1
Introduction and Overview

Software is critical to many aspects of our lives. It comes in many forms. The applications we install and run on our computers and smart phones are easily recognized as software. Other software, such as that controlling the amount of fuel injected into a car's engine, is not so obvious to its users. Much of the software we use lacks adequate quality. A report by the National Institute of Standards and Technology (NIST, 2002) indicated that poor quality software costs the United States economy more than $60 billion per year. There is no evidence to support any improvement in software quality in the decade since that report was written.

Most of us expect our software to fail. We are never surprised and rarely complain when our e-mail program locks up or the font changes we made to our word processing document are lost. The typical "solution" to a software problem of turning the device off and then on again is so encultured that it is often applied to problems outside of the realm of computers and software. Even our humor reflects this view of quality. A classic joke is the software executive's statement to the auto industry, "If GM had kept up with the computing industry we would all be driving $25 cars that got 1,000 miles per gallon," followed by the car maker's list of additional features that would come with such a vehicle:

1. For no apparent reason, your car would crash twice a day.
2. Occasionally, your engine would quit on the highway. You would have to coast over to the side of the road, close all of the windows, turn off the ignition, restart the car, and then reopen the windows before you could continue.
3. Occasionally, executing a maneuver, such as slowing down after completion of a right turn of exactly 97 degrees, would cause your engine to shut

down and refuse to restart, in which case you would have to reinstall the engine.
4. Occasionally, your car would lock you out and refuse to let you in until you simultaneously lift the door handle, turn the key, and kick the door (an operation requiring the use of three of your four limbs).

Why do we not care about quality? The simple answer is that defective software works "well enough." We are willing to spend a few hours finding a work-around to a defect in our software to use those features that do work correctly. Should the doctor using robotic surgery tools, the pilot flying a fly-by-wire aircraft, or the operators of a nuclear power plant be satisfied with "well enough"? In these domains, software quality does matter. These are examples of *high-integrity applications* – those in which failure has a high impact on humans, organizations, or the environment. However, we would argue that software quality matters in every domain. Everyone wants their software to work. Perhaps the biggest need for quality today is in the software security arena. In his newsletter article, *Security Changes Everything*, Watts Humphrey (2006b) wrote: "It is now common for software defects to disrupt transportation, cause utility failures, enable identity theft, and even result in physical injury or death. The ways that hackers, criminals, and terrorists can exploit the defects in our software are growing faster than the current patch-and-fix strategy can handle."

1.1 Obtaining Software Quality

The classic definition of the quality of a product focuses on the consumer's needs, expectations, and preferences. Customer satisfaction depends on a number of characteristics, some of which contribute very little to the functionality of the product.

Manufacturers have a different view of product quality. They are concerned with the design, engineering, and manufacturing of products. Quality is assessed by conformance to specifications and standards and is improved by removing defects. In this book, we concentrate on this defect aspect of quality.

> This is because the cost and time spent in removing software defects currently consumes such a large proportion of our efforts that it overwhelms everything else, often even reducing our ability to meet functional needs. To make meaningful improvements in security, usability, maintainability, productivity, predictability, quality, and almost any other "-ility," we must reduce the defect problem to manageable proportions. Only then can we devote sufficient resources to other aspects of quality. (Humphrey, 2006a)

Table 1.1. The software CMM (Paulk, 2009)

Level	Focus	Characteristics
1, Initial	None	*Ad hoc* or chaotic.
2, Repeatable	Project Management	The necessary process discipline is in place to repeat earlier successes on projects with similar applications.
3, Defined	Software Engineering	The software process for both management and engineering activities is documented, standardized, and integrated into a set of standard software processes for the organization.
4, Managed	Quality Processes	Detailed measures of the software process and product quality are collected. Both the software process and products are quantitatively understood and controlled.
5, Optimizing	Continuous Improvement	Continuous process improvement is enabled by feedback from the process and from piloting innovative ideas and technologies.

1.1.1 *Defect Rates*

In traditional manufacturing, quality is assured by controlling the manufacturing process. Statistical tools are used to analyze the production process and predict and correct deviations that may result in unacceptable products. Statistical process control was pioneered by Walter A. Shewhart in 1924 to reduce the frequency of failures of telephone transmission equipment manufactured by the Western Electric Company. After World War II, W. Edwards Deming introduced statistical process control methods to Japanese industry. The resulting quality of Japanese-manufactured products remains a benchmark for the rest of the world.

