

## Chapter 1

# Introduction

### 1.1 The Need for Testing

One of the common but very true jokes about the software industry tells of a software entrepreneur proudly stating that if the automobile industry had advanced in the past decade with the same rate as that of the computer industry, we would all be driving \$25 cars that run for a 1,000 miles per gallon of gasoline. To which an automobile CEO replies, that if cars were like today's software, they would crash twice a day for no reason, and when you call for service, they would tell you to shut down the engine, as well as all windows, get out of the car, lock it, and start over! Indeed, in the past decade, computer hardware has advanced significantly offering much higher processing capability and speed. Consequently, the complexity and size of software has increased exponentially too. Software is vital to almost every industry and we are becoming ever more dependent on our computers to perform every day tasks. As the complexity and size of software grow, so does the need to insure the reliability and precision of software outputs. Software errors can be fatal in numerous industries; including airlines, medical and the military. Billions of dollars are spent on software and software testing and maintenance annually by the DoD. Software errors result in hundreds of millions in losses annually from the DoD alone [1] (software errors are defined in Section 1.3.1).

Accordingly, a lot of attention is now directed towards assuring software reliability, accuracy, security and usefulness. It is crucial to dedicate a lot of resources, including time, effort and a budget during the software lifecycle to verify that the end product is useful, usable and accomplishes the tasks it was built to perform.

## 1.2 Why Does Software have Errors?

Software errors result from many reasons, from misunderstanding of the requirements, to poor design; also from tired or careless programmers to rushed deadlines [2]. Often testing is delayed until later in the software development life cycle, and little time is given for testing, debugging, and re-testing. Coverage gaps are also a problem; this happens when not all parts of the software are tested.

Some software failures happen due to a specific scenario, a sequence of events that might have been tested individually but not in that particular sequence, that causes the software failures. The operating environment, various connected hardware peripherals, other concurrently running software and different operating systems could also be a source of software failures [4].

It was proven that complete testing is impossible [3]. Meaning that it is impossible to test all values of each possible input, valid and invalid, with their respective combinations for a software application with considerable size and complexity; the number of test cases will be astronomical and the time needed for testing and verification will be almost endless. For example, a very simple program might take only three inputs, an integer Age, between 0 and 99, a character, Initial, between A and Z, and another integer representing a Month between the integers 1 and 12.

```
Age: integer [0..99]
Initial: char [A..Z]
Month: integer [1..12]
```

For this simple program, there are 100 possible valid values for Age, 26 valid values for Initial, and 12 valid values for Month. Accordingly there are  $100 * 26 * 12 = 31,200$  possible combinations of valid values. If complete testing was required, there would be 31,200 test cases that should be carried out to test the operation of this program on valid values alone. Even if the 31,200 test cases were run, this would not be considered complete testing. The number of test cases required for complete testing will be much greater, when the invalid values are added. Invalid values could be in one, two or all three inputs. The number will increase exponentially if we want to create test cases for all these cases; where only one of the three variables contains an invalid value, a combination of two of the variables are invalid, or all three. For example, the Age variable is an integer, one

can enter any integer from the negative integer boundary to the positive integer boundary. Also, one can try to enter characters, real numbers or any other value for the Age integer value and test the program behavior and how it handles exceptions. The possibilities and combinations of all those cases for this very simple three input program are thus almost endless. Also it is impossible to completely test all values of real numbers, even in a small range, and for strings representing names for example, etc. Some transient errors are revealed in complex distributed or embedded control systems after prolonged usage. This may be a result of a memory overflow, or minute system clock discrepancies that accumulate to a noticeable difference after prolonged usage. Such errors are very difficult to recognize [77], and require high volume system testing or long sequence testing techniques, and extended random regression testing [79]. Hence there is a need for better software testing techniques, which focus on testing parts of the input domain, yet, attain good coverage to verify software quality.

