Assertion Roulette, Eager Test, Indirect Testing, Mystery Guest, General Fixture, etc. They showed how these test smells reduce the maintainability of test code. To identify and remove these smells, they discussed the characteristics of the smells and refactoring technique. However, no automatic technique was provided to detect dead field because it was not identified that time.

Gabriele et al. conducted an empirical analysis to perceive the distribution and impact of unit test smells in software maintenance [5]. Two studies were conducted where one of these was exploratory study and another was controlled experiment. Sixteen open source and two industrial projects were studied in the exploratory study to know the distribution of test smells. It was found from the study that test smells were widely spread throughout the projects. Twenty masters student were asked to understand the smelly projects in the controlled experiment. The students faced difficulties in understanding the test code of the projects due to the existence of the smells. They also found the projects difficult to perform some maintenance tasks that were used in the controlled experiment. Although the empirical analysis provides a notion about the distribution and impact of test smell, the authors did not provide any technique to automatically detect dead fields in test code. The reason is that the purpose of the study was only to gain insight about the impact and distribution of test smells in open source and industrial projects.

Rompaey et al. proposed a metrics-based approach to find two different test smells - test fixture and eager test [7]. To detect test fixture, three metrics were used which were setup size, fixture size and fixture usage. The authors calculated setup size by accumulating the number of production type referred in the test class and the number of methods or attributes of non test objects. They defined fixture size as the total number of fixture elements and production types in test code. To identify eager test smell, they calculated the number of methods in the production code invoked by a test case. The proposed technique showed promising results for the experimental project ArgoUML. However, the metrics used in the approach are insufficient to detect dead field correctly. The reason is that the characteristics of dead field are different from test fixture and eager test. Satter et al. proposed and developed a static code analysis tool that can detect dead fields in a single test file which does not inherit any test fixture [11]. In the real life projects, common test fixtures are often reused and many indirect dependencies are seen [9]. The tool fails to detect all the dead fields correctly due to not considering base class test fixture, initialization of header fields, and indirect usage of setup fields by non-test methods.

Stefan Reichart et al. integrated static analysis technique in TestLint to detect several test smells such as Guarded Test, OverReferencing, Assertionless Test, Long Test, Overcommented Test, etc. [6]. To identify the smells, the technique parses the test code, constructs Abstract Syntax Tree, finds patterns, and calculates metrics. The authors used a set of rules generated from the properties of the test smells [4], [12], [13], for instance, finding at least one valid assertion statement to detect Assertionless Test, figuring out conditional branches in a test case to identify Guarded Test, etc. However, the authors did not address any rule or metric for dead field.

In order to understand the structural and maintenance properties of test code, Manuel et al. proposed a tool named TestQ [8]. A visualization module was plugged into the tool to quantify test smelliness and explore test suites from different granular levels. To detect eleven different test smells proposed by Deursen [7], the authors used a list of metrics. For example, number of assert statements for Assertionless, number of assert statements containing insufficient description for AssertionRoulette, number of invoked methods in the production code for EagerTest, and so on. The tool allows the user to customize the threshold values of the metrics that best fit for detecting the smells. It can identify the test smells and the level of smelliness. However, the authors did not provide any metric for dead field.

Usually, a test fixture defines the system under test [14]. Greiler et al. analyzed the test fixture of test code and proposed five test smells such as Test Maverick, Obscure In-Line Setup, Lack of Cohesion of Test Methods, Vague Header Setup, and Dead Field [1]. They defined dead fields as the initialized setup fields that have never been used by any test method in the test code. The authors developed a tool named TestHound to detect the smells. The tool takes the test code with the dependencies and calculates several metrics such as number of declared variables, number of header fields, number of used header fields, number of test methods, etc. Next, it identifies test smells in the code based on the metric values. The authors conducted an experimental analysis and found that the tool performed well in identifying the smells. However, it could not detect all the dead fields correctly because it could not map the initialization and the usage of setup fields properly.

Several studies showed the adverse effect of test smells in software maintenance. To increase the comprehensibility and maintainability of test code, studies suggested that test smells should be identified and removed. Manually detecting test smells in large software systems is time consuming and error-prone. Literature contains several techniques to automatically identify test smells, for example, metric-based technique [6], [7], static code analysis [1], rule-based approach [6], etc. However, none of the techniques can identify all the dead fields correctly due to not analyzing method call dependency and data dependency in test code.

IV. Proposed Technique

In this paper, a technique named Dead Field Detector (DFD) has been proposed to identify dead fields automatically.

