Design for Electrical and Computer Engineers

Theory, Concepts, and Practice

Ralph M. Ford and Christopher S. Coulston

This document was prepared with LaTeX.

0.1 About the Authors

Ralph Ford obtained his Ph.D. and M.S. degrees in Electrical Engineering from the University of Arizona in 1994 and 1989 respectively. He obtained his B.S. in Electrical Engineering from Clarkson University in 1987. He worked for the IBM Microelectronics Division in East Fishkill, NY from 1989-1991, where he developed machine vision systems to inspect electronic packaging modules for mainframe computers. Ralph also has experience working for IBM Data Systems and the Brookhaven National Laboratory. He joined the faculty at Penn State Erie, The Behrend College in 1994. Ralph has experience teaching electronics and software design, as well as teaching the capstone design course sequence in the electrical, computer, and software engineering programs. His research interests are in engineering design, image processing, machine vision, and signal processing. Ralph is currently Director of the School of Engineering at Penn State Behrend. He also serves as a program evaluator for ABET. He was awarded a Fulbright Scholarship to study at the Brno University of Technology in the Czech Republic in 2005.

Chris Coulston obtained his Ph.D. in Computer Science and Engineering from the Pennsylvania State University in 1999. He obtained his M.S. and B.S in Computer Engineering from the Pennsylvania State University in 1994 and 1992 respectively. Chris has industry experience working for IBM in Manassas, VA and Accu-Weather in State College, PA. He joined the faculty at Penn State Erie, The Behrend College in 1999. He has experience teaching design-oriented courses in digital systems, embedded systems, computer architecture, and database management systems.

Chris' research interests are in Steiner tree routing algorithms and artificial life. He is currently an Associate Professor of Electrical and Computer at Penn State Behrend and also serves as Chairperson of the program.

Contents

0.2 Preface

This book is written for undergraduate students and teachers engaged in electrical and computer engineering (ECE) design projects, primarily in the senior year. The objective of the text is to provide a treatment of the design process in ECE with a sound academic basis that is integrated with practical application. This combination is necessary in design projects because students are expected to apply their theoretical knowledge to bring useful systems to reality. This topical integration is reflected in the subtitle of the book: Theory, Concepts, and Practice. Fundamental theories are developed whenever possible, such as in the chapters on functional design decomposition, system behavior, and design for reliability. Many aspects of the design process are based upon time-tested concepts that represent the generalization of successful practices and experience. These concepts are embodied in processes presented in the book, for example, in the chapters on needs identification and requirements development. Regardless of the topic, the goal is to apply the material to practical problems and design projects. Overall, we believe that this text is unique in providing a comprehensive design treatment for ECE, something that is sorely missing in the field. We hope that it will fill an important need as capstone design projects continue to grow in importance in engineering education.

We have found that there are three important pieces to completing a successful design project. The first is an understanding of the design process, the second is an understanding of how to apply technical design tools, and the third is successful application of professional skills. Design teams that effectively synthesize all three tend to be far more successful than those that don't. The book is organized into three parts that support each of these areas.

The first part of the book, the *Design Process*, embodies the steps required to take an idea from concept to successful design. At first, many students consider the design process to be obvious. Yet it is clear that failure to understand and follow a structured design process often leads to problems in development, if not outright failure. The design process is a theme that is woven throughout the text; however, its main emphasis is placed in the first four chapters. Chapter 1 is an introduction to design processes in different ECE application domains. Chapter 2 provides guidance on how to select projects and assess the needs of the customer or user. Depending upon how the design experience is structured, both students and faculty may be faced with the task of selecting the project concept. Further, one of the important issues in the engineering design is to understand that

systems are developed for use by an end-user, and if not designed to properly meet that need, they will likely fail. Chapter 3 explains how to develop the Requirements Specification along with methods for developing and documenting the requirements. Practical examples are provided to illustrate these methods and techniques. Chapter 4 presents concept generation and evaluation. A hallmark of design is that there are many potential solutions to the problem. Designers need to creatively explore the space of possible solutions and apply judgment to select the best one from the competing alternatives.

