EXPLOITING SIMILARITY PATTERNS TO BUILD GENERIC TEST CASE TEMPLATES FOR SOFTWARE PRODUCT LINE TESTING

SURIYA PRIYA R ASAITHAMBI

NATIONAL UNIVERSITY OF SINGAPORE

2014

EXPLOITING SIMILARITY PATTERNS TO BUILD GENERIC TEST CASE TEMPLATES FOR SOFTWARE PRODUCT LINE TESTING

SURIYA PRIYA R ASAITHAMBI

M.Eng (CS- Distinction), National Institute of Technology, India

B.Eng (CS), Bharathidasan Unviversity, India

A THESIS SUBMITTED FOR THE
DEGREE OF PHILOSHOPHY
DEPARTMENT OF COMPUTER SCIENCE
NATIONAL UNIVERSITY OF SINGAPORE

JULY 2014

DECLARATION

I hereby declare that the thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.

SURIYA PRIYA R ASAITHAMBI
9 DECEMEBR, 2014

Acknowledgement

I take this opportunity to express my heartfelt thankfulness and gratitude to my research supervisor Prof. Stan Jarzabek. He introduced me to software product lines and taught me key things I needed to learn in the field of software reuse and software engineering that equipped me in pursuing this research. His mentorship, wisdom and kindness have been my source of inspiration. The academic writing and research guidance he imparted will always guide my future endeavours.

My profound thanks to the members of my thesis panel of experts Prof Khoo Siau Cheng, Prof Dong Jin Song and Prof Abhik Roychoudhury for their valuable advice and directions during various stages of my research work.

I thank all the professors, faculty and teaching staff of SoC for sharing their wisdom and knowledge during my course work as well as during my research. I also wish to record my thanks to the administrative staff members of the SoC graduate office for their kind support in various aspects of my candidature.

I wish to thank my employers ISS, the management and staff for their support and encouragement for pursuing my research ambitions. I wish to thank Dr. Venkat Ramanathan for his help in carrying out editorial review of my thesis and for his constructive comments.

I thank all my peer researchers at School of Computing for lightening my PhD years with positive words of encouragement and sharing of ideas. I thank the anonymous reviewers of my research publications for their valuable technical comments, pointers and encouraging feedback which helped me shape my PhD research work.

Finally I thank my family - my parents, mother-in-law, husband, sister and brother for being there for me at good and as well as challenging times. Importantly, I thank my son. His boundless affection gives a purpose to my life and strength to thrive this research journey with enthusiasm and peace.

Table of Contents

Summary

Software product line testing (SPLT) is more complicated than the conventional testing. Since software product lines consist of several product variants, there arises a need to test each variant thereby causing test case explosion. In this thesis we studied Android OS product line test libraries to understand the combinatorial test explosion problem. Our study reveals frequent occurrences of test code fragments which we call “test clones”. As new product variants are added to SPL, test cases from existing products are copied and modified. This leads to test clones and problems of managing large test libraries with many redundancies. In this thesis, we propose a method to avoid test clones and therefore save effort of developing and maintaining SPL test libraries.

A study of existing literature reveals that while some attempts have been made to address the test case explosion issue, most of these are heuristics, combinatorial selection or model based approaches which have known limitations when it comes to variability and heterogeneity prevalent in the software product line executable test libraries. The approach proposed in this thesis solves the problem in a way that is effective (any type of test clones can be tackled) and practical (any test library can be addressed irrespective of programming platform).

The proposed approach is based on test case reuse facilitated by test templates. Our approach constructs test libraries using templates that represent groups of similar test cases in generic adaptable form. The ***Generic Adaptive Test Template (GATT)*** structure proposed in this thesis takes advantage of common aspects and predicted variability that are present among individual test cases. The process starts with detection and grouping of test clones, provisioning for variability and then constructing hierarchical templates. Subsequently, the process provides specifications to derive the test library by binding variant points with appropriate

variant choices. This compile-time test template approach helps in test construction by adaptive generation without affecting the follow up test execution. The proposed template-based design and implementation approach helps the test engineers to handle key challenges namely variability, redundancy and heterogeneity in large scale test libraries.

The results of the experiments conducted on Android OS test libraries demonstrate that a compressed, normalized, non-redundant test library can be achieved using our proposed approach. The results also confirm our hypothesis that test library construction using template-based approach will facilitate scalability in test evolution and improve test designers' productivity.

The contributions made by this thesis is expected to create insights with reference to usefulness of generic test case template approach, which in addition to being beneficial to software product line industry would be a seed that would foster further research in this fertile area.

List of Tables

List of Figures

CHAPTER 1
Introduction

In this chapter we set the prelude for our thesis. We discuss the problems in existing software product line testing and in particular the issues pertaining to test libraries. These challenges provide the motivation for our research on developing a non-redundant representation of test libraries.

1.1. Background

Gartner special report[1] describes the *emerging technology trends* consisting of social interaction solutions, mobile computing, cloud computing and information via big-data as the "nexus of forces" that will empower organizations to drive future digital workplaces. To exploit new opportunities provided by these latest technologies software engineering practices are constantly evolving. A quick look at the practices behind emerging technologies like the mobile computing reveals the increasing demand for individualization and hence there is a constant need for augmenting software engineering practices directed towards software product line. "***Software Product Line Engineering (SPLE)*** *is a set of software-intensive systems that share a common, managed set of features satisfying the specific needs of a particular market segment or mission and are developed from a common set of core assets in a prescribed way*" [125] . Testing in a product line scenario must examine both reusable core assets (termed as domain testing) and individual products (termed as application testing).

Northrop and Clements [115] observe that increased adoption of SPLE practices in industry has yielded good results in the form of reduced implementation costs,

[1] http://www.gartner.com/technology/research/nexus-of-forces/ Last Retrieved June 2014

shorter time to market and improved quality of the derived products. However, to derive the full benefits, it is not only important for software development to be carried out rapidly, but the developed software should be rapidly tested as well; else the effort put in software development becomes sub-optimal since the products cannot be released to users.

Software Product Line Testing (SPLT) verifies and validates the confidence that any instance of the product line will operate correctly. By using managed reuse techniques product lines take advantage of feature similarities. It hence becomes important to focus on SPLT to explore for improved approaches that can contribute towards faster launching of products and their various versions. SPLT is complex because of the need to test a very large set of variations and the feature combinatorics. SPLT processes produce test artefacts that can be further classified as non-executable test artefacts (such as test plans, test model, test strategies and test reports) and executable test artefacts (such as test cases, test data sets and test scripts) as classified by Myers et al [110]. We shall refer to such executable test artefacts as test libraries hereon.

In SPLT context of reuse, Knauber et al [84] raise the following research questions encompassing both domain and application testing:

1) How to design generic reusable test cases for different testing levels, namely unit testing, integration testing and system testing?

2) How to create non-redundant representation of test libraries that positively influences the quality properties such as reliability, maintainability and testability?

3) How to increase efficiency and effectiveness of testing efforts?

In order to achieve the overall product-line goals, namely increased reuse and reduced cycle time for testing, improvements/enhancements to existing traditional testing mechanisms are required. We had mentioned above that the SPLE

leverages on “reuse” to accomplish rapid software development. Taking a cue from this we believe that such reuse approach should be embraced in software product line testing as well. Hence one of the key improvements to software product line testing would be to accomplish non-redundant test libraries with effective reusability of common test libraries and variability management. This would help in faster identification of product specific defects, which in turn would result in increased throughput in testing process, more traceability and efficient resource utilization.

1.2. Motivation

In a systematic mapping study Emilie Engström [51] states: “*Three main challenges concerning SPLT are: (i) the large number of tests, (ii) balance between testing effort for reusable components and concrete products, and (iii) handling variability in testing artefacts”*. According to McGregor [99], the key challenges while testing a product line with higher levels of reusability are variability, emergent behaviour, creation and management of reusable components. From our study on large scale test libraries it is observed that test assets are composed of multiple programming languages, configuration techniques and scripting procedures. The need for variability management among software product line test libraries is exacerbated by two key orthogonal aspects. (1) Cost of testing: Test case generation through a systemic template based variability adaptation mechanism is an important means for keeping the cost of the testing low, while guaranteeing adequate degree of dependability. (2) Testing the various layers: the test case selection, interoperability and heterogeneity problems arising due to variations in platform frameworks, operating system, kernel and hardware layers causes bugs (as reported in al [3, 4]). This remark highlights the need for variant adaptation mechanism towards product variants in different layers. Thus focusing into new research approaches for SPLT artefacts would be timely and lucrative.

1.2.1. Challenges

As product features grow, the product variations also grow and with the growth in variations, there is a corresponding growth in binding options for each of these variations. Consequently, the challenges get more complicated in SPLT due to the need for testing individual features, their respective variations and bindings. We describe some of the key challenges related to SPLT below:

1.2.1.1. Voluminous of Test Libraries

The key reason for increase in volume for SPLT is the presence of variability. As variability grows, the number of test cases needed to verify all variant points along with their respective choices increases. Typical industry product lines may consist of thousands of variable features [18]. Moreover, due to continuous evolution of projects, features get added, modified and removed over time. Key issues arising out of such continuous evolution are: (i) poor visibility of changes that have been applied to test libraries over time and (ii) the lack of generic adaptable representations to facilitate in elegant evolution. Due to this test cases get duplicates and volume increases. These problems need scientific investigation with the view to proposing an effective methodology as a solution that would address the stated issues.

1.2.1.2. Reusability in Test Libraries

The second challenge lies in designing reusable components and using them as part of test libraries. If test libraries are not well-designed, redundancy builds up over a period of time and makes test case library maintenance difficult. Testing the product line involves testing various combinations of product against the specified feature variants. To address these variations productively software engineering practices normally resort to reusability. In testing context, this may require design of generic adaptable test components. But designing reusable components is a complex and time consuming job. Thus test engineers are

challenged to balance their efforts between creating reusable tests vs creating product specific tests.

1.2.1.3. Variability Management

The breadth of the variability that must be accommodated in the product line directly impacts the testing efforts needed to adequately ensure product quality. This is a key issue that needs to be addressed. The creation and maintenance of generic test libraries in domain testing that will be subsequently reused in application testing accommodating all expected variations is a stated research problem [126]. In SPL testing context, variability is expressed as variation points at different levels with different types of interdependencies. According to Kolb [85], one of the major risk factors in testing of product lines is the verification of individual variant point to appropriate binding choices. This makes it necessary to test all the variant points and their appropriate binding choices alongside regular feature testing. Also a simple variant binding can happen at many stages e.g., at domain testing stage for one product and at application testing stage for another product. Hence in SPLT context, products can be built from core assets in many different ways, both domain and application testing require "management of variability".

1.2.1.4. Heterogeneity of Test Libraries

Contexts such as multiple programming languages, diverse OS platforms and multifarious devices cause heterogeneity in an SPLT environment. Traditional structured SPLT techniques (and associated coverage criteria) currently target a single programming language. With the advent of assorted computing trends in newer technologies, we need to be more inclusive in adopting techniques that can manage multiple programming languages and heterogeneous platforms. As the complexity and intelligent features of product line devices increase, the need for collaboration among different vendors, operating systems, versions and firmware/software/hardware components also escalates. Often such heterogeneity

causes serious incompatibilities necessitating a robust SPLT approach. Thus, seamless integration of test case management strategies targeting libraries comprising of different programming languages, test mock data objects, tools and techniques has become a crucial requirement for SPLT, which the research community cannot afford to ignore.

From the descriptions in the above sub-sections it is evident that non-redundancy, reusability, variability and heterogeneity are important qualities of well-designed SPL test library.

1.2.2. Existing SPLT Approaches

Extensive research and systematic mapping studies have been conducted in the SPLT literature to investigate and evaluate the state-of-practices [41, 51, 90] . The following are some key alternate approaches for test library creation and management:

- Model Based Testing: These techniques predominantly work with one of the modeling representation, namely, feature models, unified modeling language (UML) or object constraint language (OCL). One approach is to use UML for deriving test models. Kishi and Noda [82, 83], Bertolino Pluto [19-22]and ScenTED [134], Pohl & Metzger [125, 126] , Nebut [111-113] , Duenas [48] and Olimpiew and Gomaa [62, 116-120] have proposed derivation of test models using UML. The UML-based techniques use various algorithms to generate test models using model checking, test class diagrams, test scenario diagrams, activity diagrams, sequence charts, profiles and stereotypes of UML artefacts as guidance. Kang's technique reported in [77, 79, 93] uses feature model that naturally expresses the commonalities and variations among product line features as a tree for guidance. WeiBleder's technique [159] , uses state machines and OCL expressions.

- Formal Specification & Natural Language Based Testing: These techniques predominantly work with natural language or formal specifications. Temesghen Kahsai [77] proposes a framework that evaluates the use of specific test case selection based on formal specification. Bashardoust Tajali [7] use domain models expressed as generated contracts and use that to guide test case generation.

- Aspect Oriented Testing: These techniques predominantly work with aspect based programming constructs and dependency injection concepts. Feng [53] uses aspect oriented approach to generate unit test cases in a product line context. Knauber and Schneider [84] combine aspect oriented programming and unit testing to trace and manage small scale variability among test cases.

Other significant contributions are discussed later in literature survey (Chapter 3). A quick assessment of SPLT research literature and approaches proposed therein reveal the following limitations:

- Model based techniques aid more in test case selection, model based reusability and variability management using stereotypes/model checkers. The test libraries are generally found to be non-executable test artefacts.

- Formal Specification & Natural Language based approaches provide sound mathematical models for verification and validation of test representation. They provide abstract representations (formal notations or natural language) for variability, reusability and control test case explosion.

- Aspect oriented approaches are limited by the capabilities of the hosting container and underlying programming language expressiveness.

It is not possible for any one particular SPLT approach to mitigate all key challenges mentioned earlier. Yet, if a few approaches can work collectively by complementing each other instead of being mutually exclusive, then the chances are that such a mixed approach can better handle test library construction.

With the advent of test-driven-development and advancement in the "nexus of forces", we believe that investigating SPL test libraries in construction and management perspective is an essential research problem for the product line and service testing community. Software engineering principle of generality encourages avoiding repetitions and constructing parameterized, configurable and adaptable generic test libraries as templates that can be reused. This serves as the motivation behind our thesis. Our research work aims to contribute towards effective test library construction and management in software product lines.

1.3. Objective

The main objective of this research study is to devise a comprehensive generic adaptive test template approach for constructing reusable, variability managed, non-redundant, heterogeneous test libraries that would contribute towards improved productivity under the complex, multi-faceted software product line scenario.

The study of research literature and state of the practice in the typical software development industry reveals the following key limitations of existing approaches in the context of software product line testing:

i. Existing approaches are more focused on model based non-executable test libraries and there is very limited research in executable test libraries.

ii. Even in the current research studies where executable test libraries are being studied, there is a lack of study on variability management and evolution in software product line testing context.

iii. Study of similarity patterns in test libraries which is crucial for SPLT has not been sufficiently researched especially in large scale test libraries.

With the advent of test-driven-development and advancements in open source software platforms, we believe that investigating SPL test libraries in maintenance and evolution perspective is an essential research problem to address the gaps and meet the stated objective.

With the above in view, this thesis aims to carry out detailed research on template based variability management as a strategy and derive solutions that would act as an effective enhancement to existing software product line test construction approaches.

1.4. Proposed Solution

In a typical software development lifecycle, testing can take as much as 40% of the development effort [143]. This would include test case construction and maintenance, both of which are time consuming and manpower intensive. Over a period of time, as the software evolves multiple versions get created. In software product lines the presence of variations adds to the increase in versions and test libraries tend to explode in size. We use the term ‘test library maintenance’ to refer to the periodic changes made to the test libraries, possibly due to minor changes in requirements or features in existing products. Likewise we use the term ‘test library evolution’ to refer to long term changes that happen during the whole life span of test libraries due to reasons such as adding of new products to the SPL. For example, consider a situation where a new product is created which is similar to an existing one but with some variations from the original. To test this product a new test case has to be created. Since the new test case has commonality

with the old test case (due to similarity of the products), test designers traditionally makes a copy of the existing test case and modifies it to address the variations.

Typically test designers perform the following key steps to create the new test library for the SPL:

1) Analyse the test requirements defined for the new product.
2) Understand the similarities and differences among the new test library and earlier test library releases.
3) Select the best matching test library release from the configuration repository and use it as a baseline.
4) Customize this selected test library to fully meet the defined test requirements of the new product.
5) Execute the customized test library to validate the new product.

It can be logically inferred from the above *copy-paste-modify* approach that test libraries may have plenty of redundancies. The large presence of redundancies cause the test case explosion issue; i.e., test cases with similar code fragments replicated in variant forms, which we call *test clones*. The presence of redundancies cause hindrance to testing productivity by increasing the effort spent on maintaining these duplicated tests. Therefore in the context of software product families, the ability to achieve non-redundant test libraries would have significant impact on testing productivity. Hence, we propose a template based test construction approach that would mitigate test case explosion in software product line situation. This is the prime focus of our research work.

Existing test library construction techniques use a combination of techniques such as the use of parameterization, test patterns, test model generators and test frameworks to manage variability. The test template approach proposed in this thesis uses a synergetic merging of these existing mechanisms to represent groups

of similar test cases as generic, adaptive test templates that would improve maintainability and impart engineering qualities that are otherwise difficult to achieve. Our template-based approach pivots on flexible object oriented programming (OOP) languages such as Java and C++ to design the core test libraries comprising of test data, test fixtures, test cases, test suites and test oracles. Our study also designs a structure for generic adaptive test template (GATT) built using Adaptive Reuse Technique (ART). While the existing test case codes composed using OOP languages and XML based configuration files expresses the syntax and semantics of the test libraries, GATTs express the syntax and semantics of change.

In our proposed template approach, we identify and unify test clones within test libraries as well as help manage evolutionary changes that regularly occurs across the test library releases over time. The non-redundant template representation that we propose captures changes and their relationship within and across test libraries and provides a way of unifying test clones of any type or granularity. GATT structures are simple, tree-hierarchical, text file based and hence they can easily grow and can remain intact under pressures exerted on the SPL test libraries by multiple changes, and version creations happening over years of evolution.

1.5. Contributions

The proposed approach counters the test libraries explosion problem. The research focuses on effort reduction via systematic reuse of generic test assets. This is achieved by taking advantage of common aspects and predicted variability that are present in test cases. The proposed template based reuse approach organizes test libraries by preserving test case commonalities and provisioning for variant points. The novelty of the proposed approach is that the suggested technique is programming language/platform independent. Key contributions of this thesis are listed below:

- SPLT deals with large voluminous test libraries. First and foremost benefit of the proposed approach is that it can handle combinatorial explosion. The approach constraints test library explosion by only allowing permissible test combinations, while also preserving information relating to code and test traceability.

- The empirical study presented later in Chapter 4 establishes the presence of significant amount of redundancy among test libraries in a SPLT context. This study further uncovers the strong need for research in test clone management approaches. The findings are also expected to promote other SPLT researchers to pursue possible research approaches in the field. The study also contributes to detailed understanding of the redundancy patterns occurring in test libraries by identifying the various *types of test clones* and provides adequate examples to formalize the redundancy patterns in a practical context. Based on this, the thesis has constructed formal definitions and taxonomies for test clones present in test libraries, which is a contribution to the SPLT theories.

- The thesis also proposes and defines useful SPLT metrics that can be used in assessing quality of test library construction and management. These metrics also aid in scientifically comparing different test library construction approaches.

- The core contribution of this thesis is the formulation of a new approach for template-based representation of test libraries which we call STRAT. This approach and derived templates are capable of generating various feature combinations and versions based on test designer's choices. The proposed approach provides a new framework to identify the redundant elements, variability requirements and other feature specific details for creation of templates. Adopting this approach is expected to yield productivity gains for SPLT.

- Finally the thesis demonstrates the use of the STRAT approach by constructing working test templates for software product line test libraries where redundancy was found to be significant. The case study used the above test templates to further generate test libraries to validate the proposed STRAT approach. In addition, the generated test library was compared to the original test library for the purpose of establishing the benefits derived using STRAT approach. The results of this comparison show that using our proposed approach yields significant improvements in test library reuse.

The output produced as part of this thesis has contributed to research literature. The research hypothesis was published in SPLASH 2012 conference proceedings [5] under doctoral student research section. We received constructive suggestions for improvements and we carefully implemented them in further research studies presented in this thesis. The research outcomes of the study conducted on large scale Android platform test libraries were published in the International Conference on Software Reuse - ICSR 2013 proceedings [6].

In summary, the key contribution of our research work is the simplification that comes from non-redundancy accomplished through reduction in both test libraries size and its conceptual complexity. The study also has evaluated the benefits and trade-offs of working with non-redundant test templates.

1.6. Thesis Organization

- Chapter 2 provides a summarized view of the entire research work carried out as part of this PhD research which are presented in detail in various subsequent chapters.
- Chapter 3 discusses the existing research literature in test library context; comprehensively covering all related aspects namely general software testing, software product line testing and android platform specific testing.

- Chapter 4 describes the results of similarity analysis performed on a typical software product line with Android platform framework project's test libraries as example.
- Chapter 5 carries out an in-depth analysis of test software to define test clones and formulate appropriate taxonomies, granularities and metrics related to test clones. The chapter also illustrates these definitions with the support of examples.
- Chapter 6 proposes a new approach for test clone management which we call as Systemic Template based Reuse Approach for Large Scale Test Libraries (STRAT). The chapter also elaborates the variability management and template creation techniques involved in the proposed STRAT approach.
- Chapter 7 provides an illustrative example to demonstrate the STRAT approach described in the Chapter 6. It also details the experimental analysis to assess the gains derived through the template-based approach as well as to identify the constraints and trade-offs involved.
- Chapter 8 presents the thesis conclusions and identifies possible future areas of follow up research work.

CHAPTER 2
Overview of the Research Work

In this chapter, we provide a bird's eye view of the research work carried out as part of this PhD thesis. We set the stage by discussing relevant concepts, problems in test library construction and management, and then provide an overview of the proposed approach along with key results.

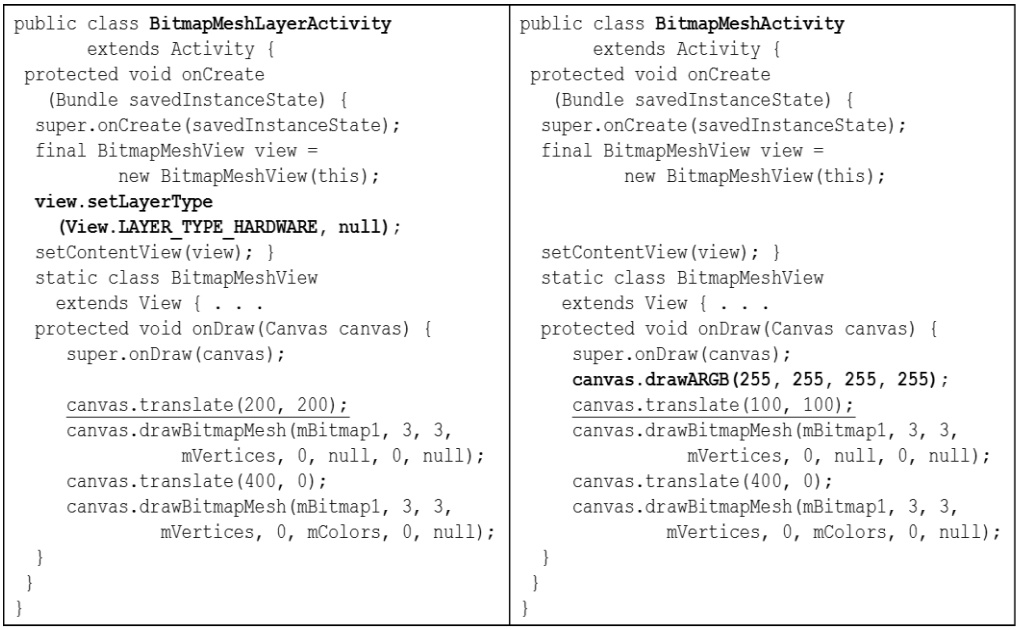
The key idea behind our reuse based generic design approach is to construct *adaptive test templates* that will provide flexibility and variability management to the entire product line. To achieve flexibility and variability management, test structures must be reused for both common and for variable portions by explicit specification mechanisms. Generic templates foster this idea by creating unified non-redundant structures and reusing them with modifications for product-specific and version-specific variant bindings.

Unlike generics programming techniques applied to traditional executable test cases, the generic adaptive test template approach aims to cater for unrestricted test case parameterization. In the proposed approach, test designers need to identify and capture only the commonalities among test clones in a product line setting. Variations can be specified as *deltas* and can be kept separate from the commonalities based generic representations. This makes the test structures concise and non-repetitive. The existing test library (in terms of its component structure) remains an integral part of the proposed solution. Any future changes imposed on the test library can be done via template structures to ensure that the test library and its template extensions are in sync with one another. Over a period of time such templates allow the test library to improve its changeability, ease of maintenance, reusability and reduced risk of anomalies.

This research work attempts to validate and demonstrate the above proposition using scientific methods. In order to execute the above research, we adopted a research methodology wherein we organized the work into phases. It is first important to establish that redundancies are present in significant numbers in large scale test libraries; this would form the motivation for exploring a generic adaptive solution. Then the research has to establish a nomenclature to classify and categorize the redundancies in terms of test clone patterns. Based on the established research motivation and using the theoretical formulations, the research work should propose a reuse-based template approach that enhances and complements existing test construction techniques. Then the research needs establish the practicality of the proposed technique through an illustrative case study. Finally, the implementation has to be validated for effectiveness and quantify the improvements offered in comparison to traditional techniques. A quick summary of each of the above research phases are presented here as an overview and elaborated in Chapters 3 through 7.

2.1. Motivational Example

Our analysis of Android test libraries reveal substantial redundancies among test cases within a test library for single Android version, as well as across these test libraries for different versions. To understand the nature of such redundant codes (test clones) consider two test clones `BitmapMeshLayerActivity` and `BitmapMeshActivity` shown in Figure 2-1. These are part of test cases that tests the *bitmap mesh* functionality in two Android product features. The partial codes for test methods `onCreate()` and `onDraw()` are provided in the left and right column of the figure.



```
public class BitmapMeshLayerActivity
       extends Activity {
 protected void onCreate
   (Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  final BitmapMeshView view =
         new BitmapMeshView(this);
  view.setLayerType
    (View.LAYER_TYPE_HARDWARE, null);
  setContentView(view); }
  static class BitmapMeshView
    extends View { . . .
  protected void onDraw(Canvas canvas) {
     super.onDraw(canvas);

     canvas.translate(200, 200);
     canvas.drawBitmapMesh(mBitmap1, 3, 3,
                mVertices, 0, null, 0, null);
     canvas.translate(400, 0);
     canvas.drawBitmapMesh(mBitmap1, 3, 3,
              mVertices, 0, mColors, 0, null);
  }
 }
}
```

```
public class BitmapMeshActivity
       extends Activity {
 protected void onCreate
   (Bundle savedInstanceState) {
  super.onCreate(savedInstanceState);
  final BitmapMeshView view =
         new BitmapMeshView(this);


  setContentView(view); }
  static class BitmapMeshView
    extends View { . . .
  protected void onDraw(Canvas canvas) {
     super.onDraw(canvas);
     canvas.drawARGB(255, 255, 255, 255);
     canvas.translate(100, 100);
     canvas.drawBitmapMesh(mBitmap1, 3, 3,
                mVertices, 0, null, 0, null);
     canvas.translate(400, 0);
     canvas.drawBitmapMesh(mBitmap1, 3, 3,
              mVertices, 0, mColors, 0, null);
  }
 }
}
```

Normal Text: identical; Underlined Text: Parametric Clones; **Bold Text: Non-Parametric Test Clones;**

Figure 2-1 Simple Test Clone Example

Test case codes shown in regular font depict exact duplication, while variations are either underlined or in bold font. Underlined text indicates test case parametric variations that can be handled using traditional programming such as generics[29]. Whereas the bold text refers to complex variations such as different API/method calls, partial names and other gapped test clones whose handling may not be feasible using traditional programming constructs. Similarly, more such redundancies are observed across successive versions of test libraries.

This example is a typical illustration of the copy-paste-modify approach adopted by test designers when they construct new test cases using existing test cases as basis. Needless to say, such duplication would result in difficulty in test code maintenance in the long run.

The motivation for our proposed research is born from the existence of such redundancies in large scale test libraries of software product lines. It can be inferred that the presence of redundancies inflate the size of test libraries. Hence, any reduction in test library size decreases both the overhead of maintenance and the number of test cases that must be rerun after changes are made to the software

product line. A good test libraries management approach or technique would facilitate both in the ease of designing new test cases as well as in eliminating unnecessary test cases. Presence of redundancies impairs the understandability and maintainability of test libraries. Further during test library maintenance, knowledge of similarity is essential to perform consistent changes. Thus, a clear awareness of recurring code portions leads to better test library development and reduced evolution efforts.

2.2. Study of Redundancies

As a first step it is important to establish the extent of redundancies within and across large scale SPL test libraries and then nature of redundancies. For this purpose, we chose to study the Android test libraries since it is a large scale open source project with executable test libraries having the features of a typical SPL[25] and hence meets our research focus. Android platform is a software stack for mobile devices that includes an operating system, relevant middleware and key applications. Moreover, Android platform is a "Code-line"[2] and exhibits the key characteristics of a typical software product line.

Additionally, being a well-built platform with suitable test automation frameworks, Android systematically hosts' more than 500 test libraries (as projects) in its main source repository. We studied a large slice of these test libraries to identify, understand and classify the nature of redundancies inside these test libraries. Android test libraries are structured hierarchical using test projects, source/configuration folders and test files. We used our lab's internal code clone tool, namely Clone Miner and Clone Visualizer along with external ccfinder tool to gain an initial understanding. Subsequently we carried out systematic code analysis to determine the nature of redundancies. Our

[2] http://source.android.com/source/code-lines.html

investigations provided clues regarding possible root causes for redundancies, i.e., test clones. One of the reasons for test clones in Android test libraries is due to the platform heterogeneity caused by (i) the need to the support diverse hosting environment such as Dalvik VM and (ii) the Linux kernel and the use of multiple programming languages like Java, C++ along with configurations through XML and text based properties. Additionally, the absence of variability mechanisms, results in test code replications. Current practice is that each vendor preserves their variations as separate test projects.

We made a quantitative analysis to estimate the volume of redundancies. Our studies indicate that around half of existing test case files (53%) contain some form of redundancy and around three-fourths of test methods (79%) exhibit some form of simple test clone similarities. In addition, the similarities found varied from simple test code snippets to higher level test structures spread across files. Test clones vary from as small as 30 tokens to as large as 1290 tokens. Average test clone length was computed as 53 tokens. Although a majority of redundant test case codes were either identical or parametric in nature, our study also uncovered instances of complex structural similarities.

In summary, this study apart from establishing the presence of high level of redundancy has also helped identify the existence of numerous types of test clones; an in depth understanding of these clone patterns is required so that our proposed template construction technique could address all types of test clones. The impact of test clones on test libraries maintenance is significant in terms of affecting testing productivity (as described in next section); therefore requires to be addressed.

Some of the above challenges can be addressed by traditional techniques using existing generic programming techniques such as parameterization, aspect oriented programming (AOP) and test design patterns. Generic parameterization approaches are used over types and variables. Iterators and collection libraries

separate out storage and traversal perspectives. AOP techniques are used to achieve separation of concerns by isolating aspects. Test design patterns welds together pre-existing exact test clones into a fixed patterns of functional components. Despite the availability of such generic programming techniques we find that these will not eliminate all types of redundancies found in the large scale software product line like the Android platform. Since we need to collectively counter multiple inducers such as heterogeneity and large volume which is only possible through a seamless technique that supports variability management and facilitates reusability.

These above reinforce the demand for a complementary template approach from a generic design perspective and hence motivate us to devise a suitable research solution.

2.3. Impact of Test Clones on Test Library Maintenance

Prior to describing the specifics of the proposed solution, we need to understand "*test clones*", a term that we have coined in this thesis drawing inspiration from the popularly understood "*code clones*". Whereas the *code clone* may refer to duplicate code fragments in software applications, *test clone* refer to duplicate code fragments in test libraries.

By test clones we mean group of test code fragments that are exactly similar or test code fragments that are similar to a large extent. For further understanding of test clones, we have conducted a thorough analysis as part of this research study and have come with formal definitions for various types of test clones typically found in test libraries. As per our classifications, test code fragments that are similar with minor parametric variations are called *simple test clones*. Likewise, test code fragments that have complex non-parametric or non-type variations with larger-granularity (higher level syntactical structure) are called *structural test*

clones. A more detailed technical description of various types of test clones is presented in Chapter 5.

Eliminating test clones benefits a test library in the following positive ways:

- Reduces maintenance efforts because it is easier to make changes across similar groups of test assets.
- Improves clarity while designing new test cases since unified test clones can be presented as shared common utility components within the test libraries.
- Enhances reusability via common features that can be used for consistently testing every product release.
- Optimizes resource utilization by carefully grouping expensive resource intensive test components such as test fixtures, test data, test stubs and automated test oracles.

In summary, eliminating or minimizing test clones helps us in improving design generality that promotes larger granular reuse, effort savings and improved quality attributes.

2.4. Generic Design Approaches

As mentioned in previous section, SPL is particularly prone to test clones. This warrants an in-depth research study on test clone patterns to come up with a simplified generic design and implementation approach. The aim of such generic design is to unify differences among test clones and represent a group of such clones in a unique, generic form.

The extent to which the test clones are present in a particular test library would reflect a failure of the existing design approach to fully exploit the potentials of generic design. From the study conducted on large scale test libraries, we notice that there is a variety of test clones that contain similarities at various levels such

as method level, file level, directory level, design level and test pattern level. The occurrence of test clones at diverse levels in a test library further complicates the test library construction and test management efforts. Before going into finer details of generic design, we outline the three key engineering benefits that a generic design has to offer:

1) Generic design promotes test library reusability. Generic design aims to unify redundancies found across test cases, test data and test processes.

2) Generic design facilitate test library understanding. By capturing redundant test structures at a single place, the generic template also captures the variability details. Hence it becomes easy for the tester to understand the test code and data spatiality form the unified generic forms.

3) Generic design reduces the number of distinct conceptual elements that a test designer has to deal with. Test library construction process depends on concepts from the underlying domain engineering. Generic design further unifies redundant test structures that could be part of either code or design level similarities. Thus such unification will reduce the overall individual conceptual element in comparison to those present in redundant representations.

Generic design approach for test libraries is a well-researched area. Several authors have proposed approaches for addressing redundancy issues.

- Parameterization Testing Approaches. There are a variety of parameterization techniques ranging from simply sending different primitive data values as parameters to sending objects as parameters to set values in configuration files. Effective parameterization allows test engineers to unify test code structures (such as methods or classes). However parameterization is limited by generality offered by the underlying programming language (Fraser et al [55, 56]).

- Aspect-oriented Testing Approaches. Aspect represents a cross-cutting concern. Aspect injections evolve from underlying testing framework and domain concepts. But when implementing testing as a concern, there are known difficulties since the mechanisms offered by the aspect languages are not always sufficient to capture all types of concerns that were encountered (Marin et al [97]) .

- Test Pattern Approaches: The use of test patterns removes test smells thereby further standardizing the test library. Consequently, test libraries built using such standard test patterns display lots of similarity. Despite the benefits from using test patterns driven design, test libraries still suffer from maintenance complexity. This is because test patterns force consistent repetitions between test components and thereby causing design level redundancy (Fraser et al [55, 56]).

To overcome the above listed limitations, we propose the use of generic test templates that would complement the existing generic design approaches. The following list explains how our template based approach proposes to address the generality:

- Generic templates use a *representative mechanism* that can compact test code fragments irrespective of the underlying programming language, by treating each of the fragments as pure text.

- Generic templates offer parameterization and control constructs that can customize test code fragments at various granularity levels and to provision for variations in a test library.

- Generic templates separate the product line variability concerns and version-based changes into a meta-layer from the test libraries thereby providing clarity and visibility for impact analysis during maintenance.

- Generic templates offer adaptive constructs that can customize the administration of scattered test clones. Such unifications can alleviate some of the combinatorial explosion of test libraries.
- Generic templates provide multi-level hierarchical representation (fragment, file or tree structures) that makes the construction flexible and scalable in any large scale test libraries context where horizontal and vertical growth is anticipated.

Thus by providing a suitable test construction mechanism we focus on a smaller base of non-redundant generic test templates; at the same time we preserve feature combinations and variant dimensions inside the templates. Test libraries can be generated based on test engineer's choice specifications and by binding these choices at compile-time using a suitable template processor.

2.5. Preview of Proposed Solution

The above discussion highlights the motivation for the generic design approaches in the light of redundancies that are present in the form of test clones. The large variety in types of test clones further highlight that traditional approaches do not address all the challenges to improve management and maintenance of test libraries. In this section we first highlight the context of where our proposed solution fits in and then provide an overview of the solution.

2.5.1. Context

Test libraries store and manage not only test cases but also other sharable components such as test data, test mocks, test stubs and test configurations. Figure 2-2 illustrates a typical product line environment where the *domain testing process* involves domain requirements, domain test models, domain test scenarios and domain test libraries. The *application testing process* will consider application specific requirements, feature models and application specific test libraries. Feature model is a compact representation of all the products of the

product line in terms of features. The domain of our research focus is shown in black dotted box in the Figure 2-2.

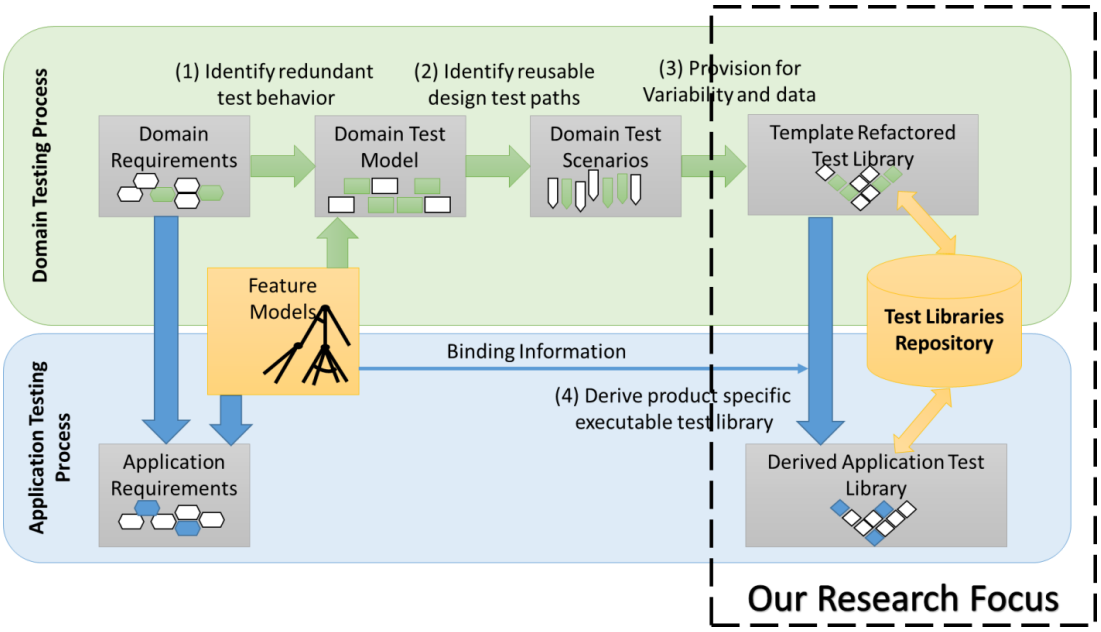


Figure 2-2 Two Testing Processing Layers in Software Product Lines

Test library management problems require suitable generic design techniques that address key problems. Among the many problems two are particularly acute; namely, explosion of look-alike test structures and poor visibility of evolutionary changes across multiple versions. These necessitate generic design techniques that would meet requirements such as:

- A concise grammar for non-redundant representations.
- Ability to support different testing levels that cater to the different features and their variations.
- A balanced approach for variability management that improves overall quality attributes of the software.

Our proposed approach uses template-based refactoring of test libraries that closely influences domain and application testing in a product line.

2.6. The Proposed Reuse-Based Approach for Test Libraries

This thesis proposes a **S**ystemic **T**emplate based **R**euse **A**pproach for Large Scale **T**est Libraries (STRAT) that aims to address the issues discussed above. Test processes share common steps frequently. The approach aims to identify such shared test code fragments using test clone detection techniques. After test clone identification, the approach aims to capture the essential commonality as isolated non-redundant template frames and template fragments. These become the reusable building blocks for the approach using which the templates are evolved by provisioning for variant points and binding options. Thus version based change management and long term test library evolution support are addressed through these templates designed as part of STRAT.

Thus STRAT works in two layers; one layer for the existing executable test library and the second higher layer is for complementing templates representations. Since STRAT uses compile time construction approach, respective variant binding for each variant point can be derived based on the test designer's binding and configuration choices. All of the above reasons make the generic adaptive test templates more exhaustive and permits intrinsically detailed variability design.

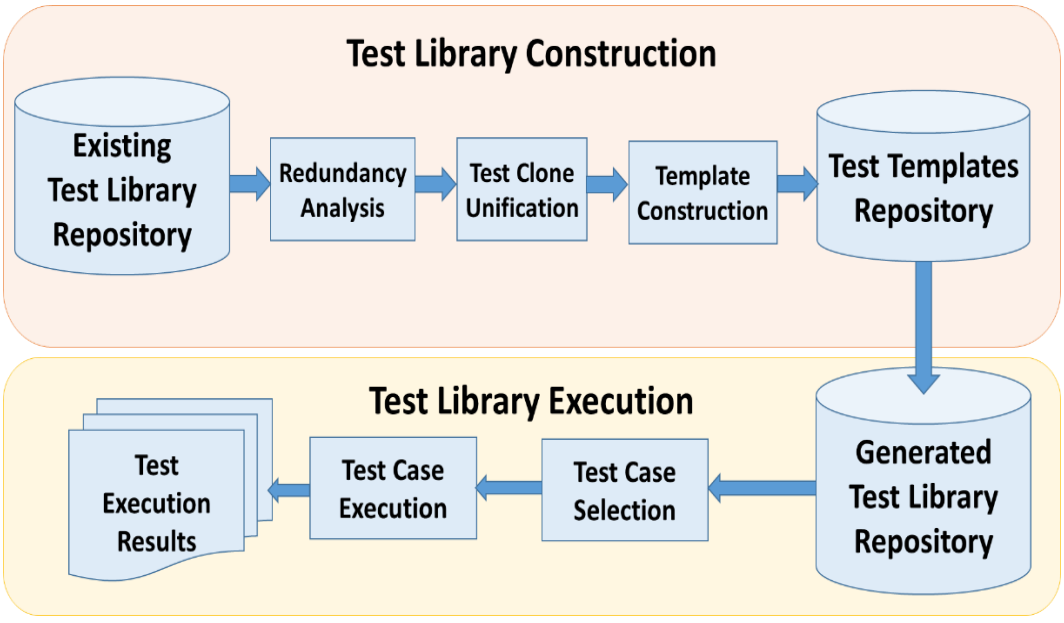


Figure 2-3 Reuse approach for test library construction and management

Figure 2-3 illustrates few key activities involved in the test construction and execution layers. In our approach, activities related to test template refactoring, composition by adaptations and augmentation is called *test library construction layer* (this also includes planning and design of test templates). *Test library execution layer* comprises of activities related to template binding choices, derivation of executable test libraries and actual test execution. Further, template file types are classified into specification files and configuration files. Both template files can define variant points, declare possible binding choices and provide constructs for selection, conditional branching, iterations and processing. Specification files capture design level adaptation decisions while configuration files capture the existing test library structure in a non-redundant form using smaller reusable template building blocks in hierarchy.

STRAT targets groups of test clones. The more complex and scattered the test clone structures are, the more effective the template based approach becomes. Since STRAT uses templates that work similar to a pre-processing macro engine, it is not constrained by the general syntactical representations of the test libraries. All design decisions are implemented as directives, thus altering and managing them becomes easier. In essence, templates capture parameterization, provide generic design constructs, allow variant point definitions and allow variability binding options. In summary, STRAT provides a compile-time variant-binding test library maintenance model.

2.7. Case Study: Implementation of Proposed Solution

In this research phase we demonstrate a practical illustrative example using a selected test library with the view of refactoring the test library using **G**eneric **A**daptive **T**est **T**emplates (GATT) and make experimental assessments of the benefits derived. The example selected is `BiDiTests` test library from Android Platform OS core test libraries repository. The case study provides a concrete

step-wise implementation to illustrate our proposed STRAT approach. Using the case study as illustration we showcase the STRAT process as described below:

- The solution to the case study starts by identifying test clones and classifying test clones based on similarity taxonomy.
- Then test clone structures were individually analysed to decide if use of templates would be suitable.
- For test clone groups that are not reducible using conventional testing techniques, a normalized template structure plan with allowance for variant points and other needed control constructs was created.
- Subsequently these non-reducible groups were refactored into template structures using various template unification schemes proposed earlier.

We also used this case study to carry out experiments to assess the strengths and shortcomings of our complementary approach in comparison to pure traditional techniques. Further, we unified three subsequent versions of the `BiDiTests` project (namely versions 16, 17 & 18) to explore multiple version management. We also studied version specific contextual interpretations, related variability refinements and change propagation. The results were examined from the perspective of non-redundant representation achieved, ease of change propagation, structural hierarchy and ability to respond to future changes. These results are discussed in the next section.

2.8. Discussion of Key results

The redundancy study on Android test libraries affirms our hypothesis that "reuse of test cases can boost testing productivity" by showing presence of redundancy in nearly half of the existing test files. Redundancies were either simple test clone or structural test clones. Most redundant simple test clones were either identical

or had parametric variations. The study uncovered significant instances of complex structures as well.

Based on detailed assessments, we conclude that some of these straight-forward test case redundancies can be rectified by traditional approaches and some of the more complex ones need a complementary generic adaptive test template approach that supports unrestricted parameterization, variability management and heterogeneous test file formats.

Firstly the case study established that the proposed STRAT approach yields a loss-less re-construction of test libraries. Further the illustrative example confirms that the template based test libraries are concise (less lines of code), non-redundant (reduction in repeated fragments) and normalized (many clones represented in single template construction).

Using the experimental observations we evaluated and reported a measure of productivity improvements in terms of reusability (expressed through reduction in executable lines of codes and ability to express many types of variability) and effort reduction in terms of maintainability (expressed through reduced number of modifications required to implement a particular change request and the ability of template hierarchy to scale along with the growth of underlying test libraries). For the selected test library, the use of GATT templates yielded 63% to 80% compression to original size. In addition the same test library when reconstructed across three subsequent versions shows a compression to 23%.

Such findings illustrate a very good potential and encouragement for using template based approach in SPL Testing.

CHAPTER 3
Literature Review

In the previous chapter we described three key landscapes related to our research. This chapter provides the background, current state of research progress and literature review in the three areas relevant to this research thesis, which are: (1) Software Testing (2) Software Product Line Testing and (3) Android Platform.

Software testing is the process of exercising or evaluating a system or system components. It is used to validate specified requirements or to identify differences between expected and actual results. Testing is the measurement of software quality. This literature review focuses on three types of software testing strategies namely model based testing, combinatorial testing and mining/learning based testing.

Testing plays an important role in the quality assurance process for software product line engineering. There are opportunities for economies of scale in the software product line testing (SPLT) activities; consequently strategies and techniques that can take advantage of these opportunities are essential. Variability in features may lead to diverse products composition possibilities. Thus the research focus in SPLT fall under various categories namely, overall challenges, planning, processes, variability management, and levels of testing, testing efforts, metrics, test case generation, selection and execution. This literature review discusses past research work in each of these areas.

To align understanding research work pertaining to the motivational and illustrative examples used in this thesis (i.e., Android mobile platform), a brief review of android related work has been included in this chapter.

The organisation of the chapter is as below:

- Section 1 provides a brief introduction of this literature review.
- Section 2 reviews the various research efforts specific to typical testing strategies from the current software testing landscape.
- Section 3 reviews the various research efforts specific to software product line testing approaches, various test construction alternatives, their strengths and limitations.
- Section 4 reviews the research published thus far specific to testing of the Android platform.
- Section 5 concludes the chapter.

3.1. Introduction

We conducted a detailed review of available research literature to understand the current state of practice in software product line test construction landscape. Sources of information are scientific evidences such as scientific research publications, industrial case studies and open sources product line code repositories. Detailed listing and description regarding the source of information is presented in Appendix A.

3.2. Landscape: Software Testing

The goal of *software testing* is to [106] ensure that the software satisfies specified requirements as well as reveals faults that may exist . Efficiency and effectiveness are important characteristics that seek a right balance across available resources. Thus, it is carried out literature review to explore, identify and study the various testing strategies and key factors in software testing automation context. We have grouped the key findings, relevant papers and summarize their empirics into four sub-sections based on the test case construction techniques used. First subsection presents the overall challenges and findings from survey related publications. The

subsequent subsections deal with testing strategies such as model based testing, combinatorial testing and mining/learning based testing.

3.2.1. Overall Challenges and Survey Publications

Antonia Bertolino [18] summarizes the many outstanding research challenges for software testing into a consistent roadmap. She identifies four ultimate and unachievable research goals called "dreams" and discusses the individual challenges related to each of these dreams. In her work she mentions the need for a more holistic approach to software testing research, where the community can find new interesting synergies spanning across the other research disciplines of software engineering.

Alessandro Orso and Gregg Rothermel [121] have recently published a research travelogue that covers key research contributions from the past fourteen years in the field of software testing. In this work the authors summarize that domain-based testing techniques (i.e., those that address new application domains such as component-based systems, web applications and mobile applications) will remain one of the largest opportunities for testing researchers. They also assert that both testing of a software product line and testing of android applications are relatively young field of research offering more opportunities.

Two other key survey publications from Quan Yang et al [157] and Mats Grindal et al [63] explain the state-of-affairs related to test case construction. Test coverage is a measure of thoroughness and an indicator for confidence in the readiness of software under test. Yang et al has compared and summarized features of 17 coverage based testing tools. Yang et al in their survey make three key conclusions as following: (1) Reliability increases with increase in test coverage (2) Test coverage provides quantifications on test progress (3) In industrial settings, test coverage is observed to improve testing efficiency. Grindal et al [63] on the other hand classify various combination strategies based

on the algorithmic computability theory and illustrate the categories using a tree diagram shown in Figure 3-1. Combination strategies define ways to select values for individual input parameters and combine them to form complete test cases. Several proposals, implementations and case study observations have been found in their survey. Particularly, the work highlights the iterative combination strategies in which the test libraries are built up gradually. In the test case based iterative combination strategies, test cases are generated one at a time and added to the test library. Thus, a tester may start the algorithm with a preselected set of test cases.