DFD comprises four steps - Call Graph Generation, Data Dependency Graph Generation, Setup Field Identification, and Dead Field Detection. Each of the steps is discussed below.

A. Call Graph Generation

A setup method may invoke one or more methods to initialize a setup field. Again, a test method may call one or more non-test methods for its execution. The test method may not have any statement that uses a setup field. However, its invoked methods may use the setup field. So, it is required to identify the invocation relationship among the methods. To find the relationship, Algorithm 1 has been devised.

Algorithm 1 Generating Call Graph and Finding Invoked Methods

```
Require: Test code (F) of a given project for which call graph will be generated
 1: procedure ConstructCallGraph(F)
 2:     parse F to find all the methods and store into M
 3:     Map < Method, List < Method >> adj
 4:     for each m ∈ M do
 5:         for each st ∈ m.body.statements do
 6:             if st is a method invocation statement then
 7:                 for each m' ∈ M do
 8:                     if m == m' then
 9:                         continue
10:                     end if
11:                     if st.name == m.name & st.argumentsTypes ==
    m.parametersTypes & st.returnType == m.returnType then
12:                         adj[m].add(m')
13:                     end if
14:                 end for
15:             end if
16:         end for
17:     end for
18:     return adj
19: end procedure
20: procedure FindInvokedMethods(m, adj)
21:     Y = ∅
22:     Queue q = ∅
23:     q.push(m)
24:     while q ≠ ∅ do
25:         x = q.pop()
26:         Y = Y ∪ x
27:         for each x' ∈ adj[x] do
28:             q.push(x')
29:         end for
30:     end while
31:     return Y
32: end procedure
```

In order to construct a call graph from the test code of a given project and find method call dependencies from the graph, Algorithm 1 has been proposed. Here, a call graph is a directed graph where each node denotes a method, and an edge from node a to b (i.e., $a \longrightarrow b$) represents a invokes b for its execution. In the algorithm, the procedure $ConstructCallGraph$ takes the test code as input for which call graph will be generated. The test code is parsed to obtain all the declared methods and store those into a list, M. A map named adj is declared in Line 3 of Algorithm 1 to store invocation relationship among the methods. For each method m in M, all the statements found in the method body are parsed to get method invocation statements as shown in Algorithm 1 from Line 5 - 10. Method name, argument type and return type are obtained from each invocation statement to match with the signature of other methods. If a matching is found for any of the declared methods, the respective method is stored in adj as an invoked method of m (Algorithm 1, Line 11-12). Next, the procedure returns adj as shown in Algorithm 1, Line 18. The second procedure named $FindInvokedMethods$ uses adj to find all the methods invoked by the given method m. In the procedure a list named Y is declared to store the invoked methods. A queue, Q is initialized with m to find all the methods that are accessible from m using adj. In each iteration of the *while* loop, Q is popped to get the unvisited method x in the call graph (adj). Next, x is added to Y as an invoked method of m. All the methods called by x are obtained and added to Q using a *for* loop in Algorithm 1 Line 27-29. After finishing the iteration, the procedure returns Y as the list of methods on which m is dependent.

Algorithm 2 Resolving Field Dependency

```
Require: Test code (F) of a given project for which data dependency graph will be
    generated
 1: procedure ConstructDataDependencyGraph(F)
 2:     statements = parse F to find all the statements in F
 3:     fields = parse F to obtain all the fields
 4:     Map < Field, List < Field >> G
 5:     for each s ∈ statements do
 6:         if s is a field declaration or initialization statement then
 7:             f = fields in s
 8:             x = declared field in s
 9:             y = f \ x
10:             G[x] = G[x] ∪ y
11:         end if
12:     end for
13:     return G
14: end procedure
15: procedure Visit(nd, G)
16:     Y = ∅
17:     Queue q = ∅
18:     q.push(nd)
19:     while q ≠ ∅ do
20:         x = q.pop()
21:         Y = Y ∪ x
22:         for each x' ∈ G[x] do
23:             q.push(x')
24:         end for
25:     end while
26:     return Y
27: end procedure
```

B. Data Dependency Graph Generation

A setup field, f may depend on another setup field, f' for its initialization. f' may not be used by any test method, but f may be used for the execution of at least one test method directly or indirectly. In this case, f' should be marked as used setup field. In order to find the dependency relationship among the setup fields, a data dependency graph is required. Each node in the graph denotes a field, and an edge from field x to y (i.e. $x \longrightarrow y$) expresses that x depends on y. To construct the graph and find all the dependent fields of a given setup field, Algorithm 2 has been devised.