The second part of the book, *Design Tools*, presents important technical tools that ECE designers often draw upon. Chapter 5 emphasizes system engineering concepts including the well known functional decomposition design technique and applications in a number of ECE problem domains. Chapter 6 provides methods for describing system behavior, such as flowcharts, state diagrams, data flow diagrams and a brief overview of the Unified Modeling Language (UML). Chapter 7 covers important issues in testing and provides different viewpoints on testing throughout the development cycle. Chapter 8 addresses reliability theory in design, and reliability at both the component and system level is considered.

The third part of the book focuses on *Professional Skills*. Designing, building, and testing a system is a process that challenges the best teams, and requires good communication and project management skills. Chapter 9 provides guidance for effective teamwork. It provides an overview of pertinent research on teaming and distills it into a set of heuristics. Chapter 10 presents traditional elements of project planning, such as the work breakdown structure, network diagrams, and critical path estimation. It also addresses how to estimate manpower needs for a design project. Chapter 11 addresses ethical considerations in both system design and professional practice. Case studies for ECE scenarios are examined and analyzed using the IEEE (Institute of Electrical and Electronics Engineers) Code of Ethics as a basis. The book concludes with Chapter 12, which contains guidance for students preparing for oral presentations, often a part of capstone design projects.

Features of the Book

This book aims to guide students and faculty through the steps necessary for the successful execution of design projects. Some of the features are listed below.

- Each chapter provides a brief motivation for the material in the chapter followed by specific learning objectives.

- There are many examples throughout the book that demonstrate the application of the material.
- Each end-of-chapter problem has a different intention. Review problems demonstrate comprehension of the material in the chapter. Application problems require the solution of problems based upon the material learned in the chapter. Design problems are directly applicable to design projects and are usually tied in with the Project Application section.
- Nearly all chapters contain a Project Application section that describes how to apply the material to a design project.
- Some chapters contain a Guidance section that represents the author's advice on application of the material to a design project.
- Checklists are provided for helping students assess their work.
- There are many terms used in design whose meaning needs to be understood. The text contains a glossary with definitions of design terminology. The terms defined in the glossary (Appendix A) are indicated by ***italicized-bold*** highlighting in the text.
- All chapters conclude with a Summary and Further Reading section. The aim of the Further Reading portion is to provide pointers for those who want to delve deeper into the material presented.
- The book is structured to help programs demonstrate that they are meeting the ABET (accreditation board for engineering programs) accreditation criteria. It provides examples of how to address constraints and standards that must be considered in design projects. Furthermore, many of the professional skills topics, such as teamwork, ethics, and oral presentation ability, are directly related to the ABET Educational Outcomes. The requirements development methods presented in Chapter 3 are valuable tools for helping students perform on cross-functional teams where they must communicate with non-engineers.
- An instructor's manual is available that contains not only solutions, but guidance from the authors on teaching the material and managing student design teams. It is particularly important to provide advice to instructors since teaching design has unique challenges that are different than teaching engineering science oriented courses that most faculty are familiar with.

- PowerPoint™ presentations are available for instructors through McGraw-Hill
- There are a number of complete case study student projects available in electronic form for download by both students and instructors and available at. These projects have been developed using the processes provided in this book.

How to Use this Book

There are several common models for teaching capstone design, and this book has the flexibility to serve different needs. Particularly, chapters from the Professional Skills section can be inserted as appropriate throughout the course. Recommended usage of the book for three different models of teaching a capstone design course is presented.

- **Model I.** This is a two-semester course sequence. In the first semester, students learn about design principles and start their capstone projects. This is the model that we follow. In the first semester the material in the book is covered in its entirety. The order of coverage is typically Chapters 1–3, 9, 4–6, 10–11, and 7–8. Chapter 9 (Teams and Teamwork) is covered immediately after the projects are identified and the teams are formed. Chapters 10 (Project Management) and 11 (Ethical and Legal Issues) are covered after the system design techniques in Chapters 5 and 6 are presented. Students are in a good position to create a project plan and address ethical issues in their designs after learning the more technical aspects of design. Chapter 12 (Oral Presentations) is assigned to students to read before their first oral presentation to the faculty. The course concludes with principles of testing and system reliability (Chapter 7 and 8). We assign a good number of end-of-chapter problems and have quizzes throughout the semester. By the end of the first semester, design teams are expected to have completed development of the requirements, the high-level or architectural design, and developed a project plan. In the second semester, student teams implement and test their designs under the guidance of a faculty advisor.
- **Model II**. This two-semester course sequence is similar to Model I with the difference being that the first semester is a lower credit course (often one credit) taught in a seminar format. In this model chapters can be selected to support the projects. Some of the core chapters for consideration are Chapters 1–5, which take the student from project