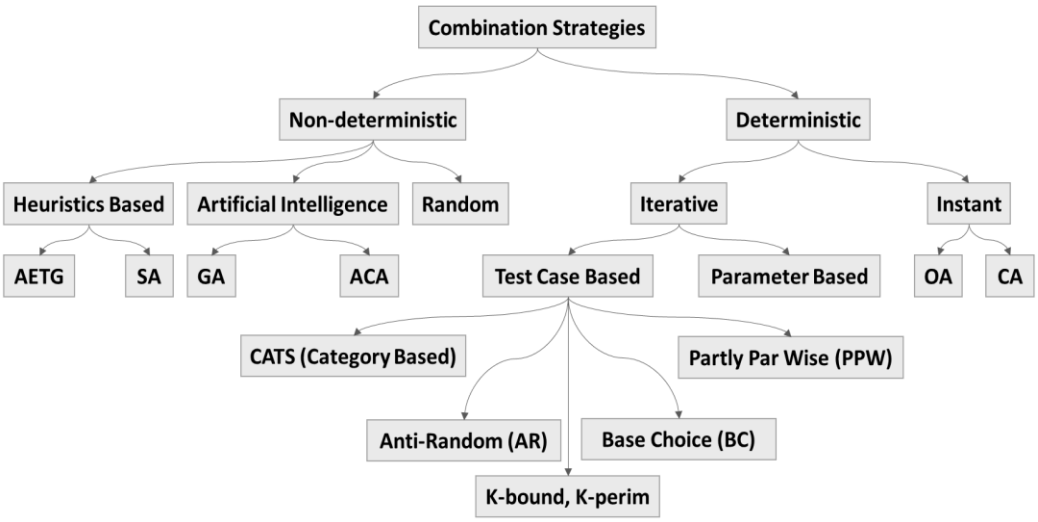


Figure 3-1 Classification scheme for combination strategies

Studies [122, 137], show that testing techniques that are used to test evolving software assume that a change consists of a modification of the code and even make specific assumptions on the invariability of the environment. The studies also conclude that heterogeneity and environment dependence which affect the ability to perform impact analysis is yet another key consideration for testing techniques that are heavily used during software maintenance phase.

In general, heterogeneity of technologies, rich contextual settings and high configurability makes modeling a test library in its entirety using single-language as self-contained units difficult. Also identifying differences between versions is

difficult; this identification is essential in understanding test library evolution and performing regression testing.

In summary, it is evident from literature that the most important challenge for researchers in the landscape of software testing is to define techniques that can go a step further and embrace real world complexities of emerging on modern software, regardless of its complexity and size.

3.2.2. Model based Testing

Model-based testing (MBT) [128, 129] is a variant of testing that relies on explicit behaviour models that encodes the intended behaviour of a 'System Under Test (SUT)' and/or the behaviour of its environment. Test cases are generated from one of these models or their combination and then executed on the SUT. MBT encompasses the processes and techniques for the automatic derivation of abstract test cases from abstract models, the generation of concrete tests from abstract tests, and the manual or automated execution of the resulting concrete test cases. One of the reasons for the success of MBT from a practical standpoint is that it has perceived advantages due to its capabilities at treating aspects such as generation, abstraction and traceability. A generic process [148] of MBT then proceeds as follows:

Step 1. A test model of the SUT directly linked with the testing objectives is built using informal requirements or existing specification documents.

Step 2. Test selection criteria are chosen to guide the automatic test generation, so that it produces a 'good' test suite—one that fulfils the test policy previously defined for the SUT.

Step 3. These test selection criteria are then transformed into test case specifications.

Step 4. Once the test model and the test case specifications are defined, set of test cases are generated with the aim of satisfying all the test case specifications.

Step 5. Once the test library has been generated, the test cases are run.

The last three decades have seen substantial research in the area of MBT. Model-based testing is extensively discussed in books by Beizer et al [17] and Utting et al [147] as well. UML diagrams such as state-chart diagrams, use-case diagrams, sequence diagrams, etc. can assist to generate test cases and this mechanism has led to Model based test case generation. Several researchers have proposed different ways of adopting or implementing MBT. Alessandra Cavarra et al [30-32, 40] utilize UML profiles and test constraints in form of directives in their generation approach. AGEDIS [65] is largely based on UML inputs. Desurvie et al [45] used a meta-model (xml based) and UML artefacts as inputs. They implemented a transformation program that generates a planner model, which aids testers during manual test construction and selection. Riebisch et al[135] utilized feature diagrams and extended the idea with UML multiplicity. S.Gnesi et al [60, 61] used formal conformance testing (UML state chart) and lambda calculus to model IOLTS (Input / Output Labelled Transition States). A recent systematic review of state-based MBT tools by Shafique and Labiche [141] gives a detailed classification of nine commercial and research MBT tools. Due to the ability of UML state machines/profiles to capture rich and detailed information, UML has been used as guidance for the automated generation of test cases.

Jeff Offutt proposes a technique that utilizes specifications of UML state charts. In this paper, the authors implement a prototype (UML test) which was later integrated into Rational Rose. The authors covered a breadth of details such as full predicate tests, statement coverage and transition pair tests. M.E.R.Vieira et al [151] proposes and implements a prototype of "Design and Specification – Based Object Oriented Testing (DAS BOOT)". Here a tester chooses class under

test, possible states charts for a class and also choose a coverage criteria. The prototype generates test cases based on these controlled inputs.

Dias-Neto et al [46] have performed a systematic review of the MBT research literature. They identified 599 relevant publications and have analysed 271 of these papers. From those publications, they identified 219 different approaches to MBT. They classified the approaches according to 29 different attributes, such as whether the models used UML or not, whether the goal was functional or non-functional testing, the testing level (system/integration/unit/regression testing), and the level of automation, the test generation process and the software development environment within which MBT was used. The authors summarize various benefits and limitations related to MBT research. One key finding from this work is that in the case of evolution, after identifying the changes carried out in a software component, the test library must be updated to reflect these changes. However the MBT techniques proposed in the above literature do not explain how to update their test models or what could be the effort do so or whether it would be feasible could be to perform this task within the given toolset. They usually presume that a new software model is ready to regenerate test cases which may introduce risks to a project.

3.2.3. Combinatorial Testing

Test case generation is the most active area of combinatorial testing (CT) research. To date, four main groups of methods have been proposed: greedy algorithm, heuristic search, mathematic method and random method. The first two groups are computational approaches while the latter two are guided by mathematical engines.

Greedy algorithms have been the most widely used method for test suite generation for CT. They construct a set of tests such that each test covers as many uncovered combinations as possible. David M. Cohen et al [36-38] have implemented AETG system that uses combinatorial design techniques. Complete automatic test

construction remains an important research area, but based on the study and literature review, it is overrated as a practical technique; most current tools that automatically generate tests produce tests that cover only trivial boundary condition test cases. CITLAB [28] tool allows importing/exporting models of combinatorial problems from/to different application domains, by means of a common interchange syntax notation and a corresponding interoperable semantic meta-model.

Heuristic search techniques such as hill climbing, great flood, tabu search and simulated annealing have been applied to τ-way covering array and variable strength covering array generation. Ali et al [2] have conducted a systematic review and empirical investigation on existing search based test construction techniques. Their compilation suggests the existence of techniques such as genetic algorithms, hill climbing, random search, static analysis, greedy algorithm and constraints-solving based construction techniques. They also catalogue presence of coverage, fault, timed and fitness based measurements that aid test asset construction. Sami Beydeda and Volker Gruhn [23, 24] suggest a dynamic path oriented approach using binary search algorithm.

Mathematical methods for computing the covering array have been widely studied by several researchers [64, 66, 67] and published in mathematical journals. The fourth group of test generation method is the random method, an ad hoc approach that randomly selects test cases from the complete set of test cases based on some input distribution. One example of these new random-testing approaches is adaptive random testing. Adaptive random testing (ART) [33] is a class of testing techniques designed to improve the failure-detection effectiveness of random testing by increasing the diversity of the test inputs executed across a program's input domain.

3.2.4. Mining and Learning Based Testing

The increased connectivity and increased computational power of today's computers enable collection of large amounts of telemetry data. This telemetric

data includes execution profiles, program spectra analysis, execution failure analysis, statistical usage, data mining, and machine learning techniques. Though the field is still in its infancy, researchers are studying ways to collect [34, 59, 81] and find ways to use these techniques to better guide the testing process [75, 94]. In this context, some important challenges mentioned by these researchers are scalability, the treatment of sensitive information and the general inability to assess whether an execution in the field terminated correctly or resulted in a (non-crashing) failure.

3.2.5. Summary

Although many methods have been proposed to generate test suite, as the problem of test suite generation is NP-hard, there is room for further improvement of these testing strategies. In addition to above specified testing strategies, researchers have also investigated ways to successfully combine testing strategies, as well as to combine testing with other types of verification techniques. A good representative work in this line is the Yogi project at Microsoft Research [114] , which is a property checking tool that picks an initial abstraction, and then based on heuristics it picks predicates for refinement to perform further optimization during testing.

In addition to research contributions, the software testing area has witnessed improvements in the state of the practice. From observing the improvements in the test automation landscape and key testing strategies we can summarize two key contributions in the recent times as:

1) Availability of sophisticated testing framework that supports creation of test fixtures, provides reusable test utilities and presents templates for standard test cases. Some examples of such test framework are JUnit, NUnit, Android Testing Framework, Robotium, Selenium Testing Framework, TestNG and Mockito.

2) Another practical contribution is the practice of Continuous Integration (CI). The idea behind CI is to commit code many times a day to integrate all of the working copies of the software. Automated regression test libraries are then run against the committed code to help ensure that the software remains stable. Thus CI ensures that subsequent engineering efforts can be performed reliably and reduces the amount of code rework needed in later phases of development, thereby speeding up overall development time.

3.3. Landscape: Software Product Line Testing

In software engineering lifecycle, testing often consumes 25% to 50% of the development costs [95]. Due to the variability within an SPL, the testing of SPLs is more challenging than single system testing. If these challenges are met by adequate approaches, the benefits outweigh the higher complexity and the increased effort of testing activities. The challenges of testing an SPL are caused by the product line variability and the systematic reuse.

In Software Product Line (SPL) engineering [125], domain engineering constructs a common product line platform by identifying commonality and variability while application engineering develops individual products based on the platform. Domain testing produces test libraries that would be reused for testing products in the product line. Domain testing includes testing for common parts related to variable artefacts that may or may not be realized during domain engineering. Meanwhile, application testing should also be able to achieve efficient reuse of domain test assets at the time it tests application specific parts.

To systematically present the literature review for SPLT, we decided to use the idea of categorizing studies in different aspects, as described by Petersen et al [124]. We classified the publication into different categories based on *Research Focus. Research focus* explains the kind of problems being solved.

We identified seven key categories of 'Research Focus' in this literature review; these are: (1) Overall Studies (2) Test Planning, Process and Management (3) Test Case Generation Approaches (4) Test Selection and Execution Approaches (5) Variability Management (6) Levels of Testing and (7) Testing Efforts and Measurements. *Overall Studies* comprises of publications that focus on overall landscape of SPLT challenges, technical reports and systematic mapping studies in the field. *Test planning, process and management* comprises of publications that focus on strategies coordinated to ensure an efficient, effective operation. It also describes testing framework seeking answers regarding how the testing activities are conducted, what are activity inter-relations, their sequence and chronology. *Test Case Generation* approaches comprises of publications focusing on methods, technologies and models used to construct test libraries (both generic and product specific). When test libraries are constructed, the product features designs are consulted as guidance for test case design. *Test selection and execution approaches* comprise of publications that focus on testing activities relating to (a) the selection of test cases from test libraries and (b) test execution performed to evaluate the quality of the product line assets. *Variability management* publications deal with commonality/variability analysis of the product line testing. *Levels of testing* publications elaborate on testing approaches, namely black-box or white-box and testing stages such as unit or integration or systems and acceptance testing. *Testing efforts and measurements* publications deal with test effort reduction strategies, test library measurements such as reuse, testability and adequacy of testing measured through coverage.

3.3.1. Overall Studies

Testing of a software product line is a complex and costly task due to the large variety of products derived from the product platform. In addition to the complexity of stand-alone product testing, product line testing also includes the dimension of what should be tested in the reusable domain platform and what should be tested in each separate product version. Clements and Northrop [35]

present the key underlying practices for software product line engineering and specifically discuss testing. McGregor brings out the need for research improvements in the area of software product line testing by outlining key challenges and discussing complex relationships between product lines and testing practices in his technical reports [99, 100]. Kolb et al [85, 86] discuss the importance and complexity of testing a software product line and component-based systems. Extensive research and systematic mapping studies conducted for investigating and evaluating the state-of-practices Software Product Line Testing have been published [2, 41, 51, 90, 92, 146] over the recent years. These studies document the general challenges, observations and comparisons among variability management approaches. Observing the above seminal research publications in SPLT reveals some key challenges are: (1) maintenance of large volume of test assets, (2) balancing testing efforts between design of reusable generic test components and tests targeting concrete products, and (3) handling variability using appropriate generation/selection strategies [49, 153].

3.3.2. Test Planning, Process and Management

The group of research publications listed in this subsection relates to processes, planning and other management activities of SPLT. Research in SPLT planning and processes pinpoints the need for guidelines and comprehensive, efficient techniques for systematically testing product lines. They also promote the idea of creating generic test cases. The research on SPLT management contains proposals and a few evaluated research works.

Condron et al [39] propose a conceptual model that supports the use of test automation frameworks for rapid test development to improve the test execution in various activities of the SPL test lifecycle. Software Engineering Institute (SEI) has published reports on series of case studies of organizations that have adopted software product line approach for developing systems. In particular, the report describes the Salion Inc. [142] case study detailing their acquisition management

solutions with mass customizations for automotive suppliers. This case study also describes dimensions that are key to testing practice, commonalty preservation strategies, benefits gained and lessons learned. W-model proposed by Li Jin-hua et al. [74] describes two V sub-models of domain test and application test and discuss the key issues and activities specific to testing software product lines with an example.

S.Kang et al [93] have come up with feature models which provides a compact representation of the product line in terms of features and their relationships. Kang et al., in one of their publications[79] , propose a for formal testing framework that creatively links concepts such as feature, variability, product line architecture, component and use case scenario to product line test concepts such as test architecture, variability for test and test scenario and by providing a systematic way for deriving product line tests. Nebut et al [105, 111-113] propose algorithms and frameworks based on SPL requirements and SPL specifications to automatically generate product-specific test cases. Publications [117, 136] presents processes and planning based on product line requirements and specifications.

Kolb and Muthig [86] discuss the importance and complexity of testing a software product line and component-based systems. Muccini et al [108, 109] highlight the risks and propose architecture based testing frameworks that emphasizes on the need for a combination of heuristic approaches in order to effectively perform testing. Denger et al [1]have studied comparison of defect detection techniques, code inspections and functional testing, in the context of product line development. There are discussions regarding review of domain design approaches [44] in the design of adaptable SPL test plans. McGregor clearly outlines steps for creating and using a fault model in SPLT context in[102].

Al dallal et al [42] focus on a testing model that considers retesting framework assets during the application engineering. Ganesan et al [57] introduce an

economic model to calculate cost-benefits for SPLT by performing Monte-Carlo simulation on various test strategies. In addition to these, publications based on industrial best practices, experiences with regard to technology, organization, and processes also provide useful insights into SPLT [149]. More specific SPTL research specifically targeting mobile and medical devices are other useful references pertaining to this thesis [71, 152].

In summary, product lines testing like all other software testing requires careful planning and well-defined processes. The major conclusion resulting from our literature review shows that most of the research in software product lines testing process involves automatically generating test cases for software product line and in the development of models to manage variability in software product lines.

3.3.3. Test Case Generation Approaches

Test case generation approaches are largely influenced by the technologies, models, tools and strategies that were used to construct other core assets. Coordination and effective construction of test libraries depend on the common reuse techniques, nature of test libraries and variation mechanism adopted by SPL. Some key contributions in the SPL test construction landscape are discussed here.

3.3.3.1. Domain Testing

Key problems in dividing SPLT as domain testing and product testing has been discussed by Tevalinna et al [146]. They highlight that complete integration and system testing in domain engineering is not feasible. They also point out that it is hard to decide how much we can depend on domain testing in the application testing. They propose four different strategies for SPLT namely (1) product-by-product testing (2) incremental product line testing (3) reusable asset instantiation (4) division of responsibilities.

3.3.3.2. Model Based Techniques

Several model-based testing approaches for software product lines exist due to the fact that models can be reused and adapted easily and are suitable to describe variability. Antonia Bertolino and Stefania Gnesi [21]propose a methodology to derive specific test cases for product families called PLUTO (Product Lines Use case Test Optimization) that derives test cases by using requirements described as product line use cases and a category partitions method; however, this work does not provide methods for reducing testing effort across products. ScenTED by Reuys et al [134] extends test case generation based on existing UML artefacts such as test scenarios, sequence diagrams and activity diagrams. Wubbeke et al [156] has published a proposal using feature models and trace them against product line requirements to guide product line testing. Dueñas et al [48]present a proposal of meta-model for testing, based on the UML profile for testing, but capable of being used in a Product Family Engineering context. Olimpiew and Gomaa [116], generated tests based on UML stereotypes and value tagging. WeiBleder [154] use state machines and OCL expressions. Kishi and Noda [82] propose a model checking technique for integration testing of a software product line.

3.3.3.3. Combinatorial Techniques

Optimization techniques such as combinatorics may be used to prune redundant test configurations that need not be explored. Test generation using combinatorial techniques involve systematic selection of a set of configurations that represent the relevant variability space and these configurations can be reused to test the individual product later. Hervieu et al [68] propose a tool to analyse feature models and automatically generate a set of configurations that cover all pair-wise interactions between features. Lamancha et al [89] present a solution that implements combinatorial testing techniques adapted to the SPL context. Gilles Perrouin et al [123] have published an experience report that applies *t*-wise techniques for SPL with two independent toolsets; one focuses on generality and

splits the generation problem according to strategies while the other emphasizes providing efficient generation. The report then evaluates the respective merits and limitations of these approaches.

3.3.3.4. Formal Specification & Natural Language Techniques

Applying formal methods and analysis techniques in SPLT is a promising way to address the variability modelling challenge. Temesghen Kahsai et al [77] propose frameworks that evaluate the use of specific test case selection based on formal specification. Bashardoust and Tajali [7], use domain models expressed as generated contracts and use that to guide test case generation. Both research provide abstract representations (formal notations or contracts) of the tested product features and guide product specific test case selection.

3.3.3.5. Aspect Oriented Techniques

Aspect oriented programming (AOP) allows testing techniques to exhibit power of parameterization, construction time constraint checking and conditional compilation to AOP languages. Feng et al [53], use aspect oriented approach to generate unit test cases in a product line context. Knauber and Schneider [84] combine aspect oriented programming and unit testing to trace and manage small scale variability among test cases.

3.3.4. Test Selection and Execution Approaches

Almost all of the proposed strategies for product line testing are idealistic in the sense that they enforce specific requirements on other parts of the development process (i.e., software codes other than the testing). Literature cited below describes many such conceptual and underlying tool dependent solution proposals in the SPL test selection and execution approaches. Some interesting mentions are listed.

3.3.4.1. Combinatorial Approaches

Similar to combinatorial test generation, McGregor in his technical report mentions the use of combinatorial selection techniques for variant selection. Muccini and Van Der Hoek [106-109] propose usage of heuristics for appropriate test case selection. Cohen and Dwyer [38] use both combinatorial selection and coverage criteria for selection.

3.3.4.2. Search Trees Based Approaches

The use of meta-heuristic search techniques for the automatic generation of test data has been of burgeoning interest for many researchers in recent years. Stephenson's [144] proposal uses effective search strategies for reduction and selection of test cases. Geppert et al [58] uses decision trees and execution traces to reduce and select test cases. F.Ensan et al [52] propose an evolutionary testing approach based on Genetic Algorithms to explore the configuration space of a software product line feature model in order to automatically generate test suites of $O(n)$ size complexity with a suitable trade-off balance between error coverage and feature coverage in its generated test suites.

3.3.4.3. Cost/Priority Based Approaches

Testing techniques select test cases based only on the testing priorities established in the test plan. The selection and execution thus determines the number of test cases for each feature based on priorities of the requirements that will provide acceptable effectiveness. Imai [70] identifies the need and importance of finding out the cost/efforts during maintenance of test cases. Dowie [47] highlights the lack of customer perspective in existing SPLT approaches.

3.3.5. Variability Management

Variability management (VM) encompasses the activities of eliciting and representing variability in product line artefacts as also covers aspects like establishing and managing dependencies among different variability. In testing

scenarios a large majority of VM approaches are based on feature modelling and/or UML based techniques. There are also approaches based mechanism such as natural language, mathematical notation and domain-specific languages. As discussed previously, the testing approaches handle variability using a wide range of approaches; many of these usually explicate variability as early as possible and use those variants to design test cases. Frequently, in model-based approaches variability is introduced into test models created through UML use cases [67, 79] , test scenarios [132-134] , activity diagrams [62, 116-120] and feature models.

Considering various research works reported in literature, McGregor summarizes the implications of variability in testing through two key observations. (1) Variation is identified at explicit, documented variation points: Each of these points will impose a test obligation in terms either of selecting a test configuration or test data. Analysis is required at each point to fully understand the range of variation possible and the implications for testing. (2) Variation among products (i.e., variation among tests between products): The test software will typically have at least the same variation points as the product software. Specifying a set of constraints is necessary to associate *test variants* with *product variants*. One solution to specifying constraints in both product and test library is to have automated scripts that represent variations and builds both the product and the tests at the same time.

3.3.6. Levels of Testing

Levels of testing refers to particular research methods and strategies performed at various stages of software product line testing (both domain and product engineering). Some interesting citations are discussed below:

3.3.6.1. Unit Testing

Unit testing verifies the smallest unit of software implementation, for example a method call. This unit (basic element) is usually be a class in most instances but

can even be a module, a function, or a software component. The granularity level of the tests depends on the strategy adopted. The purpose of unit testing is to determine whether this basic element performs as required through verification of the code produced during the coding phase. Nebut et al. [77] use parameterized use cases as contracts on which testing coverage criteria may be applied. Feng et al. use an aspect-oriented approach to generate unit tests [77].

3.3.6.2. Integration Testing

Integration testing is applied as the modules are integrated with each other or within the reference in domain level. Verification and validation of architecture calls for specific domain components to be integrated in multiple systems. This type of testing is also performed during application engineering [101]. Li et. al. [153] present an approach for generating integration tests from unit tests.

3.3.6.3. System and Acceptance Testing

System testing ensures that the final product matches the required features. It is generally observed that research in this area focuses on functionality, UI and acceptance testing. According to Geppert et al. [58] , system testing evaluates the features and functions of an entire product and validates that the system works the way the user expects it to. Acceptance testing is conducted by the customer but often the organization/department that develops the software would create and execute a preliminary set of acceptance tests. Software product line organizations often leverage on commonality among the tests pertaining to the various products to reduce costs. Yi Yu et al [158] presents a software acceptance testing technique based on knowledge acquisition and accumulation in form of an expert system. This expert system builds information over a period of time and later helps in exposing software faults based on the accumulated and processed knowledge.

3.3.7. Testing Efforts and Measurements

Test effort reduction strategies and testability factors can have significant impact on productivity and profitability. McGregor first defines SPL testability, where he describes the unique characteristics of SPL that helps in testing. Jaring et al[73] point out that the testability of a product line can be viewed as a function of the binding time of variability, and that providing early binding can increase the ability to test products early. Engstrom et al [51] also discuss the different attributes of testability, its relationship with SPL architecture and Isis Carbel et al [27] propose an approach that improves SPL testability.

3.3.8. Summary

The above literature review suggests that most of the original research work on SPL testing focused on solving specific research challenges. Thus, they present the problems at the fine-grained level with appropriate techniques and do not usually provide a perspective of the whole SPL testing process from initiation through completion. Through a review of literature in this section we have tried to highlight key contributions of the existing researches in various focus areas such as overall SPLT landscape, SPLT planning, process and management, SPLT variability management, SPLT levels, SPLT efforts and measurements and SPL test libraries construction, selection and execution.

3.4. Landscape: Android Platform Testing

Users increasingly rely on mobile applications for computational needs. Google's Android Open Source Project (AOSP) maintains a complete software stack to be ported by Open Equipment Manufacturers (OEM) and other device implementers and run on their own hardware. To maintain the quality of Android solutions, Google has a dedicated team of full-time engineers, product managers, user interface designers, quality assurance testers and other roles required to bring modern devices to market.

Google's testing team executes individual test cases on attached mobile devices or on emulators. The test cases are written in Java as JUnit tests and packaged as Android .apk files to run on the actual device targets. The compatibility test suite (CTS) test harness runs on desktop machines and manages test execution. The CTS includes the following types of test cases:

- Unit tests: Tests atomic units of code within the Android platform; e.g. a single class, such as `java.util.HashMap`.
- Functional tests: Test a combination of APIs together in a higher-level use-case.
- Reference application tests: Instruments a complete sample application to exercise a full set of APIs and Android runtime services.

From our literature review, we observe that several authors have published Android test practices as proposals or case studies in the application testing context. Amalfitano et al [3, 4] present AndroidRipper, an automated technique that tests an Android app's Graphical User Interface (GUI). Hu and Lulian [69] present an approach for automating the testing process for Android apps with a focus on GUI bugs. Takala and Jaaskelainen [72, 145] describe the model based testing that they had performed using an Android app case study. Mahmood et al [96] propose the use of cloud based performance testing for Android apps. They developed a fully automated test case generator for the non-functional security testing feature and implemented a feedback loop to ensure code coverage. Zhang et al [159] use symbolic execution to test apps. Our research work focuses on the variability management of executable test libraries including both functional and non-functional features of the SPL under test. Being a relatively new platform, there is a paucity of research investigation focusing on Android Platform. Needless to say platform specific test case generation techniques will be of interest only to concerned vendors and partners who distribute Android fortified

smart phones. Nevertheless due to the large market encroachment of Android and due to an enormous pool of third party developers contributing to Android apps, a deeper study is definitely perceived need [106] and valuable in enlarging the knowledge of researchers in mobile platforms.

3.5. Conclusion

The literature review presented in this chapter indicates that software product line testing is a virgin and fertile research area especially when viewed in the light of research already conducted in the areas of matured software engineering practice. It is also evident from this review that existing software product line testing research publications are typically conceptual solution proposals and discussion oriented. Further, the available research publications focus more on isolated techniques than presenting industry practices or real-time experience reports. That said, from the SPLiT workshop literature [116] it is clear that there is a well-established understanding about challenges. Tevanlinna et al [146] in their survey publication indicate that product line testing is a large scale effort and evaluations are expensive, which is one of the explanations behind the limited availability of empirical studies in literature.

In a majority of the SPLT publications the handling of variability is the main focus. Each product line approach advocates a different test case derivation technique based on specific ways of handling variation points. The underlying variability mechanism of the tool has powerful influence regarding the uniform handling of variant points in software as well as test libraries.

In summary, it can be concluded from the literature review that software testing in general and software product line testing in particular need new, more inclusive methodological approaches for test library construction that complements traditional techniques. Also such new methods should be generic so that they can

be supported by underlying models for their theoretical foundation, tools for their practical use and metrics for their management and evaluation.

CHAPTER 4
A Study of Redundancies in Android Platform Test Libraries

Similar software systems have similar test cases. We find much redundancy even within test cases of a single system and naturally the redundancy in software product line test libraries is accepted to be significant. In this chapter, we describe the results of similarity analysis performed on a typical software product line with Android platform framework project's test libraries as example. The results confirm our hypothesis that reuse of test cases can boost productivity at least as much as reuse of code. We also identified repetitive patterns in Android platform framework test libraries that can be represented in generic form. In this chapter, we present quantitative and qualitative findings from our study of Android platform framework test libraries.

The organisation of the chapter is as below:

- Section 1 introduces context of test library similarities and reuse opportunities studied in this chapter.
- Section 2 describes challenges involved when dealing with redundancies.
- Section 3 details the case under study, the Android platform test libraries.
- Section 4 elaborates on the research hypothesis of this study.
- Section 5 describes the step-by-step processes involved in this case study research methodology.
- Section 6 presents the results of the conducted study.
- Section 7 presents the details of research analysis performed (both quantitative and qualitative) for factual interpretation.

- Section 8 presents the possible threats to validity
- Section 9 concludes the chapter.

4.1. Introduction

Testing is an essential part of software development, since it consumes 40-60 percent of the whole software development effort [17]. Recent advancements in test automation and newer development trends emphasize more continuous integration; therefore software development teams consider early inclusion of testing in iterative development environment. As organizations strive to shorten the development time of their products while at the same time attempting to improve their quality, the need for practical, scalable testing approach is becoming increasingly important [8, 78, 91] .

Test case similarities create an opportunity for reuse and reduce the effort to both develop and maintain test libraries: Suppose that for each large enough group of similar test cases we design a 'generic adaptable test case', from which all test case instances in that group can be automatically derived. It reduces the size and cognitive complexity of test libraries. Also, instead of working at the level of individual test cases, testers would effectively work with a smaller number of generic test cases – a much simpler task.

In this chapter, we explore the scope for test case reuse with Android platform framework test libraries as our case study. First, we conducted similarity analysis of Android platform framework test libraries to assess the degree of redundancies and investigated the potential benefit of test case reuse. Then, we identified patterns of repetitions among test cases that are potential candidates for reuse. Finally, we outline hints on how possibly the repetition patterns can be practical made generic so as to realize the concept of test case reuse.

4.2. Challenges

The efficiency of testing lies in the ability to select a meaningful subset of test cases that uncover defects and thereby increasing the confidence on the software under test. Test case selection is guided by parameters namely, priority, risks and scope. Well-designed testing techniques provide adequate coverage, improved testability, faster defect discovery and promotes reusability. In the context of this research, the testing artefacts and techniques should also facilitate reuse with a view to addressing redundancies in test cases.

4.2.1. Why is redundancy a problem in test libraries?

From a test execution perspective, a test case in considered redundant if its individual coverage does not contribute to the overall product's test coverage. Similarly, from a test library perspective, duplicate test cases or duplicate code fragments within test cases are considered redundant. Presence of such redundancy serves as a hindrance and causes negative influence in the following ways:

1) During test case maintenance (i.e., day to day changes), redundancies would necessitate multiple code changes which are complex to trace.

2) During test case evolution (i.e., when new products get created), redundancies increase the efforts required to keep abreast of changing requirements and hence business agility may suffer.

3) Redundancies is considered as a hindrance during continuous delivery processes such as automated builds, continuous integration, refactoring of test cases and test execution that helps in delivery of frequent product release versions.

4) Redundancies may affect the ability to cope with change propagation due to large number of variations and dependencies and hence resulting in combinatorial test explosion problem discussed later.

Thus test construction approaches for building test libraries should provision for testing of feature variations, planned test case reuse and seamless management of various test artefacta. Such an approach is necessary to counter the negative influences of redundancies.

4.2.2. Improving Reusability in Test Libraries

Reuse is a fundamental discipline in software engineering and plays an important role in the development of new test libraries or in the maintenance of existing test libraries. It is almost impossible to have a test library that has just one version. One of the typical examples is testing of a well-established piece of software with different versions (say several versions created to meet different features catering to various users' needs). Such diversities can be productively addressed by systematic reuse approach.

Test library reuse has been an identified as a problem that requires extensive research[106]. Identical or near identical test code fragments are referred to as test clones. Presence of such large granular test clones signifies reuse opportunity. A systematic product customization allows us to reuse a common base of test libraries and at the same time evolve them into multiple generative versions in close accordance to the customers' requirements and test designers variability choices. By improving the quality of test libraries via reuse approaches there is a significantly higher chance of detecting faults and correcting them in one place without propagating the faults in all products, thereby delivering robust products with well managed release versions. The reuse of test libraries within the focus of product line helps reduce maintenance effort. Testing the modified parts of the software alone is not sufficient as changes to one part of the software may cause errors in other parts. In regression testing, test cases of older versions of a software product are reused to test a new software version.

The key idea of establishing test library reuse is to develop test cases once for the entire product line and then reuse them for multiple test library versions. To achieve a sufficient degree of reuse, test cases must be reused for common as well as for variable parts of the test libraries by explicitly specifying and managing common and variable parts. Test cases for common parts can be reused as they are and for the test cases that contain variability, the variation points can be defined and choices are bound at compile time before the test cases get executed.

4.3. Overview of Android Platform Test Libraries

Android is a software stack for mobile devices that includes an operating system, relevant middleware and key applications [26]. Since its launch in 2003, Android has captured the attention of mobile phone companies, developers as well as the general audience. Android provides open standards for mobile devices. Android's architecture inherently promotes component reuse. There have been frequent platform enhancements that introduced several releases into the market to provide new features such as account synchronization, improved media-playing performance and enhanced geo location support. Thus Android platform frequently introduces complex features and effectively testing these features is a challenge.

4.3.1. Android Platform as Research subject

The complexity and challenges in testing of mobile platforms are caused by factors such as device heterogeneity, memory fragmentation, power-conserving peripherals, sensor complexities and context awareness.

Android platform exhibits the key characteristics of a typical software product line. The Android Open Source Project (AOSP) maintains a complete software stack to be ported to original equipment manufacturers (OEMs) and other device implementers to run on their own hardware. Android maintains commonality and variability of features using separate Android project structures. Common

projects are grouped under section 'base' of the source repository. Also test libraries manage display variability using layout configuration files and resource property setting files. Similar to traditional Software Product Line Engineering (SPLE) philosophy, Android names its core-assets as base framework and tests its features using an example application. A release corresponds to a formal version of the Android platform and configuration management involves policies that separate the platform code from vendor specific capability contributions.

In summary, Android platform serves as a good test bed for all the four previously stated redundancy problems namely, test maintenance, test evolution, test automation and combinatorial test library explosion owing to feature variations to be tested. The Android open source test libraries also exhibit several variability criteria such as device diversity (displaying hardware variability), platform diversity (displaying more of software variability), sensor collection (displaying peripheral variability) and comprehensive connectivity (displaying networking protocol variability). All the above mentioned factors support the choice of Android Platform Framework test libraries as the subject for our research case study.

4.3.2. Android Platform Diversity

Android is Open Source software and thus provides transparency in aspects such as platform evolution, comprehension of its features, defect fixing and hardware portability. According to Open Signals[3] there are 11868 models of Android devices as of July 2013, with diverse screen size, display density, media and camera options, touch sensitivity, text/input devices, storage options, in-built sensors (for measuring motion, orientation and environment conditions), and connectivity devices such as Bluetooth, Near field Communication (NFC), Wi-

[3] http://opensignal.com/reports/fragmentation-2013/

Fi, USB Host and Session Initiation Protocol (SIP). The report also reveals that the Android platform currently supports forty eight billion apps in play store spread over eight versions of OS in use simultaneously. Further the platform supports software from various device manufacturers, wireless carriers and other open source platform stack. All these characteristics clearly make Android fall under SPL classification and an ideal candidate for SPLT research.

Android's software stack consists of several layers. At the heart of its stack resides the Linux based Dalvik virtual machine [50] that enables portable, optimized byte-code interpretation for operating the mobile platform. Figure 4-1 below illustrates the diverse nature of the platform and major functional distribution.

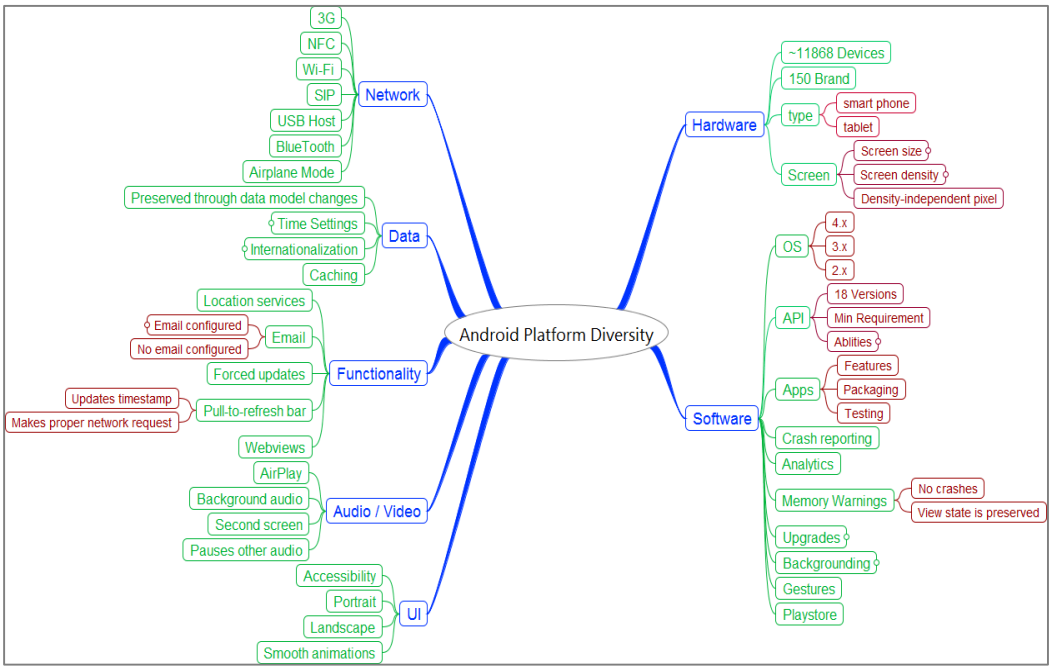


Figure 4-1 Android Platform Diversity

Android platform comprises of source codes related to both hardware and software components. Hardware related codes comprises of devices, sensors and related peripherals interface, while software components deal with kernel, operating platform, memory, processes, user interactivity and play-store functionalities. Communication protocols such as 3G, NFC, Wi-Fi, SIP, USB Host and Bluetooth are supported by the platform. Additional functionalities

include location services, email communication, messaging, screen display refresh, audio/video facilities and various UI features. Furthermore, there are many different versions of Android platform OS that are concurrently active at any one time, adding another level of complexity. Thus a versatile testing framework and efficient test library implementation are quite essential despite the fact that testing the platform and related apps across the whole range of diverse Android devices in existence can be extremely challenging and time-consuming.

4.3.3. Architecture

Figure 4-2 shows Android platform layers. The innermost layer is the Linux Kernel over which the Android Runtime Libraries are deployed. At the outer layer are the Application Framework and the Application Layer. The application layer provides core functionalities such as email, SMS, calendar, maps, browser, music, gallery and contacts while the application framework layer provides APIs for device management, context settings, application content and other programmer related services. All applications are written in Java and provide concurrent execution support. In addition to Java, the Android platform also uses C/C++ to implement its internal core libraries like Surface Manager, Graphics, Media Codecs and web browsing engine.

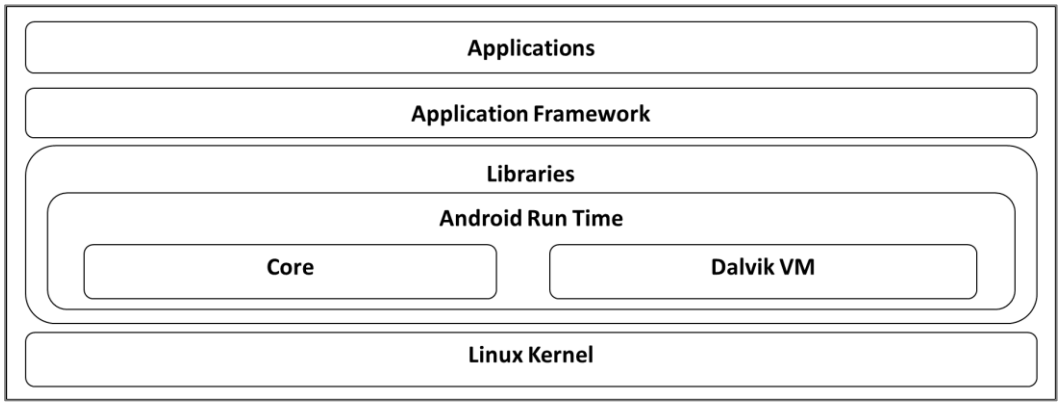


Figure 4-2 Android Layers

4.3.4. Development Tools

Android Software Development Kit (SDK) and Android Developer Tools (ADT) plug-in for Eclipse can assist in creating rich and innovative applications usually called ‘apps’. The developer environment consists of the Android SDK, the Eclipse IDE and the Java Development Kit (JDK) which has to be preinstalled prior to Android SDK and Eclipse.

4.3.5. Testing Tools and Testing Framework

The Android testing framework is an integral part of the Android software development environment. It provides the necessary architecture along with a set of powerful tools that help testing the various aspects at different testing levels starting from unit test to complete system testing. Key features of Android testing framework include:

- **JUnit**: Android test suites are based on JUnit. Either plain JUnit or Android's JUnit extensions can be used to test Android components. The general-purpose test case class named `AndroidTestCase` is useful for simple test scenarios.

- **Android JUnit extensions:** The Android JUnit extensions provide component-specific test case classes. These classes provide helper methods for creating mock objects and methods that help testers to control the lifecycle of a component.

- **Test Suites:** Test suites are contained in test packages that are similar in structure to main application packages. These test files help in organizing the test library components into logical units.

- **SDK Tools:** The SDK tools are useful in building and testing various Apps either through Eclipse with ADT or using command-line. These tools provide information about the application project under test and use

this information to automatically create the build files, manifest file and directory structure for the test package.

- **MonkeyRunner:** The SDK also contains `MonkeyRunner`, an API for testing devices with Python programs and UI/Application. `ExerciserMonkey` is an additional command-line tool for stress-testing UIs by sending pseudo-random events to a device.

- **Robotium:** Robotium is an Android test automation framework which can be used to write powerful and robust automatic black-box test cases for Android apps. Testers can write test scenarios to carry out functional testing, systems testing and acceptance testing that can span multiple Android activities.

4.3.6. Diversity Challenges while testing Android Platform

The rapid growth of the Android OS comes with an expectation of equally rapid rollouts of platform improvements and bug fixes. With the proliferation of devices such as a wide variety of mobile phone models, tablets and wearable devices, Android based providers have to manage the changing platforms variations in order to sustain the market. Vendor neutral, independent, end-to-end testing is important for the success of the Android platform OS. The mind map provided in Figure 4-3 illustrates the various functional aspects involved in testing both the android platform and the hosted apps. The diversities include hardware devices, connectivity, distribution channels, software flavours and tool stacks. Figure 4-3 illustrates the six key areas of focus for Android testing – these are hardware, software, communication protocols, tools, user interface and distribution.

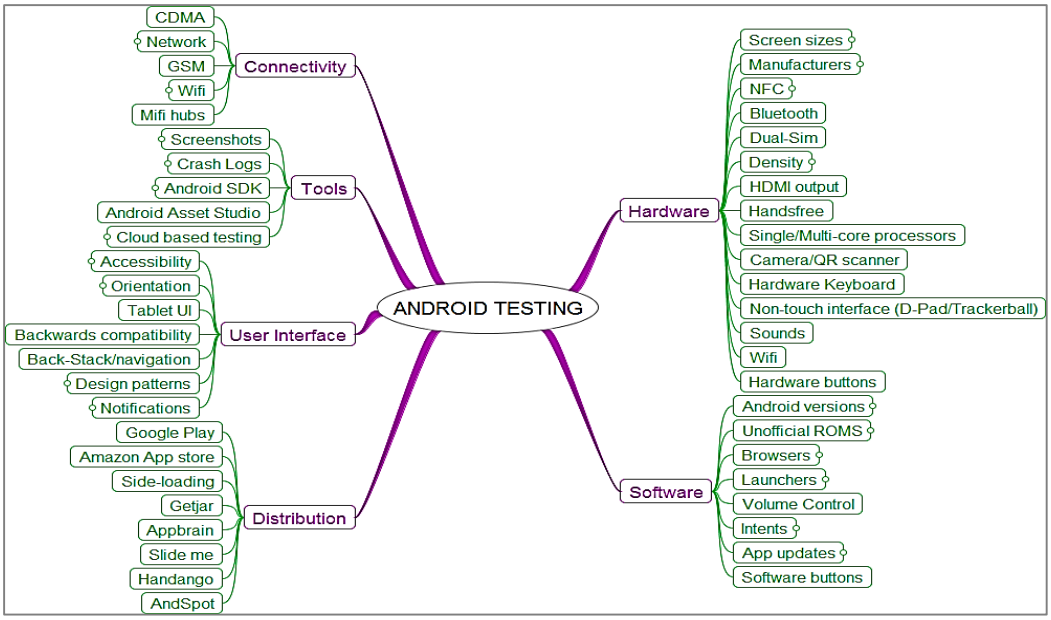


Figure 4-3 Android Testing

The discussion and analyses of Android platform described in this section provides an adequate case for the need for in depth research to evolve an approach and technique that can reduce the overall efforts required in SPL testing phase.

4.4. Research Hypothesis

The idea behind the reuse-based approach for test library maintenance and evolution is to reuse the knowledge of the past software changes to effectively implement test components that can support testing of future changes. The essence of such reuse-based evolution is to ensure clear visibility of changes, clear understanding of test library similarities/ differences at all granular levels and minimizing redundancy.

The foremost requirement to achieve test-reuse is to identify similarities and variations in test library. Hence facilitating reusability through identifying similarities and variations is a prerequisite to construct generic, adaptive and therefore reusable test libraries. Hence it is our hypothesis that: ***"Systematic reuse approaches employed in a test library can boost productivity and reduce maintenance/evolution efforts along similar lines as reuse of source code"***.

4.4.1. Research Motivation

The derivation of test cases for product families is difficult due to the presence of variability since each variation point multiplies the number of possible behaviours to be tested. ***Combinatorial explosion*** is a frequently occurring problem in testing. One instance of combinatorial explosion in testing is when systems under test have several parameters, each with so many possible values that testing every possible combination of parameter values becomes infeasible. Another instance of combinatorial explosion in testing may occur for configurable systems. When systems under test have many configuration parameters, each with several possible values, testing each configuration becomes infeasible. Examples of configuration parameters are versions of a specific software/ hardware module, different types of software/ hardware modules and number of logical or physical entities included in a computer system.

Combinatorial explosion of test libraries is caused by the need to test individual variant features. Let us assume that a particular component contains 12 variant features that may be different in different device installations. Then we might have as many as 2^{12} = 4096 combinations of these variant features in various device installations. In practice, only some of those combinations are legal. The above simple example shows that even a small number of variant features can result in a combinatorial explosion of test cases**.** Such overwhelming number of test cases can be reduced if we could exploit the fact that test cases for different product variants are similar, in the same way that the respective products are similar.

The combinatorial explosion in software product lines results in creating a large set of test cases and consequently several test classes. However, in view of the fact that these classes mostly get created due to parameter variation, a large part of test classes under the same family may be same with minor differences that cater for the parameter variations. Each such test classes are essentially ***clones***. It

therefore becomes necessary for us to take a typical large sized test library of a software product line to demonstrate the presence of clones as a first step. This requirement motivates the current experiment where we try to explore the presence of recurring similarity in test classes.

In SPLT context, most of the current research techniques anchors on the model and symbolic execution based test construction techniques as linchpins. Since the Android test libraries has not been scrutinized from a test construction and maintenance perspective, this study investigates existing redundancy problems in this large scale test libraries. By identifying and understanding the recurring similarities in the Android platform framework's large scale test library it is possible to identify strategies for redundancy removal and derive generalizations.[4]

4.4.2. Research Objectives

The objective of the study is to identify, understand and classify the nature of redundancy in test libraries. This study would lay the necessary foundations and provide the necessary technical insights which may be required to subsequently propose a strategy for reuse-based test library creation and eventually provide guidelines for designing generic adaptable test cases, simplifying test libraries and enabling test cases reuse and automation.

1) **Objective #1**: To establish the presence of large scale redundancies within Test libraries of a typical SPL (Android Platform in this study).

2) **Objective #2**: To identify and analyse similarities found in the 'Android Platform Framework Test Libraries'.

[4] *{Some excerpts of this section is published in ICSR 2013[6]}*

3) **Objective #3**: To analyse the findings with regard to redundancies and to come up with insights that would help in design of generic adaptable test cases.

4) **Objective #4**: Use the knowledge derived through this research study to enhance and formalize the test clone taxonomy/definitions/nomenclature which would contribute to the body of knowledge in testing domain.

4.4.3. Research Questions

The dichotomy in software product line testing is between testing various feature combinations and limited period of time left to execute tests. Thus efforts expended in testing activity must ensure sufficient coverage within a limited time and at the same time effectively uncover key defects of product under test. This systematic study using the Android platform test libraries as example aims to summarize the current state of redundancies and variations by proposing answers to a set of questions below. The research questions stem from the need for finding efficient reuse based approach for test library construction, maintenance and evolution. Further the focus of study is to analyse issues such as heterogeneity, managing the growth and scalability of test libraries which are encountered in typical software product line testing. The questions uncovering these SPLT challenges [51]are:

1) **Research Question #1**: What kinds of similarities are found in test libraries? How frequently do they occur and in what granularity?

2) **Research Question #2**: How are heterogeneous test assets of the current test libraries managed?

3) **Research Question #3**: How scalable are conventional test library construction techniques?

We aim to answer these questions through analysis and observations on the empirical case study and report the findings of this research in subsequent sections.

4.5. Methodology

The overall goal of the study was to identify similarities found in Android platform OS test libraries and systematically assess the empirical evidence collected from the test libraries. Although certain aspects of the methodology such as the test designer's domain expertise, platform comprehension and quality of tools employed involves subjective judgments, the methodology focused also on technical activities such as similarity findings, aggregation and inference which are tangible. Key activities involved in the study are described in study overview sub-section. Data collection explains how test library sample was retrieved from GIT public source code repositories. Analysis process was accomplished using automated tools. A final validation was conducted using manual inspection.

The experiment follows the guidelines published under the title 'Software Engineering Research Methodology Guidelines for Case Studies' by [139], which comprises of five key processes. Using this methodology, we conducted the experiment using the steps as listed below:

1) ***Systematic design*:** During design stage, the key objectives and experiment execution plans were formulated.

2) ***Data Collection*:** The study collects data from Android GIT open source code repositories. Then during the data cleansing activity, each test case was carefully scrutinized and segregated.

3) ***Analysis*:** Using the NUS SoC Software Engineering lab's *Clone Miner and Clone Analyser* tools, the experiment studied the occurrence of similarity patterns. During clone analysis the tool provides various filtering mechanisms that allow the test clones to be further scrutinized

and grouped. Additionally the similarity pattern data was stored in database and examined in detail. The test case clones identified using the tool were further classified into groups and categorized systematically through observation and inference.

4) ***Report*:** The findings and interpretations were organized into a redundancy report. The report logically details similarity occurrences patterns, interjects additional insights on possible causes and categorically summarizes the identified shortcomings that these redundancies present to test library maintenance.

4.5.1. Data Collection Process

After initial analysis of Android code repository, the experiment focused on Platform Framework Project which was selected from more than 400 similar projects based in the GIT servers (Focused library is highlighted in the Figure 4-4 below). Platform Framework Project handles system variations, security settings, graphics, multimedia and communication components. Thus the chosen platform framework project which deals with device and platform heterogeneity and provides common services and kernel interaction would be a good candidate for our study.