### 1.3 Software Testing Definitions

#### 1.3.1 Software Errors, Faults and Failures

A software error is a mismatch between the program and its specifications, provided that the specifications are accurate and complete. A software application is considered erroneous if it fails to be useful. A software fault is defined as the execution of an error [5]. Faults of commission are those resulting from incorrectly coding specifications, or coding unspecified requirements. Faults of omission result from failing to implement specified requirements. A software failure happens when a fault of commission is executed while running the software application [5]. From Figure 1.1, the goal of testing is to make sure that  $S \cap P = S = P$ . If the above equation is reached in testing a software application, the resulting product is very well tested and functional, provided that the specifications are correct, accurate, complete and do not contain any contradictions.

#### 1.3.2 Software Testing

Software testing is becoming an important area of research aimed at producing more reliable systems and minimizing programming errors [78]. Since it is impossible to completely test how a software application performs on all values of all of its inputs, as described in Section 1.2, a good strategy to test

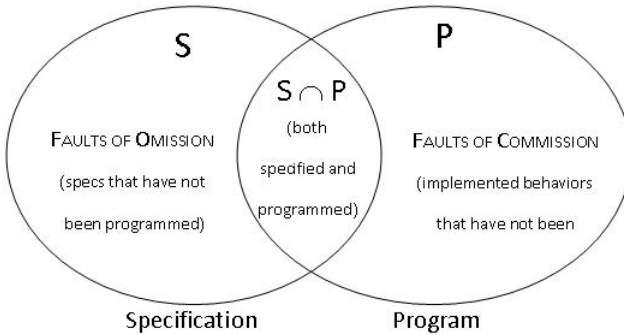


Fig. 1.1 The Relationship Between a Program Specification and its Implementation [5]

software has to be followed in order to choose test cases that test the software without coverage gaps. However, there is much more to good testing than running a program several times to check if it functions correctly [6]. In order to systematically test software, we need to model the environment in which the software is intended to run, select test scenarios that attain good system coverage, execute and evaluate the test cases, correct the errors, and measure the testing progress [4]. Some faults may result from correcting other errors, and hence regression testing is often used, which entails re-running all test cases after an error has been corrected to verify that the fix did not inject any errors in a part of the software that was previously running correctly. Software testing has been defined in many ways. Some authors define it as the process of running a software application to verify its correct execution of its valid and complete specifications, and verify that it runs correctly in the environment it was created for [5, 86]. Other authors argue that test scenarios that do not reveal errors or failures are also of great importance, since the testing team is assuring by such scenarios that the system actually implemented the required specifications correctly [6]. Software can also fail by executing too slowly or using too much memory; reasonable execution speed and memory usage are obvious requirements that might not be in the specification documents. A number of good references for software testing are [5, 6, 553]. Accordingly, it is crucial to create test cases that reflect a good coverage of all areas of the

software and determine its usability.

## 1.4 When Should Testing Start in the Software Lifecycle?

One of the most important and useful practices in software engineering is the incorporation of software testing from the very beginning of the software lifecycle. The software life cycle is described in more detail in [7, 8, 9]. Waiting until the software is implemented and then starting to test the software was proven in the last decade by many authors and software team managers to be a very costly and difficult task to manage and carryout. Testing should start as soon as the analysts finish writing the first draft of the specification document describing what the required system should be built to accomplish. Testers can then begin identifying test cases, which are called system level test cases. System level test cases are very easily interpreted and basically describe in a broad spectrum what the main functionalities of the software should be. By reading the system level test cases, both the development team and the end users can reach an early agreement and common understanding of the functionality of the software under construction.

After agreeing on the specifications, and the system level test cases, the designers can then start defining the details of the system, and answer many questions from as broad as what programming language should be used to the minute details of the functionality of each sub-routine. Testers have a very important role to play during the design phase. During that phase testers should generate the integration level and unit level test cases. Integration level test cases are targeted towards testing the correctness of the interaction between the different sub-routines (methods, functions, procedures, etc.) and the correctness of the interaction between the different modules or subsystems of the software under construction [5].