In 1987, the Software Engineering Institute (SEI), led by the work of Watts Humphrey, brought forth the notion that statistical process control could be applied to the software engineering process. SEI defined the Capability Maturity Model for Software (Software CMM) in 1991.[1] The Software CMM defines the five levels of process maturity described in Table 1.1. Each level provides a set of process improvement priorities.

There is a good deal of evidence to support the assertion that using better processes as defined by the Software CMM leads to programs with fewer defects. Figure 1.1 shows the typical rate of defects delivered in projects as

Table 1.2. Origin of defects

Study	Design and coding	Requirements and specification	Other
Beizer (1990)	89%	9%	2%
NIST (2002)	58%	30%	12%
Jones (2012, 2013)	60%	20%	20%

a function of the Software CMM level. The average rate of 1.05 defects per thousand lines of code (KLOC) obtained by engineers working at CMM level 5 appears to be a low number. However, this rate must be considered in the context of the large size of most sophisticated projects. It suggests that the typical million lines of code in an aircraft's flight management system is delivered with more than 1,000 defects. A NASA report on Toyota Camry's unintended acceleration describes the examination of 280,000 lines of code in the car's engine control module (NASA, 2011). Assuming this code was developed under the highest CMM level, the data in Figure 1.1 suggests that this code might contain nearly 300 defects. These numbers are too large for high integrity software.

To prevent or detect and remove defects before a software application is released, it is useful to understand where defects originate. Table 1.2 shows the estimates from three studies on the origins of defects in software. This data indicates that the majority of defects are created during the design and coding phases of development.

Verification and validation are names given to processes and techniques commonly used to assure software quality. Software *verification* is the process of showing that the software meets its written specification. This definition is

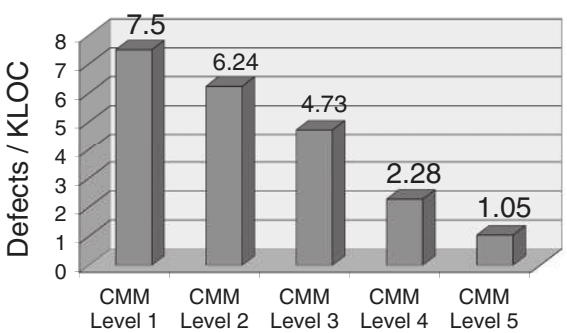


Figure 1.1. Delivered defects by CMM level (data from Jones [2000] and Davis and Mullaney [2003]).

commonly illustrated by the question, "Are we building the *product right*?" Verification is a means of demonstrating correctness. We use verification to locate and remove the defects from our design and implementation, the compiler, operating systems, and hardware on which we execute our application – defects that constitute the majority of those classified in Table 1.2.

Software *validation* is the process of evaluating an application to ensure that it actually meets the users' needs – that the specification was correct. This definition is commonly illustrated by the question, "Are we building the *right product*?" Validation is important in showing that we remove the defects originating in our specification (the third column of Table 1.2).

1.1.2 *Software Testing*

The verification strategies used to achieve the defect rates shown in Figure 1.1 are typically based on software testing. An in-depth coverage of software testing is beyond the scope of this book. For additional information, see Ammann and Offutt (2008), Black (2007), Jorgensen (2008), Kaner, Falk, and Nguyen (1999), or the classic testing book by Beizer (1990). There are two fundamental approaches to testing: black-box testing and white-box testing.

Black-box testing is based solely on the behavior of the program without any knowledge of coding details. It is also called behavioral testing or functional testing. Test cases are created from requirements given in the specification for the application. Black-box testing is usually performed on complete systems or large subsystems. It is often performed by people who did not write the software under test. These testers frequently have more knowledge of the application domain than of software engineering. There are many black-box testing tactics, including Black (2007):

- Equivalence classes and boundary value testing
- Use case, live data, and decision table testing
- State transition table testing
- Domain testing
- Orthogonal array and all pairs testing
- Reactive and exploratory testing

As black-box tests are derived entirely from the specification, they provide a means of verifying that our design, implementation, compiler, operating system, and hardware work together to successfully realize the specification. Black-box testing does not directly provide validation that our specification is correct. However, the testers' domain expertise is a valuable resource in finding errors in the specification during testing.

White-box testing is based on the actual instructions within the application. It is also called *glass-box testing* or *structural testing*. White-box tests are created from the possible sequences of execution of statements in the application. White-box testing is usually performed by programmers and may be applied to small units (unit testing) as well as to a combination of units (integration testing). The two basic tactics of white-box testing are control-flow testing and data-flow testing.

Control-flow tests are usually designed to achieve a particular level of coverage of the code. Commonly used code coverage tactics include:

- statement coverage;
- condition coverage;
- multicondition coverage;
- multicondition decision coverage;
- modified condition/decision coverage (MC/DC); and
- path coverage.

Data-flow tests add another dimension to control-flow testing. In addition to testing how control flows through the program, data-flow testing checks the order in which variables are set and used.

1.1.3 *Improving Defect Rates*

There are at least three reasons why testing alone cannot meet current and future quality needs. First, complete testing is almost always impossible. Suppose we would like to use black-box testing to verify that a function correctly adds two 32-bit integers. Exhaustive testing of this function requires 2^{64} combinations of two integers – far too many to actually test. With white-box testing, we would like to test every possible path through the program. As the number of possible paths through a program increases exponentially with the number of branch instructions, complete path coverage testing of a small program requires a huge effort and is impossible for most realistic-size programs. Good testing is a matter of selecting a subset of possible data for black-box tests and the determination of the most likely execution paths for white-box testing. That brings us to the second reason that testing alone cannot achieve the quality we need. Users always find innovative, unintended ways to use applications. We probably did not test the data entered or the paths executed by those "creative" uses of our application. Third, we now face a new category of user: one who is hostile. Our applications are under attack by criminals, hackers, and terrorists. These people actively search for untested data and untested execution paths to

exploit. As a result, we find ourselves updating our applications each time a security vulnerability is discovered and patched.

Watts Humphrey (2004) has suggested four alternative strategies for achieving defect rates below those obtained at Software CMM level 5.

Clean Room: This process was developed by Harlan Mills, Michael Dyer, and Richard Linger (1987) at IBM in the mid-1980s with a focus on defect prevention rather than defect removal. Defect prevention is obtained through a combination of manually applied formal methods in requirements and design followed by statistical testing. Quality results are ten times better than Software CMM level 5 results.

Team Software Process (TSP): A process-based approach for defect prevention developed by Watts Humphrey (2000). Quality results are more than ten times better than Software CMM level 5 results.

Correct by Construction (CbyC): A software development process developed by Praxis Critical Systems (Amey, 2002; Hall and Chapman, 2002). CbyC makes use of formal methods throughout the life cycle and uses SPARK for strong static verification of code. Quality results are 50 to 100 times better than Software CMM level 5 results (Croxford and Chapman, 2005).

CbyC in a TSP Environment: A process combining the formal methods of CbyC utilizing SPARK with the process improvements of the Team Software Process.

Both clean room and CbyC are based on formal methods. *Formal methods* are mathematically based techniques for the development of software. A formal specification provides a precise, unambiguous description of an application's functionality. Later in the development cycle, the formal specification may be used to verify its implementation in software. Although there has been much work over the years, formal methods remain poorly accepted by industrial practitioners. Reasons cited for this limited use include claims that formal methods extend the development cycle, require difficult mathematics, and have limited tool support (Knight et al., 1997).

In this book we introduce you to the SPARK programming language and how it may be used to create high-integrity applications that can be formally verified. Contrary to the claim that formal methods increase development time, the use of SPARK has been shown to decrease the development cycle by reducing testing time by 80 percent (Amey, 2002). A goal of this book is to show that with the tools provided by SPARK, the mathematics involved is not beyond the typical software engineer. We do not attempt to cover formal specification or

the software development processes defined by CbyC or TSP in which SPARK can play a critical role in producing high assurance, reliable applications.

1.2 What Is SPARK?

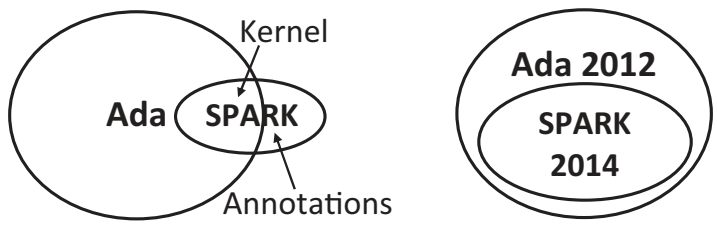
SPARK is a programming language and a set of verification tools specifically designed to support the development of software used in high-integrity applications. SPARK was originally designed with formally defined semantics (Marsh and O'Neill, 1994). *Semantics* refer to the meaning of instructions in a programming language. The semantics of a language describe the behavior that a computer follows when executing a program in that language. *Formally defined* means that SPARK's semantics underwent rigorous mathematical study. Such study is important in ensuring that the behavior of a SPARK program is unambiguous. This deterministic behavior allows us to analyze a SPARK program without actually executing it, a process called *static verification* or *formal verification*.