GIT repository path is
https://android.googlesource.com/platform/frameworks/base/

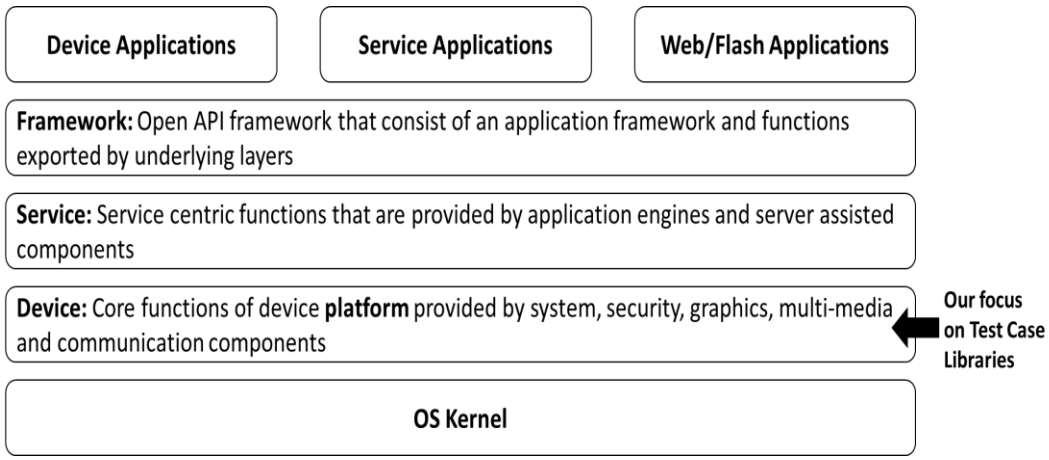


Figure 4-4 Android GIT Project Layers

4.5.2. Analysis Process

We used the Clone Miner (CM) and Clone Analyser (CA) tool for our study. CM/CA finds clones in the target software system(s) and also allows us to locate and filter clones that are of interest to our research. CM/CA helps us to find both simple and large similarities. CM/CA uses token-based techniques and data mining algorithms to find both simple test clones and related higher level structural similarities.

4.5.3. Validity Process

During the study, the CM/CA tool was limited and hence the identified similarities in test libraries were manually inspected and validated by us. The validity process was carried out using the following steps:

1) Setting up the clone miner and clone analyser tool.
2) Checking out the Android base platform code repositories from GIT server and separating test artefacts directories for further investigation.
3) Conducting similarity investigation using clone miner and analyser tool
4) Analysing the similarity investigation outputs further.
5) Reporting the outputs, challenges and research findings that will be useful in answering the original research questions defined.

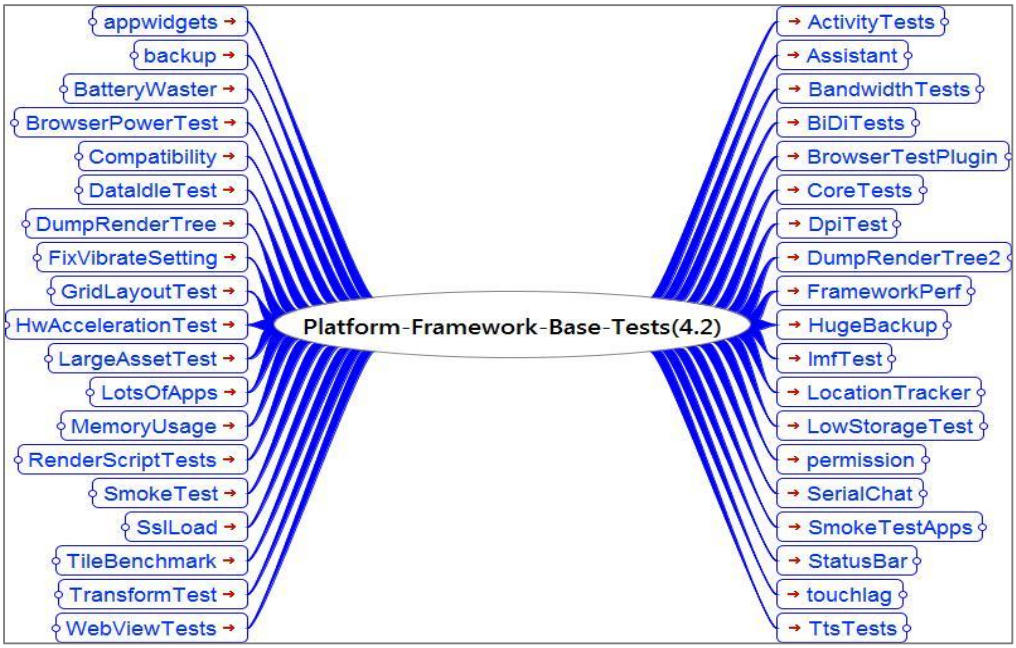


Figure 4-5 Android Platform Testing Sub Projects

We studied various test library projects under the Android platform framework. As shown in Figure 4-5 the test libraries are a collection of unit, integration and system test cases targeting the features of Android Platform under test. As our goal is to reuse test cases, we scoped our study to single language; we included Java based test libraries, excluded C++ and C test libraries.

4.6. Results

In this section, the results of the experiment performed on the Android platform framework test libraries are analysed and presented with examples. Each example provides instances of test clone redundancy of varying nature; some heavy and some with light amount of redundancies. The causes for the occurrence of redundancies are also diverse. Based on these observations, we have classified redundancies as simple or complex clones as described with examples in the following paragraphs.

4.6.1. Group 1 - Simple Redundancies

Simple redundancies are exact or similar copies of test code with variations in terms of few lines of code or parameters or attributes. Simple redundancies also

include syntactically identical lines of code (fragments), identical test classes or identical test methods except for variations in the form of parameters, return types, identifiers, literals, types, whitespace, layout and comments. A test method is a single executable test case that may share a common set-up and tear-down method for test fixtures (data). One of the key causes for such redundancies is the lack of methodical reuse among common test codes. Our observations point out that certain redundancy may also have occurred from lack of appropriate creational test case design patterns. The following examples illustrate and discuss simple redundancies.

4.6.1.1. Test Fixture Similarities

Test libraries need the setup of test environment before executing the individual test cases. This environment is usually composed of complex data structures or collections that are created and managed inside test fixtures. When the context is similar, the test fixture codes are also similar. Test fixtures also manage life cycle activities for mock objects and driver stubs. Test fixture codes include memory management and data setup/tear-down and assertion statements causing similarities. For example, in Figure 4-6, consider the test case from media group targeting testing of media thumbnails. The `MediaItemThumbnailTest` class comprises of seven identical clones shown in that validates and recycles test data. Failing to design test fixture methods as reusable components is the cause for such redundancies.

```
//. . .
for ( inti = 0; i<thumbNailBmp.length; i++)   {
     validateThumbnail(thumbNailBmp[i], outWidth, outHeight);
     thumbNailBmp[i] = null;
 }
//. . .
```

Figure 4-6 Test Code Fragment (1)

4.6.1.2. Exception based Similarities

In situations where test cases make invocation to methods that perform similar functions, similar exceptions are thrown. Thus the test cases have redundant exception managing `try-catch` block clones. For example, in the file `WindowManagerPermissionTests`, every call to the `IWindowManager` interface should throw `SecurityException`. Thus `try-catch` blocks are redundant inside the test case as shown in Figure 4-7.

```
//. . .
try {
            mWm.updateRotation(true, false);
            fail(". . .");
        } catch (SecurityException e) {
            // expected
            . . .
        } catch (RemoteException e) {
            fail(". . .");
        }
//. . .
```

Figure 4-7 Test Code Fragment (2)

4.6.1.3. Set-up/ Tear-down Similarities

Test cases are usually run by a test runner class that loads the test case class. In addition the test cases would set up the required fixtures (or data), runs and would finally tear down each test. In this context, a majority of the redundancies in test cases are found in set up and tear down methods. Our analysis of the calendar feature testing in Android repository reveals that there are fifteen test methods (with slight variations in text fixture values) occurring in `RecurrenceProcessorTest`. A sample fixture structure that gets repeated is shown in Figure 4-8.

```
 //. . .
@SmallTest
public void testMonthlyXX() throws Exception {
        verifyRecurrence( "20110703T100000" ,  "FREQ=MONTHLY;BYDAY=SA,SU;BYSETPOS=2,-2" ,
           null /* rdate */ , null /* exrule */, null /* exdate */, "20110701T000000" ,
          "20110931T235959", new String[]{"20110703T100000" ,"20110730T100000" ,
                "20110807T100000" ,"20110827T100000" , "20110904T100000" ,
                      "20110924T100000" } );
    }
//. . .
```

Figure 4-8 Test Code Fragment (3)

4.6.1.4. Mock Object Similarities

To facilitate dependency injection in testing, Android provides classes that create mock system objects such as `Context` objects, `ContentProvider` objects, `ContentResolver` objects, and `Service` objects. Test cases provide mock `Intent` objects. Testers use these mock objects both to isolate tests from the rest of the system and to facilitate dependency injection for testing. These classes that are found in the packages `android.test` and `android.test.mock.Mock` objects isolate tests from a running system by stubbing out or overriding normal operations. While testing the lifecycle events of a particular graphical component, similar mock objects are initiated and removed causing test similarities. Our analysis revealed several such multiple redundancies in test methods relating to mock objects' life cycle management codes.

4.6.1.5. Activity and Service Based Similarities

An `Activity` is a single focused task in Android context; for example, creating a pop-up dialog is an Activity. `Instrumentation framework` is the utility that allows monitoring of all interactions inside an application or platform. Activities have a complex lifecycle based on call-back methods; these methods can't be invoked directly but only through `Instrumentation`. The activity testing API base class is `InstrumentationTestCase`, which provides instrumentation to the test case subclasses that are used for testing UI activities. A `Service` object is a component that performs long operations in background

without UI intervention. Android provides a testing utility for `Service` objects that can run tests in isolation. Since the Service class is isolated, we can test a Service object without using instrumentation. Redundancies are found in both `Activity` and `Service` objects related test cases.

4.6.1.6. UI related Similarities

UI testing ensures that the framework returns the correct UI output in response to a sequence of user actions on a device, such as entering keyboard input or pressing toolbars, menus, dialogs, images and other UI controls. Functional or black-box UI testing does not require testers to know the internal implementation details of the app and it is sufficient that testers know only its expected output when a user performs a specific action or enters a specific input. The Android SDK provides tools to support automated, functional UI testing using `UIAutomator` and `TestRunner`. Redundancies are observed among test cases using `UIAutomator` and `TestRunner` utilities.

4.6.1.7. Functional similarity

One of the objectives of test cases is to verify specific *functional requirements*. To demonstrate functional similarity, we have picked an example where many functions are designed symmetrical across the screen. For example, consider testing a particular hand gesture action such as swipe or pinch. The input parameters may contain variations depending on the event under test, location and context. But the function call sequences being tested are similar. Thus test cases contain similar test codes but different input sets. This causes redundancies as evident from our analysis of test libraries. For example, Figure 4-9 shows a test case that validates whether or not a touch gesture used to grab a screen works properly. The input parameter to this test function is the content that is “touched” which could be either a hyperlink or a UI action. Consequently two sets of codes that assert the two event activities are present in the test library as seen in Figure 4-9; the code on left column for hyperlink and right for UI action.

```
public class  GridTouchSetSelectionTest extends
   Activity InstrumentationTestCase <GridSimple> {
      private GridSimplemActivity;
      private GridViewmGridView;
      ...
     @LargeTest
     public void testSetSelection()      {
       TouchUtils.dragQuarterScreenDown( this);
       TouchUtils.dragQuarterScreenUp( this);
       //... LARGE IDENTICAL CLONE....
       assertTrue( "..." , found);
     }
  ...
}
```

```
public class ListSetSelectionTest extends
   Activity InstrumentationTestCase <ListSimple> {
      private ListSimplemActivity;
      private ListViewmListView;
      ...
      @LargeTest
      public void testSetSelection()  {
        TouchUtils.dragQuarterScreenDown( this);
        TouchUtils.dragQuarterScreenUp( this);
        //... LARGE IDENTICAL CLONE....
        assertTrue( "..." , found);
      }
   ...
}
```

Figure 4-9 Test Code Fragment (4)

4.6.2. Group 2 - Complex Redundancies

Contiguous segments of parametric test cases that have intervened code portions is an example for complex redundancy. Our analysis during this study reveals several instances where similarities get spread across files, directories and projects. Complex redundancies are not constrained by syntactical boundaries.

4.6.2.1. Device and Configuration Similarities

The study reveals the existence of repetitive test codes belonging to the devices and configuration highlighting similarities across files and directories. Closer examination reveals presence of design and architectural similarities. Let us take the example of test codes pertaining to testing of access permissions. Key permissions are to be tested on activities, package managers, windows mangers, service managers, SMS (Short Message Service) managers and vibration services. These test codes are seen as intervened test clones in the test library repository. The list of java classes that contain clones pertaining to the Permission Test Cases are listed in Figure 4-10 below:

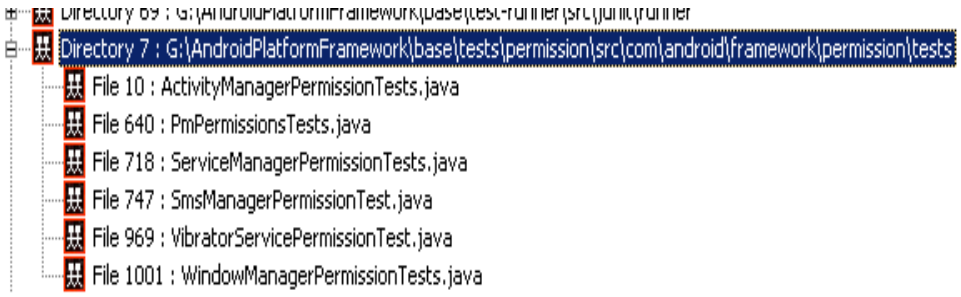


Figure 4-10 Permission Test Cases

4.6.2.2. Template

Test cases in Android repository usually had class level and package level templates. Class level templates include set up, tests and tear down. Package level templates include groups of test classes as test suites. Owing to the similarity in the domain, similarities are found in the class level and package level templates as shown in Figure 4-11. The figure shows that the testing files `HorizongalGravityTest` and `VerticalGravityTest` have method level similarities as provided in the file outline listing of the CM/CA tool.

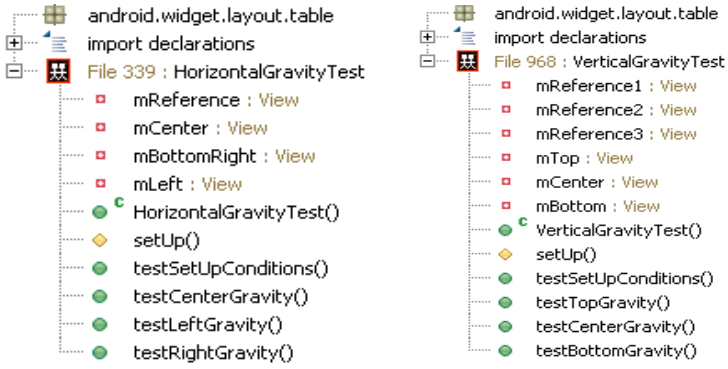


Figure 4-11 Template Similarity between two test case files.

4.6.2.3. File Level Similarities

Using CA/CM tool we applied clustering technique based on clone length and coverage metrics and located redundancies in the form of file level clones. Figure 4-12 depicts the CA/CM tool output revealing the existence of the file level gapped clone called `DownloadManagerBaseTest` in two different directories. The two instances have minor contextual differences and are managed as duplicated gapped clones.

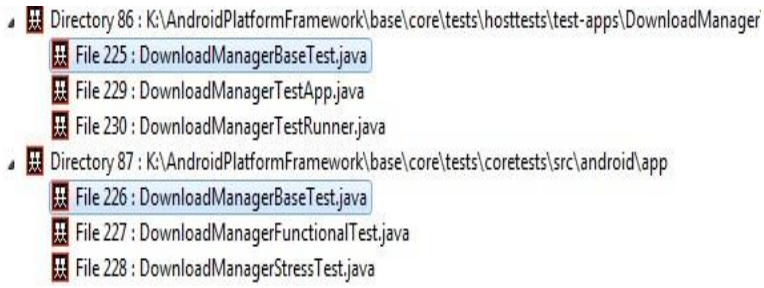


Figure 4-12 File Gapped Clone Occurrences

4.6.2.4. Call Sequence:

Android's complex functionalities get broken down into smaller activities since it uses a component based architecture. Hence test clones emerge while testing such complex functionalities and these test clones contain similar set of group of assertion statement calls as well as the sequence in which these assertion statement calls are made. An example highlighting call sequence redundancy is shown in Figure 4-13.

From the above findings it can be concluded that repetitive similarity patterns are found at various test libraries in Android platform framework. By identifying such redundancies and exploiting the similarities we can design generic adaptive test template structures that are much smaller, easier to construct and evolve than existing test libraries.

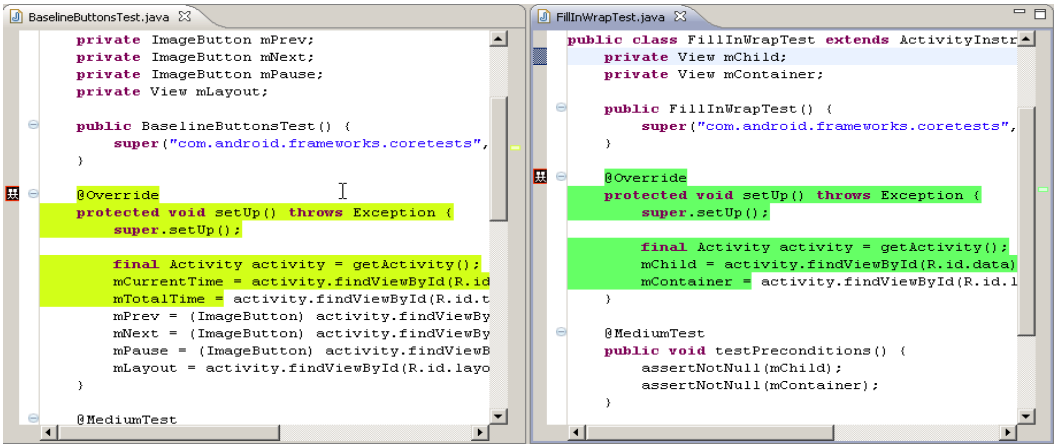


Figure 4-13 Call Sequence Similarity between Two Different Test Cases

4.7. Research Analysis

This section presents the results of research analysis of the test libraries using quantitative and qualitative analysis methods.

4.7.1. Quantitative Analysis

The following tables present a summary of quantitative findings from the study conducted on Android platform framework test libraries. The test library comprises of both unit and integration test cases. Thus the findings and insights are more applicable to white box testing approaches.

Table 1 Sample Selection

Android Platform:
Total Android Framework Code Base Size ~17 GB; Total 26767 Files; 9300824 LOC ~9300KLOC
Study Sample Focus – Framework Repository:
Framework Codes = 2.23 GB On Disk; Test Files In Framework Project: 1012 Selected Test Files For Study: 1007 Java Files (Unit /Integration/UI Tests); 12+MB;

A summary of study findings is shown in Table 2.

Table 2. Summary of Clone Analysis

Attribute	Measure
Total Directories Analysed	224 Directories
Total Test Files (Java Classes) Analysed	1007 Files
Total Methods Analysed	9728 methods
% Methods Containing Simple Test Clones	79%
% Files Containing Simple Test Clones	53%
Average Length of Test Clone	53 Tokens
Maximum Length Of Test Clone Found	1290 Tokens
Minimum Length Of Test Clone Found	30 Tokens
Simple Test Clone Class	2407 Files
Simple Test Clone Methods	7731 Methods
Parametric Test Clone Found Within File	779 Instances
Parametric Test Clone Found Across Files	335Instances
Complex Test File Clones Within Directory	12 Files
Complex Test File Clones Across Directories	11 Files

Two key observations from the above table are: (1) At least 53% of test files contain some form of redundancy. (2) At least 79% of test methods comprised of some form of redundancy.

4.7.2. Qualitative Analysis

This section presents qualitative aspects of the findings from our study conducted on Android platform framework test libraries. To perform the analysis we first grouped the test cases test clone types. Next, detailed study on similarities within each group was taken and a typical example of each type was catalogued. We also illustrate similarity groups with appropriate examples and explanations. Although the examples are not equally distributed, each group of test clone indicates a few possible causes for redundancies. The following discussion summarizes our study findings against the originally set research objectives.

1) To establish the presence of large scale redundancies within test libraries of a typical SPL (Android Platform in this study). Around 53% of test files have some form of redundancy. Test clones vary in type, complexity, token length, and variations. From the study we observe that there could be various reasons for the occurrence of test clones. Typical examples are presence of test smells, lack of reuse-based test case design and parametric combinatory explosion of test data.

2) To identify, and analyse, similarities found in the 'Android Platform Framework Test Libraries'. The research study catalogues simple similarities (Group 1) found among repeated test fixtures, exception management, test data set-up/tear-down, mock object lifecycle management, Activity & Service events, UI components and functions. The study also catalogues complex similarities (Group 2) caused among device settings, configurations, call sequences, test case template structures and test files. Removing similarities at the language level requires changes to the test libraries. In existing Android system, test

clones are tolerated in spite of their negative effect on maintenance in order to avoid the risk of breaking a functioning test library in an attempt to remove redundant clones. Different techniques can be used to realize reuse based evolution of test libraries.

3) To analyse the findings with regard to redundancies and to come up with insights that would help in design of generic adaptable test cases. To some extent these redundancies can be managed by programming level reuse techniques; but they are not exhaustive in expressing product line commonalties and variability. For example, consider the try-catch exception management structures (example discussed in section 6.1.2) that are being repeated several times in the Android platform test libraries. It is an expressive limitation of the underlying Java language. Such kind of generality demands a meta-level template composition based on planned reuse approach that makes the template structures independent from underlying platform and programming language. The proposed approach complements traditional testing techniques very well by addressing poorly supported test code scripting language paradigms.

4) Use the knowledge derived through this research study to enhance and formalize the test clone taxonomy/definitions/nomenclature which would contribute to the body of knowledge in testing domain. It is observed from the literature that current executable test libraries are maintained using test design reviews. These reviews can only identify reuse opportunities and does not suggest implementation strategies. Test libraries construction and management deals with identification of redundancies (test clones), re-construct non-redundant forms and generate actual test libraries from templates. Identification of test clones needs a formal nomenclature and taxonomy. Thus we formalize test clone definitions and taxonomy and provide a simple metric system for measuring reusability and

maintainability of test libraries in the forthcoming chapters. This will provide a means for us to compare the state of original test libraries as against the automated ones created using the previously suggested meta-level template composition technique. These formalizations are explained in Chapter 5.

McGregor [100] in his technical report states "Product line test architecture must address specific range of variability and the relevant accompanying binding times. If the range in the product line is very large, it may be reasonable to have multiple architectures and this usually happens between test points." Thus SPLT demands mechanisms that are not limited to programming language or platforms. Observations recorded by us while conducting the experiment on Android test libraries also confirms McGregor's opinion. This calls for systematic reuse approach that includes language independent variability mechanism that seamlessly connects the heterogeneous test libraries' commonalities while also preserving reuse and variants

4.7.3. Research Questions Answered

The goal of this study is to find frequently redundant test code patterns in Android platform framework test libraries and empirically evaluate the similarity patterns. Thus in summary the experimental study conducted on a typical software product line, namely the Android platform, clearly answers the research questions that we had posed initially:

1) **Research Question #1**: What kinds of similarities are found in test libraries? How frequently do they occur and in what granularity?
    - Around half of exiting test case files (53%) are found to have some form of redundancy.

- Although a majority of redundant test case codes we found were either identical or parametric in nature, our study also uncovered some instances of complex test similarities.
- We also observe that unique similarity patterns are found in situations such as exception management because of lack of error management facilities in the underlying programming language constructs.
- Finally we also observe redundancies due to configuration similarities which may be attributed to the lack of expressive ability of the underlying programming language.

2) **Research Question #2**: How are heterogeneous test assets of the current test libraries managed?
   - Currently the test libraries comprises of three types of test artefacts. Java, C++ and XML files.
   - In general, Android creates separate Java and C++ test library (project work spaces). But few instances (example codec, native and base projects) are observed to host both Java and C++ codes together.
   - XML is used for all types of configuration in all test libraries.
3) **Research Question #3**: How scalable are conventional test library construction techniques?
   - Scalability of test libraries are currently limited by two factors: underlying programming language and IDE (Integrated Developer Environment) abilities.
   - Scalability of programming language expressiveness is beyond the scope of this research study.

- On the IDE aspects, the test library scalability is guided by Eclipse project scalability settings. Some current scalability issues observed are slowing down of indexing, outline view and syntax colouring operations as the size of test library grows. These are mitigated by disabling the relevant operations from the active editor.

4.8. Threats to Validity

This study comprises several steps, combining two research methodologies: the exploratory study and the evaluation based on a tool experiment. There are a few threats to validity:

- The selection of the subject under study is based on open source license model. There could be different types of test libraries existing in the commercial sector that is the researcher is unaware of.
- Since the subject under study is a large scale test library, initial test clones were identified by the CM/CA tool. There are possibilities that some types of test clones are missed by the tool during clone detection process.

The study does not claim to have generalized all possible test similarity occurrences. There could be more causes for the test similarities (test clones) than those being listed in the analysis section. The study attempted to observe and classify as many different groups of test code clones as possible based on past study experiences and successes achieved from other studies [15, 130, 131] on software clones using the same tool. To address this issue we validated our findings seeking the expertise of clone researchers. Gapped test code similarities and other complex structural test clones are human interpreted and so it is possible that another researcher would have identified a different list of important prioritization factors. However, this is not considered a major issue since the priorities can be consolidated based on inputs from other researchers, if required.

4.9. Chapter Conclusion

This chapter presented the results of similarity analysis performed on the Android platform framework test libraries. The results confirmed that of the earlier proposed hypothesis that "reuse approaches employed in a test library can boost productivity" by affirming the presence of redundancy in nearly half of the exiting test case files. Most redundant test codes (test clones) that were detected were either identical or had parametric variations. The research study also uncovered some instances of complex and structural repetitions. Notably, the study also identified presence repetitive patterns in Android platform framework whose handling would be clearly beyond the capability of conventional testing techniques. As with the Android platform, most typical software product line test libraries usually have significant amount of such complex redundancies. Thus these redundancies necessitate a generic reuse technique with variability management. With the understanding gained from this research study, we intend to propose a systemic template based reuse approach for large scale test libraries that will exploit similarity present among test cases. The proposed approach can be further implemented by selecting a particular test library and rebuilding that test library using generic adaptive test templates to see if it confirms the proposed hypothesis fully, further analyse the benefits and shortcomings of the approach.

CHAPTER 5
Test Clones - Formulation & Definitions

In the previous chapter we had demonstrated that there are a lot of redundancies in test libraries called “test clones”. The excessive creation of test clones may become unproductive as the test library grows due to the efforts needed for maintaining the duplication. Our research work focuses on creating a strategy to manage test clones in a productive and scalable manner. However, to formulate this strategy, standardized definitions for test clone and related theory have to be identified or established. A review of available literature reveals an absence of clear definitions for test clones. We hence propose to develop the necessary theories for test clones. This chapter defines test clones and formulates a set of taxonomy, granularity and metrics related to test clones as part of this research thesis. The chapter also illustrates these definitions with the support of examples.

The organization of this chapter is as below:

- Section 1 introduces the chapter.
- Section 2 defines test clones by describing basic testing terms, building a software test system nomenclature and finally defining types of test clones.
- Section 3 provides examples for various types of test clones and further describes test clone taxonomies based on similarity or granularity.
- Section 4 defines test clone metrics to measure reusability and maintainability of a test library.
- Section 5 concludes the chapter.

5.1. Introduction

Several authors have presented guidelines, as well as examples for code clone literature [80, 138, 140]. Though there are definitions, detection techniques, taxonomy and industry experience reports in code clone literature, there is a lack of mapping between general code clone terms and test library artefacts. Hence this chapter attempts to provide definitions for general and structural test clones, build a taxonomy based on similarity patterns and suggest possible metrics that would support in scientifically assessing the influence of test clone on reusability and maintainability of test libraries.

In this chapter, we pursue three objectives. First, we formalize the test definitions and use it as a means to build test clone taxonomy. Secondly, we provide descriptions for test clone granularity in terms of physical and logical syntactical boundaries. Finally, we propose a set of metrics for reusability and maintainability of test libraries. For this discussion examples drawn from a mobile product line are used for illustrative purposes. The definitions, taxonomy and metrics would be used for the research work presented in subsequent chapters.

5.2. Test Clone Definitions

The definitions of test clone in SPL and clone literature are inherently abstract and the existing definitions are specific to the underlying clone algorithms used. It therefore becomes necessary to formally define a *test clone*. For the sake of consistent understanding throughout this thesis, this section attempts to provide formal scientific definitions for test libraries and test clone related terms.

5.2.1. Basic Terms

A **manual test** is a test that is performed by a human expert, written in readable natural language. In manual test, activities such as data inputs, analysis of the output and evaluation of results are all performed manually without any significant tool support.

Automated tests are performed without manual interaction and are usually assisted by testing tools. A semi-automated test is a test which consists of automated as well as manual parts.

The **unit test** validates the behaviour of a single component, method, or class against its input/output behaviour specified in the corresponding signature. The **integration test** validates the behaviour of two or more components that together form a configuration as specified in the architecture. **System tests** are usually conducted on complete systems to evaluate the software system's compliance against the system requirements specification.

Test Case is a sequence of executable statements (including data input and output) that stimulate a certain situation within a software system with the intent of validating the system's behaviour. Test cases describe how to operate the application, collect results and verify against expected outcomes. The test functionality structure can be broken down as:

1) Setup - acquire the necessary resources,
2) Stimuli - send one or more stimuli to the unit under test,
3) Verify - verify that the unit responds properly, and
4) Teardown - release the acquired resources.

The above set of steps of a typical test case is called the *setup-stimulate-verify-teardown (S-S-V-T)* cycle.

5.2.2. Software Test System Nomenclature

To define the software test system nomenclature we first present the constituents of a typical Software Test System which is depicted in Figure 5-1. We use the approach proposed by Van Rompaey et al [150] for unit testing as the starting

point to further enhance and modify the test clone definitions in product line context.

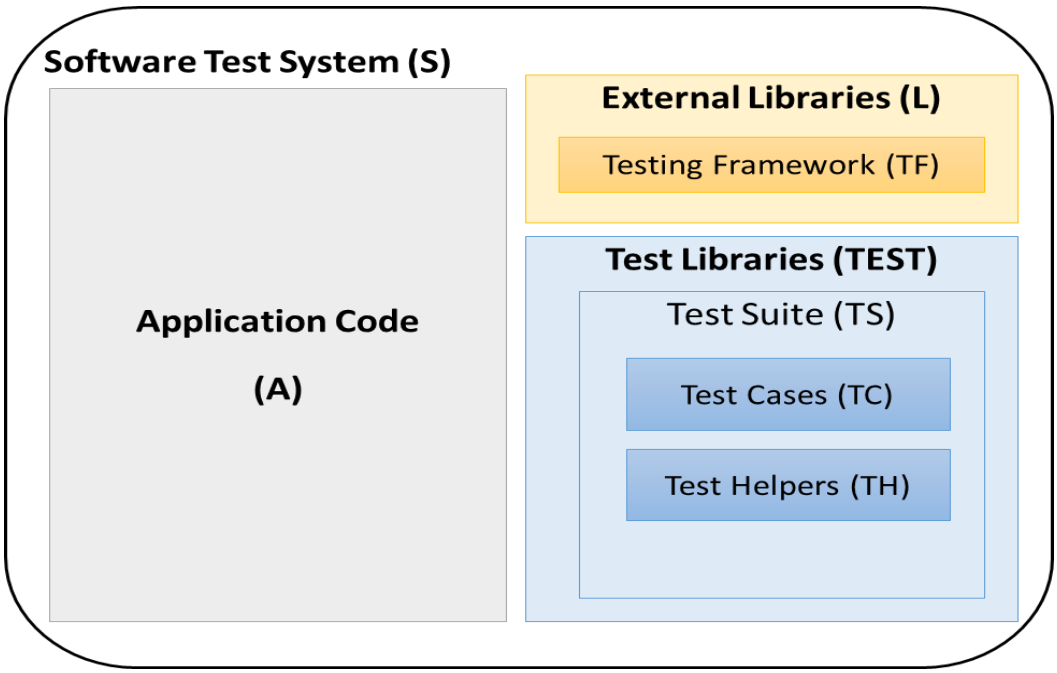


Figure 5-1 Software Test System

Software Test System:

This is the universal set that comprises of application code, external libraries and test libraries. A Software Test System is the collective set of all codes, libraries and configurations that are required for successful testing and deployment of the application to production environment.

External Libraries

These are a set of software libraries and tools that are not internally developed by the software development team but are being used by the application code. In addition to application software, the external library code would also consist of testing framework used by test libraries.

Application Code:

This is the actual application code developed by the application team and this code is the target for testing.

Test Libraries:

These are components implemented by the testing team that would be executed against the application code for defect detection, functional verification and system validation. Test libraries can be further subdivided into test suites, test cases and test helper classes. In addition, test libraries may use the testing framework of above mentioned external libraries code.

Depending on the type of testing framework employed there may be variations in terms of how the test helper classes are grouped. However this discussion ignores these minor variations and we would address them collectively as Helper Classes.

5.2.3. Test Library and Test Clone Definitions

The following definitions use set language notation to describe the various constituents in the Software Test System:

5.2.3.1. Definition 1: Software Test System (S).

An object oriented software test system (S) is composed of application code (A), external libraries (X) and test libraries (L).

$$S = A \cup X \cup L \; is \; a \; set \; of \; classes$$

5.2.3.2. Definition 2: Methods (M) and Attributes (AR).

$M(C)$ is the set of all methods of a software system C, with C expressed as a set of classes.

$Par(m)$ is the set of parameters of method m.

$AR(C)$ is the set if all attributes of system C.

$AR(c)$ for all attributes of an individual class c.

$Descendents(c)$ is the set of descendants of class c.

$M_D(c)$ is the set of methods declared in class c.

$M_I(c)$ is the set of methods implemented in class c.

$M_{OVR}(c)$ is the set of overriding methods in a class c.

$M(c)$ Stands for the set of declared or implemented methods of a single class c.

$$\text{i.e., } M(c) = M_D(c) \cup M_I(c)$$

$IM(c)$ is the set of methods invoked in a class c.

$AR(m)$ stands for the set of attributes referenced by method m.

5.2.3.3. Definition 3: Testing Framework (TF).

An object oriented testing framework TF is an external library.

$TF \subseteq L$ *is a set of classes or other libraries*

- There exists a base test case $btc \in TF$ that is the base class (provided by the testing framework) for all test cases of the system.
- $M(TF) \subseteq M(L)$ is the set of all testing framework methods and is represented as $\bigcup_{C \in TF} M(c)$
- The base test setup method bts contains the basic functionality for setting up a system under test into the right state ready for testing. Formally $bts \in M_I(TF)$, individual test case can override this method to add custom setup needs.
- A testing framework also contains a set of *test framework check methods* $TFCM \subseteq M(TF)$, used to check and report on a test's outcome by comparing the actual result with the expected outcome. Check methods

vary in the expected result, the precision requirement or the comparison mechanism.

5.2.3.4. Definition 4: Test Library (TEST).

A test library is composed of one or more test suits organized in a hierarchical fashion. Informally, we define test library (test code) as the set of classes that are either test cases, access methods or attributes of test cases. All other classes are considered production code.

- Test Library (also referred as Test Code) is defined as the union of the set of test cases and all other types that access test case methods or attributes.

$$TEST = TC \cup \{c \in C \mid ((IM(c) \cap M(TC)) \cup (AR(c) \cap AR(TC))) \neq \phi\}$$

- Production code is defined as all application code that is not test code.

$$PROD = C \setminus TEST.$$

5.2.3.5. Definition 5: Test Method (TM).

For a system under test in a certain state, a test command is a container for a single test. It is typically implemented as a method of a test case containing the stimulation and verification phases of the Setup-Stimulation-Verification-Teardown cycle. There are no parameters that influence the outcome of the test. [$TM = M(TC)$]

5.2.3.6. Definition 6: Test Suite (TS).

A test suite of test library is composed of test cases classes (TC) and test helper classes (TH). It may also comprise of project and environment settings configuration aspects. Test suite can be defined as a logical grouping of related test scenarios.

$$TS = TC \cup TH \text{ is a set of test classes}$$

5.2.3.7. Definition 7: Test Cases (TCS).

In an OO programming language context, a test case may be a single or group of classes of the test library (TL). Formally, test case is a set of classes where

$TCS = Descendents(btc) \cap C$:

- $M(TC) \subseteq M(C)$ is the set of all methods of test case classes and is represented as $U_{C \in TC} M(c)$.
- $A(TC) \subseteq A(C)$ is the set of all test case attributes and is represented as $A(TC) = U_{C \in TC} A(c)$.

5.2.3.8. Definition 8: Test Fixture (TF).

The test case fixture is the set of properties required in a test case to bring the system under test into the desired initial state. It contains both instances of the system under test and other needful shared data objects.

$$TF = A(c), (btc) \cap C$$

5.2.3.9. Definition 9: Test Setup Method (TSM).

Informally, a test setup method initializes a test case fixture into the desired state for testing. This method is invoked before every test command to reinitialize the test case fixture, resulting in isolated tests. Formally, for each test case tc, let

$$TSM(tc) = \{ m \in M_{OVR}(tc) | m \; overrides \; bts \}$$

5.2.3.10. Definition 9: Test Helper Method (THM).

A test helper method is a method in the test code that supports a set of test commands (e.g., providing an abstraction for checking common results).

5.2.3.11. Definition 11: Test Code Fragment (TCF).

A test code fragment is any sequence of code lines (with or without comments). It can be of any granularity, e.g., a class, a method, a function definition, begin-end block, sequence of statements, etc. A TCF is identified by its file name and

the block of statements indicated as begin-end line numbers in the original code base and is denoted as a triple:

$$(TCF.FileName, TCF.BeginLine, TCF.EndLine).$$

5.2.3.12. Definition 12: Test Clone (TC).

A test clone is a (consecutive) substring of a test with a certain minimal length, appearing at least twice in a test suite. A test clone relation exists between two test code clone fragments TCF1 and TCF2, *if and only if* TCF1 and TCF2 satisfy certain threshold of pre-defined similarity or identical measures. A test code fragment TCF2 is a clone of another test code fragment TCF1 if they are similar when some given definition of similarity is applied, that is, $f(TCF1) = f(TCF2)$ where f is the test functional similarity (defined later). Two fragments that are similar to each other form a test clone pair - denoted by$(TCF1, TCF2)$.

5.2.3.13. Definition 13: Test Clone Group (class).

Test code fragments that satisfy certain threshold of pre-defined similarity or identical measures are called test clone class or group - denoted by $(TCF1, TCF2, TCF3, \ldots TCFn)$. A test clone group consists of test clones within a test library.

5.2.3.14. Definition 14: Simple Test Clones (TC).

Test code fragments containing predefined amount of similarity are called General Test Clones. Simple test clone is often the result of copying a code fragment and pasting it into another location. This would include (1) Identical test code fragments that vary only in whitespace, layout and comments; (2) Syntactically identical test code fragments with variations in identifiers, literals, types, parameters and variables. (3) Test code fragments that have additional variations in terms of added, modified and removed statements.

5.2.3.15. Definition 15: Structural Test Clones (STC)

Structural test clones are a special type of general test clones. These fragments can be similar based on their test functionality structure (independent of their test code text). Test functionality structure is the S-S-V-T cycle defined previously. Structural test clone fragments are semantically or functionally identical coarse grained test clone groups that perform the same computation but implemented through different syntactic variants.

In the context of this research thesis, the term *test clone* refers to either general test clones or structural test clones; and having text based or syntactical similarity. Semantic test clones inferring to computational similarity based on program dependency graph (PDG) is not addressed in this study.

5.3. Test Clone Examples

In this section we provide some examples of test clone types with the support of test code fragments. These would be mapped against test clone granularity and unification template creation in subsequent chapters.

5.3.1. General Test Clones

5.3.1.1. Exact Test Clones

Two or more test code fragments are called exact test clones if they are identical to each other with some differences in comments and whitespace or layout. Editing activities like changing the comments, restructuring in layout i.e., changing the positions of begin, end brackets (e.g., "{" "}") or other language elements through adding/removing tabs, blanks, new lines may have been applied in the copied fragment. Figure 5-2 shows an exact test clone example (the bold text in figure shows the difference between the original and the clone).

```
package com.android.rs.image2;
. . .
public class TestBase  {
    protected final String TAG = "Img";
    protected RenderScript mRS;
    protected Allocation mInPixelsAllocation;
    protected Allocation mInPixelsAllocation2;
    protected Allocation mOutPixelsAllocation;
    // Override to use UI elements
    public void onBar1Changed(int progress) {      }
    public void onBar2Changed(int progress) {      }
    public void onBar3Changed(int progress) {      }
    public void onBar4Changed(int progress) {      }
    public void onBar5Changed(int progress) {      }
     . . .
}
```

```
package com.android.rs.image;
. . .
public class TestBase  {
    protected final String TAG = "Img";
    protected RenderScript mRS;
    protected Allocation mInPixelsAllocation;
    protected Allocation mInPixelsAllocation2;
    protected Allocation mOutPixelsAllocation;
    // Override to use UI elements
    public void onBar1Changed(int progress) {      }
    public void onBar2Changed(int progress) {      }
    public void onBar3Changed(int progress) {      }
    public void onBar4Changed(int progress) {      }
    public void onBar5Changed(int progress) {      }
     . . .
}
```

Figure 5-2 Exact Test Clone Sample

5.3.1.2. Renamed Test Clones

Renamed clones have variations in identifier names, literal values, comments or whitespace changes between the copied test code fragments. Renamed test clones are generally more inclusive of identifier changes. The example shown in Figure 5-3 below shows three test code fragments, original on the top, the clones on centre and bottom. Renamed clones are inclusive of parameterized clones (explained next) shown in centre as well as more generic clones as shown in Figure 5-3.

```
public void onClick(View v) {                                    Original
 try {
   int a = Integer.parseInt(value1.getText().toString());
   int b = Integer.parseInt(value2.getText().toString());
   if (a > b)
     Integer answer = a + b;
   else
     Integer answer = b - a;
   result.setText(answer.toString());
 } catch (Exception e) { Log.e(LOG_TAG, "Failed to add numbers", e);
}
```

```
public void onClick(View v) {                    Renamed & Parametric
 try {
   int val1 = Integer.parseInt(value1.getText().toString());
   int val2 = Integer.parseInt(value2.getText().toString());
   if (val1 > val2)
     Integer answer = val1 + val2;
   else
     Integer answer = val2 - val1;
   result.setText(answer.toString());
 } catch (Exception e) { Log.e(LOG_TAG, "Failed to add numbers", e);
}
```

```
public void onClick(View v) {                              Only Renamed
 try {
   int val1 = Integer.parseInt(value1.getText().toString());
   int val2 = Integer.parseInt(value2.getText().toString());
   if (val2 > val1)
     Integer answer = val1 + val2;
   else
     Integer answer = val1 - val2;
   result.setText(answer.toString());
 } catch (Exception e) { Log.e(LOG_TAG, "Failed to add numbers", e);
}
```

Figure 5-3 Renamed and Parameterized Test Clone Sample

5.3.1.3. Parameterized Test Clones

A parameterized test clone or p-match test clone is a renamed clone with systematic renaming. The detection techniques for parameterized test clones usually perform consistent matching of identifiers and/or literals among test code fragments. The parameter clones are a sub-type of renamed test clones. On the example previously illustrated (Figure 5-3) the original and the centre test clone fragment is a parameterized clone that has a and b identifiers renamed with var1 and var2.

5.3.1.4. Near-Miss Test Clones

Near-miss test clones are those test clones where the copied test code fragments are very similar to the original. Editing activities such as changing in comments,

layouts, changing the position of the source code elements through blanks and new lines, changing the identifiers, literals, macros may have been applied in such clones after the copy was made. Near miss test clones despite having the above said changes still have the syntactical structure of the original. However authors [138] have not provided a clear metric to quantify or qualify the accepted level of modifications allowed on the copied fragment from that of original. Incidentally, all parameterized and renamed test clones are also a subset of near-miss clones.

Figure 5-4 shows a sample of a near-miss test clone that has both parametric as well as class name modifications. Though the second clone has a copy of test code fragments that is very similar to the original, there are changes and modifications to syntactical structure as highlighted in bold.

```
public class GridTouchSetSelectionTest
 extends Activity InstrumentationTestCase
    <GridSimple> {

 private GridSimple mActivity;
 private GridView mGridView;
 ...
 @LargeTest
 public void testSetSelection()
  {
   TouchUtils.dragQuarterScreenDown(this);
   TouchUtils.dragQuarterScreenUp(this);
   //... LARGE IDENTICAL CLONE....
   assertTrue( "..." , found);
 }
 ...
}
```

```
public classListSetSelectionTest
 extends  Activity InstrumentationTestCase
    < ListSimple> {

 private ListSimple mActivity;
 private ListView mListView;
 ...
 @LargeTest
 public void testSetSelection()
  {
   TouchUtils.dragQuarterScreenDown( this);
   TouchUtils.dragQuarterScreenUp( this);
    //... LARGE IDENTICAL CLONE....
   assertTrue( "..." , found);
 }
 ...
}
```

Figure 5-4 Sample Near Miss Test Clone

5.3.1.5. Gapped Test Clones

A gap test clone code is partly similar to the original fragment. In this type of test clone, there is some difference in code portions between the test clones. The code portion that is different is called ***gap***. Let us consider a piece of original test code fragment (Figure 5-5). It can be seen that the gapped clones can have variations based on insertion, deletion or modification to original test code fragment.

```
Cursor allNotes = notesDbAdapter.fetchAllNotes();
assertEquals(1, allNotes.getCount());
allNotes.moveToFirst();
. . .                                                    Original
```

```
Cursor allNotes = notesDbAdapter.fetchAllNotes();
assertEquals(1, allNotes.getCount());
allNotes.moveToFirst();
assertEquals("my title", allNotes.getString(1));
assertEquals("the body of my note", allNotes.getString(2));
. . .                                    Gap – Test Code Insertion
```

```
Cursor allNotes = notesDbAdapter.fetchAllNotes();
allNotes.moveToFirst();
. . .                                     Gap – Test Code Deletion
```

```
Cursor allNotes = notesDbAdapter.fetchAllNotes();
assertEquals(3, allNotes.getCount());
allNotes.moveToFirst();
. . .                                  Gap – Test Code Modification
```

Figure 5-5 Gapped Test Clone Sample

5.3.1.6. Non-Contiguous Test Clones

Non-contiguous test clones are basically gapped test clones with multiple modifications and larger granularity. All the variations that were identified for gapped test clones are also applicable for non-contiguous test clones.

5.3.2. Structural Test Clones

Structural test clones are higher level clones that represent repeated structures, resulting from a repetition of a high-level design or similar feature. Structural similarity of test components (either unit, integration or system level test case codes) originates from similarities present in the test case design. Most structural test clones consist of a large number of test code fragments at the implementation level as a result of what we call ‘test clone fragmentation’. Test clone fragmentation is the phenomenon of coarse-grained clones actually manifesting as scattered patterns of fine-grained test clones, resulting in an intractable number of small test clones that we have to deal with. This fragmentation is a result of decomposition forces of the implementation technology, further exacerbated by injection of variations.

Structural test clones are similarities that could be of different granularities depending on the underlying programming language. For example, consider the java programming language; structural test clones may occur at the following boundaries:

- Declaration: class { ... }, interface { ... }
- Method: method, constructor, `static` initializer
- Statement: `if` statement, `for` statement, `while` statement, `do` statement, `switch` statement, `try` statement, synchronized statement.
- Block range surrounded with '{' and '}'.

In the same way, we can define the boundaries of structural test clones for other languages of interest. Structural test clones can comprise of various simple test clones such as exact clone, parameterized clone, renamed clones and gapped clones. The structural test clones identification process focuses on finding similar design structures after identifying the basic similarities like textual, lexical, syntactical and/or semantic similarities. While the simple test clones are based on the level of similarity between the code fragments, structural clones are based on the level of clone granularity of the language.

5.3.2.1. Functional Test Clones

Functional test clones are test code fragments where an entire test method is duplicated. Functional test clones are therefore, a subset of structural clones. For example consider the function `getview()` in `hwui` test library (Figure 5-6). The function creates and returns a text view by setting up resources as specified by layout file (i.e., adds display boundaries, padding and size. This same functional test clone call repeats itself in eight locations. Thus the need to manipulate the `View` related activities results in creation of structural clones.

```
@Override
public View getView(int position, View convertView, ViewGroup parent) {
   TextView v = (TextView) super.getView(position, convertView, parent);
   final Resources r = getContext().getResources();
   final DisplayMetrics metrics = r.getDisplayMetrics();
   v.setCompoundDrawablePadding((int) (6 * metrics.density + 0.5f));
   v.setCompoundDrawablesWithIntrinsicBounds(
         r.getDrawable(R.drawable.icon), null, null, null);
   return v;
}
```

```
ListActivity.java
TransparentListActivity.java
ViewLayersActivity.java
ViewLayersActivity2.java
ViewLayersActivity3.java
ViewLayersActivity4.java
ViewLayersActivity5.java
ViewLayersActivity6.java
```

Figure 5-6 Functional Structural Test Clones

5.3.2.2. Design Level Test Clones

Design level structural test clones are also a subset of structural clones. These higher level structural similarities are caused by design similarities among test cases. [77]. Consider the example provided in Figure 5-7 which shows two different test libraries as `fbotest` and `modelviewer`. Here the files with suffix `RS` and `View` are structurally similar to each other and spawns across different folders. These structural similarities are caused by the design similarity that accesses the open graphics library. In other words the test clones are dictated by the test case design that insist that the view must first access the rendering script which then interacts with the graphic library.

Figure 5-7 Design Level Structural Test Clones

The diagram below (Figure 5-8) illustrates the design similarity among the rendering script, view and graphic library for various entities such as FBOSync, FBOTest, SceneGraph and SimpleModel.

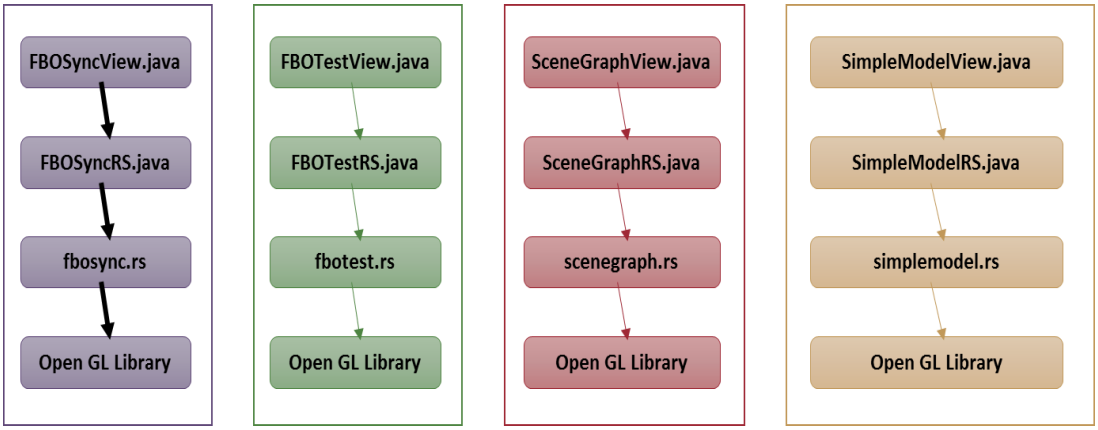


Figure 5-8 Design Similarity among Test Cases

5.3.2.3. File Level Test Clone

File level structural test clones are exact code fragments spanning the whole file syntactical boundaries with minimal variations. Variations could be few lines or package names and so on. Usually file level test clones match attributes, method codes, parentheses, quotation marks and comment delimiters.

In the following example (Figure 5-9), `BottomEditTextActivityPanScan` is an activity test case used to test *pan and scan actions* on a text graphical item.