Once implementation starts, the programmers should have both the design and the test cases that will be exercised against their implementation. Having the complete set of test cases, that cover all possible outputs of the functionality a programmer is implementing, will help the programmer to understand the design and implement it correctly. There is a strong possibility that the programmer will make sure the implemented code passes the given tests before submitting the code for integration with the rest of the system. There are many advantages for the above approach of starting testing early in the software lifecycle. In this situation, when the program-

mer is given both the design and test cases, testing starts very early in the software lifecycle. A test case will be executed multiple times to check that the system passes it correctly at the unit level as well as the system level after integration with the rest of the system modules. There will be no ambiguity in what exactly the programmer should implement, even if the programmer misinterprets the design and specified functionality, reading the test cases, and their respective expected output, will clear much of the ambiguity, and will assure a higher level of software quality, compared to that when only a design is given to the programmer. Once the implementation phase is over, the majority of the testing phase will have been implemented already. The cost of integration and system level testing will be minimal compared to that of delaying testing until the last phase in the software lifecycle. Also the overall quality of the implemented software will be much higher than might result given the same resources of time and money, but leaving testing till after implementation. It is worth noting that the waterfall model of software development where all the specifications of the software are known from the beginning is not what is generally followed in real world software development environments. Test cases should be created for all the specifications as they are gathered for the different builds or versions of the software under construction. Testing should be carried out for each of those versions to make sure that the specifications thus far collected, are all implemented correctly. Specifications should be tested to make sure that they do not contradict previously specified requirements. Google, Microsoft, and Sun Microsystems are investing \$7.5 million to fund research on alternative software development models to the waterfall model with higher reliability and faster deployment, at UC Berkeley [101].

In conclusion, it is essential that the testing process starts from the beginning of the software lifecycle and be concurrent with all the phases as a better software engineering practice.

## 1.5 Types of Testing Techniques

There are two approaches to identifying test cases: functional and structural. Functional testing, also often referred to as black box testing, is the process of testing the executable program without any reference to the source code of the software. It is called functional since we treat the software as a function, we provide inputs, and observe its output. Functional test cases can be reused even when the code changes, is translated into

another language, or its structure differs. Another advantage of functional testing is that the test cases can be created during the early phases of development, even before implementation starts, since it is based on the specifications. There are a number of functional testing techniques including boundary value analysis, equivalence class testing, and decision table-based testing. The reader can refer to [5] for a good coverage of those three techniques. Other black box testing techniques are domain driven, stress driven, specification driven, risk driven, random / statistical, function, regression, scenario, use case, user and exploratory testing. The reader can refer to [7] for an explanation of these techniques. In structural testing on the other hand, also referred to as white or clear box testing, test cases are created to execute specific branches and paths in the source code. Structural testing techniques include decision-to-decision and define-use testing. The advantage of structural testing is in its ability to confirm that all paths in the source code have been executed in test cases, and thus those testing techniques can be used as a coverage matrix to guide other functional testing techniques.

## 1.6 UML

To better understand some types of today's automated testing tools that are targeted towards testing in the requirements definition and design phases, one should understand the Unified Modeling Language (UML). The reader can refer to [11, 12, 13, 14] for UML syntax and diagrams. The majority of software development organizations today, use UML in describing a system design. System designers represent the design of a system using UML, a set of diagrams that both technical and non-technical system end users can understand and use as a basis for discussions and reassurance that the software under development is what the end-users actually have in mind. Those UML diagrams are then used as a basis for more detailed system design, implementation and testing. In their book, *Business Modeling with UML* [11], Eriksson and Penker describe the nine different UML diagrams and the use of each for business modeling. They state that 'UML standardizes notation for describing a process, but it doesn't standardize a process for producing those descriptions' [11]. This leads to the first objective, namely, to create a standard representation for storing UML or software specifications represented by a formal specification languages like Z, since UML diagrams are not machine interpretable.

## 1.7 Formal Specification Languages

Formal specifications, also known as formal methods, are a mathematical-like notation used to describe the specifications, i.e. what a system should execute, without necessarily going into the details of how the requirements should be implemented. Formal specifications were developed primarily to result in accurate, unambiguous software specifications to replace informal natural language specifications for better engineered software. The reader can refer to [15-24, 65] for a detailed description of formal methods, use, semantics and syntax.