To construct data dependency graph from the given test code F, a procedure named $ConstructDataDependencyGraph$ is shown in Algorithm 2. In the procedure, all the statements in F are parsed to identify the fields declared in the code. A map G is declared in Algorithm 2 Line 4 to store which field is dependent on which other fields. The *for* loop iterates on the statements to detect field declaration or initialization statement (Algorithm 2 Line 5-12). If a declaration statement

Algorithm 3 Finding Setup Field

```
Require: Test code FL which contains setup fields and test cases, call graph CG, and
    data dependency graph DG
 1: procedure IDENTIFYSETUPFIELD(FL, CG, DG)
 2:     p = parse setup methods from FL
 3:     M = ∅
 4:     M = M ∪ p
 5:     M = M ∪ FindInvokedMethods(p, CG)
 6:     F = parse all fields
 7:     H = parse all header fields in FL
 8:     S = ∅
 9:     S = S ∪ H
10:     for each m ∈ M do
11:         for each st ∈ m.body.statements do
12:             for each f ∈ F do
13:                 if st initializes f then
14:                     S = S ∪ f
15:                     S = S ∪ visit(f, DG)
16:                 end if
17:             end for
18:         end for
19:     end for
20:     return S
21: end procedure
```

Algorithm 4 Identifying Dead Field

```
Require: Test code F containing test cases, call graph CG, data dependency graph
    DG, a set of setup fields S
 1: procedure IDENTIFYDEADFIELD(F, CG, DG, S)
 2:     T = parse all test methods in F
 3:     U = ∅
 4:     for each t ∈ T do
 5:         M = t ∪ FindInvokedMethods(t, CG)
 6:         for each m ∈ M do
 7:             for each st ∈ m.body.statements do
 8:                 for each f ∈ S do
 9:                     if st uses f then
10:                         U = U ∪ f ∪ visit(f, DG)
11:                     end if
12:                 end for
13:             end for
14:         end for
15:     end for
16:     return S\U
17: end procedure
```

is found, it is parsed to identify the declared field x and its dependent fields y (Algorithm 2 Line 7-9). Later, y is stored into G against the field x as shown in Algorithm 2 Line 10. At last, the procedure returns the generated data dependence graph G for F (Algorithm 2 Line 13). Another procedure named $Visit$ is shown in Algorithm 2 Line 15 to traverse the data dependence graph G, and find all the fields on which the given node, nd is dependent. A set, Y is declared to store all the fields on which nd depends. To traverse G, a queue, q is declared, and nd is pushed as starting node (Algorithm 2 Line 17 - 18). The $while$ loop iterates until q is empty to find all the nodes that can be visited from nd (Algorithm 2 Line 19). In each iteration, q is popped and the corresponding field x is stored into Y. The *for* loop iterates on the adjacent nodes of x and these nodes are pushed to q (Algorithm 2 Line 22 - 24). After traversing all the nodes in G that can be visited from nd, the procedure returns Y as a set of fields on which nd is dependent (Algorithm 2 Line 26).

C. Setup Field Detection

Since unused setup fields are recognized as dead fields, all the setup fields in the test code are required to be identified for dead field detection. The process of finding setup fields is shown in Algorithm 3.

To identify setup fields, a procedure named $IdentifySetupField$ is defined in Algorithm 3. The procedure parses the given test code FL to identify the setup method as shown in Algorithm 3 Line 2. Next, it calls $FindInvokedMethods$ of Algorithm 1 to obtain all the methods invoked by the setup method directly or indirectly. All these methods are stored in M (Algorithm 3 Line 5). The procedure parses FL to obtain all the declared fields, and it stores the fields in F. Since header-fields (fields which are initialized directly without invoking the setup method) are also considered as setup fields, these fields are identified in FL and saved in S (Algorithm 3 Line 7 - 9). For each method $m \in M$, every statement $st \in m.body.statements$ is checked to determine whether st initializes any field $f \in F$ (Algorithm 3 Line 10 - 19). If such f is found, the procedure $visit$ (defined in Algorithm 2) is invoked to identify all the fields on which f is dependent directly or indirectly (Algorithm 3 Line 15). These fields are considered as setup fields because these are required for the initialization of f. All the fields are stored in S which is returned as a set of setup fields (Algorithm 3 Line 20).