`BottomEditTextActivityResize` is an activity test case used to test the *resize action* on similar text graphical item. that Both these test files have exactly similar test fixtures and manipulate identical graphical activities.

```
public class BottomEditTextActivityPanScan extends Activity
{
    private View mRootView;
    private View mDefaultFocusedView;
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        mRootView = new LinearLayout(this);
        ((LinearLayout) mRootView).setOrientation(LinearLayout.VERTICAL);
        View view = getLayoutInflater().inflate(R.layout.one_edit_text_activity,
                        ((LinearLayout) mRootView), false);
        mDefaultFocusedView = view.findViewById(R.id.dialog_edit_text);
        ((LinearLayout) mRootView).addView(view);
        setContentView(mRootView);
        this.getWindow().setSoftInputMode(
                    WindowManager.LayoutParams.SOFT_INPUT_ADJUST_PAN);
    }
    public View getRootView() {
        return mRootView;
    }
    public View getDefaultFocusedView() {
        return mDefaultFocusedView;
    }
}
```

```
public class BottomEditTextActivityResize extends Activity
{
    private View mRootView;
    private View mDefaultFocusedView;
    @Override
    public void onCreate(Bundle savedInstanceState)
    {
        super.onCreate(savedInstanceState);
        mRootView = new LinearLayout(this);
        ((LinearLayout) mRootView).setOrientation(LinearLayout.VERTICAL);
        View view = getLayoutInflater().inflate(R.layout.one_edit_text_activity,
                    ((LinearLayout) mRootView), false);
        mDefaultFocusedView = view.findViewById(R.id.dialog_edit_text);
        ((LinearLayout) mRootView).addView(view);
        setContentView(mRootView);
        this.getWindow().setSoftInputMode(
                     WindowManager.LayoutParams.SOFT_INPUT_ADJUST_RESIZE);
    }
    public View getRootView() {
        return mRootView;
    }
    public View getDefaultFocusedView() {
        return mDefaultFocusedView;
    }
}
```

Figure 5-9 File Level Test Clone Example

5.3.3. Test Clone Taxonomy

Test clones or test code duplication is generally observed to be common in large scale test libraries [5, 6]. Typically test clones can be characterized by repeated blocks of code performing similar test functionalities. Depending on context such repeated blocks can be either small or large (i.e., from as small as 30 token counts to as high as few hundred token counts). Test clone taxonomy is an attempt to measure how syntactic elements change within each test clone group. Test clone taxonomies can also be useful for test case design and test execution optimization.

For example, consider the renamed test clones that include two test code fragments, which are identical except for variations in attribute names and method

parameters. Understanding test clone fragments using a standard taxonomy would help the test designer in classifying fragments. Such a classification is useful as they contribute to better refactoring and maintenance techniques. Additionally by knowing the frequencies with which different groups of test clones occur, a test designer can concentrate his efforts by targeting test clones that have greater redundancies first. Such prioritization is expected to improve the productivity of the tester.

5.3.4. Taxonomy Based on Similarity

As stated above, a test clone relation exists between two test code fragments TCF1 and TCF2, if TCF1 and TCF2 satisfy certain threshold of pre-defined similarity. Similarity of test code fragments can further be defined in terms of lexical, syntactic or semantic structures. While machines can handle lexical and syntactic comparisons, they are unable to distinguish between semantic equivalence necessitating human intervention. Thus clone detection algorithms focus only on similarity of tokens. Token based clone detection algorithms scale well for larger libraries, have high recall and reasonable precision [88]. Human judgment is an important factor in deciding the similarity threshold. However the use of effective clone detection algorithm would largely assist in identifying the presence of similarity.

In test libraries, a given test code fragment can be classified based on physical or logical granularity. Examples of physical granularity include test method, test files or test directories. Examples of logical granularity include test stimulus code fragments, test fixture setup fragments, test helpers, test setup/teardown methods and test assertion fragments. Given this context, test clone taxonomy can be built based on syntactical boundaries such as method layout, expressions and control flow. Examples of such syntactical boundaries in a test library are test method signature, test fixture attributes, test methods and test helper methods.

Similarity relationship among test code fragments can be expressed using the following descriptions:

1) ***Equal***: Two test code fragments are considered equal if all of its associated syntactical boundaries are same.

2) ***Similar***: Two test code fragments are considered similar if the differences between the associated syntactical boundaries are less than a pre-defined delta threshold.

3) ***Distinct***: Two test code fragments are considered distinct if the differences of the associated syntactical boundaries are more than the pre-defined delta threshold.

The Table 3 summarizes the commonly used test clone terms and possible mapping to a clone type. The test clone code fragment boundaries can be of two types: fixed or free. Fixed syntax boundaries work on predefined syntactic boundaries or tokens (such as methods, begin-end block scopes etc.) while free text is not constrained by syntactical boundaries. Clone detection algorithms finds test clones depending on the boundary representations.

Table 3 Test Clone Similarity Taxonomy

Test Clone Group	***Test Clone Type***	***Boundary***	
		Fixed Syntax Boundaries	***Free Text No Boundaries***
Exact Test Clones	Simple Test Clone	X	X
Renamed Test Clones	Simple Test Clone	X	X
Parameterized Test Clones	Simple	X	X
Near Miss Test Clones	Simple	X	X
Gapped Test Clones	Simple	X	X
Non Contiguous Test Clones	Simple	X	X
Structural Test Clones	Simple and Structural	X	-
Functional Test Clones	Simple and Structural	X	-
Design Level Structural Test Clones	Simple and Structural	X	-

5.3.5. Taxonomy Based On Granularity

Another taxonomy we propose is based on test clone granularity. Test clone granularity can vary from single line of test code to complete abstract syntax trees or program dependency graphs. The structural test clones cover large granular repeated test codes. As it often happens in clone research, the actual definition of test clones is influenced by the underlying clone detection algorithm. In this research we have used the Clone Miner tool [9-16] and hence target to derive definitions that, apart from meeting generic test clone, would also address the specifics required in the Clone Miner tool approach.

Clone Miner takes a bottom-up approach towards the detection of test clones using the following hierarchy: Simple Test Clone Structures (containers being methods and files), Test Method Clone Classes, Test Method Clone Structures (containers being files and directories), Test File Clone Classes, Test File Clone Structures (containers being directories) and Test Directory Clone Classes; (abbreviated as STCS, TMCC, TMCS, TFCC, TFCS and TDCC respectively).

The higher-level structural test clones are built based on hierarchy of corresponding lower-level test clones. Thus the tool initially detects simple test clones based on the similarity of the transformed token strings generated by a lexical analyser. It subsequently groups them in terms of their container level (e.g., methods, files and directories). In the next stage, the Clone Miner detects larger recurring configurations of simple test clones using frequent item-set mining. Each of these recurring configurations of simple clones represents a possible first-level “structural test clone”, where the test clone fragments are at same container level. In the final step, the higher level structural clones that occur within higher-level containers are detected based on clues from previous lower level analysis.

The table below (Table 4) provides the taxonomy that we have created for the test libraries that would use the clone miner tool.

Table 4 Granularity Based Test Clone Taxonomy

Level	*Types of test clones found by Clone Miner*	*Exact Test Clones*	*Renamed Test Clones*	*Parameterized Test Clones*	*Near Miss Test Clones*	*Gapped Test Clones*	*Non Contiguous Test Clones*	*Structural Test Clones*	*Functional Test Clones*	*Design Level Structural Test Clones*
Level 1	**Simple Test Clone Structures (STCS)**									
	STCS Within Methods	X	X	X	X	X	X	-	-	-
	STCS Across Methods	X	X	X	X	X	X			
Level 2	**Simple Test Clone Structures (STCS)**									
	STCS Within Files	X	X	X	X	X	X	-	-	-
	STCS Across Files	X	X	X	X	X	X	-	-	-
Level 3	Test Method Clone Classes (TMCC)	X	X	X	X	X	X	X	X	X
Level 4	**Test Method Clone Structures (TMCS)**									
	TMCS Within Files	X	X	X	X	X	X	X	X	X
	TMCS Across Files	X	X	X	X	X	X	X	X	X
Level 5	Test File Clone Classes (TFCC)	X	X	X	X	X	X	X	-	X
Level 6	**Test File Clone Structures (TFCS)**									
	TFCS Within Directories	X	X	X	X	X	X	X	-	X
	TFCS Across Directories	X	X	X	X	X	X	X	-	X
Level 7	Directory Test Clone Classes (DTCC)	X	X	X	X	X	X	X	-	X

As observed from the table, general test clones are found at all levels and types as identified by the clone detection algorithm. But the structural test clones exhibit similarity only across syntactical boundaries and thus are observed at only at higher levels like test method, test file or test directory levels.

5.4. Metrics Exhibited in a Test Library

The current state-of-the-practice in testing involves the use of quality factors that pertain to test libraries. Hence an appropriate set of metrics should be defined to provide assessment for the identified quality factors. Such metrics can serve as an indicator for efficiency and effectiveness of software testing process. Analysis and evaluation of test clone metrics can identify areas for reuse and the test designers can use this input to refactor test libraries that are easy to maintain.

Thus, test clone metrics can assist in continuous improvement of test library creation and maintenance.

Based on a systematic analysis of SPLT projects and literature we have arrived at the key quality factors that measure the test libraries. These are reusability, maintainability, reliability, understandability, efficiency and testability. The framework proposed by McCall et al [98] refers to a systematic correlation between software quality factors and quality metric(s). Quality factors are further decomposed into low level attributes that are normally referred as quality criteria. Each of these quality criteria can be associated to quality factor based on the characteristics of quality factor and directly measured using a further lower level attribute known as quality metrics. Figure 5-10 illustrates the ordered relationship between quality factor, quality criteria and quality metrics.

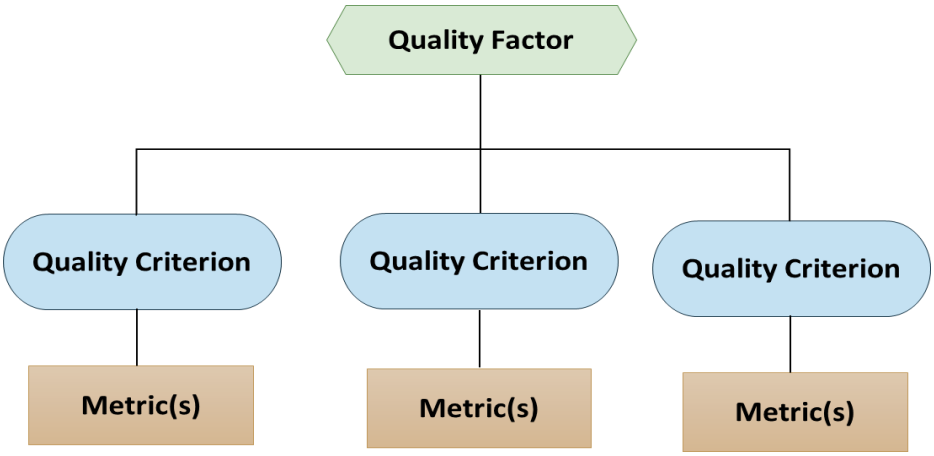


Figure 5-10 Software Quality Framework

While test library related metrics can be multi-dimensional, distributed and comprehensive, it is obvious that the effectiveness of the metrics is more important than the number of metrics used. A key point to be noted is that measurements can be in terms of both quantitative and qualitative indicators [78]. Yet any decision making should be driven by further analysis and not merely based on data. For the purpose of our research study, we have defined and analysed two key metrics exhibited by test clones that directly influences test library quality:

(1) Reusability – The extent to which a test case can be used in other test suites including packaging, variations, commonality and scope of feature under test.

(2) Maintainability – The effort required to locate and manage changes to test library that finds defects in the constantly changing application code.

The main reason for focusing on the re-usability and maintainability quality factors is that both these factors unambiguously help to measure quality of test libraries for longer life cycle, continuous execution and automated usage characteristics.

5.4.1. Test Library Reusability Metrics

Test cases are the essence of test libraries. In product line scenario, reusing available test cases is an effective strategy to improve the efficiency of the overall core assets. Reusability reduces maintenance cost, improves test comprehension and most importantly when test libraries are implemented using reusable cases properly, they preserve the unique design decisions. In order to efficiently reuse test cases, a unified standard format to describe test cases is needed.

Constructing test libraries in SPL context can be expensive and labour intensive. Thus the possibility of standardizing on the test case file format depends on the "reuse perspective" adopted. Usually reuse perspective depends on the generic features of the product lines irrespective of underlying technology used. For example, consider testing a product line of web applications, underlying test libraries contain generic test cases for common features such as login screen, forget password, registration information and page links. Identifying such common features helps to decide on the right combination of test case reuse strategy to be employed.

5.4.1.1. Three Reusability Perspectives

In testing context, reuse perspective depends on the test specification, design, generation and execution on underlying domain. Test library reuse uses three key factors: test fixtures (properties), states (of individual entities under test) and templates (pertaining to the application as a whole). Though the objective of the research work presented in this thesis confines to template based test library reusability, a brief overview of all three perspectives of test library reusability is provided below for completeness.

Fixtures Based Reusability

Test Fixture is the set of properties that are needed to bring the system under test to desired initial state. In general terms a test fixture or test context is the collection of one or more of the following items, required to perform the test: (1) preconditions; (2) particular states of test units; (3) necessary clean-up procedures. Though these tasks are encountered in many if not all test cases, what makes a test fixture different is repetition. Where a normal test case implementation does all preparatory and clean-up work by itself, a test fixture allows this to be implemented in a separate reusable unit. Since test fixtures attempts to capture reusable test data, the base programming language or the testing framework will usually provide the necessary implementation mechanisms.

State Based Reusability

Generally test suite is the set of logically connected test cases. These test cases may be manually written or generated using a common design paradigm. Different operation sequences of a test scenario with the same input data may cause different results. An operation sequence can be partitioned into two styles viz., state check and sate comparison. State check helps to determine if the state resulting from the execution of an operation sequence is as expected. While state comparison compares if different operation sequences arrives at the same state of

equivalence as provided by the specification. For example, a stack data structure can have three states: {Empty}, {Loaded} and {Full}. There can be eight possible test case scenarios based on state changes.

1. {Empty} $\xrightarrow{pop}$ {Empty}
2. {Empty} $\xrightarrow{push}$ {Loaded}
3. {Loaded} $\xrightarrow{pop}$ {Empty}
4. {Loaded} $\xrightarrow{pop}$ {Loaded}
5. {Loaded} $\xrightarrow{push}$ {Loaded}
6. {Loaded} $\xrightarrow{push}$ {Full}
7. {Full} $\xrightarrow{pop}$ {Loaded}
8. {Full} $\xrightarrow{push}$ {Full}

In many test cases design processes like above, the push method and the pop method can be reusable across test cases. Thus, the two methods could be extracted from different operation sequences and be stored in the library. State based reusability perspective takes into account such standard operation sequences and implements reusable methods or API on the testing libraries.

Templates Based Reusability

Another popular reusability strategy is the use of templates as a mechanism to formalize, preserve and reuse the test assets within test libraries. A single template fragment or file can be designed and distributed among the test library for reuse. Template based reusability ensures consistent design, ease of changes and clarity. Template based reuse approaches provide hierarchy, encapsulation and extendibility. Since test templates attempt to capture both commonality and variability in a software product line, it can be either implemented as part of test library or as a meta-layer using generative composition techniques. Each style has its own strengths and shortcomings.

5.4.1.2. Measuring Reusability

IEEE standards for software quality measurements define re-usability using quality criteria such as generality, modularity, machine independence, software system independence and self-descriptiveness. The standards further map these criteria to metrics such as unit referencing, unit implementation, modular design, modular implementation, hardware independence, software independence, effectiveness of comments and descriptiveness of language. William Frakes and Carol Terry [54] catalogue reusability metrics based on various categorization such as cost/productivity model, amount of reuse, level of reuse, reuse maturity, failure modes analysis and reuse library metrics. These discussions pertain to general software reuse.

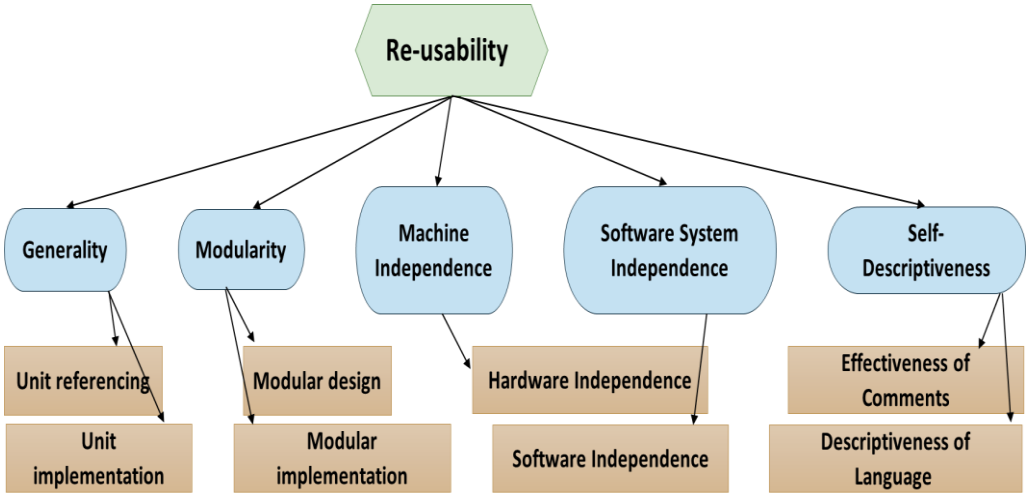


Figure 5-11 Metrics for Reusability

Along similar lines we may need to derive reuse metrics (Figure 5-11) that targets SPL testing context with the view to later using these metrics to assess the benefits and productivity gains derived by using the template-based testing approach proposed in this research. Reuse metrics targeting SPL testing can be classified into two groups. Static reuse metrics that measure reuse in terms of actual artefact (test libraries in our case) and dynamic reuse metrics that measure the run-time nature of reusable artefacts (test execution, defects and other runtime properties). Since the template based reuse approach pursued in this thesis focuses only on

test libraries, only those static reuse metrics related to productivity improvements, amount of test library reuse and level of reuse in test library will be discussed further.

Amount of Reuse

The *Amount of Reuse* metric is used to assess and monitor improvement effort by tracking percentages of reused objects over time. In an executable test library context, amount of reuse usually refers to lines of code. More precisely, amount of reuse is expressed in terms of executable lines of codes reduced in comparison to the old redundant test library by the reuse approach.

$$Amount\ of\ Reuse = \frac{amount\ of\ executable\ lines\ of\ code\ reused\ in\ the\ test\ library}{total\ amount\ of\ executable\ lines\ of\ code\ in\ the\ test\ library}$$

Hence, the *Amount of Reuse* metric denotes the percentage fraction measuring the extent of reused code lines to the total code lines. The metric is commonly expressed as a fraction or at times as a percentage. Unit of measurement is ELOC (Executable Lines of Code).

Reuse Level

The *Reuse Level* metric uses reuse improvement efforts as a dependent variable. This may vary among test libraries depending on the platform and the scripting language employed. However, reuse level measurement assumes that a test library is composed of parts that have different levels of abstraction. To measure reuse level, the levels of possible abstraction must be defined and interaction between the components (both internal and external) have to be measured. For example in the case of procedural language such as C, the components contain modules, functions and lines of code. In an OO language such as Java, the components contain test packages, test files, test methods, test fixture attributes and lines of test code fragments. A test library contains higher level components that are

composed from lower level components. For example, a test file is composed of test fixture attributes, test methods and test helper methods. The following quantities can be calculated:

$$L = The\ total\ number\ of\ lower\ level\ components\ in\ a\ higher\ level\ component$$

$$E = The\ total\ number\ of\ lower\ level\ components\ (from\ external\ libraries\ in\ a\ higher\ level\ component)$$

$$I = The\ total\ number\ of\ lower\ level\ components\ (from\ test\ libraries\ in\ a\ higher\ level\ component)$$

$$M = The\ number\ of\ components\ in\ a\ higher\ level\ component$$

$$External\ Reuse\ Level\ = \frac{E}{L}$$

$$Internal\ Reuse\ Level\ = \frac{M}{L}$$

$$Total\ Reuse\ Level = External\ Reuse\ Level + Internal\ Reuse\ Level$$

$$= \frac{E + M}{L}$$

Hence, the *Reuse Level* metric may be interpreted as the ratio between the sum of internal and external reuse level achieved by test libraries in comparison to the total reuse achieved in the software product line.

Depth of Inheritance

This metric calculates the depth of inheritance hierarchies. Shallow hierarchies forsake reusability for the simplicity of understanding, thus reducing the extent of method reuse within an application. *Depth of inheritance* indicates the extent to which a test file is influenced by the properties of its ancestors and the potential impact on its descendants. The depth of inheritance and number of children

collectively indicate the genealogy of a class. In the test template based reuse approach, this depth of inheritance metric can be converted as 'Depth of Template Tree (DTT)'. The deeper a template is in the hierarchy, the greater is the number of methods it is likely to inherit, making templates more complex. As a positive factor, deeper trees increase reusability because of inheritance feature.

$$Number\ of\ Template\ Children\ (NTC) = number\ of\ immediate\ subcomponents\ of\ a\ template$$

$$Depth\ of\ Template\ Tree\ (DTT) = maximum\ inheritance\ path\ length\ from\ leaf\ to\ root$$

In essence, the *Depth of template tress* metric denotes the count of maximum depth (from leaf node to root node) in the constructed hierarchical tree of test templates.

5.4.1.3. Reusability Maturity Model

A reuse maturity model is a core set of recommendations for planned reuse that helps organizations understand their past, current and future goals for reuse activities. The model assesses the organization's systematic reuse using an ordinal scale of reuse phases.

The Software Productivity Consortium has developed a Reuse Capability Model to serve as a basis for understanding and improving an organization's reuse capability[43]. The reuse capability model proposed has two phases: an assessment phase and an implementation phase. The assessment phase suggests a set of critical success factors (goals) that an organization can use to assess the present state of its reuse practice. The set of goals respectively identified for a software product are (1) Management (2) Application development (3) Asset development and (4) Process/Technology perspectives. The implementation phase prioritizes the above mentioned goals and builds the reusable asset in

successive stages. Research literature survey observes two other reuse maturity models but these are proposal and are not systematically validated by the authors [87].

Organizations use the 'Reuse Capability Model' to access their current reuse maturity and further to set measurable goals for improvement. Also the interpretations can be tailored in accordance to the requirements of software artefacts under consideration such as analysis models, codes and test libraries. From a test library perspective, the four stages of reuse maturity can be defined and interpreted as listed below.

1) Opportunistic: The opportunistic reuse strategy is developed for every individual test project. For example, when an initial test library is created the maturity model identifies specialized reuse tools such as refactoring wizards, test clone analyser and common test fixtures as reusable assets. This stage produces an inventory of reusable test artefacts with respect to the individual test library, which in case of Android test libraries would be test data, test methods, fixtures and layout configuration files.

2) Integrated: The integrated reuse strategy is defined and integrated for all test projects during the test development. For example, every time a new test library is needed a test project is created and connected to reusable test assets such as other common test libraries, testing frameworks and component libraries. In case of Android testing scenario, these reusable test assets would be Robotium, content providers, `UIAutomator`, `Service` and `Activity` testing components.

3) Leveraged: In leveraged reuse strategy the entire test project life cycle and is specialized for each product line. For example, the reuse performance is measured and the weakness of individual test project is identified. In case of Android testing scenario, this would be done through standard

tools such as tracing facilities available in testing tool kit and `AndroidTestCase` helper classes.

4) Anticipating: In anticipating reuse strategy, the new business ventures take advantage of the reuse capabilities and reusable assets. For example, every new test project is planned and designed to integrate with existing test projects. Here the high-payoff test projects or modules are identified and carefully reused. In case of Android testing the reuse technology is driven by the platform requirements.

Thus, the reuse maturity for a given test library can be assessed as opportunistic or integrated or leveraged or anticipating, where opportunistic is considered least and anticipating is considered the most matured state.

5.4.2. Test Library Maintainability Metrics

Prior to defining Test Library Maintainability Metrics for SPLT, this section attempts to briefly review existing literature on the Software Maintainability in SPL with a view to drawing correlations and proposing enhancements to the existing concepts to meet the test libraries metrics requirements.

Pressman[127] defines maintainability as the ease with which a software (test library in this thesis) can be understood, corrected, adapted and/or enhanced. Software maintenance metrics measure the interconnectivity of system components, efforts involved in different activities and measure how efficiently the system reacts to software change requests. Some key published research works that define metrics for maintainability in SPL context are listed below:

- IEEE [77], states: “It is reasonable to state that maintainability of software has its sub characteristics as adaptability, modifiability, testability, portability and understand ability.”

- Sanjay et al [77] map the maintainability metrics based on past research literature into set of thirty eight attributes {Adaptability, Analysability, Changeability, Cohesiveness, Compatibility, Complexity , Comprehensibility, Conciseness , Consistency , Correct ability, Documentation, Ease of Impact, Analysis , Expandability, Extensibility, Flexibility, Implementation, Install ability, Instrumentation, Integrate Ability, Level of validation and testing, Localizability, Maintainability, Compliance, Modifiability, Modularity, Perfectiveness, Portability, Process Delivery, Programming language, Readability, Reusability, Self-descriptiveness, Simplicity, Stability, Standardization, Testability, Traceability and Understandability.}.

- McCall [77] has proposed a software quality model and has defined one or more sub-characteristics for each of the quality characteristics. In his model, *maintainability quality characteristic* has correct ability, testability and expandability as the sub characteristics.

- Chidambaram and Kemerer (CK) [77] is the most referenced research for SPL maintainability. They have defined six metrics viz., Weighted Methods per Class (WMC), Response sets for Class (RFC), Lack of Cohesion in Methods (LCOM), coupling between Object Classes (CBO), Depth of Inheritance Tree of a class (DIT) and Number of Children of a class (NOC). The CK metrics can be used to analyse coupling, cohesion and complexity very well.

5.4.2.1. Four Dimensions of Maintainability

Changes are inevitable for test libraries as they need to constantly realign the test functionalities to the constantly changing application codes as well as the changing needs in business/user requirements. Thus test maintenance is difficult and is an expensive task in the SPL testing. Maintainability of test libraries would involve activities such as error correction in test scripts, enhancements of test

functionalities, deletion of obsolete test functionalities and optimization. Thus test maintenance mechanisms attempt to evaluate the changes and make controlled modifications on test libraries. Test library maintenance can be classified into four dimensions based on nature of activity involved. A brief description of the four dimensions is provided below:

1) Corrective Maintenance: Just as in software, executable test libraries are also defect prone. Identification of such defects, removal of defects and verification of correctness in test libraries is known as corrective maintenance.

2) Adaptive Maintenance: Both business requirements and the application under test constantly change. Therefore it becomes necessary to keep the related test libraries in sync with the underlying test specification and code modifications. This type of maintenance is known as adaptive maintenance.

3) Perfective Maintenance: Test libraries needs to constantly expand their test capabilities. The reason may be improved coverage, changing business priority or new features being added to the product line. Thus test libraries may need to be enhanced with additional test suites and test cases for improved abilities. This type of maintenance is known as perfective maintenance.

4) Preventive Maintenance: Test libraries are constantly improved to cater for future anticipated enhancements. This type of maintenance is known as preventive maintenance.

Although we have classified maintenance activities into four distinct dimensions, they are all in a generic sense concerned with having to make specific changes to the test library. In essence, *Maintainability of Test Library* can be defined as its *ability to adapt to changes as test library versions evolve.*

5.4.2.2. Test Library Maintainability Metrics

A study of literature with regard to SPL software maintainability reveals that researches have proposed numerous maintainability measures as discussed in Chapter 3. While drawing basis from these as inputs, we focus on metrics for SPL testing context. As was the case with reuse metrics, the maintainability metrics (Figure 5-12) in SPL context can also be classified as static and dynamic metrics. Examples for static maintainability metrics are test coverage, testing efforts, change count and modularity based measurements. Examples for dynamic maintainability metrics are testability, cohesion/coupling factors at runtime, expandability and execution coverage. Since the template based reuse approach proposed in this thesis focuses only on test libraries, this section confines to those metrics that have direct impact on test library maintenance activities. Hence, the relevant metrics dealt in this research are change request backlog, backlog management index, number of modifications per change request and change request responsiveness. Each of these metrics is described in detail in the following sub-sections.

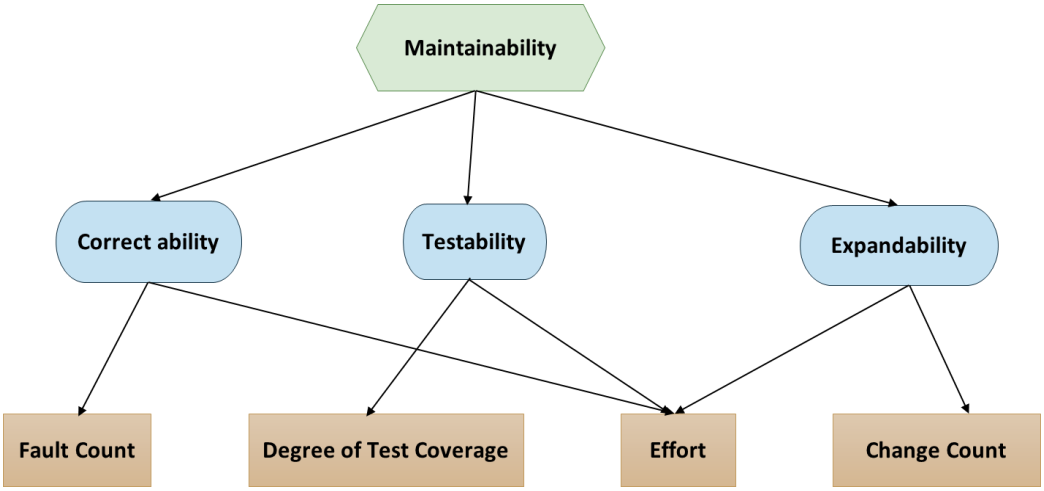


Figure 5-12 Metrics for Maintainability

Change Request Backlog and Backlog Management Index

Change request (CR) backlog is a workload statement for test library maintenance. Change request backlog depends on both the CR arrival and the

time the CR is implemented and is expressed as a simple count of CR over a period of time (week, month or year) depending on the frequency of change request activities in that project. The second metric is backlog management index (BMI) which provides a good measure of open unresolved CRs.

$$BMI = \frac{Number\ of\ CRs\ closed\ during\ the\ month}{Number\ of\ CRs\ arriving\ during\ the\ month}\ X\ 100\%$$

BMI is interpreted to indicate that the backlog is "reduced" if the ratio of completed problems to open problems is larger than 100 and likewise backlog is interpreted as "increased" if BMI is less than 100. With more data points, BMI can help investigate trends in CR arrival, completion and control limits.

Number of modifications to implement a particular Change Request

Modifications required on a test library to implement a change request can be measured. Three metrics can be useful: number of modified location, nature of modification and total number of modifications required per change request (CR). The number of files affected by each CR is denoted as #F and the number of modified locations is denoted as #L. To classify the nature of modifications, for each modification in a given file we use #M to refer to edit changes, #A for newly added changes and #D for deletions (code removals).

$$Modifications\ per\ CR = \sum_{i=1}^{\#F}\sum_{j=1}^{\#L}(\#M + \#A + \#D)$$

where, $\#F - Number\ of\ files\ affected$;

$\#L - Number\ of\ locations\ modified\ in\ each\ file$;

$\#M\ or\ \#A\ or\ \#D\ is\ Edit\ or\ Add\ or\ Delete$

$modifications\ in\ each\ indentified\ location$

Modifications per CR metric indicates the complexity involved in implementing the change. The metric measures the effort needed in terms of lines of code change

and number of locations to perform the planned enhancement or restructuring or fix.

Change Request Responsiveness Metric

It is also common to measure the mean time taken to implement a particular change request. Many organizations have established guidelines for CRs, their complexity, business priority and actual turn-around time. For high priority CRs faster responses alleviates risks, while for lower priority CR response times are not that crucial. CR responsiveness metric is usually calculated as a time measure.

$$CR\ Responsiveness = \ Mean\ time\ taken\ for\ CRs\ of\ all\ severity$$

$$to\ move\ from\ Open\ to\ Closed\ \ state.$$

While the statistical mean is used for normal distributions, median is preferred for skewed distributions. This is because if the change request data points are found to have extreme values and the request arrivals are not frequent, then median is known to yield a meaningful measure and hence used in the computation.

In general shorter time frames to implement CRs yield test productivity increase. While there may be other measures such as the test engineer's skills, technical platform capability and business interests, these are subjective and not easily quantifiable and are also contained as a subset of the Time Metric. Hence this thesis confines to Time measurements as the relevant metric.

5.5. Chapter Conclusions

In this chapter, we formalized definitions for test libraries and related terms. We proceeded to use this definitions to build test clone taxonomy. We provided formal descriptions for test clone granularity in terms of physical and logical syntactical boundaries. We analysed two key quality attributes that relate to test libraries – reusability and maintainability, as these two quality factors were observed to have greater influence on the test library construction approach.

Finally we proposed a set of metrics for reusability and maintainability in terms of test libraries. The theory developed in this chapter would form the foundations of the core research of the thesis which is to develop an approach for template based test clone management in large scale test libraries.

CHAPTER 6
Systemic Template Based Reuse Approach for Large Scale Test Libraries

In the earlier chapters we have laid the necessary foundations and built the relevant theoretical prerequisites and formulations to use as basis for the proposed testing strategy presented in this chapter.

In Chapter 4 we observed that the presence of redundancies increases testing effort, reduces productivity and introduces anomalies. To solve the redundancy problem caused by duplicate code fragments, we propose the use of generic templates. Creation of generic templates requires a systematic approach especially in Software Product Line (SPL) context were ramifications can be large. Hence, to facilitate the creation of effective generic adaptable test templates (GATT) we have formulated an approach which we call as **S**ystemic **T**emplate based **R**euse **A**pproach for Large Scale **T**est Libraries (STRAT). Testers can use the proposed STRAT approach as a framework to methodically create GATT to run it on top of the ART processor. This chapter elaborates the variability management and template creation techniques involved in the proposed STRAT approach.

The organisation of the chapter is as below:

- Section 1 provides an overview of the proposed template based testing approach.
- Section 2 explains the reasons behind the need for generic design solutions.
- Section 3 describes the proposed solution with the support of examples.

- Section 4 explores how the proposed solution helps software product lines testing.
- Section 5 summarises the benefits of template-based approach.
- Section 6 identifies some of the limitations of the template based testing approach.
- Section 7 provides the conclusions derived from this chapter.

6.1. STRAT Overview

When traditional approaches fail to construct non-redundant, large scale test case repositories, it is worthwhile to look for new research solutions that address the shortcomings. This chapter describes the proposed Systemic Template based Reuse Approach for Large Scale Test Libraries (also synonymously referred in short as "template-based approach" or "STRAT approach" or just "STRAT" in this thesis). The STRAT approach applies *variant parameterization* and *composition based adaptation* techniques to manage variability. The approach employs the proposed *Generic Adaptive Test Template* artefacts to resolve issues arising out of the presence of redundant test clone groups; these issues were analysed and presented in the previous chapter. The generic adaptive test templates are fabricated using a meta-programming technique called Adaptive Reuse Technique (ART). STRAT incorporates techniques such as parameterization, composition and adaptation of the test code fragment files that are deemed non-unifiable using traditional testing techniques.

In this chapter, we pursue three objectives. First, we provide a sketch of the proposed STRAT approach. Subsequently, we illustrate template unifications for various test clone types. Finally, we discuss STRAT related process, template lifecycle activities and the possible benefits of using STRAT approach to manage variability in a product line context.

6.1.1. Motivational Example

The Android Note app is used as the motivational example for illustrating the shortcomings of existing testing techniques and to showcase how STRAT approach overcomes these shortcomings. This Note app was chosen due to its ease of understanding and its suitability to demonstrate the research motivation.

The Note app can be used to create a new text note, add a title or save/edit/delete an existing note. Test cases for this app would first set up required data and context for the events and then simulate test situation by applying appropriate activity commands on the instrumentation instance. The test cases would, finally compare received results against expected values and report the results. The existing test libraries for this app use test file archetype comprising of resource

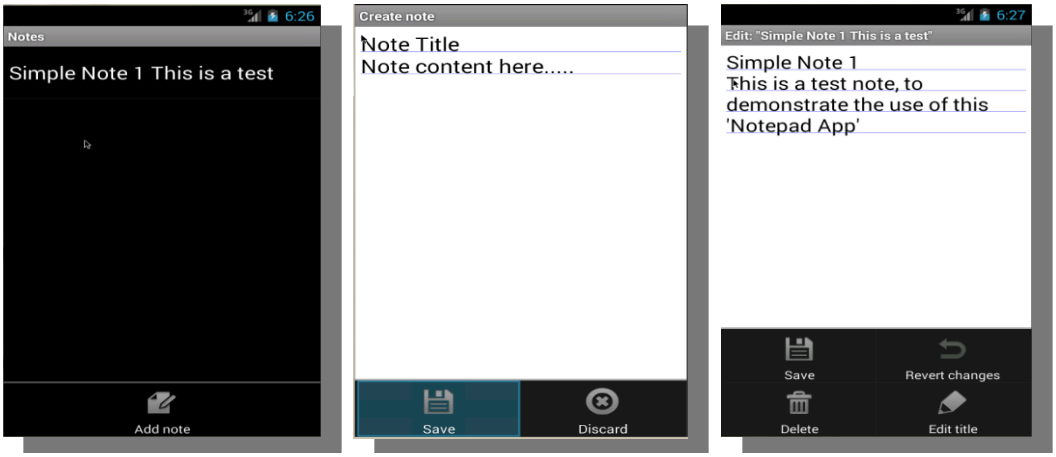


Figure 6-1 Note App (Listing, Create and Edit Screens)

setup, test cases and teardown. Examples for resource setup include common data, mock object instantiations, test fixtures and instruments from the test framework. A typical test case consists three elements namely, pre-conditions, the actual test stimulus and the assertion of expected results. To provide a clear understanding of the Test Case Scenario we have created a representative Test Library Project using the Eclipse IDE and Android development toolkit. Execution of the Note app Test Project would result in testing the three screens viz., listing, create and edit screens as shown in Figure 6-1. The test cases for above mentioned screens

employ a black box testing approach. The test cases verify the correctness of the Note app features such as creation, editing and deletion.

The Test project consists of several files and directories which are explained below. The Note app test project workspace (structure in Figure 6-2) comprises of configuration and code files. The role of a configuration file (eg: `AndroidMaifest.xml`) is to facilitate the setting up of the desired device environment and context settings (e.g., the minimum version number for platform/OS/API against which the app can be tested). The role of the Android project's code file (eg: `EditorTest.java`), is to define test cases targeting the functionality of the various Note app features. The `res` folder, contains presentation layer codes and UI related configuration values. A layout configuration defines the visual structure for a user interface such as UI for an activity or widget for an app. Examples for layout configuration can be radio button color, text box length, form size, submit button's display icon etc. The project also references the Robotium framework library.

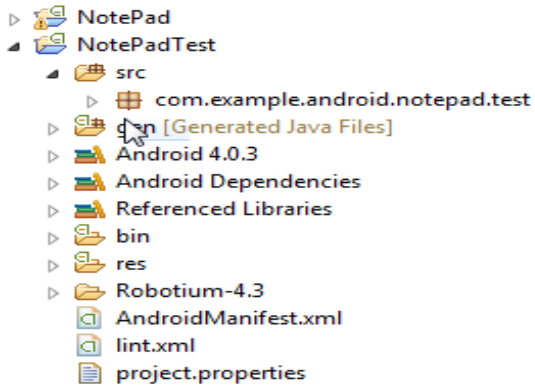


Figure 6-2 Note app test project structure

A typical Android test case constitutes of pre-conditions, the actual test stimulus and the assertion of expected results (as illustrated in example test file `EditorTest.java` in Figure 6-3). The `setup()` and `teardown()` methods are the resource management codes. The

`testPreferenceIsSaved()` method encapsulates codes for implementing a test case (preconditions, stimulus and result assertion).

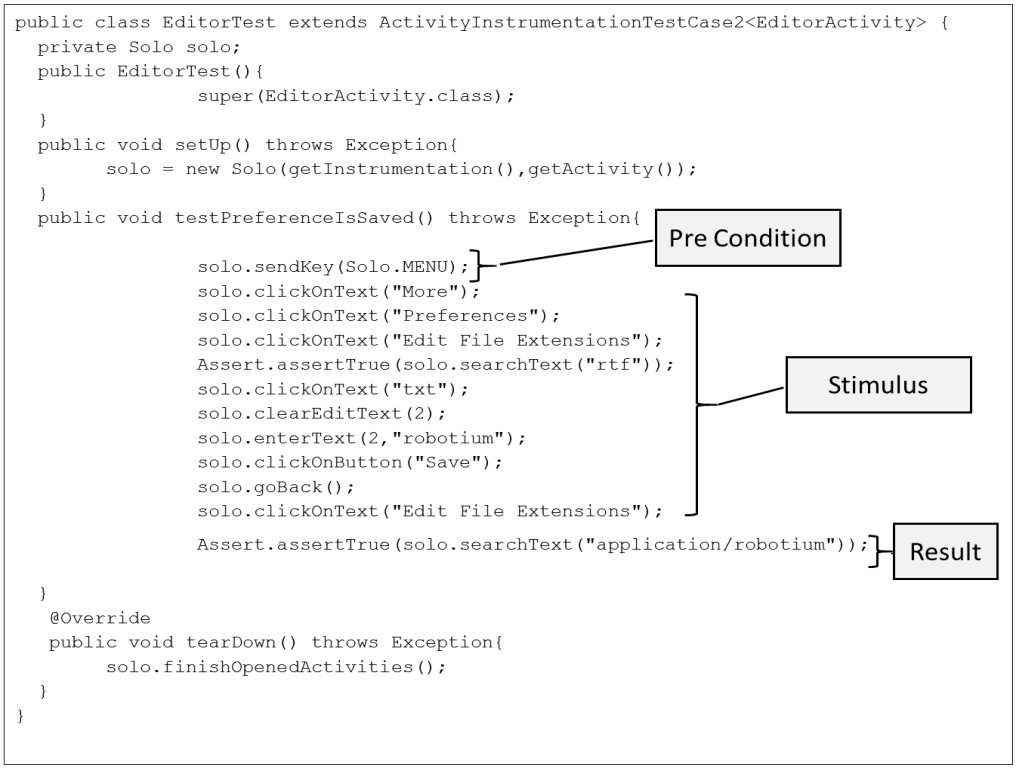


Figure 6-3 Test Archetype and Test Case Structure Example

A common observation is that test cases that test a particular feature tend to be similar even among different OS platforms. Our analysis shows that it is not uncommon to find a fair amount of similarity between certain test cases for different features. To understand the nature of such test clones and visualize how to unify them, consider two test methods shown in Figure 6-4. The Test method `testAddNote()` is provided in the left column while `testEditMode()is` shown on the right column in the Figure 6-4. As the name suggests these methods test the Add and Edit Note features.

```
public void testAddNote() throws Exception {
solo.clickOnMenuItem("Add note");
solo.assertCurrentActivity
 ("Expected NoteEditor activity",
  "NoteEditor");
solo.enterText(0, "Note 1");
solo.goBack();
solo.clickOnMenuItem("Add note");
solo.typeText(0, "Note 2");
solo.goBack();
solo.takeScreenshot();
boolean expected = true;
boolean actual = solo.searchText("Note 1")
          &&solo.searchText("Note 2");
assertEquals("Note 1 and/or Note 2"+
      "are not found", expected, actual);
}
```

```
public void testEditNote() throws Exception {
solo.clickInList(2);
solo.hideSoftKeyboard();


solo.setActivityOrientation(Solo.LANDSCAPE);

solo.clickOnMenuItem("Edit title");
solo.typeText(0, "test");
solo.goBack();
solo.takeScreenshot();
boolean expected = true;
boolean actual =solo. waitForText("(?i).
    ("note 1 test");
assertEquals("Note 1 test is not found",
     expected, actual);
}
```

Normal Text: identical; Underlined Text : Simple Parametric Clones

Bold Text: Complex Non-Parametric Test Clones;

Figure 6-4 Sample Test Clone Testing Different Features

Codes shown in regular font in depict exact duplication, while variations are either underlined or in bold font. Whereas the underlined texts indicate parametric variations that can be handled using traditional programming constructs, the bold texts indicate complex variations such as different API/method calls, partial names and other gapped test clones whose handling may fall beyond the purview of traditional programming constructs. In Software Product Line Testing (SPLT) context, test clones will directly benefit from core as well as app level test libraries reuse. Domain engineering generality principle encourages avoiding repetitions and construction of parameterized, configurable and adaptable generic test libraries. The need for an approach incorporating such generality serves as the motivation behind the proposed solution.

6.2. Need for Generic Design

In the previous chapter where we analysed the Android platform test libraries we had observed the following:

- About 30% to 80% of test codes contain similar program structures repeated within and across test libraries.

- The repeating structures range from fine grained instances such as few lines of test codes (test clones), to large granularity, such as test files/directory level patterns of test libraries (termed as structural clones previously).
- Test clones induce extra conceptual complexity and are counter-productive for maintenance.
- Scope for avoiding test clones through traditional approaches was limited or would force test developers to compromise on other important test design goals.

The above motivates the need for evolving a research based solution that would address the stated issues.

6.3. Proposed Solution

Our proposed approach uses templates to represent test libraries in a non-redundant way. Templates provide meta-level decomposition and test clone unification support mechanisms. Creation of such templates requires a formal approach consisting of the following elements:

- A mechanism for Test Clone Identification and Classification.
- A definition for Generic Adaptive Test Template Structure.
- *Template Artefact Designs* for defining various types of test clones.
- A mechanism for *Test Clone Unification* based on the underlying test clone type and generic design requirements.
- A methodology for the *Test Library Construction Process* and related Template Lifecycle Management activities.

Keeping the above in view, our solution proposes a formal identification mechanism for test clone detection and classification based on various factors such as structural granularity, feature under test, variant point representation and nature of test artefact. The template structures proposed in our solution provides for representational features such as heterogeneity (independent of underlying scripting language), scalability, extensibility and variability. To address the need for test clone unification, our solution proposes design guidelines and test clone unification schemes for different types of test library variations with adequate practical code samples. Finally to ensure that the proposed template solution can be systematically adopted, our solution outlines a methodology for template lifecycle management activities and test library refactoring.

6.3.1. Solution Design

STRAT provisions for unrestricted parameterization by implementing templates as a non-intrusive complimentary layer that captures key information such as the differences among test clone structures, change request modification design decisions, variability choices and respective binding options. The STRAT approach provides two layers; one layer for the executable test cases and the second is a higher layer for templates (GATT). For clarity of distinction, we call the meta-code level as GATT (or simply templates) and the original test cases as test libraries. STRAT utilizes a flexible, configurable and versatile variability management programming construct - ART (Adaptive Reuse Technique, art.comp.nus.edu.sg) developed in our lab. Being a compile time test construction method, the appropriate variant binding for each variant is generated based on the test designer's configuration choices. The use of templates permit more exhaustive and intrinsically detailed variability design and management. The test templates are made of two types of files: SPC (Specification Configuration) and ART (Adaptive Reuse Technique) files. The SPC files are usually in the top level acting as handles for rest of the ART template files. Both files have elements that

define variant points, possible binding choices and provide constructs for selection, conditions, iterations and processing using ART directives.

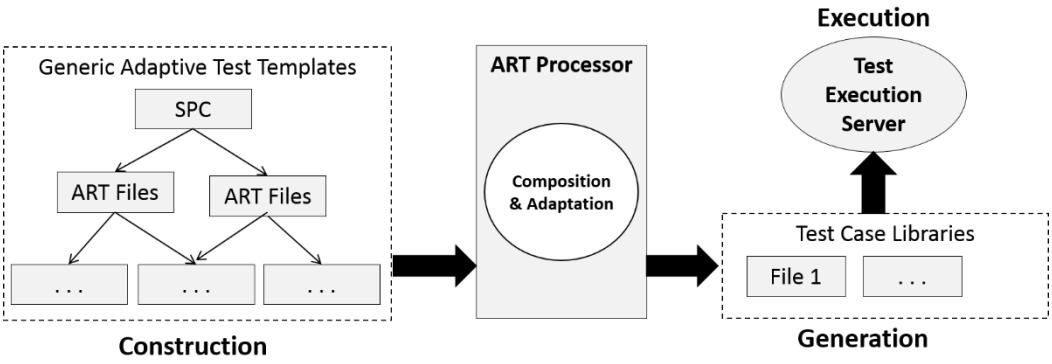


Figure 6-5 Generation of Original test libraries from GATT

Figure 6-5 illustrates the steps in deriving test libraries from templates using ART Processor. GATT design is a hierarchical arrangement of template files (*.art and *.spc file extensions). Using the ART processor, specific test cases or the complete test library can be generated from templates and subsequently the system or app can be tested in a normal way. Since the test cases created using GATT structures are pre-compiled, the proposed template based solution does not interfere with the regular test execution procedures.

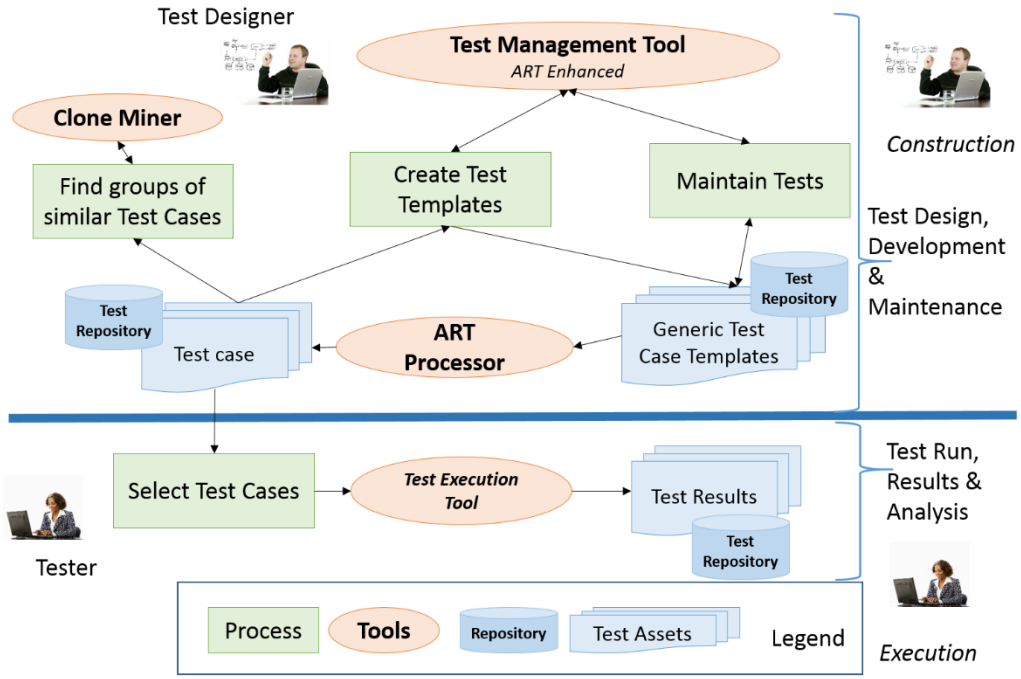


Figure 6-6 Systemic Template Approach for Large Scale Test Libraries

Figure 6-6 illustrates how STRAT approach works in two layers, namely *test construction* (planning and design of test cases) followed by *test execution*. STRAT approach involves the participation of two key roles; they are: *test designer* who is a domain expert contributing towards test case design and development; *tester* who is a test automation expert responsible for executing selected set of tests on a targeted system. Test designer identifies test clones that are candidates for building templates. Test designer selects group of test clones, isolates the common portions and variations, create templates applying the identified variability constructs. These templates along with test designer's variability binding choices are inputs that would eventually generate the original executable test library. Test designer also preserves the domain related test design decisions inside the templates.