## 1.8 Current Testing Technologies and Tools

### 1.8.1 *Types of Automated Testing Tools*

Depending on where in the software lifecycle a development team would like to introduce an automated testing tool, the types of tools can be divided into tools for the analysis and specifications gathering phase, others for the requirements definition phase, design phase, implementation phase, and finally quality assurance and debugging phase. There are also metrics and testing organization tools to aid in the process of managing the testing effort. Some tools are targeted for specific purposes like test data generators [57], simulators and prototyping tools.

There are dozens of automated software testing tools in the market today. The tools range in their functionality between functional-black box testing, to source code analysis tools. A good list can be found at the Aptest: Software Testing Resources and Tools website [25]. A lot of the available tools record and play back test scenarios the tester inputs while doing exploratory or scenario testing. Regression testing is the most commonly automated testing technique.

Some of the tools are targeted towards detecting run-time errors and memory leaks. Among those tools are Purify from Rational Software, C Verifier from PolySpace, SWAT from Coverity, Insure++ from Parasoft and GlowCode from Electric Software. Other tools are test case organization tools, to help in the organization of the testing process. There are a number of automated test data generators and a lot of research in the automation of test data creation [69-72, 80-85, 87-95]. Test data generators generate data to simulate real life data in operational systems for the purpose of testing. Simulators of test environments are also available as well as automated

stress and load testing tools. The following sections include a more detailed look at some of the features and underlying technology in several of the most widely used tools.

### 1.8.2 *Design and Visual Modeling Tools*

Design and visual modeling tools are mainly used in the analysis and design phases. These tools help the development team as well as the end users to reach a common understanding of the functionality of the system under construction. Some of the available tools also generate the system backbone from the design generated by the visual modeling tools, including the database tables, classes, modules, etc. The programmers can then write the code in the designated locations. Rational Rose and Oracle Designer are two of the common design and visual modeling tools available in the market.

#### 1.8.2.1 *Rational Rose*

Rational Rose provides a channel to communicate a system's architecture between various parties of the development team, from analysts and designers to developers and other team members. Rational Rose allows the designers to visually map system architecture to unified models and diagrams using models similar to UML. The diagrams go in their detailed level only to define the interaction between the different parts of the software. You can write code in the subroutine definition area, and thus when you choose to generate the code from Rational Rose, the code that you wrote will also exist where it should in your application. However, Rational Rose does not go into the details of the design or interpret the functionality of the software as it does with the higher level design. Accordingly, the testing tools that are now available and produced by Rational, do not depend on the specified functionality, but on the backbone, and the actual implementation and user scenarios.

#### 1.8.2.2 *Oracle Designer*

Oracle is one of the leading companies in database and information systems solutions. One of the products offered by Oracle is Oracle Designer. As stated by Oracle, Designer is "a complete toolset to model, generate, and capture the requirements and design of enterprise applications" [26]. Oracle Designer has many modeling tools to help in the design and analysis capture

of a database application. Among the tools included in Designer are the process Modeler, Dataflow Diagrammer, function Hierarchy Diagrammer, Entity relationship Diagrammer, Design Editor, Dependency Manager and Matrix Diagrammer. None of the above tools captures the detailed design and the functionality of the application, in a way that could be interpreted by an automated testing tool. The diagrams and matrixes are used to communicate system requirements and design between the different development team members, but not for an automated testing tool, that can intelligently understand the specifications and design of the software under construction and accordingly generate good test cases to test it. In the design editor, one could design the forms and the components of the application's windows, etc. One can also determine what type of triggers run, and what triggers them, but the code and the meaning of the implementation in those triggers can either be written in plain English as comments for the programmers or can be written in actual PL-SQL code. However, what is written in the triggers or stored procedures in designer, whether it is the actual code or plain English description, is stored as is to allow for it to exist after the user chooses to generate the application from the stored design in Oracle Designer.

Designer automatically generates a script for the specified database tables, constraints, modules for the application, and also forms, triggers, etc. Designer thus stores the structure of the application, whether it is the database schema architecture or the higher level process interaction and application dataflow and user interface. Accordingly, Oracle Designer lacks the feature that is proposed in this book to create a well defined representation of the detailed design of the application under construction.