selection to functional design, and Chapters 9–11 on teamwork, project management, and ethical issues. Other chapters could be covered at the instructor's discretion. The use of end-of-chapter problems would be limited, but the project application sections and example problems in the text would be useful in guiding students through their projects.

- **Model III**. This is a one-semester design sequence. Here, the book would be used to guide students through the design process. Chapters for consideration are 1–5 and 9–10, which provide the basics of design, teamwork, and project management. The project application sections and problems could be used as guidance for the project teams.

Acknowledgements

Undertaking this work has been a challenging experience and could not have been done without the support of many others. First, we thank our families for their support and patience. They have endured many hours and late evenings that we spent researching and writing. Melanie Ford is to be thanked for her diligent proofreading efforts. Bob Simoneau, the former Director of the Penn State Behrend School of Engineering, has been a great supporter of the book and has also lent his time in reading and providing comments. Our school has a strong design culture, and this book would not have happened without that emphasis; our faculty colleagues need to be recognized for developing that culture. Jana Goodrich and Rob Weissbach are two faculty members with whom we have collaborated on other courses and projects. They have influenced our thinking in this book, particularly in regard to project selection, requirements development, cost estimation, and teamwork. We must also recognize the great collaborative working environment that exists at Penn State Behrend, which has allowed this work to flourish. Our students have been patient in allowing us to experiment with different material in the class and on the projects. Examples of their work are included in the book and are greatly appreciated. John Wallberg contributed the disk drive diagnostics case study in Chapter 11 that we have found very useful for in-class discussions. John developed this while he was a student at MIT. Thanks to Anne Maloney for her copyediting of the manuscript. The following individuals at McGraw-Hill have been very supportive and we thank them for their efforts to make this book a reality – Carlise Stembridge, Julie Kehrwald, Darlene Schueller, Craig Marty, Kris Tibbetts, and Mike Hackett.

Finally, we would like to thank the external reviewers of the book for their thorough reviews and valuable ideas. They are Frederick C. Berry

(Rose-Hulman Institute of Technology), Mike Bright (Grove City College), Geoffrey Brooks (Florida State University Panama City Campus) Wils L. Cooley (West Virginia University), D. J. Godfrey (US Coast Guard Academy), and Michael Ruane (Boston University).

We hope that you find this book valuable, and that it motivates you to create great designs. We welcome your comments and input. Please feel free to email us.

Ralph M. Ford,

Chris S. Coulston,

Chapter 1

Testing

A stitch in time saves nine.—Anonymous

Most systems undergo testing throughout their development and before they are delivered to the customer. Clearly, systems should be tested to ensure that they meet the engineering requirements. In fact, one of the desirable properties of an engineering requirement is that it be verifiable, or in other words, testable. The philosophy of testing is embodied in the quote above, which means that it is better to correct errors early, rather than wait until they become much larger problems later. As we saw in Chapter 1, the cost to correct problems increases exponentially with the lifetime of the project. Thus, testing should be considered throughout system development.

Testing means different things to different people. A field service technician, assembly line worker, and designer will have their own definitions and requirements from a test. In this chapter testing is examined from the perspective of a systems designer intent on checking that the system meets the engineering requirements. Along the way fundamental testing concepts like controllability and observability are explored. Approaches to debugging systems are provided, followed by templates for building unit tests, integration tests, and acceptance tests.

Learning Objectives

By the end of this chapter, the reader should:

- Understand the concepts of black box tests, white box tests, observability, and controllability.

- Understand the principles of debugging.
- Understand when a unit test is used and how it is constructed.
- Understand when an integration test is used and how it is constructed.
- Understand when an acceptance test is used and how it is constructed.