As test cases are maintained at templates level, testers work with the representation that is smaller in size than the actual test library and may hence find it easier to understand due to non-redundant structure. Any change made to template parameters can be selectively propagated without the need to deal with each individual version thereby providing an inherent protection against unintended changes. The specification and template files captures test design decisions, variant options and other selective evolution based information. This will allow test designers to browse existing test templates at any point in time, grasp the overall similarity situation, and narrow selectively to specific templates that facilitate test designers to study the exact nature of similarity and differences in a given group of test cases.

As long as regular programming language constructs are able to unify the test clones, it remains a traditional solution. But when the complexity of variations cannot be handled, the test designer may resort to the use of STRAT approach. As the name suggests, generic adaptive test templates complement than compete with traditional techniques. While the STRAT approach proposes that the

maintenance of test libraries will be primarily handled through test templates, the test libraries are the ones that get compiled and executed. Thus at runtime, there is no difference between traditional and STRAT generated test libraries.

6.3.2. Scope of Proposed Solution

In this section we propose to identify where the proposed solution would fit within the entire testing process space. Generally software testing process (shown in Figure 6-7) comprises of four stages namely, Test Planning, Test Preparation, Test Execution and Result analysis.

Test Planning involves planning for forthcoming testing activities such as identification of tasks, resource estimation, financial and effort budgets. *Test preparation* involves test case design, implementation, selection and documentation. It also prepares necessary test fixtures and data needed for test execution later. *Test Execution* could be manual or automated and each execution results in test outcomes. *Result Analysis* uses the test outcomes and test coverage achieved to suggest whether or not to perform further testing. If testing is to be continued, it could recommence from any of the previous stages.

The STRAT approach fits into the *Test Preparation* stage and aligns closely with the test library repository. Test preparation involves test data preparation, test case design, implementation selection and documentation. STRAT works with all artefacts belonging to test libraries and recommends potential candidates for templates through identification of test clone groups and guides methodically in re-constructing the identified subjects into generic adaptive test templates.

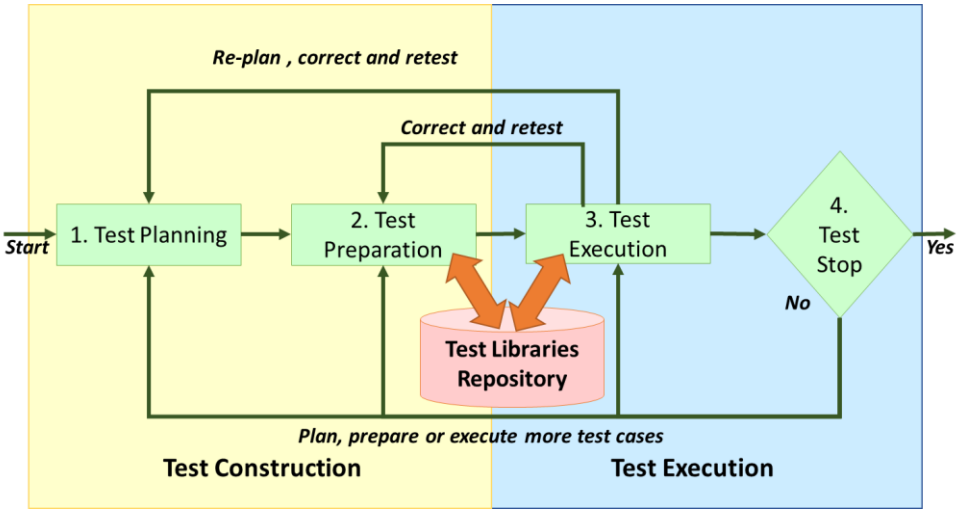


Figure 6-7 Software Testing Process

6.3.3. Generic Adaptive Test Template Derivation

Templates derived adopting generic design mechanisms allow test designers to represent and maintain test clone structures as compact customizable test code fragments. In this section we provide a conceptual overview of the derivation of the proposed GATT followed by an actual example to illustrate the concept. Consider a test library that consists of two similar test case codes T1 and T2. The conceptual diagram (Figure 6-8) illustrates how these two similar test case codes are managed. Our approach proposes that the generic form of test case code be represented as template T. Additional codes specific to the above two instances T1 and T2 to be represented in ΔT1 and ΔT2 respectively. Hence running the templates via ART processor generates exact test cases (T1 and T2) as before. Thus GATT's variability management constructs promote reuse with only modest extensions to existing test libraries.

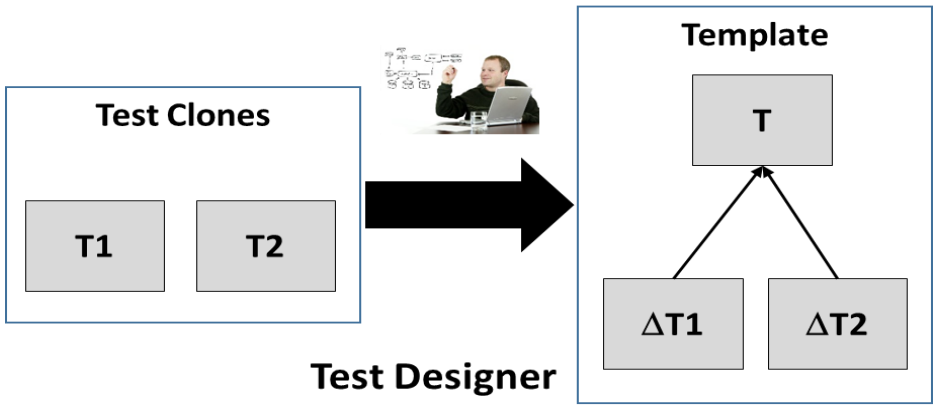


Figure 6-8 GATT for Test Clones

To demonstrate this, we derived the templates using the proposed GATT for the motivational example discussed in Section 6.1.1. The templates are shown in Figure 6-9 where the two test clone files `TestAddNote.java` represents T1 `TestEditNote.java` represents T2. The GATT template files `TestNote.art` is the main template file (T) and additional template fragments are in `TestNoteAdd.art` (ΔT1) and `TestNoteEdit.art` (ΔT2). `TestNote.SPC` is the main specification file.

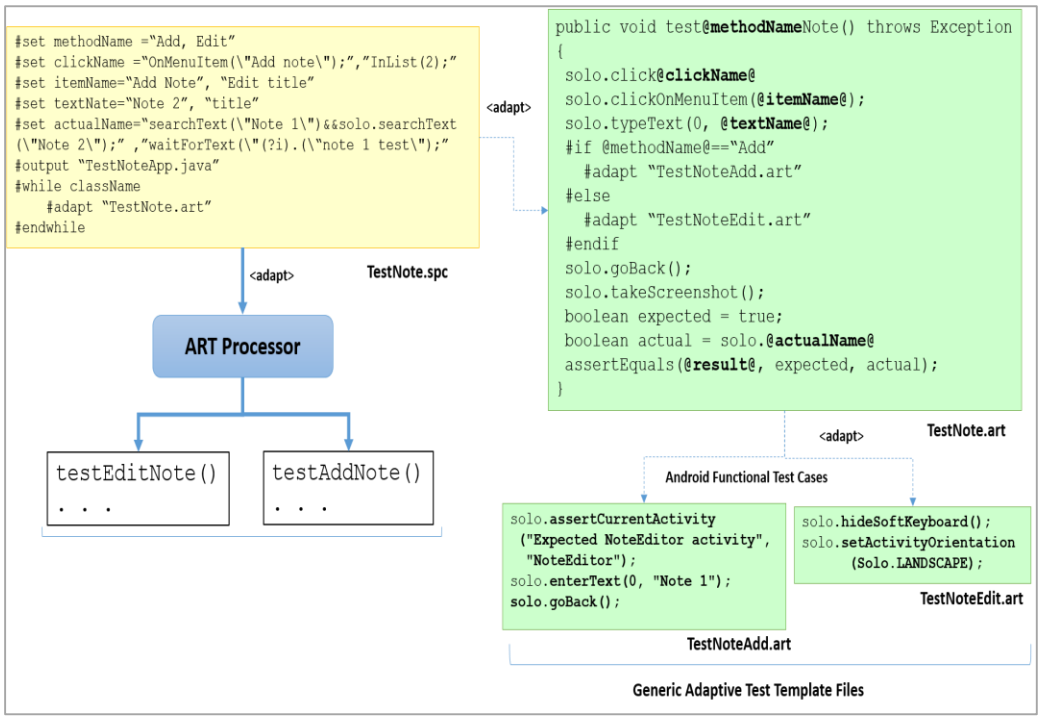


Figure 6-9 Templates for Motivational Example

The above structure is typical in most test clone situations. However, for brevity and ease of understanding, we first present a much simpler example (which confines to a single template without delta variations) to explain the full details of GATT constructs. Consider two similar test cases depicting non-parametric variation namely, `MainActivity.java` and `MainActivity2.java` (Figure 6-10). Both test cases attempt to search and invoke a menu item, and if the searched menu item is found it adds more items to an attached action bar. As part of test specifications, tests need to be performed for all orientations of the screen and hence multiple test cases are created with screen layout variations resulting in two test clones as seen in Figure 6-10. Comparing the two test clones we have identified code level variations and have shown as bold text.

```
package com.example.test;
import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.view.Menu;
import android.view.View;

public class MainActivity extends Activity {

@Override
protected void onCreate(Bundle savedInstanceState) {
 super.onCreate(savedInstanceState);
 setContentView(R.layout.activity_main);
}
public void activity1(View view){
      Intent intent = new Intent(this,com.example.
                              test.MainActivity.class);
      startActivity(intent);
   }
   …
}
```

```
package com.example.test;
import android.app.Activity;
import android.content.Intent;
import android.os.Bundle;
import android.view.Menu;
import android.view.View;

public class MainActivity2 extends Activity {

   @Override
   protected void onCreate(Bundle savedInstanceState) {
   super.onCreate(savedInstanceState);
   setContentView(R.layout.activity_main);
   }
   public void activity2(View view){
      Intent intent = new Intent(this,com.example.
                              test.MainActivity2.class);
      startActivity(intent);
   }
   …
}
```

Figure 6-10 Sample Android Activity based Test Clone Pair

Variations as above can be well managed using template-based approach since the screen layout orientation is set using test configuration files hence reducing duplicate code fragments. Rewriting the above test clone pair using GATT, we have two template files namely, `Activity.art` and `Activity.SPC` (refer Figure 6-11). The ART template file also provisions variant points that can bind to different values when generated into actual test cases later. In this example the two variant points defined are `@className` and `@methodName`. The `SPC` file

provides relevant binding choices for each of the variant points defined earlier in the `ART template` file. This can be seen through the #set directives in the SPC files that are configured for each test case combination. Finally to generate the two individual test cases, the SPC file uses an iteration structure (`#while` construct).

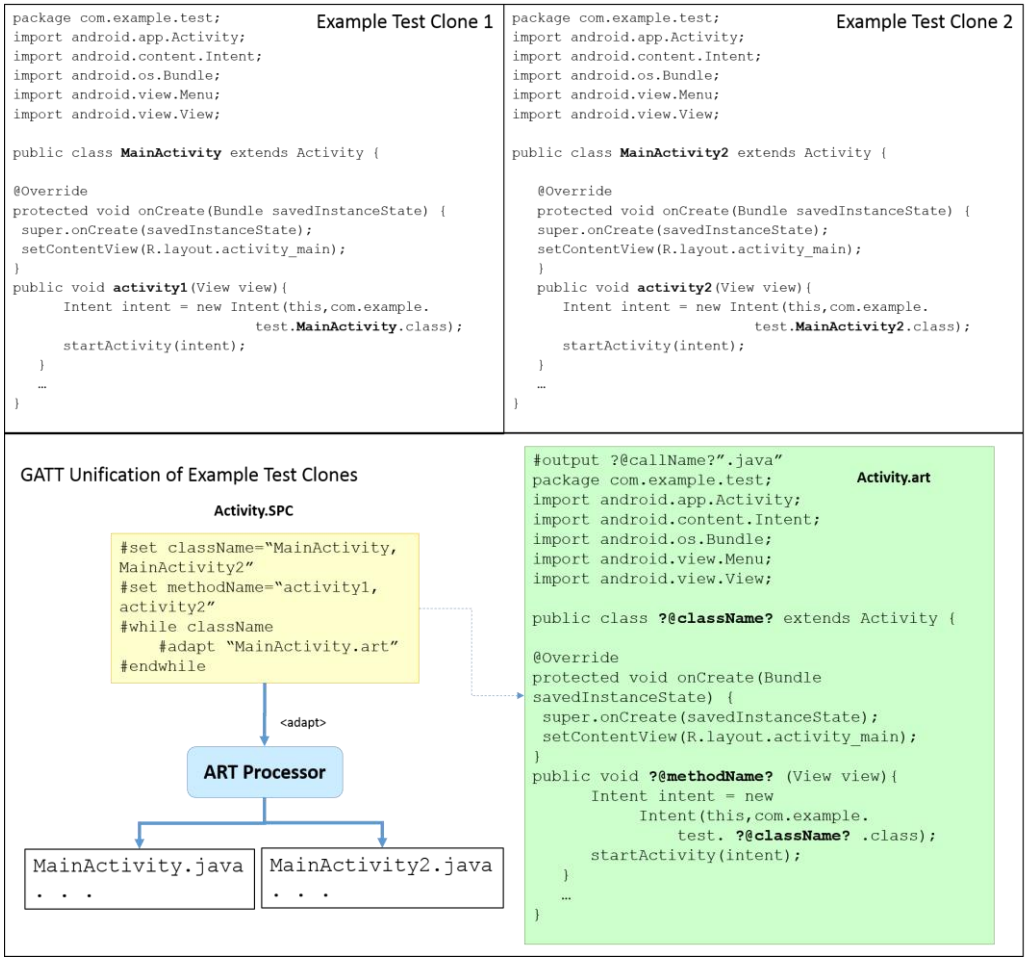


Figure 6-11 GATT Representation of Test Case Example

Thus, based on tester's choices the specification file (SPC) can be evolved to manage multiple orientations. This example demonstrates the ease with which other variations such as handling multiple platform settings and release versions can be managed as the product line evolves over time.

6.3.4. Adaptive Reuse Technique

To support the understanding of the template unification examples provided subsequently, this section briefly introduces ART structure along with the relevant ART functionalities and their syntax. ART commands are embedded in the base code resulting in a composite code file. This composite code must be executed by a ART Processor to render the original base code. By setting ART parameters, testers can derive many variants of test libraries from the base code. ART provides controls for various levels of decomposition and composition. Some frequently used ART commands are examined here (a complete set of command syntax chart is provided in Appendix B):

- Global ART parameters are set in SPC (**SP**e**C**ification) file. All other source files comprising the base code are linked to SPC by chains of **#adapt** commands. So to run the ART Processor it is enough to provide the specification files.

- ART Processor executes template commands embedded in SPC, leaving existing base test library codes untouched. Command **`#adapt f`** directs the ART Processor to the adapt file **f** in the location where the command is being called. Once file **f** has been processed, ART Processor resumes processing. Processing of ART's **`#adapt`** command is similar to cpp's **`#include`** command. The processed content of the adapted file will be placed where the **`#adapt`** command was stated. We could say adapting a file is almost the same as having its content in the place of the **`#adapt`** command. The difference is the ability to make configurations in the extended adapt body and in the scoping rules.

- ART Processor interprets template commands and generates test codes corresponding to the template commands located in the visited files. ART **`#output`** <path> command specifies the output file where the test

code should be placed. The <path> can be absolute or relative path. If output file is not specified, then ART Processor emits tests to an automatically generated default file named **defaultOutput** in the main installation.

- **#set** command declares a template variable and sets its value. **#set** command is similar to cpp's **#define** except that template variable values propagate across the files along **#adapt** links. With the **#set** command, we can declare single and multi-value variables. Expressions are written between question mark **'?'** characters. A direct reference to variable **x** is written as ?**@x?**. Each extra **'@'** symbol in front of a variable name indicates an extra level of indirection. This means we can refer to the **(value of (value of x))** with **?@@x?**, and to the **(value of (@@x))** with **?@@@x?**.

- An **#insert** command replaces all matching **#breaks** with its content. Matching is done by a name (**break X** for example).

- Command **#while** is a loop that iterates over its body and generates custom test code at each iteration. Command **#select** allows us to choose one of many customization options. **#while** and **#select** are often used together.

6.3.5. GATT Derivations for Unification of Various Test Clone Types

ART can be employed in all test clone type that we had identified, classified and described in the previous chapter. The examples given below show how the GATT structures unify various test clones samples discussed previously.

6.3.5.1. Unification for Exact Test Clones

Consider the Exact Test Clone example discussed in previous chapter. That sample of exact test clone could be unified into a template file and specification as shown in Figure 6-12. All the exact code clone portions are captured in the file `TestBase.art`, while the `TestBase.SPC` has an iterative structure to generate all clone instances and needful variations for package name differences.

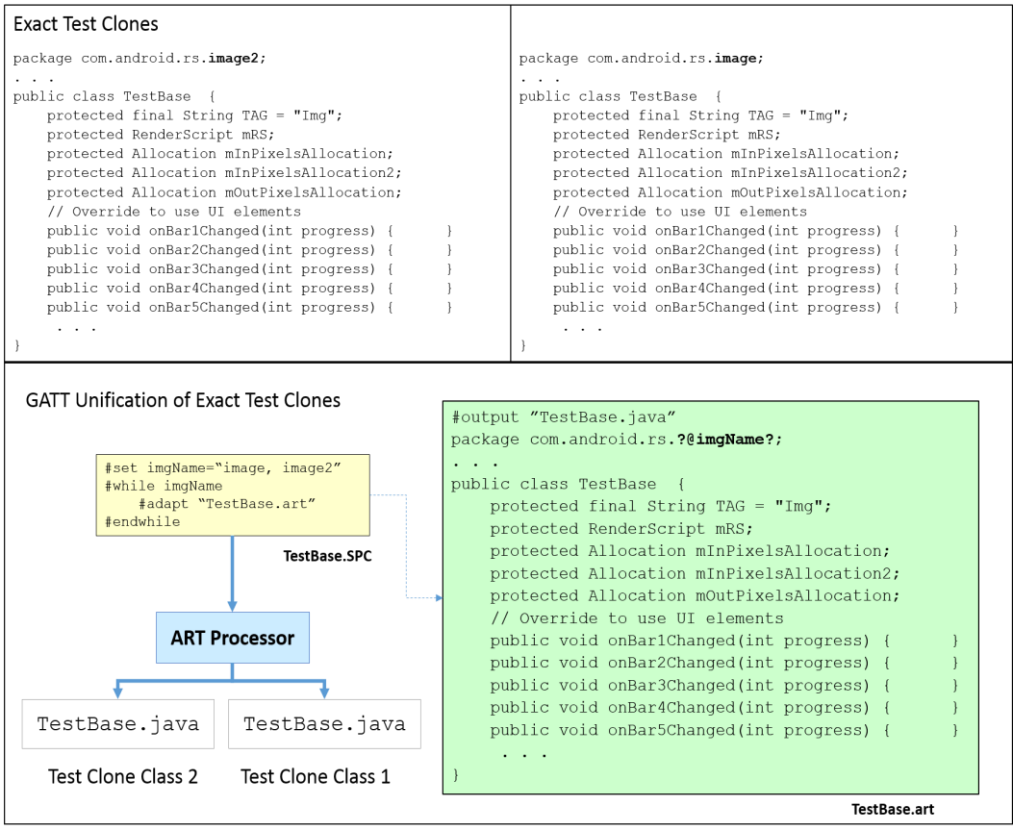


Figure 6-12 GATT Unification of Exact Test Clones

6.3.5.2. Unification of Renamed and Parameterized Test Clones

GATT deals with systematic variants for identifiers, literals and parameters. This mechanism will assist in unifying both renamed and parameterized test clones. While unifying renamed and parameterized clones, variant points and relevant granularity are the design choices involved. Figure 6-13 shows the GATT derivation for renamed and Parametrised test clones. In this test clone example, TC1 uses parameter "a" while the same parameter is named "val1" in TC2 and "val2" in TC3. The GATT derivation for this is also shown (Figure 6-13).

Renaming/parameterization is handled with appropriate settings in templates and specification files. As can be seen from the GATT, the renamed variables are configured in the ART (eg: the variables a, val1 and val2 of TC1, TC2 & TC3 respectively is named as @pos1). A similar approach is used for parameterised variables between test clones. Since an identical configuration approach is used for both parameterisation and renaming, an #if … #endif construct is used in the ART to distinguish between the two. To generate test case instances the SPC declares renamed variables as variation points that have been marked using the #set command (eg: `#set pos1Name="a, val1, val2"`). As before, to generate the three individual test cases, the SPC file uses an iteration structure (`#while` construct).

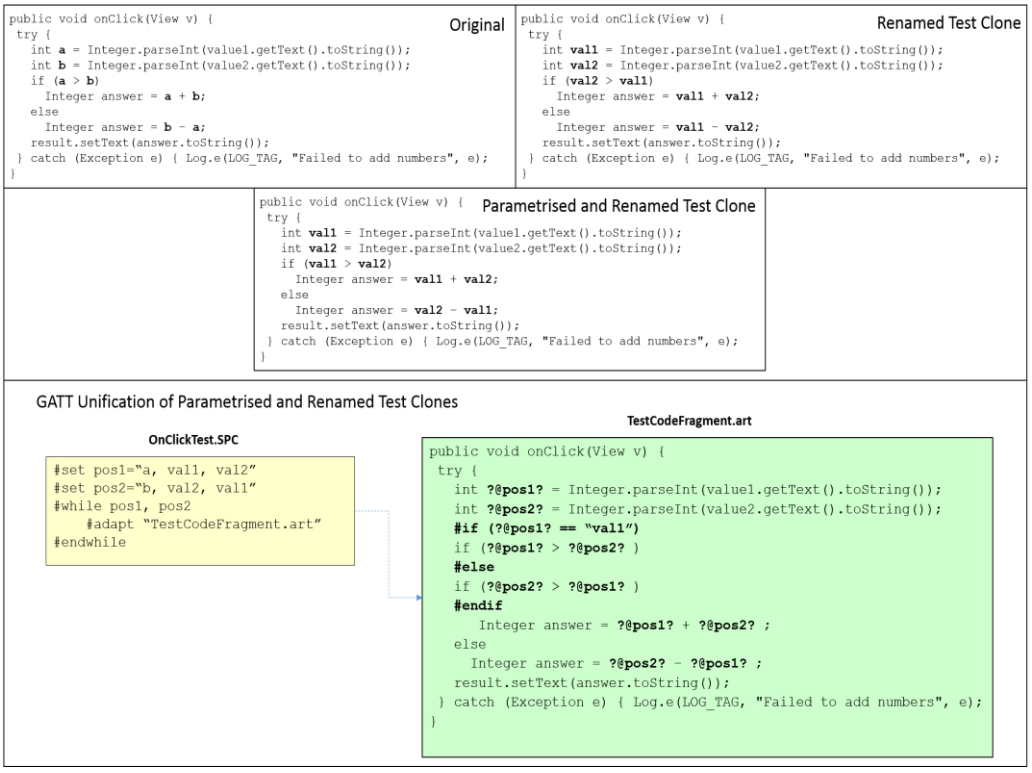


Figure 6-13 Unification of Renamed and Parametrised Test Clones

The above unification scheme will generate parametric test clones that cater for test fixture variations (@pos1, @pos2) as well as test code variations (test code fragment) as shown in Figure 6-14.

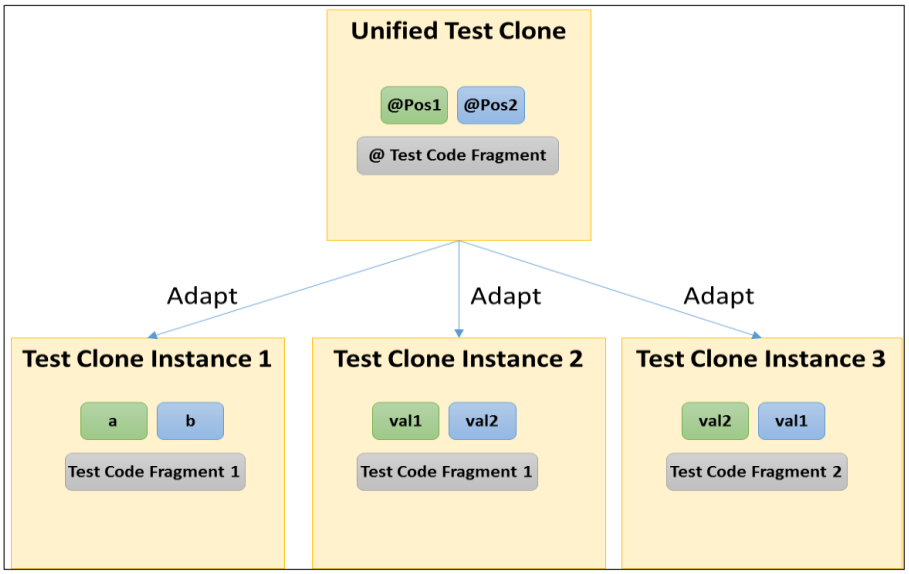


Figure 6-14 Adaptation of Parametric and Renamed Test Clones

6.3.5.3. Unification of Near Miss Test Clones

Near-miss test clone unification involves parameterizations of identifier, literals and variable variations using appropriate variant points. Consider the near-miss test clone with class name and parametric variations occurring at intervening locations in a single file. These variations get unified with the help of variants referring to both parametric as well as class syntactical name changes. The GATT derivation for near miss test clone example is given (Figure 6-15).

Figure 6-15 Unification of Near Miss Test Clones

This unification scheme will generate parametric test clones that cater for parametric variations (@var1, @var2) as well as code variations (test file class name fragments) as illustrated in Figure 6-16.

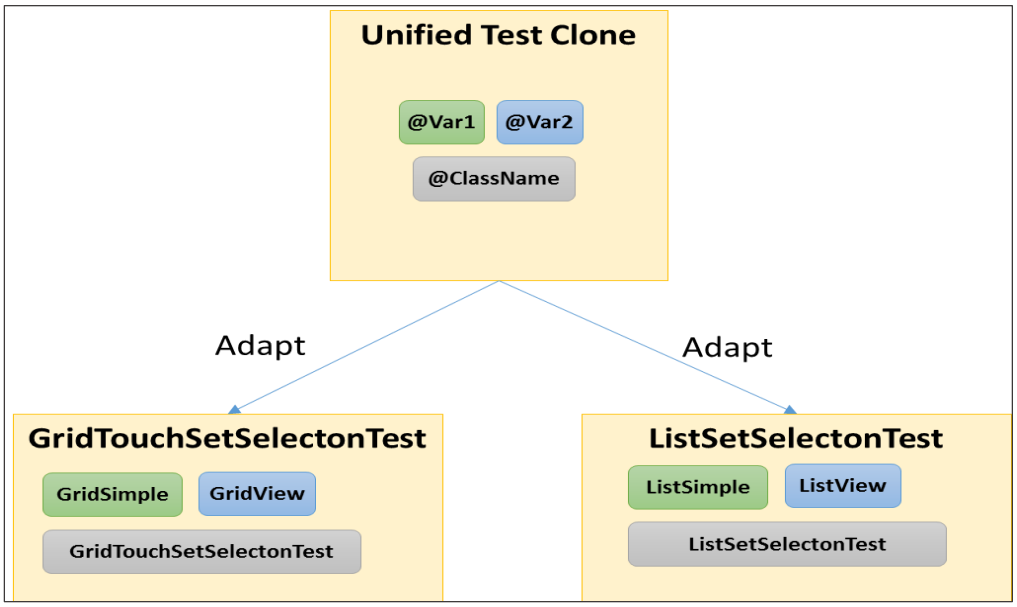


Figure 6-16 Adaptation of Near Miss Test Clones

6.3.5.4. Unification of Gapped Test Clones and Non-Contiguous Test Clones

The GATT for unifying Gapped Test Clones is shown (Figure 6-17). The file `tag.art` captures common portions of the test code fragment. The specification file sets a variant point choice with four values as original, insert, modify and delete. These tags cater for variations based on insertion, deletion or modification of original test code fragment. The file and specification uses iteration and conditional constructs to achieve the unification of gapped clones. Similar techniques are extended for the non-contiguous test clones comprising of more gapped test code fragment portions in the test clone classes.

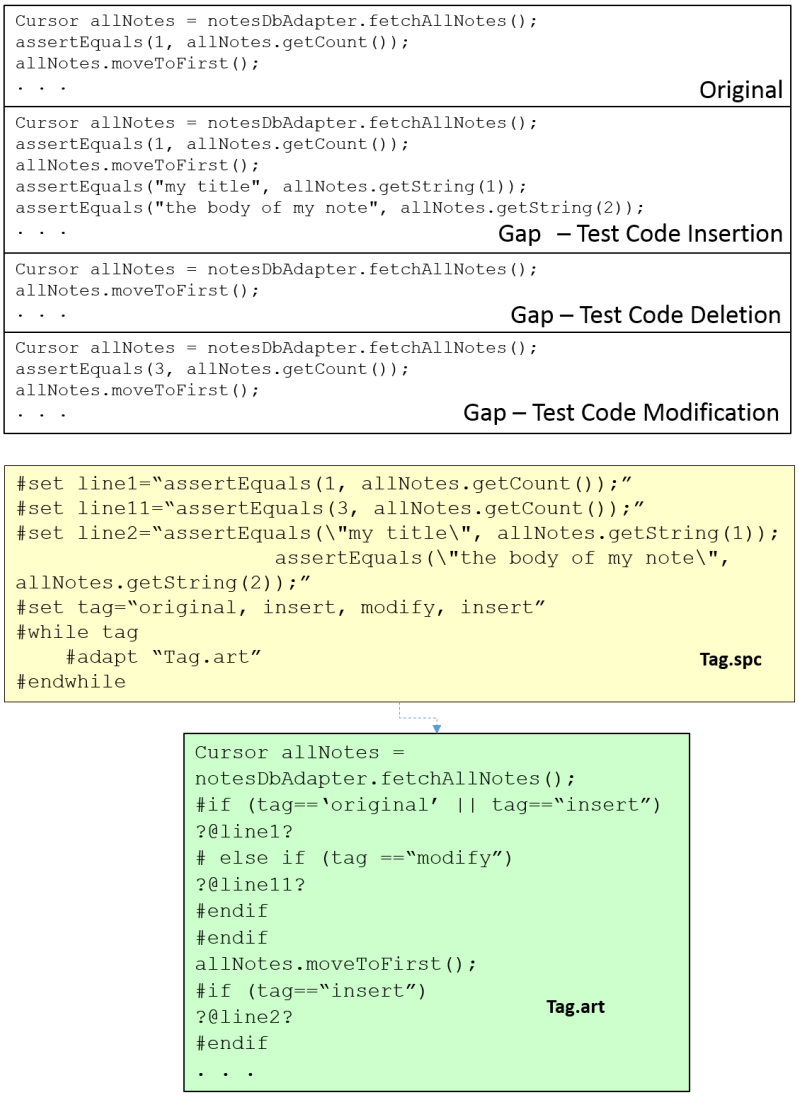


Figure 6-17 Unification of Gapped Clones

The original clone instance consists of three segments depicted as A, B, and C in Figure 6-18. Insert test clone instance adds segment D to the file end. Deletion test clone instance removes segment B. Modified test clone instance changes B to B`.

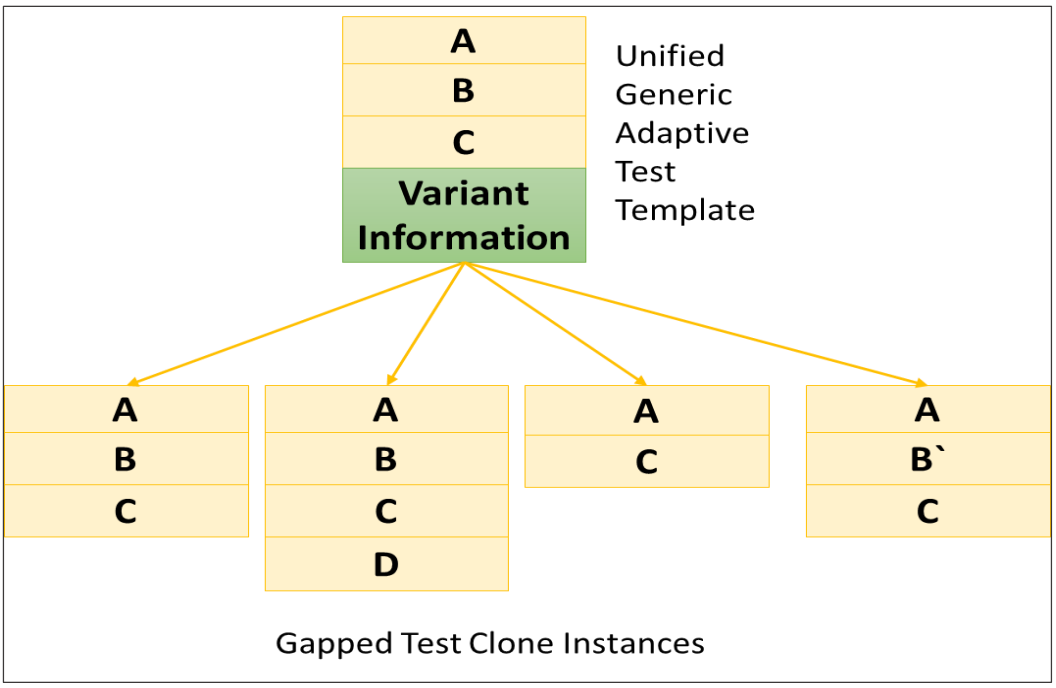


Figure 6-18 Gapped Test Clone Instances

6.3.5.5. Unification of Structural Test Clones

As explained in previous section, structural test clone (SC) class can be characterized as a compound structure that represents the various test clone fragments which can be dispersed across syntactical boundaries such as test methods, test files and test directories. Unified representation of structural test clones and their appropriate instantiation mechanisms are represented by ART files and SPC files respectively. Unlike simple test clones, the instantiation of structural test clones can be dispersed across many ART files and SPC files.

Figure 6-19 shows the partial hierarchy of the coarse-grained structural clone (named SC1) and three instances of it (named Instance 1, Instance 2, and Instance 3). The SC boundaries are marked with double line boxes. SC1 in turn consists of three lower level entities, two of which are further structural test clones themselves (SC2 and SC3) and the other, a simple test clone (A). SC2 consists of two items which are also

structural clones (SC4 and SC5). These two structural clones consist of simple test clones (B, C) and (D or E) respectively. Likewise the SC3 also has layers of low level test clones as can be seen from the figure. Simple clones seen in the SC hierarchy are A, B, C, D, E, F, G, H, I, X, Y, Z and related variant changes specified in the individual ART files.

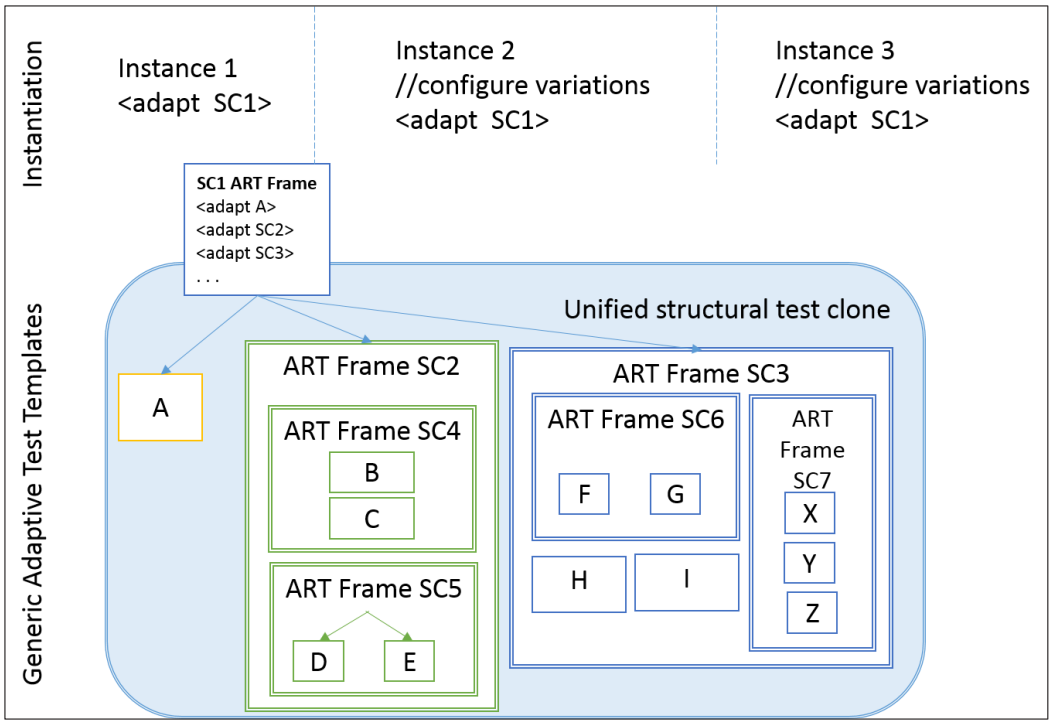


Figure 6-19 Unifying Structural Test Clones

This emphasizes the fact that more powerful structures can be generated using the proposed GATT since it provides a comprehensive syntax collection suitable for all types of variability adaptation situations.

6.3.5.6. Unification of Heterogeneous Test Clones

Heterogeneous test clones comprise of test clones with multiple file formats and programming languages. GATT constructs can unify test clones in any language/platform through heterogeneous text-based compile time test artefacts. Consider a heterogonous test clone example as shown in Figure 6-20. The four files shown (`BiDiTestGalleryLtr.java`, `BiDiTestGalleryRtl.java`, `gallery_ltr.xml` and `gallery_rtl.xml`) are exact test clone fragments with portions of non-

parametric variation in class names, layout API calls and layout orientations. These variations between clones are indicated as bold text in the figure.

Original Test files (Clones) to test Right-to-Left and Left-to-Right Bi-Directional of Gallery Layout

Java Test Clone 1

```
package com.android.bidi;

import android.app.Fragment;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.Gallery;

public class BiDiTestGalleryLtr extends Fragment {
 @Override
 public View onCreateView(LayoutInflater inflater,
   ViewGroup container, Bundle savedInstanceState) {
   View v = inflater.inflate(R.layout.gallery_ltr,
                    container, false);
   Gallery g =(Gallery)v.findViewById(R.id.galleryview);
   g.setAdapter(new BiDiTestGalleryImages
              (this.getActivity().getBaseContext()));
        return v;
    }
}
```

Java Test Clone 2

```
package com.android.bidi;

import android.app.Fragment;
import android.os.Bundle;
import android.view.LayoutInflater;
import android.view.View;
import android.view.ViewGroup;
import android.widget.Gallery;

public class BiDiTestGalleryRtl extends Fragment {
 @Override
 public View onCreateView(LayoutInflater inflater,
   ViewGroup container, Bundle savedInstanceState) {
   View v = inflater.inflate(R.layout.gallery_rtl,
                    container, false);
   Gallery g =(Gallery)v.findViewById(R.id.galleryview);
   g.setAdapter(new BiDiTestGalleryImages
              (this.getActivity().getBaseContext()));
        return v;
    }
}
```

XML Test Clone 1

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout
xmlns:android="http://schemas.android.com/apk/res/android"
        android:id="@+id/gallery_ltr"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:layoutDirection="ltr">

    <Gallery
        android:id="@+id/galleryview"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:spacing="10dip"
        />

</FrameLayout>
```

XML Test Clone 2

```
<?xml version="1.0" encoding="utf-8"?>
<FrameLayout
xmlns:android="http://schemas.android.com/apk/res/android"
        android:id="@+id/gallery_rtl"
        android:layout_width="fill_parent"
        android:layout_height="fill_parent"
        android:layoutDirection="rtl">

    <Gallery
        android:id="@+id/galleryview"
        android:layout_width="fill_parent"
        android:layout_height="wrap_content"
        android:spacing="10dip"
        />

</FrameLayout>
```

Figure 6-20 Test File Clones for BiDiTestGallery Group

Through this example, we propose to show how GATT seamlessly integrates the Java and XML files as text templates. Though optimally these test code clones can be composed into a single template file, for the sake of clarity and ease of understanding the test clones are captured into two template files (`BiDiTestGallery.art` and `XML_gallery.art`) and one specification file (Main.SPC). The skeletal codes in Figure 6-21 show the relevant fragments of the template files (`BiDiTestGallery.art` and `XML_gallery.art`). As can be seen `BiDiTestGallery.art` templates the variability of xml while the `XML_gallery.art` does the same for java. Finally when the test libraries are generated, the `Main.spc` specification file iteratively emits the Java and XML test file codes as specified by the test designer.

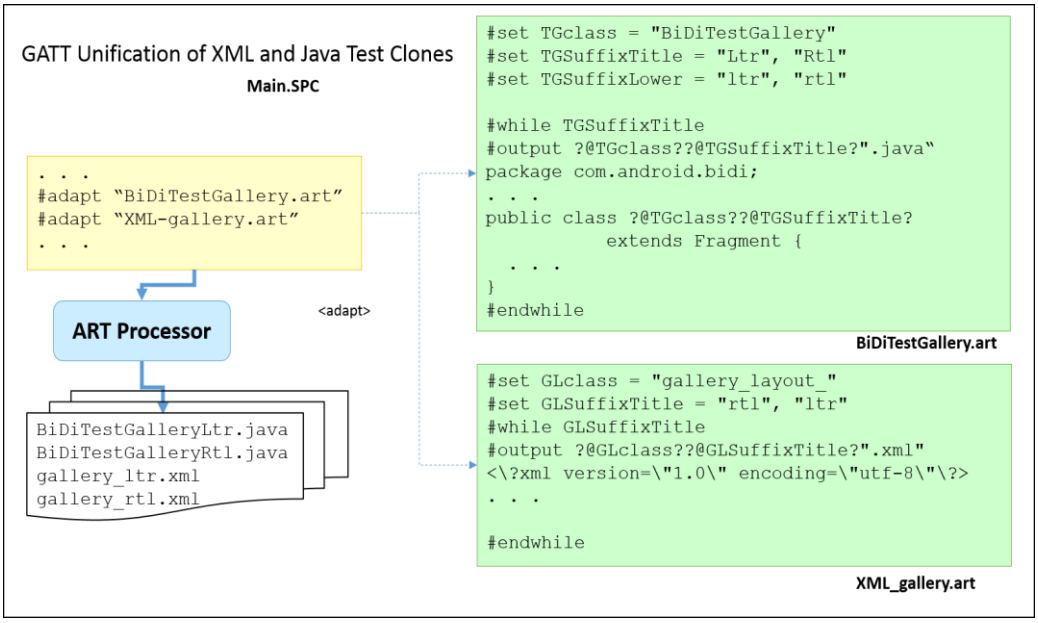


Figure 6-21 GATT Structures for Heterogeneous Test Clone Fragments

Typically, tree structures grow in a balanced manner thereby making them more scalable than other connected structures. Hence, STRAT approach recommends organizing the test templates in a hierarchical connected tree structure thus providing ease of configuration as well as rich customization. Further, ART Processor can interlink these test configurations derived through STRAT with the relevant test libraries and automate test library generation smoothly.

6.3.6. STRAT Process and Template Lifecycle Management

In this section we propose a process comprising of a series of steps that describes the underlying STRAT approach from a testing perspective, thus reducing the semantics and complexity of research problem in hand. Subsequently we provide details of the lifecycle activities involved during test template creation, maintenance and management. The defined methodology for STRAT approach comprises of the following key steps (process) and activities (template lifecycle):

- Step 1: Test Clone Identification - The first step in the approach is to identify test clones occurrences in test libraries.

- Step 2: Feasibility analysis for adopting templates – Using the inventory of test clones and classifying them in the previous step, this step checks on the suitability of template approach for unification.
- Step 3: Template construction – This step involves template creation and management. The management process carried out in stages using four lifecycle activities listed below:
    - Activity 1: Template cataloguing – Involves identification of test clones using a combination of techniques and analysis which yields categorization of test clones.
    - Activity 2: Template unification - This activity involves three tasks in sequence namely, select a technique (GATT or traditional), harmonization and template structure unification.
    - Activity 3: Repository creation – This activity involves storage and retrieval tasks related to templates and test libraries.
    - Activity 4: Evolution of templates for SPL that keep harvesting of test clones and unifies as a continuous activity throughout the test libraries life.
- Step 4: Product specific test library generation from GATT artefacts.

Further subsections deals with each of these steps and activities in detail.

6.3.6.1. The STRAT Process

Test library creation and maintenance is a key testing activity that involves a continuous life cycle methodology contributing towards the products' quality improvements. The process includes design and implementation activities that build and manage the test library artefacts. The generic test case template design process used in STRAT comprises of four stages as shown in Figure 6-22.

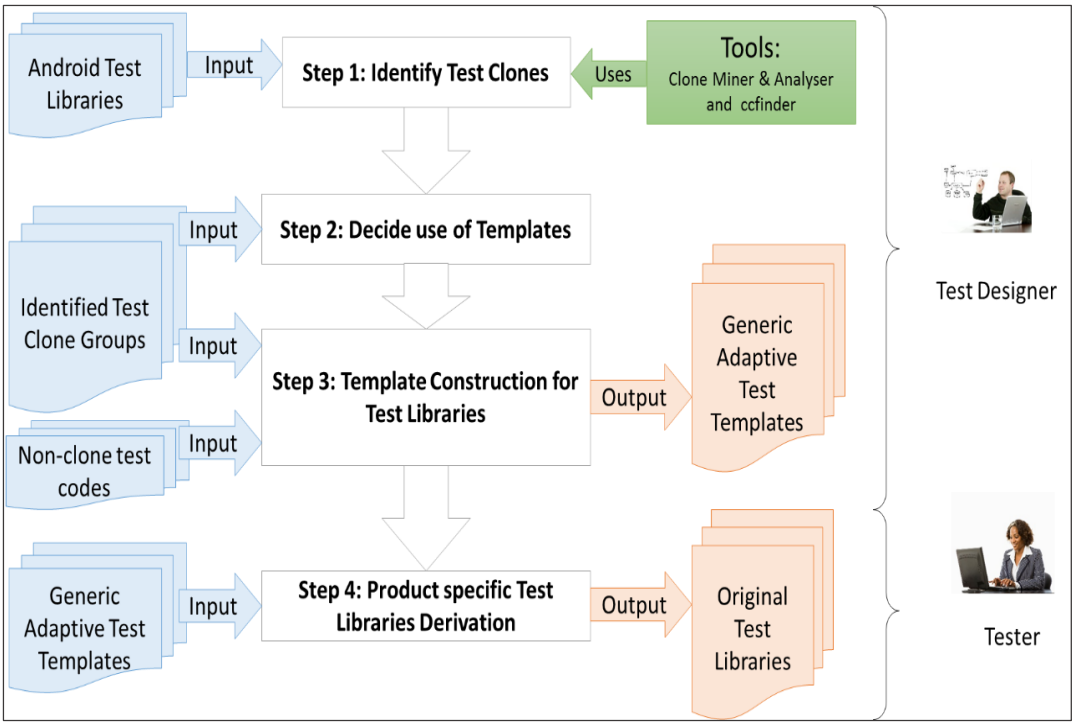


Figure 6-22 Steps in STRAT Process

Step 1: Identify Test Clones

The first step in the approach is to identify various test clones patterns occurring in existing test libraries. Input for this step is the existing test library that needs improvements. This identification starts with use of standard code clone detection techniques followed by test specific improvements. Standard code clone detection techniques help find those functions that are either an exact copy or a mutant of another function in the system. In existing test libraries, most test clones are created by copying a test function and then making a series of modifications to the copy. This is usually due to absence of an established re-use practice during the test library construction processes. Test designers generally copy an entire test suite or a part of it and then change the implementation based on relevant functionalities. In a standard SPL environment, code clone detection algorithm uses different ordinal scales of similarity relations and structural granularity for comparison. Specifically in testing context, identification of test clones requires the use of additional aspects of structural and semantic inferences. Such aspects would also depend on the test library structure and test architecture.

Clone detection tools use pattern matching algorithms and data mining techniques. The mining algorithm in Clone Miner tool used in this study is based on Repeated Tokens Finder (RTF), a token based simple clone detector algorithm for initial token based recurring pattern detection. The algorithm lists simple clones for each method or file, depending on the analysis level. The method level analysis works only when method boundaries are known and simple clones are contained within these boundaries, without straddling them. To detect higher level recurring patterns, the clone miner applies a data mining technique that aims to find all the groups of simple test clone structures whose instances occur in the targeted test libraries along with their locations (like files, classes or methods). The clone miner then normalizes the data by removing the duplicates across the multiple instances of simple test clone sets without missing out any key information.

The role of human perspective provided by an experienced test designer cannot be replaced while building test libraries. Keeping this in view, the STRAT approach streamlines this process by proposing specific improvements on top of regular clone detection techniques as described below:

- Test Feature Identification: Test features can be classified into three kinds based on the variations they deal with. STRAT classifies these features as stable, compile time variations and run time variant choices. These test features are designed and customized by both test designer and tester[5]. STRAT identifies the following special considerations for each of the stated test feature type:
    - Stable Test Features: These test features usually do not change much across testing of different SPL products or versions. These

[5] Roles are mentioned in subsection “Solution Design”

core test assets will be incorporated into templates crafted by the test designer. The main challenge lies in identifying binding options for the product line variations – this will be addressed by test clone harmonization described in sub-section "Template Lifecycle Management Activities".

- Compile Time Variant Test Features: Compile time variant test features are those that are pre-processed before test execution and vary across testing of different product lines or versions. Variants are present in all test artefacts such as test models, test plans, and test libraries. Also variants can be classified further into two groups. (1) Variant Points that are configured to manage the features under test; (2) Variant Points that manage operating platform, evolution and vendor specific variations. Both variant options are crafted by test designers. Testers bind the templates with appropriate binding choices when generating test library for a particular version or platform

- Test Functional Identification: Test clones are identified and grouped based on the encapsulated testing activity such as test fixture, test resource setup, test resource teardown, mock library usage, test functions and test helper functions. Test functional identification would facilitate in making semantic interpretation about test pre-conditions and test methods while constructing test templates.

- Test Granularity Identification: Test clones are identified and grouped based on the physical syntactical test clone granularity such as test directories, test files, test methods and test code fragment. Test granularity identification would facilitate in deciding on template structure and hierarchy.

- Test Commonality/Variability Identification: Presence of test clones indicate the presence of commonality and variability among test clone group. The commonalities among the products will translate into opportunities for reuse in the test artefacts. The variability among products will determine how much testing will be needed.

- Test Domain Influence Identification: Underlying domain perspective can be identified from test clones. This identification is essential since the domain will influence the test execution priorities of product and consequently dictate the test coverage.

Testing frameworks and related test libraries constructed from these testing frameworks combine various technologies involved in the software product line. Due to involvement of multiple technologies, the application testing logic gets spread across multiple configuration files, test scripts and test codes thereby resulting in the test code fragmentation problem. Hence in this step STRAT introduces a methodology wherein the test libraries are first analysed and all possible test clone fragments are identified and categorised in various dimensions to facilitate systematic construction of generic templates.

Step 2: Decide Use of Templates

Second step assesses the feasibility of unifying the identified test clones using generic adaptive test templates (GATT). As stated before, some test clones can be treated via traditional test case design while more complex variants need template design. As already discussed, traditional testing techniques are limited to the expressiveness and generality offered by the underlying programming language. Thus traditional techniques fail to fully exploit test library similarities to derive higher reuse benefits. To address this, STRAT proposes the preparation of a decision schema that would map the test clone types that could be handled through traditional approach or those that require templates. In this step, testers would

map the identified test clones against a set of test clone types that cannot be handled by traditional approach as an aid to justify the use of templates.

This step further decides on the choice of template unification techniques that need to be implemented. When choosing the unification technique, we compare corrective test clone management (i.e. using traditional test methods such as test fixtures, test method reuse, common setup/teardown and test utility files) against compensatory test clone management using templates. Based on the test maintenance activity classified in chapter 4 section 4.2.1 previously, test clone management can be further categorized as preventive, corrective, generative and compensatory. It is generally observed that use of traditional test construction techniques address preventive and corrective test clone management scenarios, while STRAT approach, in addition to preventive and corrective clone management scenarios, can be used to address generative and compensatory test clone management scenarios.

Factors typically considered when making the decision to use template approach are:

- Development and Verification Effort – Is the test library reconstruct-able into templates? Does it require further exploration? How much additional effort is required to create these templates? Does it require the introduction of a new technology? How much verification efforts are required?

- Impact – Is it self-contained? Do the test library changes impact other projects? Does it require changes to unrelated test files in other test libraries? Modification to widely reused test library may have wider impact.

- Risk –Test clone unification requires changes to existing test libraries. What is the risk involved in modifying executable test library? Is the risk manageable (especially when there is no additional test coverage benefit)?

By constructing templates for test clone repetitions of significant engineering importance, testers are able to abstract similarities, boost reuse levels, automate test library generation and improve development/maintenance productivity for test libraries.

Step 3: Template Construction for Test Library

The third step focuses on designing templates that would encompass variability and heterogeneity while still preserving the commonality. Construction and maintenance of test clones are labour intensive tasks. This step targets such situations by improving test clone management via use of template approach. Template construction step involves three activities namely, test clone filtering, test clone harvesting and actual construction of templates.

Test Clone Filtering:

This activity involves removal of false positives. A few of the test clone fragments identified by the clone detection tool (like Clone Miner) may not be clones (i.e., false positive). So before constructing generic test templates, the test clones are manually inspected for removal of false positives. STRAT recommends the use of clone detection tool to find out all possible test clone fragments and then subsequently extract a list of significant test clone fragments by ignoring the rest which could be false positives. STRAT recommends adding the following filters to accomplish this efficiently:

- Granularity: Granularity refers to the structural boundary measure of the test clone in terms of fragment, method, file and directories. Fixing the minimum granularity in terms of token length will guarantee that only test clones of reasonable length are considered for further template construction. For example, if the test designer specifies a minimum token length of 30 all the smaller insignificant test clones gets filtered off for the template construction.

- Test Clone Density: Test clone density refers to the number of test clones that occurs in a unit measure of test library codes expressed as a percentage. Test clone density is an indicator for focusing on more test clone prone test libraries so that unification efforts would yield more benefits from refactoring. For example the test designer can specify that testing suites that have a test clone density of 40% be unified as templates.

Test Clone Harvesting:

The term 'harvesting' is used to describe the process by which periodically test clones are identified and unified as a continuous activity throughout the test libraries lifecycle rather than one single task. Test clone harvesting is guided by domain knowledge (i.e., knowledge of domain level test similarities), test case design and test clone knowledge. Harvesting can be done in both top-down and bottom-up fashions. In the top-down approach, coarse grained test clone structures are identified and broken into further manageable finer-grained test code fragments until the entire complex structure is fully represented via simple test clone units. In the bottom-up approach, we start from simple test clones and build the hierarchy upwards until higher level coarsest test structures are achieved. STRAT approach recommends the use of bottom up approach in most cases. Top down approach would prove beneficial when harvesting is attempted with adequate test domain expertise and is recommended as preferred approach when an organisation has such domain expertise.

Construction and Evolution of Test Templates:

The test clone unification and template construction consists of four core activities namely, Cataloguing, Template Unification, Repository Creation and Template Evolution. All these activities are conducted with template artefacts and thus explained in more detail in sub-section "Template Lifecycle Management Activities" below.

Step 4: Product Specific Test Library Derivation

In software product line testing, test libraries are required to be executed for every build of an application or product. This process is time consuming, unreliable and inconsistent when performed manually. Test cases are executed using multiple test data and fixtures for the same set of actions. Test libraries often involve the use of expensive resources such as test simulators, devices, sensors and data connections. Thus automated test library derivation strategies are essential for effective testing. Using the STRAT approach, once generic test templates are crafted, the tester can choose to derive specific version of test library at any given point in time using the ART Processor. Tester would need to additionally bind the required set of variants and configurations with appropriate variant choices. More details are provided in test template lifecycle activities in the next sub-section.

In conclusion, the four step process described above improves productivity in scenarios where testing templates are derived commencing from existing test libraries. Hence in this approach existing test libraries are analysed to derive the necessary testing templates. This is the most prevalent practice in software product line engineering and has been dealt in this thesis. However, software teams that have strategic reuse plan with prior project experiences in template based test library construction may construct test libraries from scratch using forward engineering process. For this additional inputs in the form of information such as feature model, test-requirement traceability, commonality, and variability may be required to be considered in the test library construction step.

6.3.6.2. Template Lifecycle Management Activities

The STRAT approach consists of a set of logically grouped activities that contribute to the overall productivity and maintenance through the use of the proposed adaptive composition mechanisms. In this section we elaborate the Template Creation step described in previous section 4.1.3. The template lifecycle

consists of four core activities namely Cataloguing, Template Unification, Repository Creation and Template Evolution (shown in Figure 6-23).

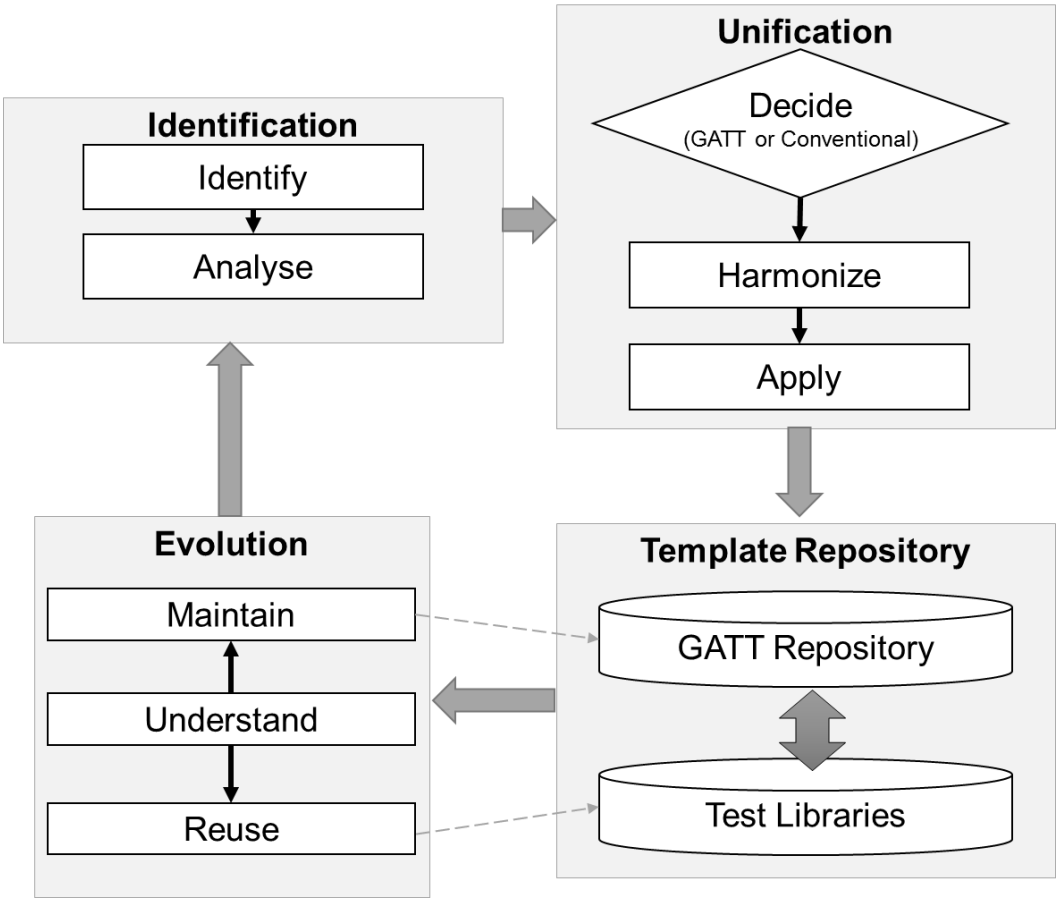


Figure 6-23 Generic Adaptive Test Template Lifecycle Activities

Cataloguing

This activity consists of two tasks, namely Identify and Analyse. Identification of test clones in test libraries can be made using a combination of techniques and is performed in the 'Identify' task of the cataloguing step. The tester may first use clone detection tools like **`CCFinder`** or Clone Miner to prepare a list of test clones. Since the test designers are the creators of test libraries that contain the test clones, their prior knowledge may also be used to draw out the test clone list. In addition, since test clones can be caused by domain, design or feature variations, the identification task may involve manual inspection to apply domain specific knowledge. Hence identification of test clones can be driven from the domain/design perspective or by harvesting existing test clones. The second task

namely ‘Analyse’ involves categorization of test clones to yield systematic cataloguing. This would include measuring test clones based on size, priority, granularity and other taxonomy. In summary, the cataloguing activity encompasses identification and analysis which is followed by filtering and prioritization.

Template Unification:

This activity consists of three tasks in sequence namely, Decide (GATT or traditional), harmonize and apply.

Decide Task:

The first task in this activity starts with a weighted decision which would help recommend if the use of templates over traditional test scripts is appropriate. Factors considered to make this decision include intangible parameters such as business priority, estimation of effort required to create templates, impact of templates in maintenance and evolution, assessment of risks of switching to templates (in the case of operational test libraries) and other trade-offs.

Harmonize Task:

Test clone unification into adaptive templates first requires harmonization. Harmonization refers to reduction of accidental complexity. Examples of accidental complexities include: removal of unintentional variations, reduction in variant points, adjusting whitespace and reordering statements. In the ‘Harmonize’ task, the test designers can select similar test cases or test clones, isolate their common and variant parts, and apply ART constructs to build test templates accordingly. The unification schemes previously listed and illustrated in sub-section “The STRAT Process” would be used in this task.

Apply Task:

Once harmonized, templates are constructed by applying ART constructs in the 'Apply' task of the unification process. This task includes creating new templates as well as managing exiting templates. The template file hierarchy can be decomposed as: specification files, adaptive template files and variant fragments. The test designer constructs templates by decomposing original test clones into tiny fragments or files depending on the variant management strategy adopted. Finally the composed template hierarchy is normalised to remove any duplicate template fragments or files. An example of such template hierarchy is shown in Figure 6-24, illustrating a composition of one specification template file, two template files and two template fragments.

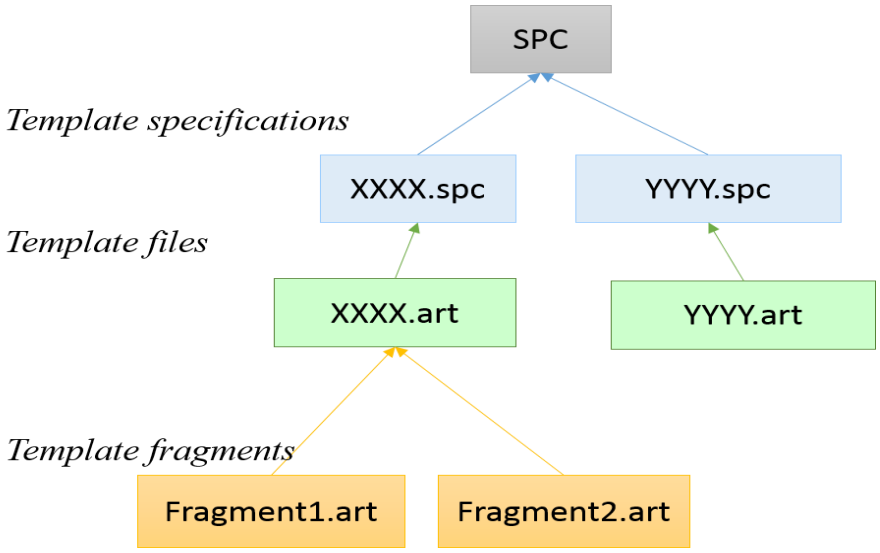


Figure 6-24 Template Hierarchy Example

In summary, template unification requires performing the following key steps:

1) Decide on usage of reuse approach based on test templates to refactor exiting or create new test libraries.

2) Harmonization of templates which include selection of template hierarchy and variant point decisions.

3) Application of templates which consists of:

a. Construction: Conduct an iterative style template construction process. The templates files are arranged in hierarchy mentioned previously:

    i. Create the master specification test template catering for global variants and binding choices managing the whole test library alongside output directory descriptions.

    ii. Implement multiple template files and frames incrementally depending on the size, priority, granularity and other taxonomy. Usually the original test library is chosen and converted into template files and fragments.

    iii. Unify test clone groups by provisioning for commonality and variability representation.

    iv. Use mark-up mechanisms that define extension points inside test templates.

    v. Review for duplicates and normalise the templates.

b. Verification: Verify the ART processor output with original test library to look for lossless translation.

c. Debug: In case the translation is defective, troubleshoot the affected templates.

d. Optimization: Carryout optimization and improvements on the constructed set of template files and fragments iteratively.

e. Further Iterations: Depending on the progress and objective the template are further improved in subsequent iterations.

Repository Creation

In this activity two repositories are created one for templates (GATT) and the second to host the core and template generated test libraries. The use of templates allows testers to store core assets into a common test case library repository. Test developers browse existing test templates to grasp the overall similarity situation (Figure 6-25) and narrow selectively to specific templates with the view to understanding the exact nature of similarity and differences in a given group of test cases.

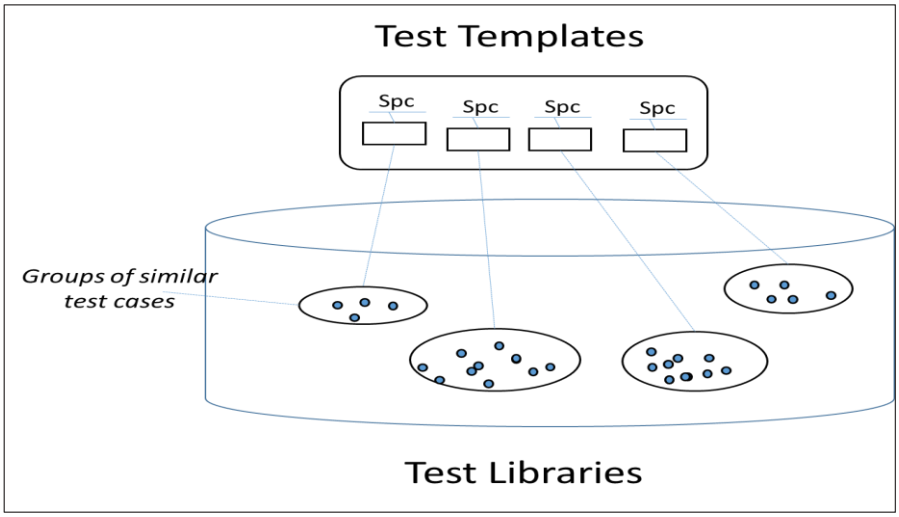


Figure 6-25 Template Repositories

The format of template storage and configuration management are crucial for the sustenance and success of the STRAT approach. The templates can be stored on the same source repository as that of the application code. The template files (SPC and ART) are text format files so that they are non-intrusive to the run-time hierarchy of test library and subsequent test execution runs. Generally template files are managed as a separate module and are not mixed with the existing test library codes; refer (Figure 6-25). Being a product line the test templates are treated as core assets necessitating the test team to establish individual configuration/workspace management procedures. Further STRAT recommends that the test templates are to be base-lined and necessary multiple asset evolution paths need to be designed and implemented.

Template Evolution

Improvements and evolution of the harvested generic adaptive test templates are the only way we can sustain the quality of the overall test library. STRAT recommends the following practices to be inculcated in the 'Template Evolution' activity:

- Matching the template construction closely to the core design and domain specialization in order to make the solution easier and concise.
- Making conscious efforts to curtail the depth of template trees and organizing the templates into logical manageable groups so as to ensure that the evolution becomes more comprehensible.
- Carrying out regular improvements on variability and variant choices management with a view to preserving the generality (generic nature) and correctness of template representation is sustained.
- Harvesting of test clones periodically and standardizing the unification procedures to ease maintenance activities in the long run.
- STRAT recognises that while the core technological solution for addressing the test clone redundancy problem is the design and implementation of GATT, to derive the full benefits of template-based approach organisations would need a life cycle management strategy. With this in view, this thesis has come up with a proposal for Life Cycle Management approach as presented in this section.

6.4. Addressing SPLT Challenges using STRAT approach

In a software product line testing context, the presence of multiple layers and variability results in: (1) voluminous growth in test libraries, (2) presence of greater redundancies in the form of test clones, (3) heterogeneous nature of test artefacts and (4) challenges to the inevitable need for scalability. STRAT attempts

to address these issues. This section illustrates how the STRAT approach counters each above mentioned challenges.

6.4.1. Countering Voluminous Growth

Voluminous growth of test libraries occurs due to the need to test combinatorial explosion of feature and variant combinations. While features are domain and application dependent, variant parameters are more product line and technology dependent. There are two means to counter this explosion. One way is to restrict test case design only to legally permissible combinations of features and variants. The other possibility is to exploit the similarity among test cases for different product variants. Generic adaptive test templates provision for both means of countering using a versatile adaptive reusable technique as could be seen from the above derivation. Thus the use of STRAT fosters construction of moderated templates.

6.4.2. Countering Redundancy

The second key challenge posed by SPLT variability is redundancy. If test libraries are not well-designed, redundancy builds up over a period of time and makes test case library maintenance difficult. A systematic and holistic reuse approach provides a significant opportunity to counter redundancy. As mentioned, STRAT approach uses templates as a formal mechanism to implement effective reuse through unifying the similar code fragments into a single template file. As can be seen from the proposed template structure derived above, the template offers necessary encapsulation and extension which are key concerns in software product lines. Thus the approach formalizes, preserves, and reuses the domain design accumulated within test libraries in a clone free (or clone minimised) format.

6.4.3. Managing Heterogeneity

With constant changes in computing platforms prevalent in software product lines, technology has become more inclusive by managing polyglot of programming languages and heterogeneous platform configurations. Thus, in order to meet adaptability to software product lines, one of the key requirements for the approach proposed in STRAT is to ensure seamless integration of language specific test libraries, platforms, and external libraries in a tool neutral way. This is possible only if the solution can function as a logical unit separate from the original test library repository. The principle of design generality advocates the importance of designing software that is free from unnatural restrictions and limitations[103, 104]. The STRAT approach meets all of these factors through designing a separate meta-layer to handle heterogeneous environment.

6.4.4. Improving Scalability

Software product lines evolve very rapidly and hence require scalability not only in the software product designs but also in testing solution designs. In addition, typical SPL test libraries are also large. To ensure test library scalability, STRAT offers primitive building blocks which are supplemented by compile time generators that can compose these blocks to yield the necessary test libraries. The template structure (GATT) derived in the proposed STRAT approach scale very well in a controlled hierarchy. The hierarchy naturally forms tree structures and thus easily scales well to large scale test libraries. Thus templates are organically grown to expand as scalable clusters.

Thus, in summary STRAT approach palliates the major shortcomings inherent in SPL testing:

i. By supporting heterogeneity through its ability to include multiple programming languages, platforms and configuration script formats.

ii. By improving scalability using a bottom up tree hierarchical organization of template.

iii. By countering redundancies and unifying them as appropriate template constructs.

iv. By curtailing volume through limiting the legally permissible feature combinations without loss of precision.

6.5. Benefits of the Approach in SPL Testing Context

Existing software product line test construction approaches handle variants at programming language level. These mechanisms are simple, cost-effective and work well as long as the numbers of variant features differentiating the products are small. In recent smart phone based computing landscape, test libraries are complex and cannot be dealt using the old singular homogenous variant management techniques. Product line configuration and customization is complex and handled through parameterization of variant point at different stages such as requirements, domain engineering and design, coding, and testing.

It was not our goal to contribute anything towards generic design, instead we used ART that has good record in other applications. From the studies conducted on Android platform, there are three prominent findings unique regarding test libraries. (1) Test libraries have higher percentages of redundancies in comparison to normal code owing to the fact that multiple test cases are being built around same component for better test coverage. (2) Android Test libraries use polyglot of programming languages as the mobile platform is composed of heterogeneous integrated. (3) Test libraries related to UI components exhibit higher redundancies resulting from the similarities among the UI components in terms of graphic type, events designed, activities triggered, fixture setup/teardown and listener actions.

The STRAT approach proposed in this thesis implements "generic test case templates" that can counter test case explosion problem using a heterogeneous variability management technique. The novelty of the proposed approach is that the generative technique is programming language and platform independent. In addition, the ART based approach can manage variations, preserve commonality

and propagate changes across versions of test library repositories. Key benefits of the proposed approach are summarised below:

- SPLT deals with large voluminous test libraries. First and foremost benefit of the proposed approach is that mixed strategies can handle combinatorial explosion. These wide input spaces of testing that usually result in combinatorial explosion are curtailed by ART template based approach. STRAT constrains testing space to only possible variant combinations, tracts and preserves the code-test case relationships inside specific meta-generative frames.

- In product line engineering, variant points and relevant binding cause major source of errors. With larger amount of variants, the complexity increases and exhaustively testing all variants in a core component prior to product assembling is practically infeasible. In this context STRAT fosters planned reuse and aims at validating only permissible variant combination, thereby minimize testing efforts and increasing the overall productivity.

- SPL test libraries comprise of heterogeneous assets with various levels of testing (unit, integration, system, and acceptance), various test artefacts (code, configurations and scripts) and various testing strategies (white box and black box testing). Adding to the above mentioned benefits, the STRAT approach being a meta-generative technique provides a systematic scalable means to handle heterogeneous test assets.

The core contribution of this research work is in the simplification that is derived from achieving non-redundancy in terms of reduction in both the size of test libraries and its conceptual complexity. An evaluation of some of these stated benefits and trade-offs has been performed which would be presented with a case example in the next chapter.

6.6. Limitations

Despite the key benefits rendered by the proposed approach, currently the STRAT approach exhibits some known limitations. The following list describes a few such limitations along with possible suggestions to address these.

- The proposed approach is useful for situations where the time required to abstract out the variability is more than compensated for by the time that would be required to create and manage individual occurrences of a test clone pattern. STRAT is not an "all-win" approach, hence it is essential to apply quantitate and qualitative evaluation of the results using control experiments, productivity metrics comparison and analytical arguments before deciding to use test template approach.

- The proposed STRAT approach requires that a complete test library be developed and then decomposed into the appropriate template chunks, categorizing these into regions of commonality. Then a set of template fragment pieces for each variant needs to be devised. The variant pieces can be produced as and when needed or produced up front for later use. Generation test libraries requires more planning and initial effort than simply constructing a test library directly. This additional effort for template construction is expected to be offset through a reduction in effort for testing subsequent products and release versions if a proper implementation strategy is used.

- Though the GATT structures are organized into template hierarchy of specification and template files, the test designer will have a learning curve to create and manage two layers of artefacts inside the test assets. While an initial disciplining and learning effort is incurred, it pays off when the test library scales and evolves.

- The proposed approach is yet to be ready for massive industry adoption. Integration with existing tools, a focused template editor and clone detector tools of industry standards are possible extensions.

6.7. Chapter Conclusions

In summary, the proposed approach provides a compile-time based test asset maintenance model for large scale test libraries. In the context of this thesis, the term "compile-time" refers to test library construction/selection phase. The template based derivation provides standardized processes to perform test activities in a product line that makes test routines consistently common across multiple applications/products. Thus the use of STRAT approach yields the following:

- Saves time in creating new extensions to existing test libraries (tester need not reinvent the wheel).
- Provides variability breakpoint facilities using the versatile generic adaptive test templates for test library evolution.
- Simplifies the process for changing common test codes without the need for multiple modifications or re-linking.
- Facilitates management of sticky change repercussions on template based test library since the proposed technique is easier than using traditional testing techniques.
- Curtails explosion of redundant test clones thereby promoting reuse in test libraries.
- Improves the selection and customization of baseline test libraries making the processes formalized, automated and productive.

It would be of research interest to demonstrate and assess the benefits resulting from the STRAT approach proposed here as a logical research follow up. In the next chapter a practical example using a test library selected from Android Platform OS is taken up with the view to rebuilding the library using generic adaptive test templates and to make some experimental assessments of the benefits derived.

CHAPTER 7
Case Study: Generic Adaptive Test Templates for BiDiTests Library

This chapter provides a case study to illustrate the STRAT approach formulated in the previous chapter. By using the STRAT process, we derive generic adaptive test templates using examples from a selected Android OS Platform test library. This chapter discusses in detail the steps involved in constructing the test templates using the STRAT approach. Subsequently the chapter presents an experimental analysis to assess the productivity gains derived through the template-based approach as well as to identify the constraints and trade-offs involved. The chapter also presents the experimental outcomes on test evolution and demonstrates the benefits of STRAT in test library maintenance using selected metrics.

The organization of this chapter is as follows:

- Section 1 describes the purpose of this research experiment.
- Section 2 sets the context for this case study
- Section 3 describes the selection process for the case study from an identified pool of test library projects.
- Section 4 introduces the classes and variants in BiDiTests case example.
- Section 5 describes nature of redundancies present in BiDiTests and their GATT representations.
- Section 6 describes how the templates (GATT) are constructed for BiDiTests test library.
- Section 7 performs details experimental evaluation for the case study and reports on details of productivity measurements observed.
- Section 8 presents some possible adaptations of the template-based approach to other areas.
- Section 9 concludes this chapter.

7.1. Purpose

Software product line engineering offers a plethora of variability techniques and strategies for core and product-based code maintenance; but offers little guidance for test libraries resulting in significant test case redundancies and lack of scalability. Thus the key challenge in SPL testing is to achieve reuse. The proposed STRAT approach targets this by introducing a template-based approach that effectively manages test library commonality and variability.

To assess the effectiveness of the proposed approach we present a case study. The purpose of this case study is to:

1) Demonstrate the STRAT approach using an illustrative example and develop generic adaptive test templates for the test libraries in the chosen example.

2) Use the developed templates to generate test library and demonstrate lossless translation of test libraries is achieved using the proposed generic adaptive test template structure.

3) Compare the GATT based test libraries source codes with the original for the purpose of establishing the benefits derived using STRAT approach

Thus the aim of this case study is to provide a concrete step-wise illustration of the STRAT approach regarding how the previously identified redundant test clone groups are systematically unified into generic adaptive test templates. The illustrative example also helps in providing additional contextual interpretations, refinements related to evolutionary changes, tactics for change propagation visibility and assistance for managing multiple version releases.

7.2. Context

Android is the leading open source platform for mobile computing and it promotes component architecture. The popularity of android devices increases the need for

testing that assures robust and reliable software stack. Android software stack can be further subdivided into five layers: Linux kernel, native libraries, platform framework, runtime and applications. Being an emergent platform facing rapid development, new major releases are delivered frequently (at least once every few months). This enormous growth necessitates sustainable and repeatable automation efforts towards test library maintenance aspects. This research work focuses on a test library example that performs testing of the platform framework layer. The platform framework layer is an important component as it comprises of the key Android product line core assets. The platform layer provides abstractions for the underlying native libraries and kernel. Testing this layer involves several constraints and challenges, and hence is an ideal candidate for experiment and research evaluation.

7.3. Selection of Case Study

The main aim of this example is to investigate if the proposed STRAT approach produces test libraries that are easier to manage and evolve in comparison to traditional techniques. Thus, for this illustrative example, we use the "one factor with two treatments" research design technique[76] . In the context of our case study, the factor under investigation is the test library while the two treatments investigated are the traditional techniques and the new STRAT approach. To make a comprehensive, practical and unbiased comparison between the two techniques, a structured mechanism to select a truly representative candidate is necessary. For the selection of the candidate Test Library we used a methodical process following the well accepted "balanced design principles"[139] which consists of the following steps:

- *Identifying Sample Space:* This step involves defining the sample space from which the candidate would be selected. In our case we commenced from a universal set of 500 Android test libraries to arrive at a sample space of 40 projects using the process explained in subsequent sections.

- *Selection Criteria:* This step identifies the key selection criteria that characterize an ideal representation of the research subject. In our study, the focus is to select a candidate that would comprehensively illustrate all facets of the STRAT approach. The selection criteria must also include factors that would help to measure the productivity changes in terms of reusability and maintainability metrics that were previously defined in chapter 5.

- *Methodology:* This step involves the design, implementation and execution of a systematic selection method. In our case, during this step we consistently evaluate all identified test library project candidates. The selection methodology formulates steps related to qualifying potential candidates (test library projects) and judging its fitness against defined selection criteria.

- *Selection:* Final selection involves the identifying of one typical candidate. In our study this step yielded a smaller set of test projects that pass the fitness judgment and eventually we narrowed to "BiDiTests" as the ideal candidate for the illustrative example that would best meet all criteria as well as help focus on comparing the two approaches.

A detailed discussion on how we applied the above steps in the Android platform and how we arrived at the `BiDiTests` as the candidate case example is described in detail in the following sub sections.

7.3.1. Identifying Sample Space

The empirical study presented in Chapter 4 has established presence of redundancies in Android test libraries. Android (universal set) software stack is logically grouped as kernel, tools, devices, accessories and platform. Here the platform group consists of the commonly used features (core assets) across all Android devices and is divided into further sub-groups such as Dalvik virtual

machine, external libraries, framework, hardware, packages, SDK (software development kit), system and tools. The framework subproject has all types of test libraries (e.g., unit, integration, and UI test libraries) and we decided to conduct the Chapter 4 study on test libraries redundancies using platform group and framework sub-group. But this selection comprises of more than 500 types of test libraries. Since we are more interested in generic core assets, we narrowed our search to the test libraries in base repository of framework comprising of 40 test projects.

These 40 projects have mixed representations from unit, integration and UI test cases. Unit test libraries verify an `Activity` in isolation using the `ActivityUnitTestCase` class. Unit test cases for this context allow verification of layout of the activity and also to check if intents are triggered as planned. Integration tests verify interaction with different components. The communication with the Android infrastructure is done via the `Instrumentation` class which can be accessed via the `getInstrumentation()` method. UI testing ensures that the platform returns the correct UI output in response to a sequence of user actions on a device, such as entering keyboard input or pressing toolbars, menus, dialogs, images, and other UI controls. From this chosen sample space of 40 projects, we intend to pick a representative archetype test library to illustrate the STRAT approach alongside practical considerations involved in the iterative construction of generic adaptive test templates (GATT).

7.3.2. Selection Criteria for an Ideal Test Library (Illustrative Example)

In order to better utilize the effort spent, it is important to ensure that the selected example satisfies both the intent for template construction and also facilitate comparison against traditional techniques. To ensure such characteristic matching

to be used in a scientific selection process, the criteria that an ideal case example should satisfy was first defined.

In test library context, the selection criteria must be representative of the underlying similarity and granularity expressed as test clones. Such test clones could be both structural and semantic. Identifying semantic similarity in a non-deterministic research problem has been excluded from the scope of our study since detecting true semantic similarity is undecidable problem. Even in the case of generic and structural test clone, certain parameters require domain knowledge and outcomes cannot be measured in clear quantitative terms (e.g., intangible benefits). Test libraries/cases that have such parameters were excluded. The selection process adopted in this research uses criteria based on the test clone taxonomy discussed previously in Chapter 5 and the relevant definitions are briefly recalled here for reference:

- Test Clone Type: Test clones can be of general, structural or sematic types. This thesis restricts itself to general and structural test clones. Based on these two types of test clones we can further describe the test clone type as one of the following:
    - Simple Test Clones: Exact Test Clones, Renamed Test Clones, Parameterized Test Clones, Near Miss Test Clones, Gapped Test Clones and Non Contiguous Test Clones
    - Structural Test Clones: Simple Structural Test Clones, Functional Test Clones and Design Level Structural Test Clones
- Test Clone Granularity: Based on structural boundaries test clone granularity is classified into simple, method, file and directory. Using the clone hierarchy we can form a list of clone types that grows in granularity from few lines of similarity to directory level similarity. The hierarchy is listed as: Simple Test Clone Structures (granularity being fragments,

methods and files), Test Method Clone Structures (granularity being methods, files and directories), and Test File Clone Structures (granularity being files and directories).

- Test Clone File Formats: The Android test library projects comprise of two file formats. The test codes are in Java programming language and test configuration is presented as XML (eXtensible Markup Language) files.

An ideal test library candidate should meet various criteria (and its subtypes) in terms of test clone type, granularity and file formats. In summary, the ideal candidate should meet the following criteria:

a) The chosen test library should have both general and structural test clone examples.

b) The test clones should possess various granularity such as simple, method, file and directory.

c) Test clone files formats should comprise of both java and xml.

7.3.3. Selection Methodology

Selection of the ideal example (test library) is a crucial step in this research, since the selection is closely connected to the generalization of the results by comparing STRAT approach with the traditional techniques. In order to generalize the results to test libraries in general, the example must be representative of an archetype of typical test library. The size of the test library also impacts the results when generalizing. The use of a larger test library ensures results are more representative by minimizing errors in interpretation the results with limited data. Thus the selection methodology must be based on purposive sampling and not a random choice. The selection methodology we use has three key stages, guided

by generic research design principles as suggested by [155]. The key steps in the selection methodology are purposive sampling, filtering and balancing.

- **Purposive Sampling**: A purposive sample is a judgmental sample that is selected based on the knowledge of the test project's fitness against a defined set of selection criteria and the purpose of the research. In this case study, the test library selection is based only on similarity perspective and not on any other parameters. To meet this requirement we use purposive sampling. The selection of test library (research subject) will thus be representative of various types of test clone redundancies.

- **Filtering**: The test libraries used in this case example have different types of test case redundancies. Some of these can be solved (treated) using traditional testing techniques. Since the effect and reuse treatment of such generic test clones is known, filtering is used to exclude samples that can be treated through traditional techniques.

- **Balancing**: Selecting a test library that has significant numbers of all clone types would yield an experimental design which would ensure unbiased interpretation of results. Balancing is desirable because it strengthens the statistical analysis of the case study. We adopted a selection methodology that uses balancing by selecting a test library that comprises of test clones spread across all categories and reasonably present in equal proportions.

The resulting candidate test library is expected to be typical, revelatory and be illustrative of the STRAT approach. The selected example serves as descriptive instance for reconstructing a test library using templates by introducing the reuse concepts and variability management techniques relevant to test libraries.

7.3.4. Selection from Android Platform Test Repository

Mobile computing functions amidst constraints such as low-powered-CPU, small-memory, limited display area and power supply. These constraints demand well

established, matured testing tools and techniques. The Android testing platform is an automated regression test library repository that facilitates special test instruments targeting peripheral devices, special sensors, graphics, geo-location, communication protocols and other media. The selected sample of forty test library projects is used in testing the Android platform UI facilities. This package was chosen because the test cases are system testing representative of typical black box strategy based test libraries and not specific to the operating or coding platform. The projects we selected as candidates for investigation are being presented as a feature diagram below.

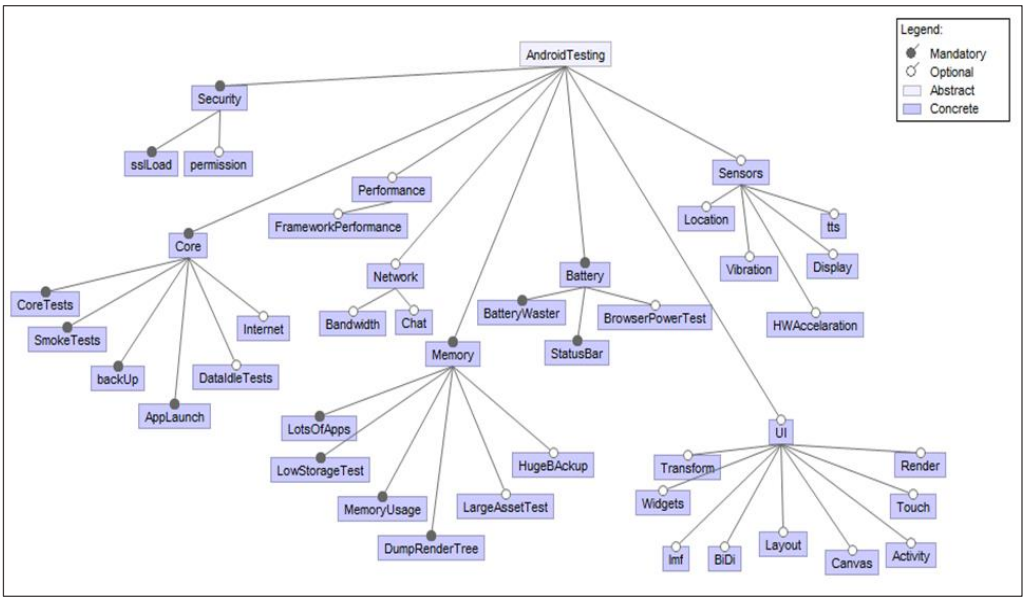


Figure 7-1 Android Platform Framework test libraries as a Feature Model

As can be observed, the main classification among the test libraries are based on the features named core, security, performance, networking, memory, battery, sensors and UI related test libraries. The core library deals with tests related to core platform functionalities, instrument setup, application launchers, internet connectivity and smoke tests. Security test libraries verify SSL (secured socket layer) and user certificate permissions. Network test libraries focus on internet bandwidth settings and chat related protocols. Memory related platform functionalities are tested against usage, storage, backup and rendering abilities.

Sensors based test libraries deal with verification of location, text-to-speech, display, device vibrations and other hardware devices. Battery test libraries checks for wastage, consumption and display status correlation. UI test libraries are system test libraries dealing with layout, display orientation, touch sensitivity, rendering facilities and other UI related activities.

A systematic sampling of significant test clone groups occurring in each of the forty test library projects are given in Table 5. All the test projects from the sample have been grouped and listed as rows while test clone similarity measurement are provided in columns depicting Simple Test Clone Classes (STCC), Simple Test Clone Structures (STCS), Test Method Clone Classes (TMCC), Test Method Clone Structures (TMCS), Test File Clone Classes (TFCC), Test File Clone Structures (TFCS) and Test Directory Clone Classes (TDCC). Though simple clones are identified using the Clone Miner tool, manual inspection was used for higher level test clone structures identification and analysis. The latter columns of the table listing sophisticated structural test clones are more meaningful in the context of refactoring into templates and are discussed in detail with respect to BiDiTests Project test libraries.

Table 5 Test Clone Analysis for Android's Core Test Library Projects

Group	Project	Java Files	Project Directories	Total Lines Of Code (TLOC)	Executable Lines Of Code (ELOC)	Total Methods	STCC	STCC Instances	TMCC	STCC Files	STCS Files	TMCC TMCS Files	TFCC TFCS Files	TDCC Files	%Clone Files
Core	App Launch	1	6	352	279	11	0	0	0	0	0	0	0	0	0%
	backup	2	5	173	120	1	0	0	0	0	0	0	0	0	0%
	Browser Test Plugin	7	9	893	499	1	0	0	0	0	0	0	0	0	0%
	Compatibility	2	5	283	160	8	0	0	0	0	0	0	0	0	0%
	Core Tests	16	3	6372	4045	193	40	101	42	5	2	5	0	12	75%
	Data Idle Test	1	6	150	91	5	0	0	0	0	0	0	0	0	0%
	Shared Library	4	19	328	194	5	0	0	0	0	0	0	0	0	0%
	Smoke Test	3	10	217	121	21	1	3	1	1	0	0	0	1	33%
	Smoke Test Apps	3	6	168	94	1	3	3	3	3	0	0	0	3	100%
	Web View Tests	9	7	2529	1696	187	38	149	30	1	0	3	0	4	44%
Memory	Dump Render Tree	19	9	4791	3523	263	13	30	24	7	4	3	0	14	74%
		24	14	5731	3896	313	19	41	26	13	5	1	0	19	79%
	Huge Backup	2	8	639	357	10	0	0	0	0	0	0	0	0	0%
	Large Asset Test	1	9	211	141	4	0	0	0	0	0	0	0	0	0%
	Lots Of Apps	1	7	734	713	1	0	0	0	0	0	0	0	0	0%
	Low Storage Test	1	8	242	188	5	1	2	2	1	0	0	0	1	100%
	Memory Usage	2	6	338	240	15	1	2	2	1	0	0	0	1	50%
Network	Bandwidth Tests	2	9	323	230	9	1	2	2	1	0	0	0	1	50%
	Serial Chat	1	7	244	189	7	0	0	0	0	0	0	0	0	0%
Performance	Performance	9	13	3521	2991	162	15	39	27	2	0	0	0	2	22%
Power	Battery Waster	1	8	306	238	12	1	2	2	1	0	0	0	1	100%
	Browser Power Test	3	5	452	323	26	0	0	0	1	1	0	0	2	67%
	Status Bar	4	12	1764	1559	4	0	0	0	0	0	0	0	0	0%
Security	permission	6	7	1062	719	33	47	183	22	3	1	0	1	5	83%
	ssl Load	1	5	138	100	4	0	0	0	0	0	0	0	0	0%
Memory	Dpi Test	2	28	718	539	16	2	4	4	1	1	0	0	2	100%
	Fix Vibrate Setting	1	10	212	161	8	0	0	0	0	0	0	0	0	0%
	Hw Acceleration Test	87	14	8162	5861	397	66	182	102	58	14	0	0	72	83%
	Location Tracker	11	11	1815	1138	92	1	2	1	1	1	0	0	2	18%
	Tts Tests	3	6	412	259	21	1	2	1	1	0	0	0	1	33%
User Interface	Activity Tests	6	11	198	159	22	17	56	1	1	0	0	0	1	17%
	Assistant	1	14	174	122	1	0	0	0	0	0	0	0	0	0%
	app widgets	5	23	720	507	22	2	4	2	2	0	0	0	2	40%
	BiDi Tests	41	10	6733	5325	59	6	14	12	7	15	2	5	29	74%
	Canvas Compare	8	10	1822	1460	105	3	11	4	2	2	1	0	5	63%
	Grid Layout Test	11	9	762	472	21	3	7	7	4	2	0	0	6	55%
	Imf Test	31	15	2597	1473	74	10	24	13	13	6	4	3	26	84%
	Render Script Tests	83	93	5648	3904	630	100	273	148	56	15	4	4	79	95%
	Tile Benchmark	6	11	1422	1109	76	5	12	10	3	0	0	0	3	50%
	touchlag	1	1	295	240	0	0	0	0	0	0	0	0	0	0%
	Transform Test	1	10	239	179	11	0	0	0	0	0	0	0	0	0%
	Total	421	479	63890	45614	2856	396	1148	488	189	69	23	13	294	16

Using the selection methodology steps stated earlier (i.e., sampling, filtering and balancing), we arrived at the `BiDiTests` project as the representative candidate for this research case study.

- Using purposeful sampling we chose forty projects and at this step all were considered and given equal chance to participate in the experiment.
- Using the filtering step the test clones were colonized into general and structural test clone types.
- Using the balancing principle, only those projects containing samples from all clone types were selected.

Thus the choices were limited to (1) `lmfTests` (2) `BiDiTests` and (3) `RendererScriptTests`. After detailed analysis of forty system testing projects in Android platform test libraries, we selected the `BiDiTests` Project because of larger size 5325 ELOC (41 Java test files and relevant 37 XML configuration files), presence of similarities are various granularity (at test code, method and file levels) and being representative of various types of redundancies (generic and structural test clones).

7.4. Introduction to 'BiDiTests' Test Library

The `BiDiTests` project represents a typical system test library, an archetype of traditional user interface (UI) testing in smart phone platform. Information from the developer forum, API documentation and GIT repository version details were used to understand the underlying domain and maintenance aspects across subsequent release version. This information was later used to measure change requests related metrics.

`BiDiTests` Library comprises of UI based functional test cases designed to test the bi-directional layout of the device screen orientation using a black box approach. `BiDiTests` has both Java and XML based test codes. These test cases

validate the bidirectional functioning of selected `View`, `Widget` and `Fragment` UI that are part of the System API. `View` class is the basic building block for UI components. `View` occupies a rectangular area on the screen and is responsible for drawing and event handling. `View` is the base class for `Widgets`, which are used to create interactive UI components (buttons, text fields, etc.). A `Fragment` is a self-contained component with its own UI and lifecycle; it can be-reused in different parts of an application's user interface depending on the desired UI flow for a particular device or screen. Test classes in `BiDiTests` project create test components using the three UI (`View`, `Widget` and `Fragment`) on various layout settings to test the left-to-right and right-to-left orientations of the UI components. A layout defines the visual structure for a user interface, such as the UI for an activity or widget. Figure 7-2 shows the participating class names of the `BiDiTests` project under the `TableLayout`. Each test layer focuses on verifying a particular combination of feature variants (graphic type, sub-type and orientation). The class hierarchies for other types of layout namely `LinearLayout`, `FrameLayout`, `GridLayout`, and `RelativeLayout` are similar to `TableLayout`.

The project defines different types of layouts explained earlier using XML vocabulary. The advantage of such external declarative definition is that it enables better separation of presentation from the platform behaviour. The external declaration allows modification and adaptation of display orientation without having to modify source code or recompile codes. `BiDiTests` project thus creates XML layouts for different screen orientations, different device screen sizes, and different languages. Additionally, the test scripts also verify proper visualization of the UI components structure.

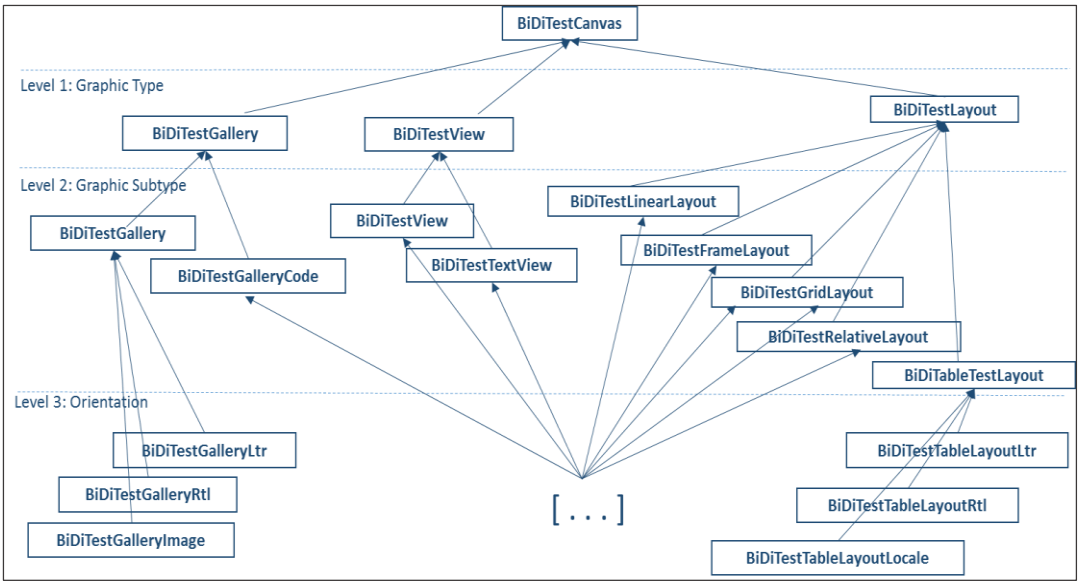


Figure 7-2 BiDiTests Partial Class Diagram

`BiDiTests` Libraries include tests for three types of graphical entities namely, layout, view and gallery. Each entity has unique properties such as size, colour, appearance, position, visibility and other behavioural properties associated. The test cases related to each graphical entity exhibits a similar/standardized structure in terms test fixtures, test data setups, tear downs, events and action lifecycle testing methods. Analysing from the class hierarchy and feature based testing perspective, `BiDiTests` classes have three levels (as shown earlier in Figure 7-2).

1) At the top level, testing focuses on graphical type. `Canvas` is the basic UI entity because all other graphical entities are added on top of canvas. Apart from canvas, there are `Activity` files, utility and constants classes. The `Activity` initializes the logical test suite and executes all other test case classes.

2) Analysis of the second level reveals that `Canvas` is further divided into sub-types (example: various layouts) or specialties (example: text view is a special type of view). For instance, let us consider the subtype `Layout` which figures in the UI test. `Layout` is further divided into five subtypes

namely, `frame`, `grid`, `linear`, `relative` and `table`. In a graphical scenario, the entity named `View` is further sub-classed into `TextView` (specialty). Test attributes related to the normal view are not sufficient to describe the special attributes for text view. Thus additional set of tests are designed to cater for text view.

3) At the third level, each graphical entity has to be tested for various orientations namely, `left-to-right` (ltr), `right-to-left` (rtl) and `locale`. To provide an example consider a graphical entity *Layout* and subtype *Table* with three orientations (classes `BiDTestTableLayoutLtr`, `BiDTestTableLayoutRtl` and `BiDTestTableLayoutLocale` shown in Figure 7-2). Let us consider few more such examples. The test file named `BiDiTestRelativeLayoutRtl.java` tests the graphic canvas for relative layout and right-to-left orientation. `BiDiTestTextViewLtr.java` would test the graphic canvas for Text View UI and left-to-right orientation. Thus in the BiDiTests test files, feature variant layers are explicitly declared in the file names in order.

7.5. Study of redundancies in 'BiDiTests' Test Library

This section discusses the redundancies found in the `BiDiTests`. According to the test granularity defined earlier in Chapter 5, a typical test library comprises of test codes fragments, test methods, test fixtures, test files and test directories.

The primary motive of presenting this illustrative case study is to provide an example for GATT. In view of the large size of the solution and for illustrative purposes we have confined ourselves to a few representative code fragments to showcase our solution in this thesis. These examples were chosen to demonstrate the presence of a cross-section of test clones belonging to different types namely, simple and structural test clones. In addition to the java examples, to demonstrate

the heterogeneity of the proposed GATT structure we have also provided an example where XML based test clone fragments were present.

For the scope of this case study, only test clones that have at least 30 tokens similarity are considered (A token size of 30 is considered representative of generic test clone patterns) [138]. This example contains 6733 TLOC (refers to Total Lines of Code measured in physical lines including comments) and 5325 ELOC (refers to Executable Lines of Code excluding comments).

In order to provide an illustrative example on constructing a template based solution we have chosen the Android `BiDiTests` project for the case study as mentioned before. The `BiDiTests` project consists of test cases built using both java codes and XML configuration files. The `BiDiTests` test library consists of 41 java classes and 37 XML configuration files (78 files in total) representing about 53 test cases (file name listing is provided in Appendix C). The analysis of the test library has revealed the presence of 83 test clones of varying sizes and test clone types. The details of the test clones present among java files are shown in Table 6.

Table 6 BiDiTests Test Clone Types Identified

Level	***Types of test clones found***	***Exact Test Clones***	***Renamed Test Clones***	***Parameterized Test Clones***	***Near Miss Test Clones***	***Gapped Test Clones***	***Non Contiguous Test Clones***	***Structural Test Clones***	***Functional Test Clones***	***Design Level Structural TC***	***Total Clones***
Level 1	**Simple Test Clones**										
	Simple Test Clone Codes (STCC)	4	0	0	0	2	0	-	-	-	**6**
Level 2	**Structural Test Clone Fragments**										
	Simple Test Clones Structures (STCS)	3	2	0	0	1	0	-	-	-	**6**
Level 3	Test Method Clone Classes (TMCC)	2	0	0	0	3	0	26	4	0	**37**
Level 4	**Test Method Level Structural Test Clone Fragments**										
	Test Method Clone Structure (TMCS)	0	4	0	0	3	0	3	4	0	**14**
Level 5	Test File Clone Classes (TFCC)		4	0	0	0	0	4	-	0	**8**
Level 6	**Test Method Level Structural Test Clone Fragments**										
	Test File Clone Structures (TFCS)	0	0	0	0	0	0	6	-	0	**6**
Level 7	Directory Test Clone Classes (DTCC)	0	0	0	0	0	0	6	-	0	**6**

An analysis of the java codes reveal that variability in `BiDiTests` test cases are currently being handled using two approaches:

1. Test code customization using object oriented programming language constructs and its type-free generics libraries
2. Test configuration using xml files and related dependency injection constructs.

In spite of such implementation mechanism, we still observe test case redundancies. The following sub-sections discuss a typical examples of test clone types that were observed in `BiDiTests` test library and illustrate how GATT construct can handle these test.

7.5.1. Simple Test Clones

Simple test clones refer to simple redundancies comprising of test code fragments that have similarity such as test method or test fixture declaration methods. In `BiDiTests` project, we focused on test code fragments that either participated in specific roles of bidirectional display orientation testing or participated in eventual test library modifications and evolution. In this sub-section we illustrate a typical example of simple test clones and their equivalent GATT structures.

Consider the following piece of gapped test code fragment extracted from two different java test files namely `BitmapMeshLayerActivity.java` and `BitmapMeshActivity.java` as illustrated in Figure 7-3.

```
public class BitmapMeshLayerActivity
  extends Activity {
    protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      final BitmapMeshView view = new BitmapeshView(this);
      view.setLayerType(View.LAYER_TYPE_HARDWARE, null);
      setContentView(view);
    }
  ...
}
```

Test Clone Fragment 1

```
public class BitmapMeshActivity
  extends Activity {
    protected void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      final BitmapMeshView view = new BitmapeshView(this);
      setContentView(view);
    }
  ...
}
```

Test Clone Fragment 2

Figure 7-3 Gapped Test Clone Example

An analysis of these test clones reveal that the two classes are contiguous segments of redundant test scripts that have intervened code portions that are not parametric. The above example is gapped test clone pair with test codes for the graphic components `View` and `TextView`. The test clone pair is similar in the way they are created and tested with slight variations in properties. The cause for duplication in the above example is the property setting variations of the graphical component under test. This gapped test clone redundancy cannot be handled at current programming language constructs level. Figure 7-4 illustrates the GATT solution.

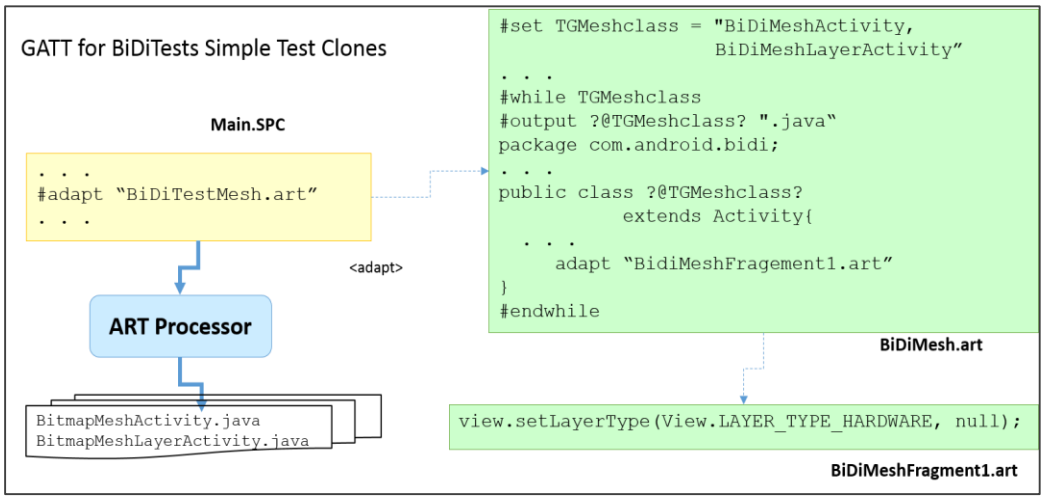


Figure 7-4 GATT Constructs for BiDiTests Simple Test Clones

This example illustrates that text based template fragments can easily adapt gapped lines of test codes.

7.5.2. Structural Test Clones

Structural test clones are higher level clones that represent repeated structures, resulting from a repetition of a high-level design or similar feature. In `BiDiTests` test code fragments of duplicated test cases/suites are seen to be present as part of a bigger replicated test library structure demonstrating the presence of larger granularity similarities i.e., structural test clones. Locating structural test clones in `BiDiTests` can help us to build significant test library understanding, evolution, reuse, and reengineering which is paramount in an ever evolving test library like `BiDiTests`.

Consider an example drawn from `BiDiTests` that depict non-parametric variation in a structural test clone (Figure 7-5). Test files `BiDiTestCanvas.java` and `BiDiTestCanvas2.java` test two different canvas views. The variations between the test clones are being highlighted in bold font in the figure.

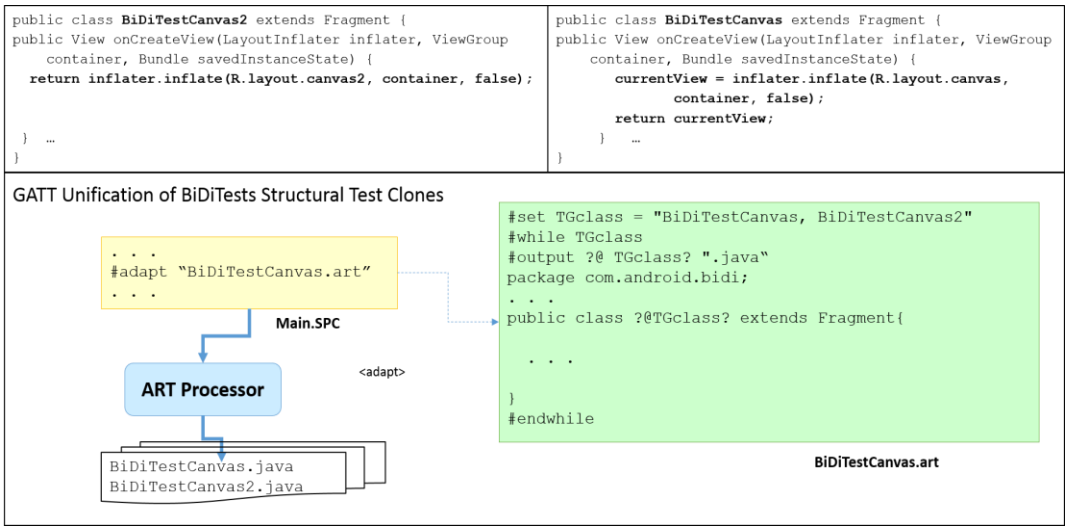


Figure 7-5 GATT Construct for BiDiTests Structural Test Clones

The study of BidiTests reveals the presence of structural clones due to variations occur in variable names, method names, class name or API calls. In addition several structural clones in `BiDiTests` are as also due to the extra or missing test code fragments between similar program structures. Not all of these can be handled through traditional means. Such structural test clones are reconstructed using GATT constructs using `BiDiTestCanvas.art` template files.

7.5.3. Heterogeneous Test Clones

The heterogeneous test clone example discussed here is of non-type variation is expressed partially in code (Java) and partially in configuration (XML). To create and use similar UI entities, different testers create different test data structures. Consider the heterogeneous test clone group (`textview_rtl.xml`, `textview_ltr`, `textview_locale.xml`, `textviewdrawables_rtl.xml`, `BiDiTestView.java`, `BiDiTestViewDrawText.java & textviewdrawables_ltr.xml`) consisting of codes that verifies a `View` graphic item and layout. It is possible that the codes were handled by more than one tester in the `BiDiTests` team and one tester has chosen to define text size as `int` while another tester chose `float`. In addition, we also observe that the test codes differ in the way variables are scoped and memory managed (`final` and `static`). In such non-type based variant situations, java generics are inadequate. Defining a template using the available Java `<T>` syntax for such diverse layout values is not possible. Since such kind of clones cannot be unified using traditional techniques because the variations cannot be expressed as user defined object type or a primitive/Wrapper types, the testers appear to have no option but to create duplicates in the form of clones in this situation.

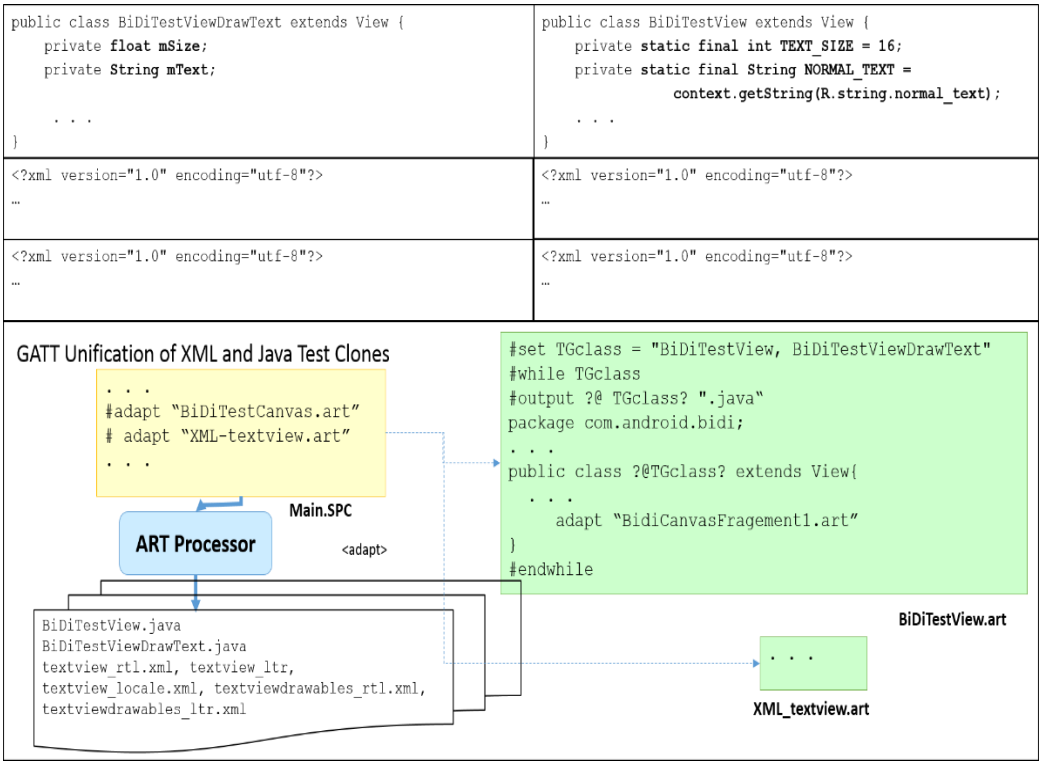


Figure 7-6 GATT Construct for BiDiTests Heterogeneous Test Clones

The GATT solution for this structural test clone example was constructed as shown in Figure 7-6. The figure show relevant skeletal code fragments for the structural clones (`BiDiTextView.art` and `XML_textview.art`).

7.5.4. Other Variations

Although type variations in test code fragments are the ideal targets for reuse using generics, the limitations of conventional template implementations (i.e., limitation in java generic implementation in this case example) usually hinders even in ideal situations. Three such limitations which apply to `BiDiTests` due to its implementation platform (namely Java) are described below.

- For example, generic parameterization using primitive types (`int`, `short`, `long`, `double`, etc.) is not allowed in Java. This is a restriction imposed by Java type system. We can get around this problem by replacing primitive types with corresponding wrapper types (`Integer`, `Short`, `Long`, `Double`, etc.).

- Another tricky situation arises in situations where test cases make invocation to methods that perform similar functions. The code structure and exceptions are thus redundant. As a result, there are numerous `try-catch` block redundancies found in **`BiDiTests`**.

- Test cases are usually run by a test runner class that loads the test class by setting up the required set of data or fixtures, executes the test and finally tears down each test.

In the light of the above stated situations and examples we notice that the traditional testing approaches are inadequate to handle test clone redundancy and variability management in totality. Most of the techniques are restricted by factors such as expressiveness, type management and data structure management of the underlying programming language. This motivates us to apply the STRAT approach on top of existing test libraries for addressing this issue of variability management and reduction of redundancy.

7.5.5. Possible Causes for test clones in BiDiTests

In this section we identify some possible causes that could have led to the presence of test clones in `BiDiTests`. It is difficult to identify all the root causes for test clone occurrences in the Android platform test libraries. There could be myriad of reasons for the root causes ranging from lack of testing skills to presence of bad test smells in the test libraries. Since `BiDiTests` is a smaller base of test classes all root causes for test clones can be thoroughly analysed for the BiDiTests. Such analysis would provide an understanding of the nature of redundancies that needs to be tackled by test templates later on. In `BiDiTests` project context, complexity arises from the fact that testing involves crafting test cases for testing of different combinations of feature variants in relation to bidirectional display orientations. Some reasons are:

- Similarities among various orientations, i.e., left-to-right, right-to-left and layout orientation features result in test clones in the related test cases.
- Similarities among the five types of layouts causes similarity among the test cases created to test these layout's common and varying graphical properties, causing test clones.
- Similarities among the event handling and action lifecycle of graphical components are naturally reflected as test clones in the respective test libraries.
- Similarities among screen layout configuration files are another cause for test clones. This is so because most layout configuration files aspire for consistent look and feel there by exhibiting redundancies.

Most of the similarities in BiDiTests project arise from test feature similarities. Test methods for common features cannot be implemented independently of each other in separate units. Thus variations appear in variant forms addressing feature combinatorial testing. Whenever test library structures cannot be parameterized to unify variant forms, similar test code structures appear as redundancies.

7.6. Construction of Test Templates for BiDiTests

In this case study we have reconstructed the test libraries using the template based approach. The GATT based test library source consists of 30 template hierarchies. In physical terms our solution consists of 1 SPC and 29 ART files (details in Table 7). Further experimental analysis to understand the benefits yielded by the template based solution is based on the above constructed solution.

Table 7 BiDiTests Template Count

Description	*Templates*
Canvas Testing Files	2
Layout Testing Files	8
Gallery Testing Files	3
Text View Testing Files	6

View Testing Files	4
Miscellaneous	7
Total	**30**

This section provides detailed explanation on the process descriptions for template construction. We use the **`BiDiTests`** classes (three subsequent versions) to illustrate the template construction.

7.6.1. Version Sampling

In this case example we initially built three versions of GATT corresponding to three Android Platform OS test library versions (namely versions 16, 17 & 18). The purpose of this expansion to consider subsequent versions (three versions) is to demonstrate that the GATT can be unified into one single template specification and also generate multiple version releases at the same time based on the test designer's binding choices. Test library codes were collected from GIT repository, similar to previous experiment. Subsequent versions make the analysing of change request implementation and identifying of evolution patterns easier. Thus three subsequent versions (API 16, 17 & 18) of BiDiTests project's test cases were analysed for redundancies and were reconstructed into test templates. Only Java based test files and xml based layout configuration files were considered in the scope of study since the selected case example did not directly deal with the Dalvik VM or Linux kernel. Consequently the properties and other IDE dependent files were ignored from the scope of this research experiment. There were no C++ test files in the project. The feature highlights of three versions analysed were:

a) Jelly Bean 4.2 (API Level 16) based on Linux kernel 3.0.31, which is an incremental update with the primary aim of improving the functionality and performance of the user interface.

b) Jelly Bean 4.2 (API Level 17, based on Linux kernel 3.4) - features include "**photosphere**" collection of panorama pictures, UI/ accessibility improvements, and messaging notifications.

c) Jelly Bean 4.2 (API level 18, based on Linux kernel 3.4 GNU v2) – features include notification bar and quick settings, better managed battery life, camera AE/AF lock and high performance graphics via OpenGL ES 3.0.

Domain engineering generality principle encourages avoiding repetitions and construction of parameterized, configurable and adaptable test libraries. GATT serves as a means to create such hierarchical structures in the suggested STRAT approach.

7.6.2. Template Construction Process

In this section we use the proposed STRAT approach to rebuild the existing test files using generic adaptive test templates. The case example solution prototyping was carried out using the following steps:

7.6.2.1. Step 1: Identify Test Clones

The first step is to identify various test clones patterns occurring in the test library. Input for this step is the existing BiDiTests test project that needs improvements. To achieve this, the clone miner and clone analyser tool was installed. BiDiTests projects was checked out from **github** server and further investigated for various test clone patterns. The clone analyser tool is employed for formal detection and cataloguing of noticed similarities. These similarities were also manually asserted for commonality and variations.

7.6.2.2. Step 2: Decide Use of Templates

Second step assesses the feasibility of unifying the identified test clones from BiDiTests using GATT. We categorized clones into generic test clones and

structural test clones. Also test clone groups can be further classified into two: Reducible test clone group and Non-Reducible test clone group. Reducible test clone groups can be re-engineered for reuse using both traditional testing techniques and template approach, while the non-reducible test clone group can only be tackled by the template approach. In the beginning, we focused on simpler test clones as they were more localized and easier to tackle. Then we widened our focus to complex structural test clones and managed to unify those test clones using content specific restructuring strategies. The logical grouping of similarities and decisions for building GATT hierarchy for BiDiTests specific test artefacts are detailed in Section 6.3 of this chapter later.

7.6.2.3. Step 3: Template Construction for Test Library

The third step focuses on designing GATT for logical groups of similarities in BiDiTests project that would preserve commonalities, encompass variability and manage file heterogeneity. This activity consists of selecting appropriate GATT structure, harmonizing clones and template unification finally. The activity starts with a weighted decision which would help recommend if the use of templates over traditional test scripts is appropriate. We then apply generative technique to build GATT structures to unify similarity patterns for which traditional testing techniques fail to provide effective generic solutions. By applying START approach, we turn the test libraries built with traditional testing approaches into a generic test libraries based solution that offers substantial productivity gains in test library construction and maintenance. Finally we unify test clones using a combination of the following techniques:

- Configuring the SPC and ART level variants using set commands and initialized appropriate variant binding values.
- Extracting duplicated test code fragments into template fragments.

- Unifying largely similar functions using conditional branches and iterative control constructs from the GATT building blocks.
- Converting similar java test files and XML layout configurations into equivalent template files (ART files)
- Finally enhancing these by applying more intensively composite template constructs to unify higher level test clone commonalities.

Observations from the previous analysis step were used as guidance to find logical cluster of templates. Then non-redundant template hierarchy (template specifications, template files and template fragments) were constructed into GATT iteratively with refinements carried out at each interaction. This was repeated till all the templates get normalized to become non-redundant. Templates generated from each iteration were further validated and verified for accuracy.

7.6.2.4. Step 4: Product Specific Test Library Derivation

In software product line testing, the test library needs to be executed for every build of an application or product. The test templates derived using STRAT approach were verified for loss-less translation. Additionally **`WinMerge`** (http://winmerge.org/) was used to assist in folder comparison (based on timestamp/content) to cross verify. The test templates derived using STRAT approach were verified for loss-less translation as mentioned in the construction process.

7.6.2.5. Step 5: Template Evolution:

Improvements and evolution of the harvested generic adaptive test templates are the only way we can sustain the quality of the overall test library in the long term. Observations from the previous analysis step were used as guidance for template modifications. The modified change-set for every version release was checked out from github servers (code repositories) based on the list of change requests completed. The identified change requests and modified change-sets were further

analysed to identify the impact on the template hierarchy. The exiting non-redundant template hierarchy is altered with additional specification files that manage every change request as a change-set that includes related modified test files. This was repeated till all the change requests were included in the normalized non-redundant template hierarchy. Templates generated from each iteration were further validated and verified for accuracy.

7.6.3. Non-reducible Test Clone Groups

Using the examples presented above we identified test clone groups for which redundancies cannot be reduced using conventional testing techniques. The purpose of grouping is to unify all the test clones within a group into a single template (GATT). After a detailed analysis of BiDiTests project, the following were identified as the non-reducible groups of test clones with complex variations.

1) `BiDiCanvas[T].java`: refers to the Java test files. Tests that verifies proper display orientation of embedded graphical canvas. T refers to number of canvas instances.

2) `BiDiTest[U]Layout[V].java` refers to the Java test files. Tests target layout graphical entities. U refers to the particular layout under test such as frame, grid, linear, relative and table layouts. V refers to one of the display orientation choices such as Ltr, Rtl and Locale. `[U]_layout_[V].xml` refers to the configuration files for the various layout options.

3) `BiDiTestGallery[W].java` refers to the Java test files. Tests target gallery graphical entities. W refers to one of display orientation choices such as Ltr, Rtl and Images in gallery. `Gallery_[W].xml` refers to the configuration files for the gallery.

4) `BiDiTestTextView[X][Y].java`: Java test files. Tests target test view graphical entities. X refers to one of display directions or UI

component drawn, Y refers to one of display orientation choices such as Ltr or Rtl. `Test_view_[X]_[Y].xml` refers to the configuration files for the various view options. `BiDiTestTextView[V].java`: Java test files. Tests target view graphical entities. V refers to one of display orientation choices such as Ltr, Rtl and Locale. `Text_view_[V].xml` refers to the configuration files for the various text view options. `BiDiTestView[Z].java` Java test files. Tests targeting view graphical entities. Z refers to one of margin and padding settings such as padding, padding mixed, group margin, and draw text. `View_[z].xml` refers to configuration files for the various views.

5) Rest of the test files (both java and configuration) inclusive can be drafted into miscellaneous frames.

The five groups of redundancies described in the list above can be logically clustered into five GATT templates named as Canvas, Layout, Gallery, Views and Text Views, respectively.

7.6.4. The Construction Iterations

In our case study, we have designed and constructed three different versions of `BiDiTests` project implementations (see Figure 7-7). The conversion process was executed in three iterations. The first iteration named $\texttt{BiDiTests}_{\texttt{Simple}}$ was based on a simple design and construction, with little attention to minimizing special redundancies. In the second iteration named $\texttt{BiDiTests}_{\texttt{Optimized}}$, redundancies are unified by understanding test smells, refactoring non-parametric and non-type variants, and also applying suitable testing patterns if needed to the previous $\texttt{BiDiTests}_{\texttt{Simple}}$. Finally in $\texttt{BiDiTests}_{\texttt{Unified}}$ iteration, we unified the three subsequent versions which in our judgment, were worth the effort because maintenance of three versions of test libraries is now made possible

with a smaller set of variant provisioned generic adaptive test templates. The list of test files in API version 17 is clustered based on feature variants.

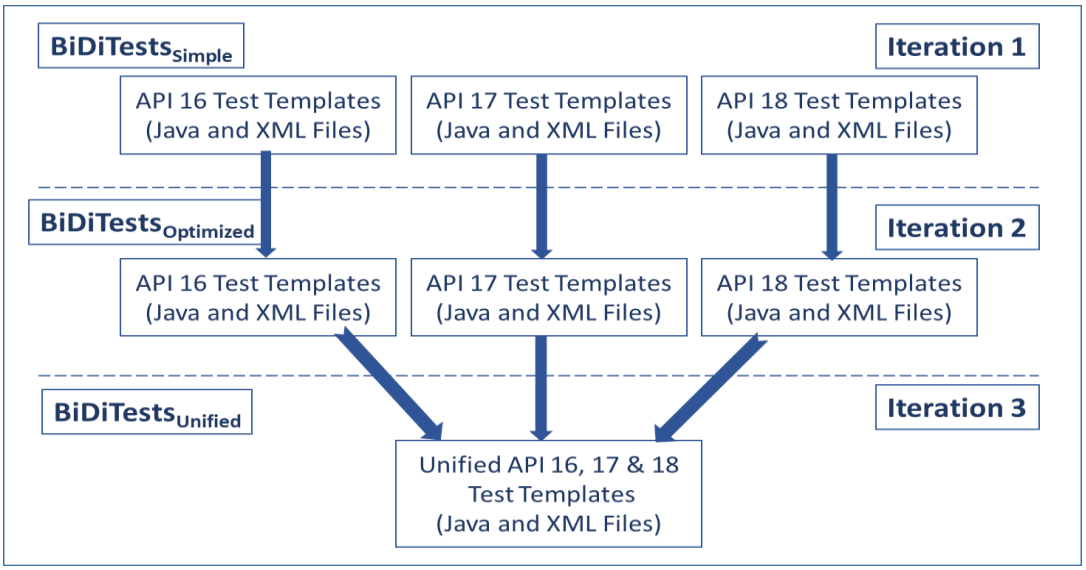


Figure 7-7 Iterative Template Construction

7.6.4.1. BiDiTests Simple

By keeping simplicity and conversion possibility in mind, we focused on converting the system test cases into simple redundant templates when implementing `BiDiTests`$_{Simple}$. The initial version is a draft first-cut solution to test the idea of implementing a meta-layer. Meta layer captures special types of domain choices and product line variants that are not dealt by conventional construction techniques. Emphasis for the iteration was neither redundancy removal nor maintainability concerns as yet. It is a simple proof-of-concept to demonstrate that two-layer management of test libraries is possible. Such separation yields added advantages of capturing test case design information concerning feature combinatory of variants under test in a mobile product line context. A high level template was created for every respective test file (java) found in the project. The iteration verified the correctness, validity and no-loss conversion by comparing the original test case with the ART processor generated codes. Then on, the verification and validation (both manual and tools based) were

repeated at the end of each subsequent iteration to ensure that the qualities as those of the original test libraries are retained.

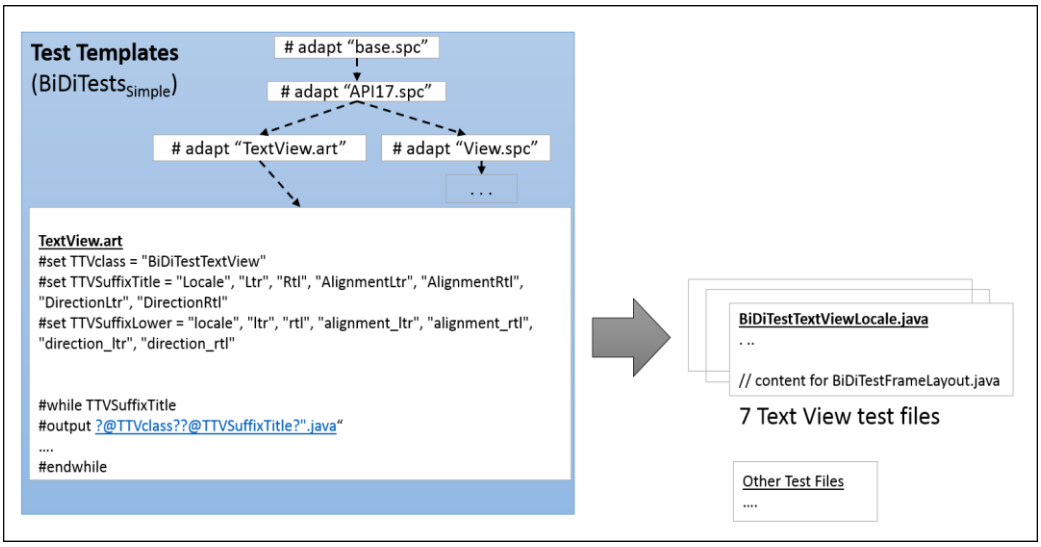


Figure 7-8 BiDiTests (simple) Example

The above (Figure 7-8) example demonstrates the template construction for TextView test clone group. Seven java test files related to TextView were normalized into four frames. Global variables are handled at specification files level (`TextView.spc` and `View.spc`). Template files `TextView.art` and `View.art` manages the java and XML file codes respectively.

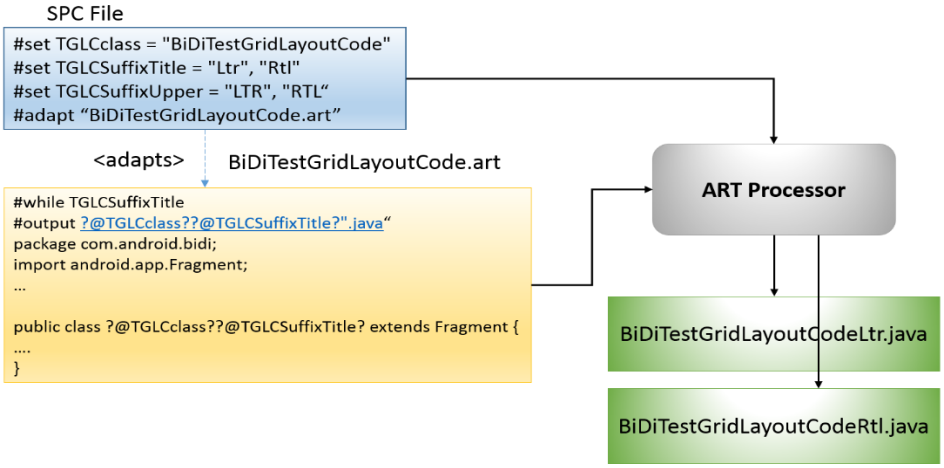


Figure 7-9 Grid Layout Unification (Simple)

Figure 7-9 illustrates unification of test clone files using test templates (GATT) – `BiDiTestGridLayoutCodeLtr.java` &

`BiDiTestGridLayoutCodeRtl.java`. We constructed two files namely `BiDiTestGridLayoutCode.spc` and `BiDiTestGridLayoutCode.art`. SPC files manage variant points class names, title and suffixes. It also adapts the art template using a `#while-#endwhile` construct to generate the original test files.

7.6.4.2. BiDiTests Optimized

While the previous iteration focused on creating template frames, this iteration focuses on more optimized template unification leading to higher reuse. Using the initial similarities derived using the clone miner and visualization tool, we were able to identify feature variations as mentioned in the Section 7.4 BiDiTests class diagrams. In this optimization iteration, we unified a variety of generic and structural clones using the variation among features as guidance. Additionally this iteration identified and unified test clones with test smells by applying test templates. These test clones could not be handled by traditional testing techniques such as generics, modularization using method extraction, object inheritance/interfaces and test case design patterns. We were able to inject feature specific test cases variation into generic modules. Since our central focus is demonstration of generic adaptable template construction for test libraries comprising of special variants, we do not focus on regular software test case design and construction issues such as: test double patterns, test organization, test refactoring, test smells, strategy and fixture/test data patterns. These are managed at scripting language level using test patterns.

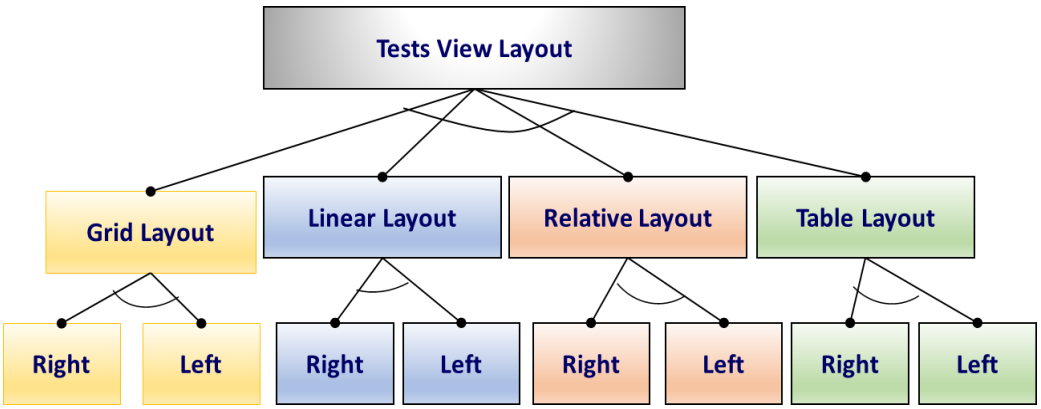


Figure 7-10 Similarity across Layout Test Files

Another interesting example is the templates that were constructed for Layout related test files. As shown in Figure 7-10, there are five layout types and two orientations each. By constructing one specification file named, `BiDiTestsLayout.spc`, the test designer sets the appropriate binding choices for planned variant points regarding layout, prefix-suffix and listing of test classes. For example consider the variant points in the template specification file of Figure 7-11 starting from `ListOfLayout` to `TFLclass`. A generic ART file (`BiDiTestXXXXXLayout`) represents the commonalties among the nineteen tests. Using two concise specification and configuration template definitions we were able to generate of 19 test files after proper variant binding. Figure 7-11 illustrates the scenario.

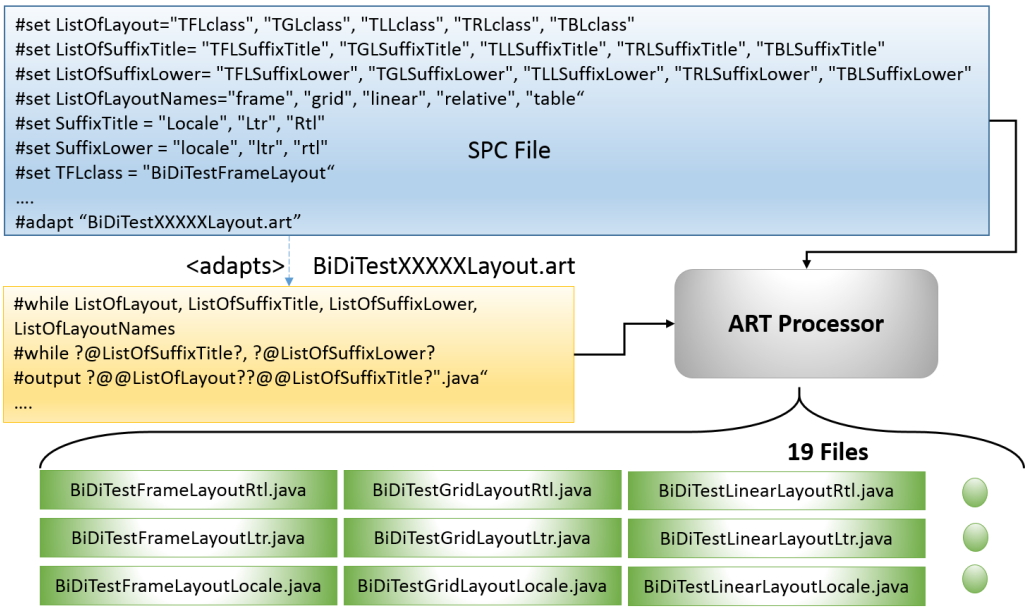


Figure 7-11 Layout Test File generation using GATT

7.6.4.3. BiDiTests Unified

The last iteration focused on changes from evolution across versions and an all-out effort to unify any remaining clones. In this iteration we merged three consecutive versions of the `BiDiTests` projects from consecutive Android platform code named Jelly bean versions with respective OS API Level 16, 17 and 18. After all the intra-module similarity patterns and regular test smells were treated in the previous iterations, evolutionary changes were analysed across versions. Unification of these change requests helped in achieving immediate close to one-third reduction in size. Research evaluation's change propagation subsection provides more details.

7.7. Research Evaluation of GATT

For the purpose of evaluation, we focus only a subset of activities from the general testing process (Figure 7-12) related to test libraries repository creation and evolution. The STRAT approach to create and maintain test libraries is aimed at maximizing productivity, ensuring repeatability in test library maintenance, reduce influence of external factors, preserve test designer's domain expertise and test design choices inside templates.

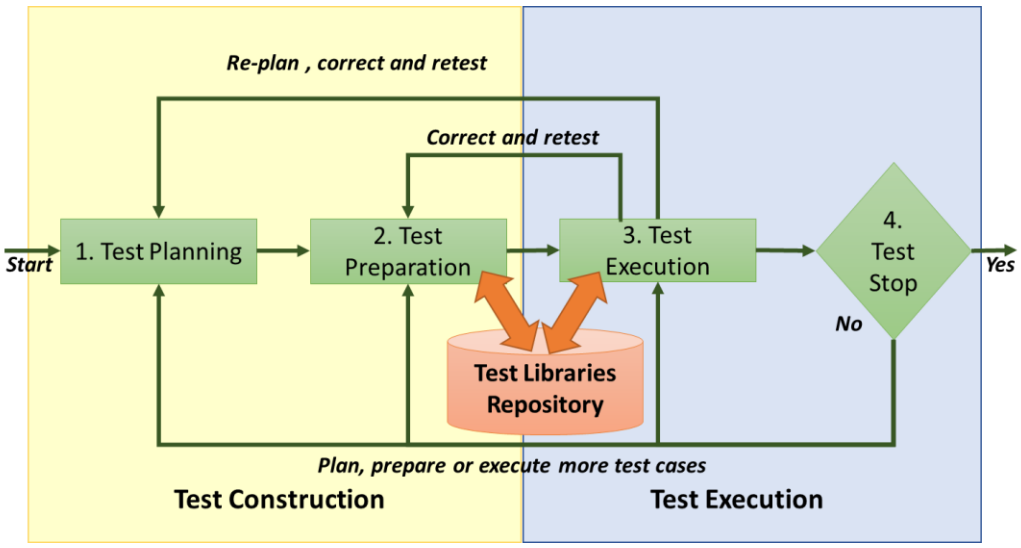


Figure 7-12 Software testing process

In Chapter 5 we described the key metrics that can measure reusability and maintainability of test libraries. These metrics are used in the following subsections to evaluate the productivity gains that the STRAT approach offers in comparison to the original test libraries creation/maintenance. In brief, the key quality measures are:

- Reusability quality factor which is influenced by test library size reduction and ability to express and manage various types of variability.
- Maintainability quality factor which is influenced by number of modifications needed to implement a particular change request and how the templates scales and sustains to the variations in growth of test libraries.

To assess the benefits derived through the template based approach we use a set of metrics in this section. Using these metrics we discuss the reusability improvements achieved through variability management and effective change propagation. We also discuss the non-intrusive nature of GATT layer, benefits of template approach and the threats to validity.

7.7.1. Lossless Translation of Test Libraries to GATT Constructs

One of the objectives of this experiment is to prove loss-less conversion of test clones in test libraries into GATT constructs and vice versa. As a scientific reuse approach, STRAT is capable of translating a variety of test clone types found in this case study into GATT constructs. It is very important and essential to verify the templates created for loss-less translation of test libraries. The verification process was carried out using the following steps:

1) Setting up the code examination tools.
2) Comparing the original and generated test libraries using the identified code examination tools.
3) Using the clone miner tool to analyse the original and generated test libraries for further investigation.
4) Conducting similarity investigation using clone analyser tool. Analyse and affirm the equivalence of test codes.
5) Reporting the outputs and findings that will be useful in answering the questions regarding equivalence of both test libraries.

We used two key techniques for this verification process of comparing the original test library and the GATT generated test library: (1) Code Examination and (2) Clone Detection.

7.7.1.1. Code Examination

Code examination was carried out using a token-based regular expression pattern comparison tool (called **`Total Commander`**) to show that original and generated test libraries are identical. Additionally, we used **`WinMerge`** (http://winmerge.org/) to assist us with folder comparison based on timestamp/content and to cross verify folder comparison we used **`ccfinder`** (http://www.ccfinder.net/). These tools compared the original and generated test libraries based on text content ignoring white space characters.

7.7.1.2. Clone Detection

We ran the **`Clone Miner (CM) and Clone Analyser (CA)`** tool for affirming that the test clones originally found in the test libraries are generated without loss by the **`ART processor`**. We ran the clone detection on the generated test library to show exactly same number of clones are found - so that the GATT actually recreates the test library in its original form. **`CM/CA`** found exact equivalence between all types and occurrences of test clones in both the original and generated `BiDiTests` test libraries. Being a token-based technique clone miner also verified similarities between simple test clones and structural test clones between the original and the generated test libraries.

The above verification process asserts that the GATT ensures loss-less translation. This also confirms that GATT based test libraries are accurate so that the further experimental analysis and inferences would be accurate.

7.7.2. Improving Productivity by Reuse

Current mechanisms of Test Library creation already provide scope for achieving reuse (e.g., reuse via test method, fixture, data, state management, and event management with test patterns, support using Activity and Instrumentation). In this case example, we applied GATT method over and above the conventional methods with a view to unifying the test clones in situations where the conventional techniques fail to provide effective reuse solutions. Our experiment shows that more than 70% of the test files exhibited some form of redundancy which could be eliminated at a meta-level. This elimination comes with additional benefits of better maintainability and reduction in cognitive complexity of the test libraries. In the BiDiTests project, the original test libraries amounted to ≈18EKLOC (executable kilo lines of code as test projects) forming three subsequent API versions. We were able to reconstruct these test libraries without any loss of quality as a non-redundant, variant preserving ≈4KLOC of generic test case templates. Effectively the process has eliminated more than 77% of the

redundancy among the test project codes. From original code base of 218 files (both Java and XML inclusive), the generic templates where compacted into 30 template files. Summary of the template iterations are shown in the Table 8 below.

Table 8 BiDiTests Project Consecutive Three Version Statistics

BiDiTests Project Consecutive Three Version Statistics			
Description	API16	API 17	API 18
# of Java Test Files	34	41	39
# of Java Test Methods	61	72	59
# of XML Configuration Files	31	37	36
Total Files	65	78	75
# Files Containing Redundancies	46	59	56
% Files Containing Redundancies	71%	76%	74%
File Size (Kbytes)	416	528	512
# Lines Of Code	6556	8422	8102
# of Executable Lines Of Code	4877	6393	6191
Total Executable Lines Of Code			17461
# ART Templates	29	29	29
Each Versions (Executable LOC)	3924	4024	4063
ART compared to Original ELOC (%)	80%	63%	66%
Merged Frames (Executable LOC)			4063 (30 template Files)
ART compared to Original ELOC (%)			23%

Table 9 logically groups templates and quantitatively describes how redundant test clones were compressed into GATT structures.

Table 9 BiDiTests Unification Metrics

BiDiTests Project Unification Metrics		
Description	# of Files	BiDi (Optimized)
Canvas Testing Files	4	2
Layout Testing Files	37	8
Gallery Testing Files	5	3
Text View Testing Files	18	6
View Testing Files	8	4
Miscellaneous + SPC	6	6
Total	78	29

7.7.3. Change Propagation

Firstly, there is a significant drop in the size of test code to be maintained from original test libraries to $BiDiTests_{Unified}$. In the $BiDiTests_{Optimized,}$ there is a 20%, 37% and 34% reduction in test script (executable lines of codes) for API kernel version 16, 17 and 18 respectively. The overall system has dropped by much more (by 77%) largely due to unification of three API kernel versions to one.

Secondly, test templates reduce the risk of update anomalies. To study this, we considered three change requests, namely Grid Change, Text View Change and Drop Extra Canvas. Table 10 below shows the distribution of the impact of the three evolutionary change requests which was available in the original github source code repository.

We carried out a controlled experiment to verify the hypothesis that test templates improve productivity and reduce update anomalies. We have collected complete details of change request and updates done on the three versions and recreated the same scenarios using test templates based test libraries. From kernel API version 16 to 17 seven files were added to satisfy change requests CR1 & CR2 shown in Table 10. Likewise, from versions API 17 to 18 two files were deleted to satisfy the change request #3.

Table 10 Change Request List

Grid Change (CR1)	Update BiDiTests app for adding 'Grid Layout' unit tests
Text View Change (CR2)	Add tests to view text alignment
Drop Extra Canvas (CR3)	Clean up code for Test View, 'Canvas Layout' and related code flags

Table 11 presents the comparison of change prorogation between original test library and concise template representations. #F denotes the number of files affected by the change request while #L denotes the number of modified locations. The number of files affected was further sub-classified as newly added files (#A), Deleted (removed) files (#D) and modified files (#M) as observed from

the distribution of impact recorded in Table 11. The effort and changes that need to be performed in order to implement these change requests are observed to have decreased in terms of modification counts from $BiDiTests_{Simple}$ test libraries to $BiDiTests_{Unified}$. Thus as testers navigate to non-redundant representation of test libraries using test templates, the chances for inconsistency reduces during updates and improves productivity since change propagation is made more systemic via reuse.

Table 11 Comparison of change propagation

CR	#F	#L	Simplified			Optimized			Unified		
			#A	#D	#M	#A	#D	#M	#A	#D	#M
CR1	8	9	7	0	1	2	0	1	1	0	1
CR2	5	5	4	0	1	2	0	1	2	0	1
CR3	4	4	0	3	1	0	2	1	0	2	1

There are many ways to design **`BiDiTests`** that are different from our change propagation strategy shown Table 11. Test templates are observed to easily capture test case design, domain analysis and test maintenance decisions inside the test template specification files. Some additional benefits achieved are feature under test to testing capability mapping, rapid evolution of test library across versions and ability to construct test libraries by binding variants of tester's choice. Releasing platforms and apps faster provide competitive advantage to smart phone companies as it makes them agile to market needs.

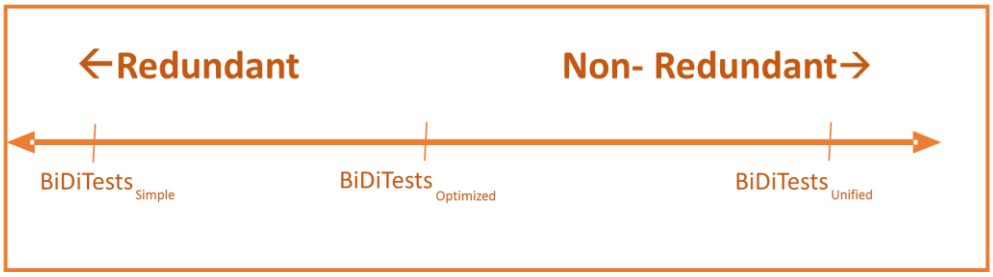


Figure 7-13 Improvement towards non-redundancy with iterations

From Figure 7-13 it is can be seen that non-redundant representation of test templates improves from being a more repetitive representation to conciseness of

test templates as the iterations progressed from simple to unified iteration. There are additional test design comprehension, indirection and template layering as we move from simple to unified representations of BiDiTests test library. This would require more planning, analysis and modeling. Despite the drop in test library size (calculated in ELOC), the construction efforts seem to increase. However as the product line starts to expand the productivity gains may outweigh the construction complexity. Also, as seen from the above control experiment test templates solution complements the conventional test libraries construction approach with modest additional foot prints.

7.7.4. Scalability

GATT is scalable and can handle large test libraries and heterogeneous test assets. To be scalable, test libraries must offer more primitive building blocks and be accompanied by generators that can compose these blocks to yield the required data structures used by programmers. We believe that our GATT generative approach is required in order to address the needs of scalable test libraries by offering primitive building blocks (GATT constructs) accompanied by compile time generators (ART Processor). Even though our empirical studies of Android domain's `BiDiTests` were of a small scale, they clearly demonstrated that the idea of GATT method was feasible and have engineering merits in adopting the generative approach. GATT provides a generic representation for both data-structures and algorithms in its meta-layer. GATT also addresses concept of vertical parameters (i.e. layered components) which is an essential ingredient for scalable Testware. The GATT related configuration constructs work as pre-generators and offers support to building of test case library in a hierarchical fashion at the level of meta-layer, thereby guaranteeing scalability. Preliminary experimental evidence presented in the BiDiTests library shows that GATT prototype does not compromise productivity and performance of the generated test cases.

7.7.5. Non-Intrusiveness

Testers work with test case code wrapped in GATT definitions. ART Processor expands directives to generate the actual test cases from templates in the similar way that the C pre-processor generates code by expanding cpp directives. When the templates are instantiated, the implementation details of the test case are resolved by binding values to the parameters of the template at compile time. For example, for a template representing a group of similar test files, ART Processor generates code for those files based on specifications of delta differences between the template and each of those files. GATT uniformly manages variability in various test library assets such as test code, test configurations and test data. In Software Product Line (SPL) context, GATT streamlines and automates customization of reusable components, improving productivity gains due to reuse. GATT's design or re-configurable and adaptable test case design is well contained within the meta-layer of test construction in the Android product line and thus is non-intrusive with the executable test libraries which the testers work with.

7.7.6. Other Benefits and Trade-offs

The reduction of test libraries size by 77% is consistent with measurements/feedback that both effort and difficulty in maintaining the original test library were significantly more than what is now required for the multiple versions of the test libraries. Many ART concepts, such as parameterization, selection and iteration are similar to programming language concepts. Even though there are specifics of ART that must be understood, it was found relatively easy to start template construction with ART as compared to conventional programming languages or other environments where testers in addition to the language must be familiar with additional support APIs such as mock and other utility libraries. Our reverse engineered extractive approach proved to be an effective way to build the initial template versions, and further reactive approach

helped us gradually refine templates as we addressed new types of test clones. The advantage of such an approach, as compared to proactive approach, is low effort investment and faster in yielding results.

GATT structures organize design and code at the meta-level for enhanced generality and changeability using ART. As a result, testers must additionally manage the ART constructs using SPC, and x-frames (ART, Java, C++ and XML). This additional complexity may have an impact on comprehension. One more trade-off is that currently there is no tool support for debugging. Proper tool support can help testers' better cope with such comprehension issues. Future studies should focus on how to apply GATT in larger testing projects, using full-fledged software engineering processes.

7.7.7. Threats to validity

The above case study discussed steps, merits and limitations from the experience of re-constructing test templates using GATT approach for BiDiTests project. There are a few threats to validity such as the choice of BiDiTests project over others in repository, the nature interpretation of test clones as found by the tool, and use of text token based tools for comparison. Our work does not claim to have generalized all possible test clone occurrences. There could be more causes for the test clones than those being listed in the analysis section. Our estimates on efforts/change metrics for change requests are based on GIT repositories entries. It would strengthen our case if this experiment was extended to measure the actual effort to maintain test templates versus original test cases. While we believe that GATT approach can be smoothly accommodated into existing software testing processes, this needs certainly be verified in industrial project settings with a particular Android vendor like Samsung or Nexus.

7.8. Adapting Test Templates to other similar situations

This research case study shows that it is technically feasible to implement generic adaptive test templates to unify test clones occurring in BiDiTests project, without compromising any characteristics of the original test case library.

This STRAT approach of variability management can be further explored using typical open source apps, device-specific variations and vendor specific variations. Android is only one of technologies used for smart phone platforms and apps. It is possible to relate our observations to alternative technologies such as iOS or Windows Mobile. Object C™ and .NET™ are two similar advanced platforms for implementing mobile apps. They also provide rich sets of API programming and device facilities (e.g. for tablets, phones, sensors and packaging). Therefore, ART implemented on the .NET or Object C is likely to follow the same high-level architecture proposed in this thesis.

7.9. Key Takeaways & Inferences

Due to paucity of code-based reuse approaches observed in literature, this research attempts to provide a solution, to improve the maintainability of large scale test libraries using generic adaptive test templates. Traditional test maintenance approaches do not capture the domain relationship between program code and its relevant test libraries. In traditional approaches, such domain relationships (if any) are implicitly inferred and not explicitly implemented. To address this lacuna we have created generic adaptive test templates. The proposed STRAT approach is not constrained by the programming language of the product or its configuration set up.

To demonstrate above mentioned research intentions, we implemented BiDiTests test library to illustrate the GATT construction and evolution process. This illustrative example confirms both the purposes defined earlier.

1) Demonstrate the STRAT approach using an illustrative example and develop generic adaptive test templates.

2) The case study clearly demonstrated that compressed, normalised, non-redundant test library can be achieved using STRAT approach without any loss of quality and at the same time preserving all product line variations. The results show that test library achieved a compression of 23% to its original size.

3) Use those developed templates to generate test library and compare with the original for the purpose of establishing the benefits derived using STRAT approach

4) Productivity improvements have been assessed in terms of reusability (expressed through reduction in executable lines of codes and ability to express many types of variability) and effort reduction in maintainability (expressed through reduced number of modifications required to implement a particular change request and the ability of template hierarchy to scale along with the growth of underlying test libraries).

Thus we believe that the proposed STRAT approach can be particularly useful in mobile and service computing, since multiple versions of apps and multiple service clients' leads to the explosion of test libraries in such environments. Based on the results achieved, we feel positive about the productivity increase the STRAT approach has to offer in SPLT.

CHAPTER 8
Conclusions

During the initial phase of our research we identified and studied the redundancies that occur within and across large scale test libraries in open source software. Redundancies often occur because of the copy-paste-modify approach used by testers when creating test cases for similar products within a software product line. We provided concrete evidence for the presence of test clones among large scale test libraries by cataloguing various types, granularity and occurrence examples. We also explained that the presence of test clones would increase effort and complexity during test library creation, maintenance and evolution. The complexity is caused due to difficulties in performing multiple concurrent modifications, variability management and combinatorial explosion of test cases.

In the next phase, we conducted a systematic review of SPLT research literature and this literature survey reveals the following:

- Existing test model based approaches reported in literature predominantly work with one of the standard modelling representation, namely, feature models, UML or OCL. These techniques aid in model based reuse, test case selection for execution and variability management using stereotypes/model checkers. These test libraries are generally found to be non-executable test model artefacts and studies with regard to executable test libraries are not reported.

- Existing formal specification and natural language based approaches provide sound mathematical models for verification and validation of test representation. They provide abstract representations (formal notations or natural language) for achieving reusability, managing variability and controlling test case explosion.

- Aspect oriented approaches are limited by the capabilities of the hosting container and underlying programming language expressiveness.

- Key problems in dividing responsibilities between domain testing and application testing have been reported [146]. Surveys highlight that complete integration and system testing in domain engineering is not feasible. They also infer that it is hard to decide how much we can depend on domain testing in the application testing.

The above findings of the literature survey revealed a gap in existing research that tackles test clone redundancy and variability management motivating further study in this thesis.

A thorough search of existing literature indicates an absence of clear definitions for test clones and also affirms the lack of mapping between general code clone terms and test libraries. Hence, in this research thesis, we have formalized definitions for simple and structural test clones in the initial phase of the research towards building a generic solution. Further we built a simple taxonomy and formulated metrics to scientifically measure the influence of test clones on reusability and maintainability aspects.

One way to tackle test clones is to represent them in some generic form. Traditional generic programming techniques such as parameterization, aspect orientation and test design patterns can alleviate issues surrounding test clones. For situations where traditional approaches fail, it is worthwhile to look for new research solutions that address existing shortcomings. Based on experiences gathered above, we understood that test clones can be well managed if reuse is planned early in product line engineering.

Thus in the next phase, we addressed the test clone challenge using a reuse-based approach towards building non-redundant generative test libraries. We propose a template based approach as a solution. For this we formulated *Generic Adaptive*

Test Template (GATT) which is an unrestricted, non-intrusive layer of templates that can be added to the existing base of test libraries. To formalise the steps and activities involved in adopting the template we came up with an approach which we have named *Systemic Template based Reuse Approach for Large Scale Test Libraries (STRAT).*

In the final phase, we illustrated the proposal using a SPL case study. The case study demonstrates that clear, compressed, normalised, non-redundant test library can be generated using our STRAT without any loss of quality and at the same time provisioning for variability. The results of the case study implementation were analysed qualitatively and quantitatively and the proposed template based approach is found to demonstrate measurable productivity improvements in terms of test library maintainability by effort reduction and improved reusability by unification of test clones.

8.1. Contributions

The following summarizes the key contributions and outcomes of our research work:

- *Study of Redundancies*: The thesis systematically documents the empirical study conducted on the large scale test libraries to establish significant presence of test clones and analyse these test clones in the light of test maintenance and rapid evolution conditions. Based on this analysis, we came up with the following key findings:
    - Test libraries are found to have various forms of test clones. In general, around half the test cases exhibited some form of test clone similarities. Majority of the simple test clones had either ‘exact text replication’ or ‘identifiable parametric variations’. Further the study reveals significant diversity in structural test clones; i.e., they are present in various proportions and granularity.

- The study also inferred possible causes for test clones – these range from simple causes such as copy-paste-modifications to other complex reasons such as lack of expressiveness of underlying programming language, test assets spanning multiple file formats, limitations of underlying programming language, test case design similarities and use of wizards test case generation tools.

- The study reveals that event based test case (namely, activity, service and other related event API's as discussed in section 4.6.1.5 to 7)had the most repetitive structures. This was due to the fact that event setup and lifecycle management methods have common structures among similar components (Examples: UI components such as radio buttons, text boxes and drop down lists). Test cases also exhibit redundant patterns with reference to the way in which test data gets initialized or garbage gets collected.

- The study concludes that though certain types of the identified test clones can be treated with existing software testing research and industry practices, certain complicated similarity patterns cannot be treated using current language level generative techniques.

- *New reuse-based test template approach*: Finally, the thesis proposed a reuse-based approach for developing and managing test libraries in a product line context. This template approach addresses a few of the previously mentioned shortcomings by targeting to design and build a non-redundant template representation for existing test libraries.

    - The template structure forms the heart of the proposed solution and this thesis has devised GATT structure such that the templates preserve and explicitly configure the program-code to test-case relationships, which are otherwise non-comprehensible and

implicit in traditional testing approaches. Key contribution that GATT offers:

- GATT structures have been designed to meet key requirements such as non-redundant representations, text based processing, variant provisioning and ease of test library derivation.
- GATT ensures managing heterogeneity among test file formats and GATT semantics provisions for variability management.

- GATT facilitates in retention of domain expert's knowledge of variant points.To facilitate test designers to create GATT in a methodical way, this thesis has formulated *Systemic Template based Reuse Approach for Large Scale Test Libraries* comprising of guidelines for test clone identification, deciding on usage of templates, template construction by test clone unification schemes and test library derivation. This helps practitioners to adopt our proposed template based approach in a systematic way.

- *Case Study - validation and experimental evaluation of proposed solution*: To illustrate the STRAT approach we implemented a typical test library using the proposed *Generic Adaptive Test Templates*. Key contributions and findings include:
  - Demonstrating the feasibility of constructing test templates that are concise, non-redundant and normalized.
  - Demonstrating that template based libraries exhibit higher compression rates achieved in terms of executable lines of code.

- Demonstrating that the use of GATT helps in increasing productivity measured across various metrics such as executable line of codes, depth of the template tree, etc.
- Demonstrating that the use of GATT improves the maintainability as established by the number of modification metric needed to implement a change request.

8.2. Future Extensions

The following list suggests future extensions of our research work:

- The culture of the underlying domain will directly influence the product line testing priorities of individual features and ultimately the levels of test coverage. For example, a medical device that integrates hardware and software requires far more evidence of the absence of defects than the latest video game. In order to formalize such influences, we need to collect more empirical evidence on specific domains to better access the trade-offs involved. STRAT approach currently does not formalize or mandate mechanisms to capture domain or application testing design decisions into the constructed test templates. This area of research could be extended in the future.
- Currently our proposed approach focuses only on test libraries with executable test cases. Future extensions on STRAT approach can study the influence of test clones and efficiency of template approach in test libraries with other similar test artefacts such as test models, test plans, test documentation and test reports.
- A drawback of the proposed template based approach is the effort required to achieve the appropriate level of abstraction. Another possible extension for this research work would be to implement a simplified tool kit that integrates clone mining tools and test related tools with a new GATT

editor workbench inside popular IDEs (Integrated Development Environment) to further improve test designer productivity.

- The proposed research approach uses compile-time variability binding technique and hence the generated test libraries can be statically verified for correctness. Since template approach works at text level, it can be used with any technologies even without concept of variation. The generator replaces the variability in a template with information from the specification. This approach can be extended to run time variability with necessary alterations in the run-time container and generator using inputs from the program-dependency-graphs of executable test libraries. Thus the research study can be extended to incorporate dynamic-run time variability mechanism in product line testing using test execution traces as guidance.

8.3. Closing Remarks

Based on this research, we conclude that while some of the straight-forward redundancies caused by test clones can be rectified by traditional approaches, redundancies of complex nature commonly occurring in large scale product line environment would need a complementary generic adaptive test template approach to supports unrestricted parameterization, variability management and heterogeneous test artefacts. While this thesis has demonstrated the benefits of the GATT approach through Android platform test libraries as case example, the principles of operation using generic adaptive test templates are same for any test library that contain redundancies. Thus the contributions made by this thesis is expected to benefit SPLT community for improving testing productivity as well as researchers to gain greater insights into using generic approaches for test library construction and maintenance.

Bibliography

[1] "Android Fragmentation Report July 2013 %U http://opensignal.com/reports/fragmentation-2013."

[2] Ali, S., Briand, L. C., Hemmati, H., and Panesar-Walawege, R. K., "A systematic review of the application and empirical investigation of search-based test case generation," *Software Engineering, IEEE Transactions on*, vol. 36, no. 6, pp. 742-762, 2010.

[3] Amalfitano, D., Fasolino, A. R., and Tramontana, P., "A gui crawling-based technique for android mobile application testing," *Software Testing, Verification and Validation Workshops (ICSTW), 2011 IEEE Fourth International Conference on*, IEEE, 2011, pp. 252-261.

[4] Amalfitano, D., Fasolino, A. R., Tramontana, P., De Carmine, S., and Memon, A. M., "Using GUI ripping for automated testing of Android applications," *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, ACM, 2012, pp. 258-261

[5] Asaithambi, S. P. R., and Jarzabek, S., "Generic adaptable test cases for software product line testing: software product line," *Proceedings of the 3rd annual conference on Systems, programming, and applications: software for humanity*, ACM, 2012, pp. 33-36

[6] Asaithambi, S. P. R., and Jarzabek, S., "Towards Test Case Reuse: A Study of Redundancies in Android Platform Test Libraries," *Safe and Secure Software Reuse*, pp. 49-64 Springer, 2013.

[7] Bashardoust-Tajali, S., and Corriveau, J. P., "On extracting tests from a testable model in the context of domain engineering," *Engineering of Complex Computer Systems, 2008. ICECCS 2008. 13th IEEE International Conference on*, IEEE, 2008, pp. 98-107

[8] Basili, V. R., Caldiera, G., and Rombach, H. D., "Experience factory," *Encyclopedia of software engineering*, 1994.

[9] Basit, H. A., Ali, U., Haque, S., and Jarzabek, S., "Things structural clones tell that simple clones don't," *Software Maintenance (ICSM), 2012 28th IEEE International Conference on*, IEEE, 2012, pp. 275-284

[10] Basit, H. A., and Jarzabek, S., "A case for structural clones," *Proc. Int. Workshop on Software Clones (IWSC 09)*, 2009.

[11] Basit, H. A., and Jarzabek, S., "A data mining approach for detecting higher-level clones in software," *Software Engineering, IEEE Transactions on*, vol. 35, no. 4, pp. 497-514, 2009.

[12] Basit, H. A., and Jarzabek, S., "Detecting higher-level similarity patterns in programs," *ACM SIGSOFT Software Engineering Notes*, ACM, 2005, pp. 156-165

[13] Basit, H. A., and Jarzabek, S., "Efficient token based clone detection with flexible tokenization," *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, ACM, 2007, pp. 513-516

[14] Basit, H. A., Rajapakse, D. C., and Jarzabek, S., "Beyond Generics: Meta-Level Parameterization For Effective Generic Programming," *benefits*, vol. 3, pp. 17, 2005.

[15] Basit, H. A., Rajapakse, D. C., and Jarzabek, S., "Beyond templates: a study of clones in the STL and some general implications," *Proceedings of the 27th international conference on Software engineering*, ACM, 2005, pp. 451-459

[16] Basit, H. A., Rajapakse, D. C., and Jarzabek, S., "An Empirical Study on Limits of Clone Unification Using Generics," *SEKE*, 2005, pp. 109-114.

[17] Beizer, B., *Software testing techniques*: Dreamtech Press 2003.

[18] Bertolino, A., "Software testing research: Achievements, challenges, dreams," *2007 Future of Software Engineering*, IEEE Computer Society, 2007, pp. 85-103

[19] Bertolino, A., Fantechi, A., Gnesi, S., and Lami, G., "11 Product Line Use Cases: Scenario-Based Specification and Testing of Requirements," vol. Software Product Lines, pp. 425-445, 2006.

[20] Bertolino, A., Fantechi, A., Gnesi, S., and Lami, G., "Product line use cases: Scenario-based specification and testing of requirements," *Software Product Lines*, pp. 425-445 Springer, 2006.

[21] Bertolino, A., and Gnesi, S., "Pluto: A test methodology for product families," *Software Product-Family Engineering*, pp. 181-197 Springer, 2004.

[22] Bertolino, A., and Gnesi, S., "Use case-based testing of product lines," *ACM SIGSOFT Software Engineering Notes*, vol. 28, no. 5, pp. 355-358, 2003.

[23] Beydeda, S., and Gruhn, V., "Test case generation according to the binary search strategy," *Computer and Information Sciences-ISCIS 2003*, pp. 1000-1007 Springer, 2003.

[24] Beydeda, S., and Gruhn, V., "Test data generation based on binary search for class-level testing," *Book of Abstracts: ACS/IEEE International Conference on Computer Systems and Applications*, 2003, p. 129.

[25] Bosch, J., "From Software Product Lines to Software Ecosystems," *Proceedings of the 13th International Software Product Line Conference*, SPLC '09, Carnegie Mellon University, 2009, pp. 111–119.

[26] Burnette, E., *Hello, Android: introducing Google's mobile development platform*: Pragmatic Bookshelf %@ 1934356492, 2009.

[27] Cabral, I., Cohen, M. B., and Rothermel, G., "Improving the testing and testability of software product lines," *Software Product Lines: Going Beyond*, pp. 241–255: Springer, 2010.

[28] Calvagna, A., Gargantini, A., and Vavassori, P., "Combinatorial interaction testing with CitLab," *Software Testing, Verification and Validation (ICST), 2013 IEEE Sixth International Conference on*, IEEE, 2013, pp. 376-382

[29] Carmichael, L., Damarla, T., McHugh, P., and Chung, M. J., "Issues involved in reuse library for design for test," *AUTOTESTCON'95. Systems Readiness: Test Technology for the 21st Century. Conference Record*, IEEE, 1995, pp. 84-93

[30] Cavarra, A., "Data flow analysis and testing of abstract state machines," *Abstract State Machines, B and Z*, pp. 85-97 Springer, 2008.

[31] Cavarra, A., Crichton, C., Davies, J., Hartman, A., Jeron, T., and Mounier, L., "Using UML for automatic test generation," *Proceedings of ISSTA*, 2002.

[32] Cavarra, A., Davies, J., Jeron, T., Mournier, L., Hartman, A., and Olvovsky, S., "Using UML for automatic test generation," *Proceedings of ISSTA*, 2002.

[33] Chen, T. Y., Tse, T. H., and Yu, Y.-T., "Proportional sampling strategy: a compendium and some insights," *Journal of Systems and Software*, vol. 58, no. 1, pp. 65-81, 2001.

[34] Clause, J., and Orso, A., "A technique for enabling and supporting debugging of field failures," *Proceedings of the 29th international conference on Software Engineering*, IEEE Computer Society, 2007, pp. 261-270

[35] Clements, P., and Northrop, L., *Software product lines: practices and patterns*: Addison-Wesley Reading, 2002.

[36] Cohen, D. M., Dalal, S. R., Fredman, M. L., and Patton, G. C., "The AETG system: An approach to testing based on combinatorial design," *Software Engineering, IEEE Transactions on*, vol. 23, no. 7, pp. 437-444, 1997.

[37] Cohen, D. M., and Fredman, M. L., "New techniques for designing qualitatively independent systems," *Journal of Combinatorial Designs*, vol. 6, no. 6, pp. 411-416, 1998.

[38] Cohen, M. B., Dwyer, M. B., and Shi, J., "Coverage and adequacy in software product line testing," *Proceedings of the ISSTA 2006 workshop on Role of software architecture for testing and analysis*, ACM, 2006, pp. 53-63

[39] Condron, C., "A domain approach to test automation of product lines," *International Workshop on Software Product Line Testing*, Citeseer, 2004, p. 27.

[40] Crichton, C., Cavarra, A., and Davies, J., "Using UML for automatic test generation," *Proc. of the Intern. Conf. On Automated Software Engineering, ASE*, 2001.

[41] da Mota Silveira Neto, P. A., Carmo Machado, I. d., McGregor, J. D., De Almeida, E. S., and de Lemos Meira, S. R., "A systematic mapping study of software product lines testing," *Information and Software Technology*, vol. 53, no. 5, pp. 407-423, 2011.

[42] Dallal, J. A., and Sorenson, P., "Testing software assets of framework-based product families during application engineering stage," *Journal of Software*, vol. 3, no. 5, pp. 11-25, 2008.

[43] Davis, T., "The reuse capability model: a basis for improving an organization's reuse capability,", *Selected Papers from the Second International Workshop on Software Reusability, 1993. Proceedings Advances in Software Reuse*, 1993, pp. 126-133.

[44] de Souza Filho, E. D., de Oliveira Cavalcanti, R., Neiva, D. F. S., Oliveira, T. H. B., Lisboa, L. B., de Almeida, E. S., and de Lemos Meira, S. R., "Evaluating domain design approaches using systematic review," *Software Architecture*, pp. 50-65 Springer, 2008.

[45] Desurvire, H., Caplan, M., and Toth, J. A., "Using heuristics to evaluate the playability of games," *CHI'04 extended abstracts on Human factors in computing systems*, ACM, 2004, pp. 1509-1512

[46] Dias-Neto, A. C., and Travassos, G. H., "A picture from the model-based testing area: concepts, techniques, and challenges," *Advances in Computers*, vol. 80, pp. 45-120, 2010.

[47] Dowie, U., Gellner, N., Hanssen, S., Helferich, A., Helferich, A., and Schockert, S., "Quality assurance of integrated business software: an approach to testing software product lines," *ECIS 2005 Proceedings*, pp. 152, 2005.

[48] Dueñas, J. C., Mellado, J., Cerón, R., Arciniegas, J. L., Ruiz, J. L., and Capilla, R., "Model driven testing in product family context," *First European Workshop on Model Driven Architecture with Emphasis on Industrial Application*, Citeseer, 2004, pp. 91-96.

[49] Edwin, O. O., “Testing in software product lines,” *Master's thesis, Department of Software Engineering and Computer Science, Blekinge Institute of Technology, Sweden*, 2007.

[50] Ehringer, D., “The dalvik virtual machine architecture,” *Techn. report (March 2010)*, 2010.

[51] Engström, E., and Runeson, P., “Software product line testing–a systematic mapping study,” *Information and Software Technology*, vol. 53, no. 1, pp. 2-13, 2011.

[52] Ensan, F., Bagheri, E., and Gašević, D., "Evolutionary search-based test generation for software product line feature models," *Advanced Information Systems Engineering*, Springer, 2012, pp. 613-628.

[53] Feng, Y., Liu, X., and Kerridge, J., "A product line based aspect-oriented generative unit testing approach to building quality components," *Computer Software and Applications Conference, 2007. COMPSAC 2007. 31st Annual International*, IEEE, 2007, pp. 403-408

[54] Frakes, W., and Terry, C., “Software Reuse: Metrics and Models,” *ACM Comput. Surv.*, vol. 28, no. 2, pp. 415–435, 1996.

[55] Fraser, G., and Zeller, A., "Exploiting Common Object Usage in Test Case Generation," *2011 IEEE Fourth International Conference on Software Testing, Verification and Validation (ICST)*, 2011, pp. 80-89.

[56] Fraser, G., and Zeller, A., "Generating parameterized unit tests," *Proceedings of the 2011 International Symposium on Software Testing and Analysis*, ACM, 2011, pp. 364-374

[57] Ganesan, D., Knodel, J., Kolb, R., Haury, U., and Meier, G., "Comparing costs and benefits of different test strategies for a software product line: A study from testo ag," *Software Product Line Conference, 2007. SPLC 2007. 11th International*, IEEE, 2007, pp. 74-83

[58] Geppert, B., Li, J., Rößler, F., and Weiss, D. M., "Towards generating acceptance tests for product lines," *Software Reuse: Methods, Techniques, and Tools*, pp. 35-48 Springer, 2004.

[59] Glerum, K., Kinshumann, K., Greenberg, S., Aul, G., Orgovan, V., Nichols, G., . . . Hunt, G., "Debugging in the (very) large: ten years of implementation and experience," *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, ACM, 2009, pp. 103-116.

[60] Gnesi, S., Latella, D., and Massink, M., "Formal test-case generation for UML statecharts," *Engineering Complex Computer Systems, 2004. Proceedings. Ninth IEEE International Conference on*, IEEE, 2004, pp. 75-84

[61] Gnesi, S., Latella, D., and Massink, M., "Model checking UML statechart diagrams using JACK," *High-Assurance Systems Engineering, 1999. Proceedings. 4th IEEE International Symposium on*, IEEE, 1999, pp. 46-55.

[62] Gomaa, H., and Olimpiew, E. M., "Managing variability in reusable requirement models for software product lines," *High Confidence Software Reuse in Large Systems*, pp. 182-185 Springer, 2008.

[63] Grindal, M., Offutt, J., and Andler, S. F., "Combination testing strategies: a survey," *Software Testing, Verification and Reliability*, vol. 15, no. 3, pp. 167-199, 2005.

[64] Hartman, A., "Software and hardware testing using combinatorial covering suites," *Graph Theory, Combinatorics and Algorithms*, pp. 237-266 Springer, 2005.

[65] Hartman, A., and Nagin, K., "The AGEDIS tools for model based testing," *ACM SIGSOFT Software Engineering Notes*, ACM, 2004, pp. 129-132

[66] Hartman, A., and Raskin, L., "Problems and algorithms for covering arrays," *Discrete Mathematics*, vol. 284, no. 1, pp. 149-156, 2004.

[67] Hartmann, J., Vieira, M., and Ruder, A., "A UML-based approach for validating product lines," *Intl. Workshop on Software Product Line Testing (SPLiT), Avaya Labs Technical Report*, Citeseer, 2004, pp. 58-64.

[68] Hervieu, A., Baudry, B., and Gotlieb, A., "Pacogen: Automatic generation of pairwise test configurations from feature models," *Software Reliability Engineering (ISSRE), 2011 IEEE 22nd International Symposium on*, IEEE, 2011, pp. 120-129

[69] Hu, C., and Neamtiu, I., "Automating GUI testing for Android applications," *Proceedings of the 6th International Workshop on Automation of Software Test*, ACM, 2011, pp. 77-83

[70] Imai, T., Kataoka, Y., and Fukaya, T., "Evaluating software maintenance cost using functional redundancy metrics," *Computer Software and Applications Conference, 2002. COMPSAC 2002. Proceedings. 26th Annual International*, IEEE, 2002, pp. 299-306

[71] Jaaksi, A., "Developing mobile browsers in a product line," *IEEE software*, vol. 19, no. 4, pp. 73-80, 2002.

[72] Jääskeläinen, A., Takala, T., and Katara, M., "Model-based GUI testing of Android applications," *Experiences of Test Automation: Case Studies of Software Test Automation*, pp. 253, 2012.

[73] Jaring, M., Krikhaar, R. L., and Bosch, J., "Modeling variability and testability interaction in software product line engineering," *Composition-*

Based Software Systems, 2008. ICCBSS 2008. Seventh International Conference on, IEEE, 2008, pp. 120–129.

[74] Jin-Hua, L., Qiong, L., and Jing, L., "The w-model for testing software product lines," *Computer Science and Computational Technology, 2008. ISCSCT'08. International Symposium on*, IEEE, 2008, pp. 690-693

[75] Jones, J. A., Orso, A., and Harrold, M. J., "Gammatella: Visualizing program-execution data for deployed software," *Information Visualization*, vol. 3, no. 3, pp. 173-188, 2004.

[76] Juristo, N., and Moreno, A. M., *Basics of software engineering experimentation*: Springer Publishing Company, Incorporated %@ 1441950117, 2010.

[77] Kahsai, T., Roggenbach, M., and Schlingloff, B. H., "Specification-based testing for refinement," *Software Engineering and Formal Methods, 2007. SEFM 2007. Fifth IEEE International Conference on*, IEEE, 2007, pp. 237-246

[78] Kan, S. H., *Metrics and models in software quality engineering*: Addison-Wesley Longman Publishing Co., Inc. %@ 0201729156, 2002.

[79] Kang, S., Lee, J., Kim, M., and Lee, W., "Towards a Formal Framework for Product Line Test Development," *CIT*, vol. 7, pp. 921-926, 2007.

[80] Kim, M., Sazawal, V., Notkin, D., and Murphy, G., "An empirical study of code clone genealogies," *ACM SIGSOFT Software Engineering Notes*, ACM, 2005, pp. 187-196

[81] Kinshumann, K., Glerum, K., Greenberg, S., Aul, G., Orgovan, V., Nichols, G., . . . Hunt, G., "Debugging in the (very) large: ten years of implementation and experience," *Communications of the ACM*, vol. 54, no. 7, pp. 111-116, 2011.

[82] Kishi, T., and Noda, N., "Formal verification and software product lines," *Communications of the ACM*, vol. 49, no. 12, pp. 73-77, 2006.

[83] Kishi, T., Noda, N., and Katayama, T., "Design verification for product line development," *Software Product Lines*, pp. 150-161 Springer, 2005.

[84] Knauber, P., and Schneider, J., "Tracing variability from implementation to test using aspect-oriented programming," *International Workshop on Software Product Line Testing*, Citeseer, 2004, p. 36.

[85] Kolb, R., "A risk-driven approach for efficiently testing software product lines," *Fraunhofer Institute for Experimental Software Engineering (IESE)*, 2003.

[86] Kolb, R., and Muthig, D., "TECHNIQUES AND STRATEGIES FOR TESTING COMPONENT-BASED SOFTWARE AND PRODUCT

LINES," *Development of Component-based Information Systems*, pp. 123, 2006.

[87] Koltun, P., and Hudson, A., "A reuse maturity model," 1991.

[88] Koschke, R., Falke, R., and Frenzel, P., "Clone detection using abstract syntax suffix trees," *Reverse Engineering, 2006. WCRE'06. 13th Working Conference on*, IEEE, 2006, pp. 253-262

[89] Lamancha, B. P., and Usaola, M. P., "Testing product generation in software product lines using pairwise for features coverage," *Testing Software and Systems*, pp. 111-125 Springer, 2010.

[90] Lamancha, B. P., Usaola, M. P., and Velthius, M. P., "Software Product Line Testing," *A Systematic Review. ICSOFT (1)*, pp. 23-30, 2009.

[91] Lazić, L., "The Integrated and Optimized Software Testing Process," PhD Thesis, School of Electrical Engineering, Belgrade, Serbia, 2007.

[92] Lee, J., Kang, S., and Lee, D., "A survey on software product line testing," *Proceedings of the 16th International Software Product Line Conference-Volume 1*, ACM, 2012, pp. 31-40

[93] Lee, K., Kang, K. C., and Lee, J., "Concepts and guidelines of feature modeling for product line software engineering," *Software Reuse: Methods, Techniques, and Tools*, pp. 62-77 Springer, 2002.

[94] Liblit, B., Naik, M., Zheng, A. X., Aiken, A., and Jordan, M. I., "Scalable statistical bug isolation," *ACM SIGPLAN Notices*, ACM, 2005, pp. 15-26

[95] Ludewig, J., and Lichter, H., *Software Engineering: Grundlagen, Menschen, Prozesse, Techniken*: dpunkt. verlag %@ 3864911680, 2012.

[96] Mahmood, R., Esfahani, N., Kacem, T., Mirzaei, N., Malek, S., and Stavrou, A., "A whitebox approach for automated security testing of Android applications on the cloud," *Automation of Software Test (AST), 2012 7th International Workshop on*, IEEE, 2012, pp. 22-28

[97] Marin, M., Van Deursen, A., and Moonen, L., "Identifying aspects using fan-in analysis," *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*, IEEE, 2004, pp. 132-141

[98] McCall, J. A., "Quality factors," *encyclopedia of Software Engineering*, 1994.

[99] McGregor, J., "Testing a Software Product Line," *Testing Techniques in Software Engineering*, Lecture Notes in Computer Science P. Borba *et al.*, eds., pp. 104-140 Springer Berlin / Heidelberg, 2010.

[100] McGregor, J., "Testing a software product line," 2001.

[101] McGregor, J. D., "Building reusable test assets for a product line," *Software Reuse: Methods, Techniques, and Tools*, pp. 345-346 Springer, 2002.

[102] McGregor, J. D., "Toward a Fault Model for Software Product Lines," *SPLC (2)*, 2008, pp. 157-162.

[103] Miller, G. A., and Chomsky, N., "Finitary models of language users," 1963.

[104] Miller, G. A., and Johnson-Laird, P. N., *Language and perception*: Belknap Press, 1976.

[105] Mishra, S., "Specification based software product line testing: a case study," *Concurrency, Specification and Programming*, 2006.

[106] Muccini, H., Di Francesco, A., and Esposito, P., "Software testing of mobile applications: Challenges and future research directions," *Automation of Software Test (AST), 2012 7th International Workshop on*, IEEE, 2012, pp. 29-35

[107] Muccini, H., Dias, M., and Richardson, D. J., "Software architecture-based regression testing," *Journal of Systems and Software*, vol. 79, no. 10, pp. 1379-1396, 2006.

[108] Muccini, H., Dias, M. S., and Richardson, D. J., "Towards software architecture-based regression testing," *ACM SIGSOFT Software Engineering Notes*, ACM, 2005, pp. 1-7

[109] Muccini, H., and Van Der Hoek, A., "Towards testing product line architectures," *Electronic Notes in Theoretical Computer Science*, vol. 82, no. 6, pp. 99-109, 2003.

[110] Myers, G. J., Sandler, C., and Badgett, T., *The art of software testing*: John Wiley & Sons %@ 1118133153, 2011.

[111] Nebut, C., Fleurey, F., Le Traon, Y., and Jézéquel, J.-M., "A requirement-based approach to test product families," *Software Product-Family Engineering*, pp. 198-210 Springer, 2004.

[112] Nebut, C., Le Traon, Y., and Jézéquel, J.-M., "System testing of product lines: From requirements to test cases," *Software Product Lines*, pp. 447-477 Springer, 2006.

[113] Nebut, C., Pickin, S., Le Traon, Y., and Jézéquel, J.-M., "Automated requirements-based generation of test cases for product families," *Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference on*, IEEE, 2003, pp. 263-266

[114] Nori, A. V., and Rajamani, S. K., "An empirical study of optimizations in YOGI," *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, ACM, 2010, pp. 355-364

[115] Northrop, L., Clements, P., Bachmann, F., Bergey, J., Chastek, G., Cohen, S., . . . Little, R., "A framework for software product line practice, version 5.0," *SEI.-2007*, 2007.

[116] Olimpiew, E., and Gomaa, H., "Reusable system tests for applications derived from software product lines," *International Workshop on Software Product Line Testing (SPLiT 2005)*, Citeseer, 2005.

[117] Olimpiew, E. H. G., and Gomaa, H., "Customizable requirements-based test models for software product lines," *International Workshop on Software Product Line Testing*, 2006.

[118] Olimpiew, E. M., "Model-based testing for software product lines," 2008.

[119] Olimpiew, E. M., and Gomaa, H., "Model-based testing for applications derived from software product lines," *ACM SIGSOFT Software Engineering Notes*, vol. 30, no. 4, pp. 1-7, 2005.

[120] Olimpiew, E. M., and Gomaa, H., "Reusable model-based testing," *Formal Foundations of Reuse and Domain Engineering*, pp. 76-85 Springer, 2009.

[121] Orso, A., and Rothermel, G., "Software testing: a research travelogue (2000–2014)," *Proceedings of the IEEE International conference on Software Engineering (ICSE), Future of Software Engineering*, 2014.

[122] Orso, A., Shi, N., and Harrold, M. J., "Scaling regression testing to large software systems," *ACM SIGSOFT Software Engineering Notes*, ACM, 2004, pp. 241-251.

[123] Perrouin, G., Oster, S., Sen, S., Klein, J., Baudry, B., and Le Traon, Y., "Pairwise testing for software product lines: comparison of two approaches," *Software Quality Journal*, vol. 20, no. 3-4, pp. 605-643, 2012.

[124] Petersen, K., Feldt, R., Mujtaba, S., and Mattsson, M., "Systematic mapping studies in software engineering."

[125] Pohl, K., Böckle, G., and Van Der Linden, F., *Software product line engineering: foundations, principles, and techniques*: Springer, 2005.

[126] Pohl, K., and Metzger, A., "Software product line testing," *Communications of the ACM*, vol. 49, no. 12, pp. 78-81, 2006.

[127] Pressman, R. S., and Jawadekar, W. S., "Software engineering," *New York 1992*, 1987.

[128] Pretschner, A., "Model-based testing," *Software Engineering, 2005. ICSE 2005. Proceedings. 27th International Conference on*, IEEE, 2005, pp. 722-723.

[129] Pretschner, A., Prenninger, W., Wagner, S., Kühnel, C., Baumgartner, M., Sostawa, B., . . . Stauner, T., "One evaluation of model-based testing and its automation," *Proceedings of the 27th international conference on Software engineering*, ACM, 2005, pp. 392-401

[130] Rajapakse, D. C., and Jarzabek, S., "An investigation of cloning in web applications," *Web Engineering*, pp. 252–262: Springer, 2005.

[131] Rajapakse, D. C., and Jarzabek, S., "Using server pages to unify clones in web applications: A trade-off analysis," *Software Engineering, 2007. ICSE 2007. 29th International Conference on*, IEEE, 2007, pp. 116–126.

[132] Reuys, A., Kamsties, E., Pohl, K., and Reis, S., "Model-based system testing of software product families," *Advanced Information Systems Engineering*, Springer, 2005, pp. 519-534.

[133] Reuys, A., Reis, S., Kamsties, E., and Pohl, K., "Derivation of domain test scenarios from activity diagrams," *Proceedings of the International Workshop on Product Line Engineering–The Early Steps–Planning, Modeling and Managing (PLEES'03), Fraunhofer IESE, Erfurt, September*, Citeseer, 2003.

[134] Reuys, A., Reis, S., Kamsties, E., and Pohl, K., "The scented method for testing software product lines," *Software Product Lines*, pp. 479-520: Springer, 2006.

[135] Riebisch, M., Böllert, K., Streitferdt, D., and Philippow, I., "Extending feature diagrams with UML multiplicities," *Proceedings of the Sixth Conference on Integrated Design and Process Technology (IDPT 2002), Pasadena, CA*, 2002.

[136] Robinson-Mallett, C., Grochtmann, M., Wegener, J., Kohnlein, J., and Kuhn, S., "Modelling requirements to support testing of product lines," *Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on*, IEEE, 2010, pp. 11-18

[137] Rothermel, G., and Harrold, M. J., "A safe, efficient regression test selection technique," *ACM Transactions on Software Engineering and Methodology (TOSEM)*, vol. 6, no. 2, pp. 173-210, 1997.

[138] Roy, C. K., Cordy, J. R., and Koschke, R., "Comparison and evaluation of code clone detection techniques and tools: A qualitative approach," *Science of Computer Programming*, vol. 74, no. 7, pp. 470-495, 2009.

[139] Runeson, P., and Höst, M., "Guidelines for conducting and reporting case study research in software engineering," *Empirical software engineering*, vol. 14, no. 2, pp. 131-164, 2009.

[140] Saha, R. K., Asaduzzaman, M., Zibran, M. F., Roy, C. K., and Schneider, K. A., "Evaluating code clone genealogies at release level: An empirical study," *Source Code Analysis and Manipulation (SCAM), 2010 10th IEEE Working Conference on*, IEEE, 2010, pp. 87-96

[141] Shafique, M., and Labiche, Y., "A systematic review of model based testing tool support," *Software Quality Engineering Laboratory, Department of Systems and Computer Engineering, Carleton University, Technical Report SCE-10-04*, 2010.

[142] Shaulis, C. L., "Salion's Quality Confident Approach to Testing Software Product Lines," *International Workshop on Software Product Line Testing*, Citeseer, 2004, p. 78.

[143] Sommerville, I., *Software Engineering. International computer science series*: Addison Wesley, 2004.

[144] Stephenson, Z., Zhan, Y., Clark, J., and McDermid, J., "Test Data Generation for Product Lines–A Mutation Testing Approach," *International Workshop on Software Product Line Testing*, Citeseer, 2004, p. 13.

[145] Takala, T., Katara, M., and Harty, J., "Experiences of system-level model-based GUI testing of an Android application," *Software Testing, Verification and Validation (ICST), 2011 IEEE Fourth International Conference on*, IEEE, 2011, pp. 377-386

[146] Tevanlinna, A., Taina, J., and Kauppinen, R., "Product family testing: a survey," *ACM SIGSOFT Software Engineering Notes*, vol. 29, no. 2, pp. 12-12, 2004.

[147] Utting, M., and Legeard, B., *Practical model-based testing: a tools approach*: Morgan Kaufmann, 2010.

[148] Utting, M., Pretschner, A., and Legeard, B., "A taxonomy of model-based testing," 2006.

[149] van der Linden, F. J., Schmid, K., and Rommes, E., *Software product lines in action*: Springer 5, 2007.

[150] Van Rompaey, B., Du Bois, B., Demeyer, S., and Rieger, M., "On The Detection of Test Smells: A Metrics-Based Approach for General Fixture and Eager Test," *IEEE Transactions on Software Engineering*, vol. 33, no. 12, pp. 800-817, 2007.

[151] Vieira, M. E., Dias, M. S., and Richardson, D. J., "Object-Oriented specification-based testing using UML statechart diagrams," *Proceedings of the Workshop on Automated Program Analysis, Testing, and Verification (at ICSE'00)*, 2000.

[152] Weingärtner, J., "Product family engineering and testing in the medical domain—validation aspects," *Software Product-Family Engineering*, pp. 383-387 Springer, 2002.

[153] Weiss, D. M., "Software product-line engineering: a family-based software development process," 1999.

[154] Weißleder, S., Sokenou, D., and Schlingloff, B.-H., "Reusing state machines for automatic test generation in product lines," *Model-Based Testing in Practice (MoTiP), Fraunhofer IRB Verlag*, pp. 19-28, 2008.

[155] Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A., *Experimentation in software engineering*: Springer %@ 3642290442, 2012.

[156] Wübbeke, A., "Towards an Efficient Reuse of Test Cases for Software Product Lines," *SPLC (2)*, 2008, pp. 361-368.

[157] Yang, Q., Li, J. J., and Weiss, D. M., "A survey of coverage-based testing tools," *The Computer Journal*, vol. 52, no. 5, pp. 589-597, 2009.

[158] Yu, Y., and Wu, F., "A software acceptance testing technique based on knowledge accumulation," *VLSI, 1999. Proceedings. Ninth Great Lakes Symposium on*, IEEE, 1999, pp. 296-299.

[159] Zhang, L., Liu, Y., and Guo, W., "Research on Diversified Designing Methods and User Evaluation of Smartphone Interface," *Computational Intelligence and Design (ISCID), 2010 International Symposium on*, IEEE, 2010, pp. 10-13

Appendix A

Journal and Conference Listing

- ✓ Abstract State Machines 2004. Advances in Theory and Practice
- ✓ ACM SIGSOFT Software Engineering Notes
- ✓ ACM Transactions on Software Engineering and Methodology (TOSEM)
- ✓ Advanced Information Systems Engineering
- ✓ Automated Software Engineering, 2003. Proceedings. 18th IEEE International Conference
- ✓ Automation of Software Test (AST), 2012 7th International Workshop
- ✓ Books
- ✓ Book Sections
- ✓ CHI'04 extended abstracts on Human factors in computing systems
- ✓ Communications of the ACM
- ✓ Computational Intelligence and Design International Symposium
- ✓ Computer and Information Sciences-ISCIS
- ✓ Computer Science and Computational Technology, 2008. ISCSCT'08. International Symposium
- ✓ Computer Software and Applications Conference
- ✓ Development of Component-based Information Systems
- ✓ Discrete Mathematics
- ✓ Electronic Notes in Theoretical Computer Science
- ✓ Empirical software engineering
- ✓ Engineering Complex Computer Systems
- ✓ Experiences of Test Automation: Case Studies of Software Test Automation
- ✓ Formal Foundations of Reuse and Domain Engineering
- ✓ Fraunhofer Institute for Experimental Software Engineering (IESE)
- ✓ Graph Theory, Combinatorics and Algorithms
- ✓ IEEE Software
- ✓ IEEE Transactions on Software Engineering
- ✓ Information and Software Technology
- ✓ International Conference on Evaluation and Assessment in Software Engineering (EASE)
- ✓ International Conference on Empirical Assessment & Evaluation in Software Engineering

- ✓ International Conference on Software Engineering and Knowledge Engineering (SEKE)
- ✓ International Conference on Software Reuse
- ✓ ACM SIGSOFT Software Engineering Notes
- ✓ IEEE International Conference on Computer and Information Technology
- ✓ Communications of the ACM
- ✓ European Workshop on Model Driven Architecture
- ✓ International Conference on Software and Data Technologies, Proceedings
- ✓ Information and Software Technology
- ✓ International Workshop on Software Product Line Testing
- ✓ Journal of Systems and Software
- ✓ International Software Product Line Conference
- ✓ International Workshop on Software Product-Family Engineering,
- ✓ Joint European Software Engineering Conference (ESEC) and SIGSOFT Symposium on the Foundations of Software Engineering (FSE-11)
- ✓ Journal of Combinatorial Designs
- ✓ Journal of Software Engineering and Knowledge Engineering
- ✓ Journal of Systems and Software
- ✓ Proc. Int. Workshop on Software Clones
- ✓ Reverse Engineering Working Conference
- ✓ Software, IEEE

Appendix B

Essential ART Syntax

Adaptive Reuse Technique (**ART**) follows a pre-processor style syntax and helps testers to incorporate variability as base test case codes for family of test library variants. ART organises and instruments test templates for ease of adaptation and reuse. Following summary of Adaptive Reuse Technique (ART) syntax as was adopted from the ART website [http://art.comp.nus.edu.sg/].

<table>
<tr><th colspan="2">ART Syntax</th></tr>
<tr><td colspan="2">File Types (SPC and ART)</td></tr>
<tr><td>Execution Sequence</td><td>ART processor starts processing the test templates with the specification file (has a *.spc extension). The processor executes statement by statement and reaches end of the SPC file. Additional configuration input files (*.art file extension) can be created and adapted by calling from the spc file.
To understand the execution sequence, consider the example shown in figure below. ART processor processes the TypeTest.SPC file line by line. When the processor encounters adapt command it starts processing TypeTest.art followed by moreMethods.art. Conditionally for Byte type Byte moreMethods.art is adapted in sequence.
 
</td></tr>
<tr><td colspan="2"># adapt command</td></tr>
<tr><td>Syntax</td><td>#adapt: file
<customizations>
#endadapt</td></tr>
<tr><td>Attributes</td><td>file : File name to be adapted</td></tr>
<tr><td>Description</td><td>Whenever ART processor encounters the ”#adapt file-A” command, processing of the current file is suspended and the processor starts processing file-A. Once processing of file-A is completed, the processor resumes processing of the current file</td></tr>
</table>

	for statements just after `#adapt file-A`. The syntax and scoping rules for commands used under `#adapt` command are the same as outside the `#adapt` command.	
Additional	A chain of `#adapt` commands must not lead to recursion, i.e., no file can adapt itself directly or indirectly.	
# output command		
Syntax	`#`**`output`** `<path>`	
Attributes	The `<path>` can be absolute or relative path.	
Description	Output command specifies the output file where the source code from the test template needs to be placed. If output file is not specified, then processor emits code to an automatically generated default file named *defaultOutput* in the main installation folder of the processor.	
# set command		

Syntax	`#`**`set`** `<var> = "value"`
	OR
	`#`**`set`** `<var> = "value1", "value2", "value 3"…`	

Attributes	`<var>` Single or multi valued variable.	
Description	`#set` command declares a test template variable and sets its value. With the `#set` command, we can either declare single and multi-value variables.	
Expressions		
Syntax	***?<<expression>>?***	
Description	Expressions are written between question mark '?' characters. There are three types of expressions, namely *name expression*, *string expression* and *arithmetic expression*. Note: A direct reference to variable x is written as?@x?. 1. A *name expression* can contain *variable references* (example ?@x?), and combinations of *variable references* (example ?@x@y@z?). 2. A *string expression* can contain any number of *name expressions* intermixed with character strings. To evaluate a *string expression*, we evaluate the *name expressions* from the left to the right of the *string expression*, replace *name expressions* with their respective values and concatenate with character strings. 3. An *arithmetic expression* can contain any mathematical expression. When an arithmetic expression is a well-formed, the processor recognizes it as such and evaluates its value. An arithmetic expression can contain '+', '-', '*', '/' operators and nested parenthesis. Usual operator precedence rules as in programming languages such as Java is applicable.	
Additional	Arithmetic and String expression cannot be mixed together. An expression is either purely string or purely mathematical in nature.	
The insert-break mechanism		

Syntax	`#`**`insert`** `breakX`
	`  content`
	`#`**`endinsert`**
		`#`**`break`** `breakX`
		`OR`
		`#`**`break:`** `breakX`
		`  default content`
		`#`**`endbreak`**

Description	An `#insert` command replaces all matching `#break` with its content. Matching is done by a name (`breakX` in the example). `#break` commands in all files reached via `#adapt` chain can be affected.
Loops and Selections	

Syntax	`#`**`while`** `mul-val-var1, mul-val-var2 . . .`
	`content`
	`#`**`endwhile`**

Description	Command `#while` is a generation loop that iterates over its body and generates custom text at each iteration. The `#while` command is controlled by one or more multi-value variables. The i^{th} value of each of the control variables is used in i^{th} iteration of the loop. This implies that all the control variables should have the same number of values, and their respective number of values determines the number of iterations of the loop.

Syntax	`#`**`select`** `<control-variable>`
	`#option-undefined`
	`% this will be executed if <variable>`
	`% is not defined`
	`. . .`
	`#endoption-unindefined`
	`#option <value>`
	`% this will be executed if value of <variable>`
	`% is the given <value>`
	`. . .`
	`#endoption`
	`#option <value2|value3>`
	`% this will be executed if value of <variable>`
	`% is <value2> OR <value3>`
	`. . .`
	`#endoption`
	`. . .`
	`#otherwise`
	`% this will executed if <variable> is defined,`
	`% and none of the options corresponds to value`
	`% of <variable>`
	`. . .`
	`#endotherwise`
	`#`**`endselect`**

Description	Command `#select` allows us to choose one of many customization options. With the `#select` command we can select one of many options, depending on the value of a control variable. The processor selects and processes in turn all the `#options` whose values match the value of the control variable. `#option-undefined` is processed if control variable is undefined. `#otherwise` is processed if none of the `#options` can be selected.

Additional	#while and #select are often used together. #while command is often used for test code generation. For instance, generating test case for testing database tables, user interface buttons etc.
Comments	
Syntax	`% comments`
Description	Text following `%` is considered a comment. In order to ignore a `%` symbol a tester can use?
#setloop command	
Description	Keeping track of corresponding values becomes troublesome in while loop, especially when variables have many values that are often changed. Any mismatch of values may cause an annoying error. `#setloop` command alleviates this problem by allowing us to organize the values of control variables to be used in a while loop in a more intuitive and less error prone way than multi-value variables do. The basic usage scenarios for this command can be directly translated into `#set` commands that control `#while` in the usual way. `#setloop` command organizes values of loop control variables into a table, where rows are formed by loop iteration and columns by values of control variables.

Appendix C

BiDiTests File Listing

Java Files	XML Files
BiDiTestActivity.java	attrs.xml
BiDiTestBasic.java	basic.xml
BiDiTestCanvas.java	canvas.xml
BiDiTestCanvas2.java	canvas2.xml
BiDiTestConstants.java	custom_list_item.xml
BiDiTestFrameLayoutLocale.java	frame_layout_locale.xml
BiDiTestFrameLayoutLtr.java	frame_layout_ltr.xml
BiDiTestFrameLayoutRtl.java	frame_layout_rtl.xml
BiDiTestGalleryImages.java	gallery_ltr.xml
BiDiTestGalleryLtr.java	gallery_rtl.xml
BiDiTestGalleryRtl.java	grid_layout_code.xml
BiDiTestGridLayoutCodeLtr.java	grid_layout_locale.xml
BiDiTestGridLayoutCodeRtl.java	grid_layout_ltr.xml
BiDiTestGridLayoutLocale.java	grid_layout_rtl.xml
BiDiTestGridLayoutLtr.java	linear_layout_locale.xml
BiDiTestGridLayoutRtl.java	linear_layout_ltr.xml
BiDiTestLinearLayoutLocale.java	linear_layout_rtl.xml
BiDiTestLinearLayoutLtr.java	main.xml
BiDiTestLinearLayoutRtl.java	main_menu.xml
BiDiTestRelativeLayout2Locale.java	relative_layout_2_locale.xml
BiDiTestRelativeLayout2Ltr.java	relative_layout_2_ltr.xml
BiDiTestRelativeLayout2Rtl.java	relative_layout_2_rtl.xml
BiDiTestRelativeLayoutLtr.java	relative_layout_ltr.xml
BiDiTestRelativeLayoutRtl.java	relative_layout_rtl.xml
BiDiTestTableLayoutLocale.java	strings.xml
BiDiTestTableLayoutLtr.java	table_layout_locale.xml
BiDiTestTableLayoutRtl.java	table_layout_ltr.xml
BiDiTestTextViewAlignmentLtr.java	table_layout_rtl.xml
BiDiTestTextViewAlignmentRtl.java	textview_alignment_ltr.xml
BiDiTestTextViewDirectionLtr.java	textview_alignment_rtl.xml
BiDiTestTextViewDirectionRtl.java	textview_direction_ltr.xml
BiDiTestTextViewDrawablesLtr.java	textview_direction_rtl.xml
BiDiTestTextViewDrawablesRtl.java	textview_drawables_ltr.xml
BiDiTestTextViewLocale.java	textview_drawables_rtl.xml
BiDiTestTextViewLtr.java	textview_locale.xml
BiDiTestTextViewRtl.java	textview_ltr.xml
BiDiTestView.java	textview_rtl.xml
BiDiTestViewDrawText.java	view_group_margin_mixed.xml
BiDiTestViewGroupMarginMixed.java	view_padding.xml
BiDiTestViewPadding.java	view_padding_mixed.xml
BiDiTestViewPaddingMixed.java	