D. Dead Field Detection

After identifying all the setup fields from the previous step, usage of these fields are analyzed for dead field detection. A test method may have a statement in its body that uses a setup field. Again, it may invoke another non-test method that may have an usage statement of the field. These cases are checked for finding the usages of the setup fields. After finding all the used setup fields, unused setup fields are separated for recognizing the dead fields. Algorithm 4 has been presented to deal with these cases.

In Algorithm 4, the procedure $IdentifyDeadField$ takes the test code F containing test cases, its corresponding call graph CG, data dependency graph DG, and the identified setup field set S to detect dead fields in F. It parses all the test methods (i.e. test cases) from F and stores these in a list, T to find the usages of the setup fields (Algorithm 4 Line 2). An empty set U is declared to store all the used setup fields as shown in Algorithm 4 line 3. A nested *for* loop is defined where the outer *for* loop iterates on the test methods T to determine the used setup fields. For each method $t \in T$, M is used to store t and its invoked methods through calling $FindInvokedMethods$ (defined in Algorithm 1). The reason is that t may not use a setup field directly but its invoked methods may use the field. The field should be marked as used setup field, because t indirectly depends on the field for its execution. For each method $m \in M$, every statement st in m is checked whether st uses any setup field $f \in F$ (Algorithm 4 Line 6 - 9). If such f is found, f and all other fields on which f is dependent are marked as used setup fields and inserted into U. This is because a setup field may not be used by any test method but it may be used for the initialization of another

setup field which is used by at least one test method. So, the procedure $visit$ (defined in Algorithm 2) is invoked to identify the list of setup fields on which f is dependent. After that, it marks these fields as used setup field (Algorithm 4 Line 10). After identifying all the used setup fields, unused setup fields are separated from S and these are considered as dead fields. The procedure returns the set of dead fields obtained through set subtraction operation, that is $S \backslash U$ (Algorithm 4 Line 16).

V. Implementation and Result Analysis

This section focuses on the evaluation of DFD in terms of accuracy in dead field detection. For the evaluation, the technique was implemented in Java and the source code is available in *https://tinyurl.com/ybk29dss*. A popular open source project named eGit(*http://www.eclipse.org/egit/*) was used as the experimental dataset. eGit contains 130K lines of code, 85 test classes and 10 modules. Another tool named TestHound (TH) [1] was used for the comparative analysis with DFD. At last, Manual Inspection (MI) was carried out to make sure the correctness of the results provided by DFD. For MI, twenty masters students and five professional Java developers were employed. They scrutinized the dataset to find the setup fields and dead fields, and cross-checked the results. Detailed results for each module of eGit along with the comparative analysis are explained below.

Comparative result analysis for org.eclipse.egit.core.test.op: According to TABLE I, there are 19 test classes in this package where 105 setup fields and 14 dead fields have been identified by DFD. On the other hand, TH has detected 92 setup fields and 5 dead fields in this package. Here, most of the test classes use the test fixture of *GitTestCase* but this class contains a header field named *testUtils* which has not been used by any test case of the test classes extending it. Since DFD has detected all the header fields of super class and considered these as setup fields, the outcome of DFD is the same 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* that extends another class named *GitTestCase*. DFD and TH both have identified all the setup fields in *RebaseInteractivePlanTest* as shown in TABLE II. However, there is a header field named *testUtils* which has not been considered as setup field by TH. Thus, TH could not detect this dead field. Since DFD takes all the header fields declared in both parent class and child class, it has found 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 package which has a parent class named *GitTestCase*. In TABLE III, the number of dead fields identified by DFD and TH is the same. However, there is a single value difference between the number of setup fields identified by DFD and TH. The reason is that DFD takes the whole test fixture of the super class whereas TH ignores initialized header fields in the super class.

TABLE I: Comparative Result Analysis for org.eclipse.egit.core.test.op

Class Name	No. of Setup fields			No. of Dead Fields		
	TH	DFD	MI	TH	DFD	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
ConnectProviderOperation Test	2	3	3	0	1	1
CreatePatchOperationTest	5	6	6	0	1	1
DiscardChangesOperation Test	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
RemoveFromIndex Operationtest	6	7	7	2	2	2
ResetOperationTest	4	5	5	0	1	1
RewordCommitsOperation Test	4	5	5	0	1	1
SquashCommitsOperation Test	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 II: Comparative Result Analysis for org.eclipse.egit.core.test.rebase

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

TABLE III: Comparative Result Analysis for org.eclipse.egit.core.test.internal.mapping

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

Comparative result analysis for org.eclipse.egit.core.test: In TABLE IV, comparative results for 13 test classes of this package has been depicted where TH has identified 53 setup fields, and DFD has detected 64 setup fields as well as 9 dead fields. This is because of not considering header fields in super class as explained earlier. Besides, there are some classes like *EclipseGitProgressTransformerTest*, *LinkedResourcesTest*, and these classes have not used the super class fixture. DFD, TH and MI have produced the same results for the test class *EclipseGitProgressTransformerTest*. On the other hand, DFD has identified 4 dead fields for *LinkedResourcesTest*, but TH could not find any dead fields. The reason is that DFD has identified all initialized fields in that class through generating data dependency graph and resolving field dependency. TH has ignored the field initialization statements and field dependencies.

TABLE IV: Comparative Result Analysis for org.eclipse.egit.core.test

Class Name	No. of Setup fields			No. of Dead Fields		
	TH	DFD	MI	TH	DFD	MI
AdaptableFileTreeIterator Test	4	5	5	0	1	1
CommitUtilTest	6	7	7	0	1	1
ContainerTreeIterator ResourceFilterTest	3	4	4	0	0	0
EclipseGitProgressTransformerTest	2	2	2	0	0	0
FileDeleteHookTest	5	6	6	0	1	1
GitProjectSet CapabilityTest	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
SubmoduleAndContainer TreeIteratorTest	9	9	9	0	0	0
UtilsTest	0	0	0	0	0	0

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* have been presented in TABLE V and TABLE VI, respectively. Both packages comprise a single test class each and these classes extend the same super class *GitTestCase*. In *GitSynchronizeDataTest*, TH has identified 3 setup fields and no dead field, but actually there are 4 setup fields. Among these fields, one is dead field which has been identified by DFD. There is a single dead field in *GitBlobStorageTest* and DFD has detected the field. However, TH could not identify the dead field. The reason is that TH has not considered the header fields in the parent class. However, DFD has identified all the header fields and parent classs setup fields through traversing the data dependence graph and resolving data dependency.

TABLE V: Comparative Result Analysis for org.eclipse.egit.core.synchronize.dto

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

TABLE VI: Comparative Result Analysis for org.eclipse.egit.core.storage

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

Comparative Result Analysis for org.eclipse.egit.core.securestorage, org.eclipse.egit.core.internal.indexdiff, and org.eclipse.egit.core: Comparative results for these packages have been shown in TABLE VII, TABLE VIII, and TABLE IX, respectively where setup fields and dead fields detected by TH, DFD and MI are the same. Each test class in these packages has its own fixture defined within the class. So, there is no fixture dependency with any super class. Besides, setup methods of these classes have not invoked any other method which indicates that all the setup fields have been initialized in the setup methods. Since, TH and DFD both can identify header fields and setup fields that are initialized directly by setup method, the results provided by these approaches are the same.

TABLE VII: Comparative Result Analysis for org.eclipse.egit.core.securestorage

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

TABLE VIII: Comparative Result Analysis for org.eclipse.egit.core.internal.indexdiff

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

TABLE IX: Comparative Result Analysis for org.eclipse.egit.core

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

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

TABLE X: Comparative Result Analysis for org.eclipse.egit.core.synchronize

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

DFD and TH, both can identify dead fields in the test code. However, TH cannot detect all the dead fields correctly due to not handling all the cases properly such as setup fields initialization 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, DFD can appropriately deal with those and as a result, it detects dead fields correctly in the test code.

VI. Threats to Validity

In this section limitations of the experimental study are discussed in terms of internal, external and construct validity.

A. Internal Validity

In the experiment, twenty five subjects were employed to identify dead fields manually in the dataset. Since there is no control over the skills of the subjects, it may happen that some dead fields and setup fields may be missed or incorrectly identified. To decrease the risks of the threat, each subject was asked to identify the dead fields and setup fields, and verify the results with other. When any inconsistency was found, they again scrutized the test code. Cross-validation was carried out multiple times to ensure the correctness of the results. Besides, to make sure that all the dead fields were correctly identified, the subjects were asked to remove the fields manually and check whether the test code runs and there is no impact on the functionalities of test cases.

B. External Validity

The dataset used in the experiment may not generalize the population of open source software. However, eGit is commonly used to analyze test code and test smells [15]. It was also used for the evaluation of TestHound [1].

C. Construct Validity

Existing dead code detection techniques may be used to identify dead fields but dead fields are different from dead code. A field is marked as dead code if it is declared but not used or accessed by any other field or method [3]. Since dead field is accessed (i.e., initialized) in the setup method, it cannot be considered as dead code.

It has not been shown the number of setup fields and dead fields incorrectly identified in the result analysis due to the space limitation. However, the subjects carefully examined the results of DFD and found no field that was incorrectly identified as dead field or setup field. The source code of the developed tool has been open sourced so that other can use it.

VII. Conclusion

The presence of a dead field reduces the maintainability of a software system by creating misapprehension among the developers and adding unnecessary code. To remove this test smell, dead fields are required to be located in the test code. Manually identifying dead fields in a large software system is time consuming and error-prone. In this paper, an automatic technique named Dead Field Detector (DFD) has been proposed to correctly detect dead fields. The technique was also implemented in the form of a software tool.

DFD parses the test code to identify the declared fields and setup method. It constructs call graph of the code to find the methods invoked by the setup method. It identifies all the fields x initialized by the methods. Fields y on which x depends are identified by generating and traversing data dependence graph. x and y are then marked as setup fields. The technique finds the used setup fields by checking the usage statements in the test methods and their invoked methods obtained from the call graph. Finally, the detected unused setup fields are marked as dead fields.

In order to assess DFD, an experimental analysis was conducted on an open source project named eGit. An existing technique named TestHound (TH) was also used in the experiment, and Manual Inspection (MI) was carried out to ensure correctness of the results. The comparative result analysis shows that DFD has detected 14.03% more setup fields and 60.98% more dead fields than TH in eGit. The results of DFD has been compared to MI, and no false positive has been seen for DFD. In future, the experiment will be conducted on industrial projects to assess the behavior of DFD.

References

[1] Michaela Greiler, Arie van Deursen, and Margaret-Anne Storey. Automated detection of test fixture strategies and smells. In *Proceedings of the 16th International Conference on Software Testing, Verification and Validation (ICST)*, pages 322–331. IEEE, 2013.

[2] Kent Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.

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

[4] Arie van Deursen, Leon Moonen, Alex van den Bergh, and Gerard Kok. *Refactoring test code*. CWI, 2001.

[5] Gabriele Bavota, Abdallah Qusef, Rocco Oliveto, Andrea De Lucia, and David 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*, pages 56–65. IEEE, 2012.

[6] Stefan Reichhart, Tudor Gîrba, and Stéphane Ducasse. Rule-based assessment of test quality. *Journal of Object Technology*, 6(9):231–251, 2007.

[7] Bart Van Rompaey, Bert Du Bois, Serge Demeyer, and Matthias Rieger. On the detection of test smells: A metrics-based approach for general fixture and eager test. *IEEE Transactions on Software Engineering*, 33(12):800–817, 2007.

[8] Manuel Breugelmans and Bart 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] Michele Tufano, Fabio Palomba, Gabriele Bavota, Massimiliano Di Penta, Rocco Oliveto, Andrea De Lucia, and Denys Poshyvanyk. An empirical investigation into the nature of test smells. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, pages 4–15. ACM, 2016.

[10] A. van Deursen, L. Moonen, A. van den Bergh, and G. Kok. Refactoring test code. In *Proceedings of the 2nd International Conference on Extreme Programming and Flexible Processes (XP2001)*, pages 92–95. University of Cagliari, 2001.

[11] Abdus Satter, Amit Seal Ami, and Kazi Sakib. A static code search technique to identify dead fields by analyzing usage of setup fields and field dependency in test code. In *CDUD@ CLA*, pages 60–71, 2016.

[12] Gerard Meszaros. *xUnit test patterns: Refactoring test code*. Pearson Education, 2007.

[13] Gerard Meszaros, Shaun M Smith, and Jennitta Andrea. The test automation manifesto. In *Extreme Programming and Agile Methods-XP/Agile Universe 2003*, pages 73–81. Springer, 2003.

[14] Lucas Pereira da Silva and Patrícia Vilain. Execution and code reuse between test classes. In *Proceedings of the 14th International Conference on Software Engineering Research, Management and Applications (SERA)*, pages 99–106. IEEE, 2016.

[15] Michaela Greiler, Hans-Gerhard Gross, and Arie Van Deursen. Understanding plug-in test suites from an extensibility perspective. In *Proceedings of the 17th Working Conference on Reverse Engineering (WCRE)*, pages 67–76. IEEE, 2010.