### 1.8.3 *Automated Testing Tools*

It was essential to survey the current automated testing tools in today's market, in order to identify where improvements can be made. The conclusions reached after surveying the market guided much of this research. The products of the most popular vendors of automated testing tools are discussed in this section, including Mercury Interactive, Rational Software, Segue and Compuware.

In the United Kingdom, Mercury Interactive holds at least 50% of the market share in software testing tools. They have a number of automated test tools, including TestDirector [45], WinRunner [46], QuickTest [47] and LoadRunner [48].

Rational Software offers the most complete lifecycle toolset, including testing for the windows platform. They are the market leaders in object-oriented development tools. The leading testing tools available from Rational Software include Rational Robot, Functional Tester, Performance Tester and Rational Purify. Rational Robot is a test management tool that gives test cases for objects such as lists, menus, and bitmaps. It also provides test cases specific to objects in the development environment. As described by Rational [41] it is a "General purpose test automation tool for client/server applications". Functional Tester, on the other hand is an "automated functional and regression testing of Java, Web and VS.NET WinForm-based applications", as described by Rational. It aids the process of test script writing and managing.[42]. Performance Tester verifies application response time and scalability, it aids in emulating a real time web-based environment where lots of users are concurrently using the software. Rational Purify aids in the detection if memory leak and memory corruption for Linux and UNIX [44].

Compuware's major product is QARun. Similar to other automated testing tools, QARun depends on scenario testing, by recording the user's actions and respective system responses into test scripts to test some of the application's functions [52]. Compuware also has a load testing tool.

Segue is another organization that provides some comparable software testing tools to Mercury Interactive and Rational Software. Among the products that it offers are SilkCentral Test Manager, SilkTest 5, and SilkPerformer. Comparable to Rational Robot, SilkCentral is a test management and organization tool [49]. SilkTest is comparable to QuickTest, and Rational Functional Tester. It is a regression testing tool that records and plays back the test cases [50]. SilkPerformer is an automated load, stress and performance testing tool [51], comparable to Rational's Performance Tester and Mercury's LoadRunner.

All four major automated testing tools vendors have products similar in functionality to the products of their competitors. The tools available are mainly categorized into four major types; namely, test management, capture and replay, functional and regression testing and performance or stress testing tools. Table 1.1 summarizes the functionality of the different testing tools. Test management tools like Mercury's TestDirector, Rational TestManager, SilkCentral by Segue and QACenter by Compuware all organize the testing process. Test management tools as described by Mercury, thus provide a consistent, repeatable process for gathering requirements, planning and scheduling tests, analyzing results, and managing defects and

Organization	Scenario Management	Capture and Replay	Functional & Regression Testing	Performance Testing
Mercury Interactive	TestDirector	WinRunner	QuickTest	LoadRunner
Rational Software	TestManager	Rational Robot	Functional Tester	Performance Tester, Purify
Segue	SilkCentral	SilkTest	SilkTest5	SilkPerformer
Compuware	QACenter	QARun	TestPartner	QACenter

issues [45]. Testers can create test cases and log them in the test management tools to organize the testing process.

Capture and replay tools usually automatically generate test scripts from test scenarios captured during a tester's navigation through the software. Examples of the commonly used capture and playback tools are Mercury's WinRunner, Rational Robot, SilkTest by Segue, and QARun by Compuware.

Functional and regression testing tools are also common tools in software test automation. Regression testing tools run test scripts after issues have been resolved, to determine whether the fixes caused other problems in other parts of the software that were previously running correctly. Examples of regression and functional testing tools are Mercury's QuickTest, Rational Functional Tester, Segue's SilkTest5, and Compuware's TestPartner.

Performance and stress testing tools emulate an environment with thousands of users concurrently using the system. Examples of performance and stress testing tools are Mercury LoadRunner, Rational Performance Tester, Segue SilkPerformer and Compuware QACenter. Rational Purify checks for memory leaks.

#### 1.8.4 *Disadvantage of Record and Playback Test Automation Tools*

In Section 1.8.3, the functionality of some of the most common software testing tools were outlined. However, none of those tools automatically and intelligently create test cases based on an understanding of what the software under construction is intended to execute. Some of those available tools record and playback test scenarios produced by the navigation of testers through the product while testing the product. In this case, coverage

relies on how well the tester or user goes through the different and possibly numerous system scenarios. The system responses have to be judged by the tester who determines whether the system passed the test or not, based on his understanding and knowledge of the specifications. The tester thus, also acts as the oracle who decides whether the system passed that specific test, generated by the scenario in question.

## 1.9 Related Literature

A number of researchers in the field of software testing emphasize the need and the advantages of having a machine readable mapping of software specifications [31, 34, 38]. Other researchers describe techniques to compare software to its formal specifications as a good means of software testing [35,39]. Other techniques, such as those found in [37, 40] can be deployed to work efficiently and automatically, given a software's specification in the form of a machine-readable format, as that described in this book.

In order to automatically test database operations, Chays et. al. in [27, 28] derived the software specifications from the database schema and program code, and a tester inputting relevant heuristics. The authors noted that it is not possible to fully automate the testing and verification process unless the formal specification is given in full, which is the focus of this book. A similar approach to testing automation was illustrated by [36], in which the authors depended on programmed embedded SQL statements in the code, assuming it to be the correct source upon which to generate test cases. The weakness of this approach and the latter, lie in the fact that the test cases are generated based on the resulting software not from the specifications, as would be the result of adopting the techniques in this book.

In [33], Hung expanded on the work of Chays et al. in [28], designing an input generator for database applications. Hung emphasized the need to compute the expected output automatically. Similar to Chays et al. in [28], Hung generated test cases based on the testers values for the system inputs in their respective equivalence classes. The test cases were not generated automatically based on the specifications. The research in [58, 59, 60, 61] did not address the problem of automatically checking the results of the tests to verify whether the results are correct compared to the specifications. A number of researchers investigated the automation of testing web database applications, including [62]. Also regression testing of database

applications was researched in [68]. Define-use path testing was used in [63] to test database driven applications. In [64] Cabal and Tuya tested the SELECT operation in database applications. Automated test case generation techniques were introduced in [32] to test the GUI of applications.

### **1.9.1 *AGENDA: A Test Generator for Database Applications***

Chays, Deng and Frankl carried out a research titled "AGENDA: A Test Generator for Relational Database Applications" [27, 28, 54, 55, 56]. The tests generated in their research were targeted towards database operations or application queries (insert, delete, update and select statements). There were no test cases targeted to test the application code itself, in which these operations are embedded. The tests generated in Chays' research depend on database input and output states, and do not compare the input or output to those specified by the design. That research does not tackle other types of operations where the output database state is not changed, but rather files, printed reports, magnetic card programming, etc. are the forms of output. Real world applications that use numerous types of inputs and outputs, besides database state changes will benefit only partially from AGENDA. AGENDA depends on the tester to specify whether the test passed or not. Accordingly, the tester is the oracle for AGENDA. AGENDA generates tests based on application code not the specifications and design of the software, and thus would not recognize errors of omission (parts that are designed but never programmed). When the tests are derived from the actual code, the tool will also not recognize errors of commission where the programmers wrote parts of code that were never included in the design document. Finally, AGENDA uses two types of functional testing techniques, namely, boundary value analysis and equivalence class testing. Not all faults in the software are revealed by these two types of testing alone. Boundary value analysis and equivalence class testing are good techniques to use in combination with other techniques to best test the software.

### **1.9.2 *Executable Software Specifications***

One of the major goals of this book is to represent software specification in a way that they can be reasoned with by automated software testing tools, to generate intelligent test cases. Intelligent test cases are a minimal set of useful test cases that collectively cover all the software and are capable of

identifying errors. Accordingly, it was necessary to review the literature for automated software specifications. In [96], Corning describes how Microsoft internally test their systems by using a technique that automates the execution of human entered software test case specifications. Accordingly, what is automated is the generation of test cases based on the specifications of each of the test cases. In other words, the test cases are not created using a computationally intelligent technique to generate interesting and important test cases, because the software specifications are not executable (though the test cases specifications are executable). In this situation, the testers are the ones that create test cases, which can be automatically re-run with different parameters. The parameters of each test case are the test case specifications, or test specifications in short. Those parameters are executable. Accordingly, automated tools can read the specification of a test case, including the input variable names, possible values, pre-conditions, as well as other factors, and create multiple test cases that match those specifications. Those multiple test cases can then be run automatically.

Formal specifications are not readily executable [97]. Fuchs believes that formal specifications should be executable if possible [98] and argues against Hayes and Jones in [97] who wanted to make the distinction between human-readable specifications and machine-executable prototyping. Fuchs also indicates that the executable specifications in his research [98] are at the same high level of abstraction, the aim of the possibly executable specifications is for rapid prototyping and to facilitate the validation process. In the early nineteen nineties, automated testing and automated test case generation was not the purpose of those techniques of trying to execute the specifications, the arguments were mainly targeted towards rapid prototyping, not automated testing.

In [99, 100], the authors describe a mechanism to transform formal specifications to executable code. The result is not however, a means of representing the specifications to an automated tool to reason about and generate intelligently test cases from it.

## 1.10 Objectives of this Book

After surveying today's market for available software testing tools, none were found to generate the tests independently from specifications. The available tools are either code-coverage tools or capture-playback tools . They all rely on the implemented software either to create tests based on

white box testing and the implemented code in question, or on the testers to provide good scenarios or test cases and their corresponding expected output. Accordingly, one can fairly make the assumption that currently the term "automatic testing" is misused to refer to automatic test execution and often misapplied to techniques that only partially automate portions of the testing process. After surveying the available tools, it was found that none of the tools in fact automatically generates the test cases for the software; meaning that, none of the available tools creates test cases based on its ability to interpret what the software under construction is implemented to achieve. The only tools close to automatically generating test cases, are those for stress testing [29], which create test cases to see how the software will react when a huge number of users log-in at the same time for example, or for database systems, creating a huge database with very big tables, etc. These tools, do not test the actual functionality of the software and whether it performs its tasks correctly or not, they just generate and execute stress tests. Basically, none of the available tools can understand the specifications and the software design, and accordingly create its own set of test cases that intelligently test and achieve good test coverage of the software.

The main reason why such an intelligent test case generator does not exist is that there does not exist a means of representing the specifications, requirements, and design in such a way that it could be interpreted, understood and used by an intelligent tool to generate test cases. Creating this machine readable and machine-interpretable representation of a system specifications and design is one of the main goals of this book. The new concepts introduced in this book are discussed in this section.

The goal of the concepts introduced in this book is thus to create a unified, machine readable representation of a software application's unit's design and specifications; whereby an automated testing tool could interpret and understand the software's units' design and specifications and automatically generate intelligent test cases to test the specified units of the software and determine whether the actual output matches the expected specified output; and to use the machine readable specifications in automated testing tools to show its use. One of those tools is to automatically generate constraints and enforce business rules for database applications at the database level. Another one of the automated testing tools is to use the specification representation to show errors of commission using a reverse engineering approach.

An automatic test case generator using the unified specification repre-

sentation (SpecDB) will guarantee good coverage of the software, since it can generate test cases for all the specified functionalities of the software, utilizing the specifications not the implementation. Such an intelligent test case generating tool can easily find errors of omission, where the programmers did not implement parts of the software. It can also spot errors of commission caused by programming unspecified functionality in the software. Both of the latter types of errors would never be spotted if the tests were not generated based on the program specifications and design. Finally, such an intelligent test-case generator can spot faults of commission caused by errors in programming a specified functionality incorrectly.

The goal of the concepts introduced in this book should not be misinterpreted as a requirements verifier. Requirements verifiers test the requirements document for clarity, consistency and completeness [30], however, they do not check the actual implemented software based on these requirements. A first step before starting to represent the specifications and the design in a way interpretable by a testing tool is to use a requirements verifier to check the accuracy of the specifications and the requirements [66]. Hence, the goal of the book is to represent (formal) software specifications in a machine readable format to aid in the testing and test result verification processes and to create useful testing tools that make use of the machine readable software specifications to intelligently guide the testing process and assure better software quality.

Accordingly, the first objective is to create a standard process for producing and storing UML or formal specifications and other software specification and design constructs and detailed functionality specification. In order to create an intelligent automated test case generation tool for software testing purposes, there has to be a way for the tool to understand what the software is built for, its functionalities, inputs, outputs, processing scenarios, etc. Accordingly, the objective of this book is to provide that standard way of representing specifications, in a way that could be interpreted by an automated test case generator. This representation of specifications is detailed in Chapter 2, SpecDB: A Database Representation of Software Specifications.

This book includes extensions to the work by Chays, Deng and Frankl on AGENDA[27,28]. One of the uses of SpecDB, is to create an automated tool to generate intelligent test cases from SpecDB based on its understanding of the functionality of the software under test, and test the application units containing both imperative-like code (programming constructs like if, while, etc. and abstract data types, etc.) and also including the database

operations. This proposed solution can compare actual program outputs to the specified design, stored in SpecDB. Therefore, the tool can automatically make the decision whether or not the test passed, based on its understanding of the available querable design, represented in SpecDB. Based on the specification, automated testing tools using SpecDB can spot errors of omission and commission immediately. In their research [27], Chays and his research team, wrote: "Unless there is a formal specification of the intended behavior of the application, including a specification of how the database state will be changed, it is not possible to fully automate checking of the application's results." Again, representing the formal specification they referred to, in a machine readable format, is the first objective of of this book, to make it possible to automate the checking of the software's results.

The second objective is to use the representation of the design and specifications to create an intelligent business rules generator for database applications. The proposed representation of the analysis and design of software will be based on a database structure that can be easily queried. Business rules and constraints will automatically be generated to enforce those constraints at the database level, not the application level.

The final part of the book is dedicated to studying the feasibility and effectiveness of structural test case generation, based on the level of detail given by the software specification and design. From the actual implementation, a reverse engineering approach is used to show all the software output, and post conditions after a test is executed. The output and post conditions are compared to the specified behavior if represented in SpecDB and accordingly, errors are automatically spotted and code segments that caused the errors are isolated. The automated structural testing techniques used in the reverse engineering automated testing approach are dataflow testing and path testing.

In Figure 1.2 the main concepts introduced in this book are represented. The design of SpecDB is detailed in Chapter 1.10. The software specifications are stored in SpecDB manually or automatically using the formal specification translation algorithm in Chapter 2.5.1. The usefulness of the SpecDB representation for automatic constraint generation and testing based on reverse engineering is demonstrated using two applications described in Chapters 3.5 and 4.3, automatic constraints generator and the reverse engineering testing tool, respectively. Finally in Chapter 5.4, enhancements to available tools in the literature like AGENDA, using SpecDB are proposed.

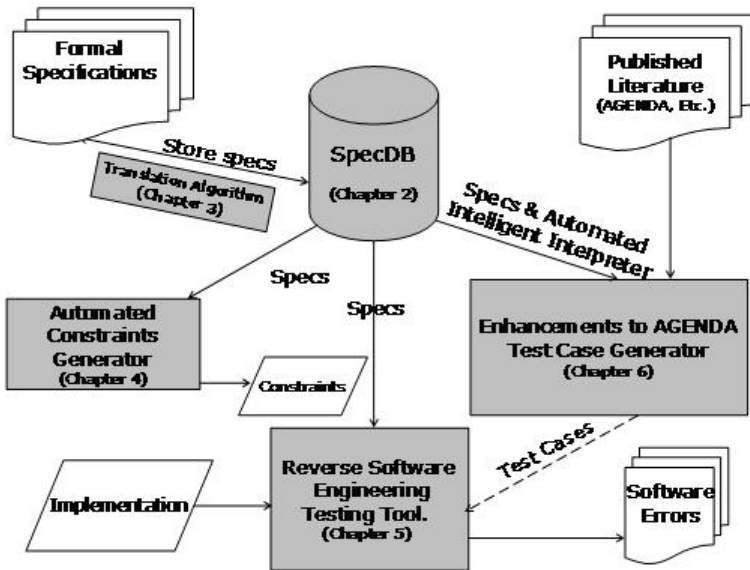


Fig. 1.2 Entities and Chapter Layout