The information provided by the static verification of a SPARK program can range from the detection of simple coding errors such as a failure to properly initialize a variable to a proof that the program is correct. *Correct* in this context means that the program meets its specification. Although such correctness proofs are invaluable, they provide no validation that a specification is correct. If the formal requirements erroneously state that our autopilot software shall keep the aircraft upside down in the southern hemisphere, we can analyze our SPARK program to prove that it will indeed flip our plane as it crosses the equator on a flight from the United States to Brazil. We still need validation through testing or other means to show that we are building the right application.

In addition, verification of a SPARK program cannot find defects in the compiler used to translate it into machine code. Nor will SPARK find defects in the operating system or hardware on which it runs. We still need some verification testing to show that it runs correctly with the given operating system and hardware. But with a full analysis of our SPARK program, we can eliminate most of the verification testing for defects in the design and implementation of our application – the defects that constitute the majority of those listed in Table 1.2.

SPARK is based on the Ada programming language. SPARK's designers selected a restricted, well-defined, unambiguous subset of the Ada language to eliminate features that cannot be statically analyzed. They extended the language with a set of assertions to support modular, formal verification.

SPARK has evolved substantially over its lifetime. The first three versions, called SPARK 83, SPARK 95, and SPARK 2005, are based on the corresponding



(a) SPARK 83, SPARK 95, and SPARK 2005 (b) SPARK 2014

Figure 1.2. Relationships between Ada and SPARK.

versions of Ada (Ada 83, Ada 95, and Ada 2005). This book describes the current version – SPARK 2014 – which is based on Ada 2012.

The complete set of goals for SPARK 2014 is available in the *SPARK 2014 Reference Manual* (SPARK Team, 2014a). Some of the more important goals include the following:

- The SPARK 2014 language shall embody the largest subset of Ada 2012 to which it is currently practical to apply automatic formal verification.
  Prior to this version, SPARK executable statements were a small subset of Ada called the *Spark kernel*. A special non-Ada syntax was used to write *annotations* – formal statements used for the static verification of the program. SPARK 2014 uses the syntax available in Ada 2012 to write both executable statements and static verification statements called *assertions*. Preconditions, postconditions, and loop invariants are examples of assertions we shall look at in detail. The two Venn diagrams in Figure 1.2 illustrate the relationships between Ada and SPARK.
- SPARK 2014 shall provide counterparts of all language features and analysis modes provided in SPARK 83/95/2005.
- SPARK 2014 shall have executable semantics for preconditions, postconditions, and other assertions. All such expressions may be executed, proven, or both.
- SPARK 2014 shall support verification through a combination of testing and proof. Our programs can be written as a mix of SPARK 2014, unrestricted Ada 2012, and other languages. We can formally verify or use testing to verify those parts written in SPARK 2014. We must use testing to verify those parts not written in SPARK 2014.

Throughout this book, we use the name SPARK to refer to SPARK 2014.

1.3 SPARK Tools

SPARK comes with a set of tools for developing SPARK programs. A full description of the tools is available in the *SPARK 2014 Toolset User's Guide* (SPARK Team, 2014b). In this section we list and provide a very brief summary of these tools. More detailed descriptions of each tool are given in later chapters when appropriate.

1.3.1 *GNAT Compiler*

The GNAT compiler performs the tasks of a typical compiler:

- Checks that the program is in conformance with all of the Ada syntax and semantic rules.
- Generates the executable code.

The *SPARK 2014 Toolset User's Guide* (SPARK Team, 2014b) recommends that our first step in developing a SPARK program is to use the GNAT compiler *semantic check* tool to ensure that the code is valid Ada. Once we have completed the formal verification of our SPARK program, our final step is to use the GNAT compiler to generate the executable code.

For testing purposes, we can request that the compiler generate machine code to check any assertions (preconditions, postconditions, etc.) while the program is running. Should any assertion be found false, the exception `Assertion_Error` is raised. This capability allows us to perform tests of our assertions prior to proving them.

1.3.2 *GNATprove*

GNATprove is the verification tool for SPARK. It may be run in three different modes:

Check: Checks that a program unit contains only the subset of Ada that is defined for SPARK.

Flow: Performs a flow analysis of SPARK code. This analysis consists of two parts: a *data-flow analysis* that considers the initialization of variables and the data dependences of subprograms and an *information-flow* analysis that considers the dependencies or couplings between the values being passed into and out of a subprogram.[2]

Proof: Performs a formal verification of the SPARK code. Formal verification will point out any code that might raise a runtime error such as division by zero, assignment to a variable that is out of range of the type of the variable, incorrect indexing of arrays, or overflow of an arithmetic