1.1 Testing Principles

The design process is really a continual increase in specificity from engineering requirements to the detailed design. We now consider the question of how to test that the resulting system meets the design requirements. One answer is based on a common testing model, the "test vee", shown in Figure 1.1. This model starts with the engineering requirements, proceeds to the implementation, and then onto testing. It emphasizes that every level of design has a corresponding level of test. What is not so clear from this model is that the testing process is actually split between the two halves of the test vee. Students typically think of testing as being exclusively confined to the right half of the test vee – build it then you test it. However, each test performed in the right half of the test vee must be carefully engineered during the development of the system in the left side of the test vee. An acceptance test plan should be written with the Requirements Specification, integration tests defined and written during the system design, and so forth.

In our enthusiasm to complete a project many of us all too often rely on a "smoke test" – turn on a system to see if it works. The name of this test is a reference to what may happen to the system if the test fails – it burns up and smokes. Beyond being a potentially expensive way to test a system, a smoke test is not a systematic approach to verify that the system behaves as expected. Customer are not be impressed with "hey it didn't catch on fire!" as the test result. Clear tests need to be developed because:

- The test cases define exactly what the module must do.
- Testing prevents feature creep, since the development of a module is complete when its test is passed.
- Test cases motivate developers by providing immediate feedback.
- Test cases force designers to think about extreme cases.
- Test cases are a form of documentation.

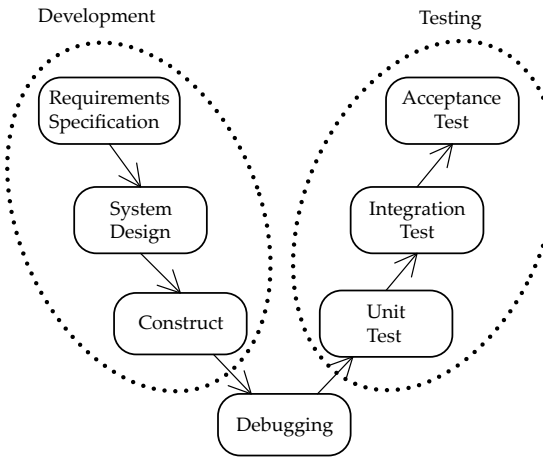

Figure 1.1: The Test Vee. Design stages are on the left and corresponding tests are on the right.

- Test cases force the designer to consider the design of the module before building it.

The test suite and its accompanying documentation contain important information about the behavior and organization of a system and its module. This gives tests a value beyond a role in showing that the system and its modules do not fail the tested conditions. The individual test cases show others engineers how to properly interface to a module, making that module more reusable. In addition, test documents can be used by other individuals in the organization like technical writers, maintenance technicians, and technical trainers.

How should testing be done? Given enough time, a system could be tested by simply enumerating every conceivable input and observing the outputs. While in some cases this might be possible, in general it would take an unreasonable amount of time to perform such a test. Instead, tests are crafted in order to maximize the likelihood of finding errors.

1.1.1 Types of Testing, Observability, and Controllability

Tests fall into two general types—black box and white box tests. ***Black box tests*** are those that are performed without any knowledge of the systems internal organization. In a black box test, the testing is typically conducted by changing the inputs and comparing the system outputs to their expected

values. The input and output values can be classified as typical, boundary, extreme, and invalid. These categories are illustrated by considering a system which converts Celsius temperatures to Fahrenheit. Typical inputs are values experienced during normal operation, say room temperature. Boundary values are encountered whenever the input or output changes in some significant way. For example, 0°C and -33.3°C mark the transition between positive to negative temperatures in Celsius and Fahrenheit respectively. Absolute zero represents an extreme value, because things can't get any colder. While these tests could be accomplished by enumerate every possible input to the system and observing the output, this would take an unreasonable amount of time. Hence, the test writer must elect candidate inputs to represent the behavior of the system over a range of possible inputs. An important goal of the test writer is to minimize the number of these equivalence classes while maximizing coverage of the input domain. Without a clear understanding of the internal organization of the system this is a challenging goal.

White box tests are conducted with knowledge of the internal working of the system. The idea of white box testing is to build tests which target specific internal nodes of the system to check that they are operating as expected. The tests should be written to check the node can handle typical, boundary, extreme and illegal situations.

One of the many goals in designing a system is to increase its testability. A design is ***testable*** when a failure of a component or subsystem can be quickly located. A testable design is easier to debug, manufacture, and service in the field. One way to increase the testability of a system is to increase controllability and observability. ***Controllability*** is the ability to set any node of the system to a prescribed value. ***Observability*** is the ability to observe any node of a system. In black box testing, both controllability and observability are low. In white box testing, controllability and observability may be higher depending on the design.

Let's examine this further via the example of a simple transistor amplifier shown in Figure 1.2. The purpose of this circuit, known as the common-emitter amplifier, is to amplify the input signal, v_i, to produce a linearly proportional output signal, $v_o = Axv_i$. The rectangular boundary in the figure represents a black box view of the system. In this view, the system power, V_{cc}, and ground would be applied to activate the circuit. The black box testing would consist of checking supply and ground voltages, varying the input signal, and observing the output signal. Again, this is a low controllability and low observability situation.

White box testing utilizes knowledge of the internal workings of the

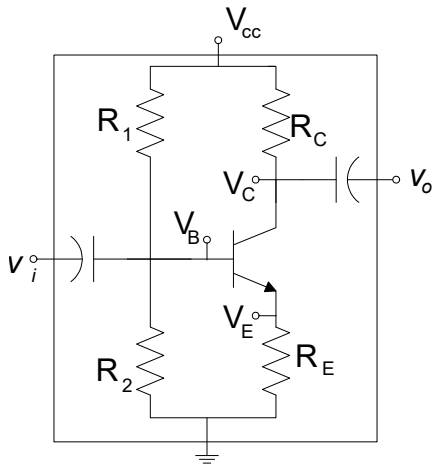

Figure 1.2: Transistor amplifier design.

design. When designing a transistor amplifier, there are two major points to consider—the DC bias voltages in the circuit and its AC, or time varying, amplification behavior. The two behaviors are related since the AC behavior depends upon proper DC biasing of the circuit. During detailed circuit design, the expected DC voltages for different nodes in the circuit would be determined. Thus, a white box test would consist of first checking the power supply and ground voltages as was done in the black box case. The next step would be different in that the node voltages (V_B, V_C, V_E) would be checked to see if they meet the expected design values. This indicates a high degree of observability. However, the controllability is not significantly better than in the black box case. This is because the internal DC node voltages in the circuit cannot arbitrarily be changed without negatively changing the operation of the circuit.

1.1.2 Stubs

A ***stub*** is a device that is used to simulate a subcomponent of a system. This might be done for two reasons, either the subcomponent has not yet been built, or the risk of damaging the subcomponent warrants using a stand-in. Typically, stubs are used to simulate inputs or monitor outputs of the ***unit under test*** (UUT). Both hardware and software stubs can be used when designing a system. In software testing stub routines are developed to either call other functions or act as those to be called by the unit under test.

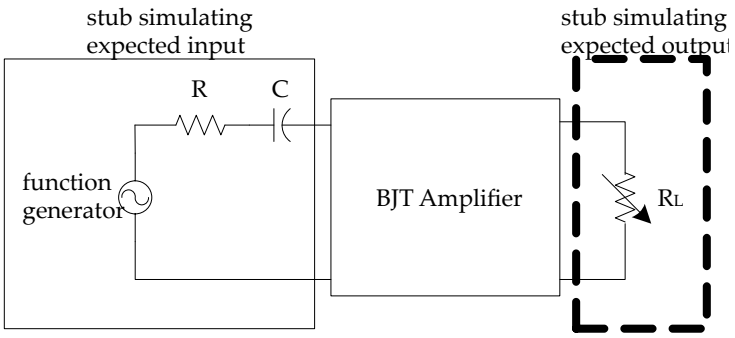

Figure 1.3: The use of stubs for testing a transistor amplifier circuit. The function generator, resistor (R), and capacitor (C) model the expected behavior of the input source in the final system implementation. The variable resistance (R_L) models the load that would be attached to the output.

Consider a hardware example, the transistor amplifier in Figure 1.2. Assume that the circuit is ultimately to be integrated into a larger system. The input to this system is a time-varying source with certain resistive and capacitive characteristics, while the output is connected to another system with a known input resistance range. The stubs used for testing in this system are shown in Figure 1.3. On the input side is a function generator, an off-the-shelf component, connected to a resistor and capacitor that models the expected characteristics of the final system. The stub on the output side is simply a resistor, whose value can be varied over the expected load.

1.1.3 Test Case Properties

As we go through the different levels of testing we will need to build effective test cases. Effective test cases share some common attributes regardless of their level. Dianne Runnels [Run99] defined the following properties for effective test cases:

- *Accurate.* The test should check what it is supposed to and exercise an area of intent.
- *Economical.* The test should be performed in a minimal number of steps.
- *Limited in complexity.* Tests should consist of a moderate number (10-15) of steps.

- *Repeatable.* The test should be able to be performed and repeated by another person.
- *Appropriate.* The complexity of the test should be such that it is able to be performed by other individuals who are assigned the testing task.
- *Traceable.* The test should verify a specific requirement. The corresponding requirements for the different types of test are derived from the associated development stages in the test vee in Figure 7.1.
- *Self cleaning.* The system should return to the pre-test state after the test is complete.

1.2 Constructing Tests

This section presents the four different types of test shown in Figure 7.1 - debugging, unit testing, integration testing, and acceptance testing. This is presented in reverse order from the order in which test should be created, as the reader is probably most familiar with basic test techniques such as debugging. Thus the presentation is from the most familiar to the more abstract. The next section presents a case study that proceeds in the opposite direction from acceptance testing to unit testing.

1.2.1 Debugging

At some point in the design process, the implementation level must be reached, where tasks such as constructing circuits, wiring integrated circuits, and writing code are carried out. Applying the functional decomposition paradigm introduced in Chapter 5 should provide a clear idea of the inputs, outputs, and behavior of the modules that are being built. Inevitably, there will come a point during the construction of a component when it will not function as expected. This is commonly referred to as a ***bug***. It requires the application of debugging skills to determine the root cause of the problem and correct it. You have undoubtedly run across a variety of bugs in your day, and it is a good guess that your bugs fell into one of two camps—Bohrbugs and Heisenbugs.

Bohrbugs are named after the Bohr model of the atom that assumes that electrons have a distinct position in space. Bohrbugs are reliable bugs, in which the error is always in the same place. This is analogous to the electrons having a definite position. Given a particular input, a Bohrbug will always manifest itself in the same way and in the same place. Finding

a Bohrbug is a matter of laying the correct trap. A good trap is simple to set-up, quickly causes an error, and reveals the source of the error. This is a tall order, but one which experience hones.

Heisenbugs are named after the Heisenberg Uncertainty Principle, in which the position of an electron is uncertain. Analogously, Heisenbugs may not always be reproducible with the same input. They seemingly move around within a system and are consequently difficult to locate. Finding a Heisenbug requires you to think outside the box because they usually result from unanticipated mechanisms. An example of a Heisenbug is a computer program with a pointer error that occasionally overwrites the system stack. This can cause return values from a subroutine to be incorrect. In such a case, the subroutine would appear to have a problem, since it is returning the wrong value. However, testing the subroutine by itself would confirm that the subroutine works properly. Another good example is a circuit that works fine on some days, but doesn't work on others (typically when a professor is nearby). Insidious problems such as a floating ground line often are to blame.

Regardless of the bug type, the debugging process is iterative. You must run tests and depending on the results, go back and run new tests. With this in mind, you should enter into the debugging process with a strategy in mind. This strategy is often similar to programming an if-then structure—"if the test is negative, then I'll pursue this line of attack; otherwise the error could be in another subsystem." In general, the debugging process is much the same as the scientific method. The steps of the debugging process are:

- Observe. Observe the problem under different operating conditions.
- Hypothesize. Form a hypothesis as to what the potential problem is.
- Experiment. Conduct experiments to confirm or eliminate the hypothesized source of the problem.
- Repeat. Repeat until the problem is eliminated.

When hypothesizing, make sure to check the simplest and easiest potential problems first. There are two good reasons for this—they are easy to perform and more tests can be performed in a given period of time. In addition, designs should be verified from the lowest levels of abstraction to the highest. For example, voltages should be verified as correct before moving

to higher levels of functionality. The reason for this heuristic is obvious—the higher level of functionality cannot operate correctly unless all the lower levels are working.

1.2.2 Unit Testing

A ***unit test*** is a complete test of a module's functionality. In order to be a complete check, a unit test consists of a set of test cases each of which establishes that the module performs a single unit of functionality to some specification. Test cases should be written with the express intent of uncovering undiscovered defects. For example, consider a hardware module which converts an input Celsius temperature into an output Fahrenheit temperature. Let the operation of the module be represented by the following pseudo-code.

```
if (16 < input < 32)
        output = ROM[input -16];
else
        output = (2 * input) + 32;
```

When the input temperature is between 16 and 32, the output is determined by a lookup operation in a ROM, otherwise the input is converted using an approximation to the familiar Celsius to Fahrenheit conversion. Each test case for this hardware module should exercise a single area of intent. Clearly, we need to have at least two test cases, one for the "if" clause and one for the "else" clause. In addition, it would be a good idea to check the boundary conditions separating the "if" and "else" clauses. Finally, we should consider the extreme values of the input. For example, if the input were a signed 8-bit number then we should check -128, and 128. If the input is a signed value then 0 is also a boundary value that should be checked.

This example illustrates the concept of a ***processing path*** – a sequence of consecutive instructions or states encountered from the beginning to the end of a computation pr process. The temperature conversion example has two processing paths, one where the "if" statement is taken and one when the "else" statement is taken. Each such processing path through the system represents a potential test case. The extent to which the test cases cover all possible processing paths is called the ***test coverage.*** It is desirable to design test sets that have the highest coverage as possible in the fewest number of test cases. The ultimate in coverage is achieved by ***path-complete coverage*** where every possible path has a test. However

this level of coverage may not be possible because the number of processing paths goes up exponentially with the number of nested branches. In cases where there are more paths than it is possible to check, you must be satisfied with partial path coverage. In such cases, those paths that which are though to most likely reveal an error should be tested.

Clearly documenting unit tests has added importance because the test cases are generally written by one person or group and performed by a separate group. In order to organize the test cases they can be organized as matrices, step-by-step tests or automated scripts.

Matrix Tests

A ***matrix test*** is a test that is best suited to cases where the inputs submitted are structurally the same and differ only in their values. The test procedure is then "factored out" leaving a list of inputs and their expected outputs. Since the tests are written by one group and performed by another the test writer must leave space in the test document for the tester to make comments and observations regarding the system behavior.

Lets consider a test for the analog-to-digital converter (ADC) that was used in the temperature measuring system in Chapter 5 (Section 5.7, Figure 5.11). Assume that the ADC's clock frequency is 10 kHz and the input ranges from 0 to 5 volts. The unit test will consist of submitting different inputs to the ADC and verifying the outputs. Since each test only varies the input, with no change in the testing procedure, the test matrix in Table 7.1 was created.

This test case exercises each bit of the ADC's output independent of the other output bits. Other test cases should examine extreme inputs as well as illegal inputs. Care should be taken that illegal inputs do not stress the ADC beyond the manufactures recommendations; otherwise the tests might accidentally damage the ADC.

Table 7.1 A matrix test for an analog-to-digital converter.

Step-by-Step Tests

A ***step-by-step test*** case is a prescription for generating the test and checking the results. These descriptions are most effective when the test consists of a complex sequence of steps. The test template for a step-by-step test has the all information contained in the matrix test template the difference

Table 1.1:

Test Writer: Sue L. Engineer				
Test Case Name:	ADC unit test			
Description:	Verify that each bit of the output can be set independently of the other outputs.			
Tester Information				
Name of Tester:				
Hardware Ver:	1.0			
Setup:	Isolate the ADC from the system by removing configuration jumpers. Connect the			
Test	V_T	**Expected output**		
1	0.000 V	0	0x000	
2	0.004887 V	1	0x001	
3	0.00977 V	2	0x002	
4	0.01955 V	4	0x004	
...	...	...	...	
10	2.502 V	512	0x200	
Overall test result:				

being the addition of a column in the test section describing what action the tester should perform at each step in the test process.

As an example, recall the state diagram for the vending machine in Chapter 6 (Figure 6.2) that accepts nickels and dimes and dispenses candy when a total of $0.25 (or more) is submitted. The state machine has different processing paths, depending upon the combination and order of coins deposited. Test cases can be written for each of the processing paths through the system, and an example is shown in Table 1.2 for one particular processing path.

Automated Test Scripts

An ***automated test script*** is a sequence of commands provided to the UUT without user intervention. The outputs are usually automatically compared against the expected outputs to determine if the module contains an error. Automated scripts are executed from a device referred to by many different names like test harness, test fixture, and test bench.

While automated scripts carry a lot of up front cost in terms of the time required putting them together they pay dividends when performing

Table 1.2: A step-by-step test for a vending machine.

Test Writer: Sue L. Engineer					
Test Case Name:	Finite State Machine Path Test #1				
Description:	Simulate insertion of money with a mix of nickels and dimes. Verifies FS				
Tester Information					
Name of Tester:					
Hardware Ver:	1.0				
Setup:	Make sure that the system was reset sometime prior and is in state $0.0				
Step	**Action**	**Expected Result**	**Pass**	**Fail**	**Comments**
1	Strobe Nickel	State should go to $0.05			
2	Strobe Dime	State should go to $0.15			
3	Wait	State should remain $0.15			
4	Strobe Nickel	State should go to $0.20			
5	Strobe Dime	State should go to $0.25			
6	Nothing	State should go to $0.00			
Overall test result:					

regression testing. Regression testing is the process of retesting a module following a modification in any related part of the system to ensure that no errors were inadvertently introduced. Reducing the time spent on regression testing will has a positive effect on the overall development time. Hence, the benefit of automated scripts is realized later in the testing cycle. In addition, design decision can have an effect on the amount of time spent on regression testing. It stands to reason that systems with highly coupled modules require more extensive and consequently more time consuming regression testing.

The template for the matrix tests could be used to describe what an automated test script does. However, the specifics of how the automated scripts perform these actions are implementation specific. For example, in hardware description languages the stimulus and responses of the UUT are processed by a test bench. The test bench is itself a piece of hardware coded in the same hardware language used to describe the UUT.

1.2.3 Integration Testing

After the individual subsystems have undergone their unit tests, they are then integrated into large subcomponents leading eventually to the construction of the entire system. Hence ***integration testing*** checks that the major modules of the overall system operate correctly together. The test cases for integration testing must be traceable to the high-level design, and the test cases are written based on characteristics of the design architecture. Test cases for integration can be derived from the following questions:

- Have all the execution paths through the system been exercised?
- Have all the modules been exercised at least once?
- Have all the interface signals been tested?
- Have all interface modes been exercised?
- Does the system meet timing requirements?

The integration tests themselves can be documented using either the matrix or step-by-step template outlined for unit tests.

1.2.4 Acceptance Testing

An ***acceptance test*** is a formal document stipulating the conditions under which the customer will accept the system. It generally consists of a suite

of test cases that exercise the systems according to the user's environment. The test cases are constructed to ensure that the engineering requirements are met. The four attributes of a good requirement (abstract, unambiguous, traceable, and verifiable) are important in building a good acceptance test. An unambiguous requirement will result in a test which everyone can agree on. A verifiable requirement sets an objective pass/fail criterion on the acceptance test. Tests based on a traceable requirement imply they are directly assessing the needs of the project. However, an acceptance test goes far beyond an enumeration of the test cases. It typically includes the following sections:

- Testing Approach. The types, level and methods employed to test the system.
- Test Schedule. Start and end dates for the individual tests.
- Problem Reporting. How the test results will be recorded.
- Resource Requirements. The hardware, software, and people requirements needed to perform the tests.
- Test Environment. The setup required to run the tests.
- