Improving Software Testing: Effective Error Detection with Dual Driven Tools

Chloe Scott[1], Lucas Scott[1], Olivia Perez[2]
[1] Department of Sociology, University of Southern California, USA
[2] School of Public Health, University of California, Los Angeles, USA

ABSTRACT

Source code of large systems is iteratively refined, restructured and evolved due to many reasons such as correcting errors in design, modifying a design to accommodate changes in requirements, and modifying a design to enhance features. Many studies reported that these software maintenance activities consume up to 90 percent of the total cost of a typical software project. This paper considers code-smells detection as a distributed optimization problem. The idea is that different methods are combined in parallel during the optimization process to find a consensus regarding the detection of code-smells. To this end, we used Parallel Evolutionary algorithms (P-EA) here many evolutionary algorithms with different adaptations (fitness functions, solution representations, and change operators) are executed, in a parallel cooperative manner, to solve a common goal which is the detection of code-smells. An empirical evaluation to compare the implementation of the cooperative P-EA approach with random search, two single population-based approaches and two code-smells detection techniques that are not based on meta-heuristics search. The Dual Driven Software Tool analysis of the obtained results provides evidence to support the claim that cooperative P-EA is more efficient and effective than state of the art detection approaches based on a benchmark of nine large open source systems where more than 85 percent of precision and recall scores are obtained on a variety of eight different types of code-smells.

Keywords: *Code Generation, Error Code Detection, Parallel Evolutionary, Dual Driven Software Tool.*

I. INTRODUCTION

The general definition of software engineering is still being debated by practitioners today as they struggle to come up with ways to produce software that is "cheaper, better, faster". Cost reduction has been a target of the IT industry since the 1990s. Total cost of ownership represents the costs of more than just buying. It includes things like capacity impediments, upkeep efforts, and resources needed to support infrastructure. The Software Engineering Institute offers certifications on specific topics like Security, Process improvement and Software architecture. Apple, IBM, Microsoft and other companies also sponsor their own certification examinations. Many IT certification programs are oriented toward specific technologies, and managed by the vendors of these technologies. These certification programs are tailored to the institutions that would employ people who use these technologies.

Many software engineers work as employees or contractors. Software engineers work with businesses, government agencies (civilian or military), and non-profit organizations. Some software engineers work for themselves as freelancers. Some organizations have specialists to perform each of the tasks in the software development process. Other organizations require software engineers to do many or all of them. In large projects, people may practice in only one role. In small projects, people may fill several or all act at the same time.

For many papers, developers are overwhelmed by the volume of incoming bug reports that must be addressed. For example, in the Eclipse project, developers receive an average of 115 new bug reports every day; the Mozilla and IBM Jazz projects get 152 and 105 new reports per day, respectively. Developers must then spend considerable time and effort to read each new report and decide which source code entities are relevant for fixing the bug. This task is known as bug localization, which is defined as a classification problem: Given n source code entities and a bug report, classify the bug report as belonging to one of the n entities. The classifier returns a ranked list of possibly relevant entities, along with a relevancy score for each entity in the list. An entity is considered relevant if it indeed needs to be modified to resolve the bug report, and irrelevant otherwise.

The developer uses the list of possibly relevant entities to identify an entity related to the bug report and make the necessary modifications. After one relevant entity is identified using bug localization, developers can use change propagation techniques [1], [6] to identify any other entities that also need to be modified. Hence, the bug localization task is to find the first relevant entity; the task then switches to change propagation. Fully or partially automating bug localization can drastically reduce the development effort required to fix bugs, as much of the fixing

time is currently spent manually locating the appropriate entities, which is both difficult and expensive. Current bug localization research uses Information Retrieval (IR) classifiers to locate source code entities that are textually similar to bug reports.

However, current results are ambiguous and contradictory: Some claim that the Vector Space Model (VSM) provides the best performance, while others claim that the Latent Dirichlet Allocation (LDA) model is best, while still others claim that a new IR model is needed . These mixed results are due to the use of different datasets, different performance metrics, and different classifier configurations.

A classifier configuration defines the value of all the parameters that specify the behavior of a classifier, such as the way in which the source code is preprocessed, how terms are weighted, and the similarity metric between bug reports and source code entities. So performing a large-scale empirical study to compare thousands of IR classifier configurations is necessary.

By using the same datasets and performance metrics, we can perform an apples-to-apples comparison of the various configurations, quantifying the impact of configuration on performance, and identifying which particular configurations yield the best performance. It is found that configuration indeed has a large impact on performance: Some configurations are nearly useless, while others perform very well.

Researchers have explored the use of IR models for bug localization. For example, Lukins et al. [16], [17] compare the performance of LSI and LDA using three small case studies. The authors build the two IR classifiers on the identifiers and comments of the source code and compute the similarity between a bug report and each source code entity using the cosine and conditional probability similarity metrics.

Nguyen et al. [12] introduce a new topic model based on LDA, called BugScout, in an effort to improve bug localization performance. BugScout explicitly considers past bug reports, in addition to identifiers and comments, when representing source code documents, using the two data sources concurrently to identify key technical concepts. The authors apply BugScout to four different projects and find that BugScout improves performance by up to 20 percent over LDA applied only to source code.

The study of querying for text within a collection of documents is termed as Information retrieval. The Search engines uses IR techniques to help users find snippets of text in web pages. The “Search Documents” function in a typical operating system is based on the same theory, although applied at a much smaller scale. IR-based bug localization classifiers use IR models to find textual similarities between a bug report (i.e., query) and the source code entities (i.e., documents).

For example, if a bug report contains the words, “Drop 40 bytes off each image Request object,” then an IR model looks for entities which contain these words (“drop,” “bytes,” “image Request,” etc.).When a bug report and entity contain many shared words, then an IR-based classifier gives the entity a high relevancy score.

Several parameters will control the behaviors of IR-based classifiers. Specifying a value for all parameters fully defines the configuration of a classifier. Common to all IR-based classifiers are parameters to govern how the input textual data are represented and preprocessed:

- How should the source code and bug report be preprocessed? Should compound identifier names (e.g., imgRequest) be split? Should common stop words be removed? Should words be stemmed to their base form?
- Which parts of the source code should be considered: the comments, identifier names, or some other representation, such as the previous bug reports linked to each source code entity?
- Which parts of the bug report should be considered: the title only, description only, or both?

After these parameters are configured, each IR model has its own set of additional parameters that control term weighting, reduction factors, similarity metrics, and other aspects.

The objectives of this paper is,

- To apply Software testing code generation scenario.
- Along with Detection of code-smells, to provide suggestions for correction of code-smells.
- During implementation of the software to Log code-errors.

- To provide validation code for forms present in the projects (applications).

II. RELATED WORKS

Carolina Salto and Enrique Alba [1] This paper investigates a new heterogeneous method that dynamically sets the migration period of a distributed Genetic Algorithm (dGA). Each island GA of this multi population technique self-adapts the period for exchanging information with the other islands regarding the local evolution process. Thus, the different islands can develop different migration settings behaving like a heterogeneous dGA. The proposed algorithm is tested on a large set of instances of the Max-Cut problem, and it can be easily applied to other optimization problems. The results of this heterogeneous dGA are competitive with the best existing algorithms, with the added advantage of avoiding time consuming preliminary tests for tuning the algorithm.

Marouane Kessentini et al., [2] In this article, presented a novel approach to the problem of detecting and fixing design defects. Typically, researchers and practitioners try to characterize different types of common design defects and present symptoms to search for in order to locate the design defects in a system. In this work, we have shown that this knowledge is not necessary to perform the detection. Instead, we use examples of design defects to generate detection rules. After generating the detection rules, we use them in the correction step. In fact, we start by generating some solutions that represent a combination of refactoring operations to apply. A fitness function calculates, after applying the proposed refactorings, the number of detected defects, using the detection rules. The best solution has the minimum fitness value. Due to the large number of refactoring combination, a genetic algorithm is used. Our study shows that our technique outperforms DECOR [5], a state-of-the-art, metric-based approach, where rules are defined manually, on its test corpus. The proposed approach was tested on open-source systems and the results are promising. As part of future work, we plan to extend our base of examples with additional badly-designed code in order to take into consideration more programming contexts.

Mark Harman [3] This paper describes work on the application of optimization techniques in software engineering. These optimization techniques come from the operations research and metaheuristic computation research communities. The paper briefly reviews widely used optimization techniques and the key ingredients required for their successful application to software engineering, providing an overview of existing results in eight software engineering application domains. The paper also describes the benefits that are likely to accrue from the growing body of work in this area and provides a set of open problems, challenges and areas for future work.

Jian Ren et al., [4] This paper presents an approach to Search Based Software Project Management based on Cooperative Co-evolution. Our approach aims to optimize both developers' team staffing and work package scheduling through cooperative co-evolution to achieve early overall completion time. To evaluate our approach, we conducted an empirical study, using data from four real-world software projects. Results indicate that the Co-evolutionary approach significantly outperforms a single population evolutionary algorithm. Cooperative co-evolution has not previously been applied to any problem in Search Based Software Engineering (SBSE), so this paper reports the first application of cooperative coevolution in the SBSE literature. We believe that co-evolutionary optimization may fit many applications in other SBSE problem domains, since software systems often have complex inter-related subsystems and are typically characterized by problems that need to be co-evolved to improve results.

A.E. Eiben and S.K. Smit [5] In this paper we present a conceptual framework for parameter tuning, provide a survey of tuning methods, and discuss related methodological issues. The framework is based on a three-tier hierarchy of a problem, an evolutionary algorithm (EA), and a tuner. Furthermore, we distinguish problem instances, parameters, and EA performance measures as major factors, and discuss how tuning can be directed to algorithm performance and/or robustness. For the survey part we establish different taxonomies to categorize tuning methods and review existing work. Finally, we elaborate on how tuning can improve methodology by facilitating well-funded experimental comparisons and algorithm analysis.

Karim Dhambri et al., [6] This paper proposes a semi-automatic detection approach that combines automatic pre-processing and visual representation and analysis of data. Our approach is complementary to automatic approaches for anomalies whose detection requires knowledge that cannot be easily extracted from the code directly. The

detection is seen as an inspection activity supported by a visualization tool that displays large programs (thousands of classes) and allows the analyst navigating at different levels of the code. More specifically, we detect occurrences of anomalies by viewing them as sets of classes playing roles in predefined scenarios. Primary roles are identified first, and starting from them, secondary roles are located. This strategy allows to reduce the search space which compensates for human intervention. Although our case study showed interesting results, it revealed that there is room for improvement. The performance variability of subjects still needs improvement. To reduce this variability, we are defining features to better guide the analyst when exploring the search space. Combining our approach with an automatic one is another direction we are investigating. Three ways of combination are possible. First, we can use our tool to explore the results of automatic detection and accept/reject the detected occurrences. We can also use the two approaches in parallel, depending on the required knowledge to detect the anomalies.

Haiyu Hou, Gerry Dozier [7] This presented a constraint-based AIS, evaluated its performance on problems with high-dimensional space, which has been shown to be difficult for NSA defined over Hamming shape-space. This experiment shows that our AIS does notsuffer from serious scaling problem and our experimental results verify that constraint-based detectors with good coverage can be generated in reasonable time. The varied performances of the different types of detectors and matching rules suggests thatunderstanding the characteristics of data and the representation will help formulating the best strategy for solving problems.

Giorgos Karafotias, Mark Hoogendoorn, A.E. Eiben Parameter control mechanisms in evolutionary algorithms (EAs) dynamically change the values of the EA parameters during a run. Research over the last two decades
has delivered ample examples where an EA using a parameter control mechanism outperforms its static version with fixed parameter values. However, very few have investigated why such parameter control approaches perform better. In principle, it could be the case that using different parameter values alone is already sufficient and EA performance can be improved without sophisticated control strategies raising an issue in the methodology of parameter control mechanisms' evaluation. This paper investigates whether very simple random variation in parameter values during an evolutionary run can already provide improvements over static values. Results suggest that random variation of parameters should be included in the benchmarkswhen evaluating a new parameter control mechanism. When setting up an evolutionary algorithm (EA) one aspect that needs to be addressed is defining appropriate values for the various parameters of the algorithm.

In case inappropriate values are chosen the performance of the EA can be severely degraded. The uestion whether a certain value of a parameter is appropriate is far from trivial as different phases in an
evolutionary run could require different values. In fact, there are two principal options to set such values [4]: (1) trying to find fixed parameter values that seem to work well across the entire evolutionary run (parameter tuning), and (2) finding a suitable control strategy to adjust the parameter values during a run (parameter control). Furthermore, we can distinguish three forms of parameter control: (a) deterministic parameter control, which uses a fixed control scheme without using any input from the state of the process; (b) adaptive parameter control, utilizing information from the state of the process to determine good parameter values, and (c) self-adaptive whereby the parameter values are part of the evolutionary process itself. In the literature a variety of evolutionary algorithms equipped with sophisticated parameter control strategies have been shown to outperform their static counterparts and many have acknowledged that dynamically adjusting parameter values is a very good idea.

In the majority of work presenting parameter control mechanisms, the value of the controller is assessed only by comparing its performance to the static version of the EA that keeps parameter values fixed. The motivation of this paper is based on the idea that such performance benefits observed when using parameter control mechanisms over using static parameter values might be in fact a result of simply the variation of the parameter and not the intelligent strategy itself. Some authors have made some weak hints in this direction, see the next Section, however, none have performed a rigorous analysis. If it is possible that variation on its own (without some intelligent strategy) might improve performance, a methodological issue is raised: when evaluating a parameter control mechanism the actual contribution of the intelligent strategy to the performance gain should not be taken for granted but should be explicitly assessed by also including 'naive variation' in the set of benchmarks used. The goal of this paper is to investigate whether (nonintelligent) 'variation' alone might indeed improve EA performance as compared to keeping parameter values fixed. To this end, we implement a few simple random methods to vary parameter values during the run of an EA and investigate their impact on a set of standard test problems. In particular, we use a uniform distribution and a Gaussian distribution and compare the resulting EAs with an EA whose parameter values

are fixed (by a powerful tuning algorithm) and with an EA whose parameters change by a sine wave based schedule (enabling increasing and decreasing the values).

A.E. Eiben and S.K. Smit is present a conceptual framework for parameter tuning, provide a survey of tuning methods, and discuss related methodological issues. The framework is based on a three-tier hierarchy of a problem, an evolutionary algorithm (EA), and a tuner. Furthermore, we distinguish problem instances, parameters, and EA performance measures as major factors, and discuss how tuning can be directed to algorithm performance and/or robustness. For the survey part we establish different taxonomies to categorize tuning methods and review existing work. Finally, we elaborate on how tuning can improve methodology by facilitating well-funded experimental comparisons and algorithm analysis.

The main objectives of this paper are threefold. We want to present a conceptual framework behind parameter tuning, provide a survey of relevant literature, and argue for a tuning-aware experimental methodology. The conceptual framework is comprised of the pivotal notions regarding parameter tuning, arranged and presented in a certain logical structure. It also embodies a vocabulary that can reduce ambiguity in discussions about parameter tuning. However, we are not aiming at mathematical rigor, as we are not to present formal definitions and theorems. Our treatment is primarily practical.

We consider the design, or configuration, of an evolutionary algorithm (EA) as a search problem in the space of its parameters and, consequently, we perceive a tuning method as a search algorithm in this space. We argue that parameter tuning can be considered from two different perspectives, that of• configuring an evolutionary algorithm by choosing parameter values that optimize its performance.

Analyzing an evolutionary algorithm by studying how its performance depends on its parameter values. To this end, it is essential that a search algorithm generates much data while traversing the search space. In our case, these data concern a lot of parameter vectors and corresponding values of algorithm performance. If one is only interested in an optimal EA configuration then such data are not relevant– finding a good parameter vector is enough. However, if one is interested in gaining insights into the EA at hand, then these data are highly relevant for they reveal information about the evolutionary algorithm's robustness, distribution of solution quality, sensitivity etc. Adopting the terminology of Hooker, we refer to these options as competitive and scientific testing, and discuss how scientific testing is related to the notion of algorithm robustness. We also show that there are multiple definitions of robustness obtained through extending the above list by.

- Analyzing an evolutionary algorithm by studying how its performance depends on the problems it is solving, and
- Analyzing an evolutionary algorithm by studying how its performance varies when executing independent repetitions of its run, and then we discuss how these definitions are related to parameter tuning.

Karim Dhambri, Houari Sahraoui , Pierre Poulin Design anomalies, introduced during software evolution, are frequent causes of low maintainability and low flexibility to future changes. Because of the required knowledge, an important subset of design anomalies is difficult to detect automatically, and therefore, the code of anomaly candidates must be inspected manually to validate them. However, this task is time- and resource-consuming. We propose a visualization-based approach to detect design anomalies for cases where the detection effort already includes the validation of candidates. We introduce a general detection strategy that we apply to three types of design anomaly. These strategies are illustrated on concrete examples. Finally we evaluate our approach through a case study. It shows that performance variability against manual detection is reduced and that our semi-automatic detection has good recall for some anomaly types.

Design anomalies introduced in the development and maintenance processes can compromise the maintainability and evolvability of software. As stated by Fenton and Pfleeger [5], design anomalies are unlikely to cause failures directly, but may do it indirectly. Detecting and correcting these anomalies is a concrete contribution to softwarequality improvement. However, detecting anomalies is far from trivial [10]. Manual detection is time- and resource consuming, while automatic detection yields too many false positives, due to the nature of the involved knowledge [12]. Furthermore, there are additional difficulties inherent to anomaly detection, such as context-dependence, size of the search space, ambiguous definitions, and the well-known problem of metric threshold value definition. In this paper, we propose a semi-automatic approach to detect design anomalies using software visualization. We model design anomalies as scenarios where classes play primary and secondary roles. This model

helps reducing the search space for visual detection. Our approach is complementary to automatic detection as defined in [10, 12]. Indeed, we specifically target anomalies that are difficult to detect automatically. We use detection strategies that combine automated actions with user-oriented actions. The judgement of the analyst is used when automatic decisions are difficult to make. Such decisions are related to the application context, low-level architecture choices, variability in anomaly occurrences, and metric threshold values.

Anomaly detection and correction is a concrete and effective way to improve the quality of software. This paper proposes a semi-automatic detection approach that combines automatic pre-processing and visual representation and analysis of data. Our approach is complementary to automatic approaches for anomalies whose detection requires knowledge that cannot be easily extracted from the code directly. The detection is seen as an inspection activity supported by a visualization tool that displays large programs (thousands of classes) and allows the analyst navigating at different levels of the code. More specifically, we detect occurrences of anomalies by viewing them as sets of classes playing roles in predefined scenarios. Primary roles are identified first, and starting from them, secondary roles are located. This strategy allows to reduce the search space which compensates for human intervention. Although our case study showed interesting results, it revealed that there is room for improvement. The performance variability of subjects still needs improvement. To reduce this variability, we are defining features to better guide the analyst when exploring the search space. Combining our approach with an automatic one is another direction we are investigating. Three ways of combination are possible. First, we can use our tool to explore the results of automatic detection and accept/reject the detected occurrences. We can also use the two approaches in parallel, depending on the required knowledge to detect the anomalies.

III. P-EA and DDF METHODOLOGY

A high-level view of our P-EA approach to the code-smells detection problem is introduced, in which codes are detected which contains zero function, one function and multifunction classes. It constructs methods, which represents detection rules (metrics combination) for artificial code that represents pseudo code-smell examples.

It also generates rules and code elements to find single function, multi function and zero function classes. The existing system parses the code files saved by the user and finds the required inefficient code scenario. Finally, developers can use the best rules and detectors to detect code-smells on any new system to evaluate.

In addition with all the existing system algorithm implementation approach, the proposed system contains codes are parsed out for code-smells types such (Functional decomposition, Data class, Long parameter list and Shotgun surgery (SS) types) are software testing code generation and validation code generation mechanism so that code-smells are eliminated. The remedies for code-smells i.e., error correcting suggestions are also found out and provided.

The first step in the software development life cycle is the identification of the problem. As the success of the system depends largely on how accurately a problem is identified. At present the existing system parses the code files saved by the user and finds the required inefficient code scenario. Finally, developers can use the best rules and detectors to detect code-smells on any new system to evaluate. Software testing code generation scenario is not applied. Since there is no application with this feature to better software development, this project identifies that and helps to solve the problem through the application. Testing for functional correctness of Dual Driven Software Tool (DDF) such as stand-alone GUI and Web-based applications is critical to many organizations.

These applications share several important characteristics. Both are particularly challenging to test because users can invoke many different sequences of events that affect application behavior. DDF research has shown that existing conventional testing techniques do not apply to either GUIs or Web applications, primarily because the number of permutations of input events leads to a large number of states, and for adequate testing, an event may need to be tested in many of these states, thus requiring a large number of test cases (each represented as an event sequence).

Researchers have developed several models for automated GUI testing and Web application testing; despite the above similarities of GUI and Web applications, all of the efforts to address their common testing problems have been made separately due to two reasons. First is the challenge of coming up with a single model of these applications that adequately captures their event-driven nature, yet abstracts away elements that are not important for functional testing. The absence of such a model has prevented the development of shared testing techniques and

algorithms that may be used to test both classes of applications. It has also prevented the development of a shared set of metrics that may be used to evaluate the test results of these types of applications. Second is the unavailability of subject applications and tools for researchers.

- The generic prioritization criterion is applicable to both GUI and Web applications.
- The new system provides a criterion that gives priority to all pairs of event interactions did well for GUI and Web applications
- The new system also provides another criterion that gives priority to the smallest number of parameter value settings did poorly for both. It shows the application is generic for both GUI and web applications.
- The results show that GUI and Web-based applications, when recast using the model, showed similar behavior, reinforcing the belief that these classes of applications should be modeled together.

A. Add GUI application folder

In this section, a GUI application id, name and folder path is keyed in and stored in the database. This record will be used to retrieve the files and extracting objects and event information from the files. If the .cs file is chosen from the open file dialog control, then the associated .designer.cs file also copied from the original location to 'TestFolders' folder in the project root folder.

B. Find defects

In this section, the folder is selected. Then all the C# files are taken and which class contains no functions, one function, multiple functions, long parameters function and functions calling the functions of other classes are found out and displayed separately. Various tab pages are provided for no functions, one function, multiple functions, long parameters function and functions calling the functions of other classes are provided in a tab control with text box control for output details.

C. Objects controls, events extraction

When the GUI file is selected, then the file is read using the stream reader object and the controls added in the form are extracted from .designer.cs file. Then the event handlers are extracted and also checked whether the coding for the event is written in the .cs file. All the objects properties if changed in that coding, they are also gathered and stored. Likewise, the web files are extracted in the same manner.

D. Setting parameters and values

In this section after the objects, their properties and events are extracted, then the properties are treated as parameters, their values are treated as values here and listed out. These details are converted into XML file which is used for test suite module.

E. Test suite

In this section, the GUI file and web file is selected, then the validation code for empty data in the controls are generated and displayed in the text box controls. The test case generation is also being done and saved in the file which will save in the bin\Debug folder of the project. The validator control tags are also created for all ASP.Net validator controls.

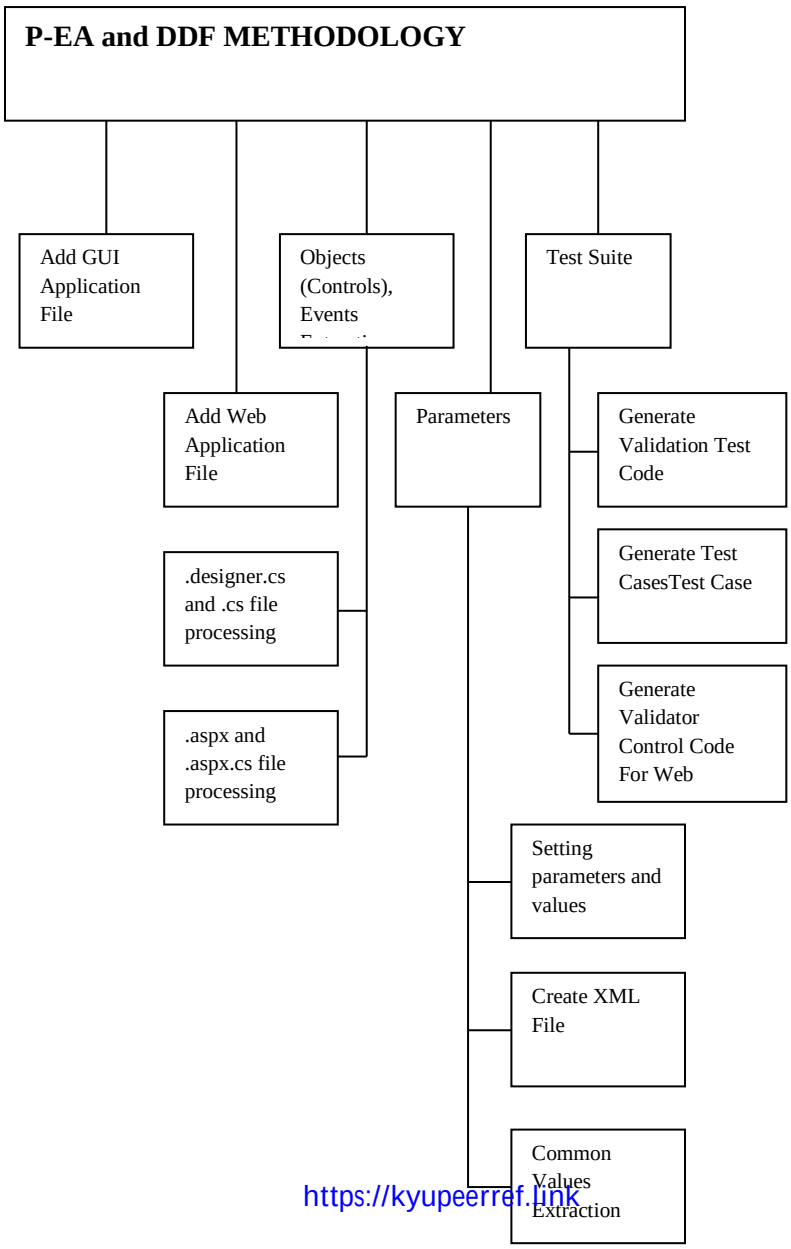
P-EA and DDF METHODOLOGY
Add GUI Application File
Objects (Controls), Events
Test Suite
Add Web Application File
Parameters
Generate Validation Test Code
Generate Test CasesTest Case
.designer.cs and .cs file processing
Generate Validator Control Code For Web
.aspx and .aspx.cs file processing
Setting parameters and values
Create XML File
Common Values Extraction

Fig 3.1 P-EA and DDF METHODOLOGY

IV. CONCLUSION

This work solves the problems of finding defects in programming constructs in OOPS designing level. For that, the projects finds zero, one and multiple function written classes are found out. Also long parameter list functions are also tracked. Function calling most of the functions of other classes are also found out. In addition, the validation code generation is also taken place. Like wise, the testing case need to be generated by test engineers also generated here so that their time is saved. The application is tested well so that the end users use this software for their whole operations. It is believed that almost all the system objectives that have been planned at the commencements of the software development have been net with and the implementation process of the project is completed.

A trial run of the system has been made and is giving good results the procedures for processing is simple and regular order. The process of preparing plans been missed out which might be considered for further modification of the application. The following enhancements are should be in future.

- ✓ The application if developed as web services, then many applications can make use of the records.
- ✓ It should validate our approach with additional code-smell types in order to conclude about the general applicability of our methodology.
- ✓ May extend the approach by automating the correction of code-smells, i.e., suggesting what the corrections to be made in the code.
- ✓ Cross project level code defect analysis can be carried out.

REFERENCES

1. *Salto and E. Alba, "Designing heterogeneous distributed Gas by efficiently self-adapting the migration period," Appl. Intell. vol. 36, no. 4, pp. 800–808, 2012.*
2. *M. Kessentini, W. Kessentini, H. A. Sahraoui, M. Boukadoum, and A. Ouni, "Design defects detection and correction by example," in Proc. IEEE 19th Int. Conf. Program Comprehension , 2011, pp. 81–90.*
3. *M. Harman, "The current state and future of search based soft-ware engineering," in Proc. Future Softw. Eng. , 2007, pp. 342–357.*
4. *Ren, M. Harman, M. Di Penta, "Cooperative co-evolutionary opti-mization of Software project staff assignments and job scheduling," in Proc. Int. Symp. Search-Based Softw. Eng., 2011, pp. 127–141.*
5. *E. Eiben and S. K. Smit, "Parameter tuning for configuring and analyzing evolutionary algorithms,"*
6. *Swarm Evol. Comput., vol. 1,no. 1, pp. 19–31, 2011.*
7. *K. Dhambri, H. A. Sahraoui, and P. Poulin, "Visual detection of design anomalies," in Proc. Int. Softw. Maintenance Reeng., 2008,pp. 279–283.*
8. *H. Hou and G. Dozier, "An evaluation of negative selection algo-rithm with constraint-based detectors," in Proc. 44th Annu. ACM Southeast Regional Conf., 2006, pp. 134–139.*
9. *Li and X. Yao, "Cooperatively coevolving particle swarms for large scale optimization," IEEE Trans. Evol. Comput., vol. 16, no. 2,pp. 210–224, Apr. 2012.*
10. *Karafotias, M. Hoogendoorn, and A. E. Eiben, "Why parameter control mechanisms should be benchmarked against random var-iation," in Proc. IEEE Congr. Evol. Comput., 2013, pp. 349–355.*
11. *Ferrucci, M. Harman, J. Ren, F. Sarro, "Not going to take this anymore: Multi-objective overtime planning for software engi-neering projects," in Proc. Int. Conf. Softw. Eng., 2013, pp. 462–471.*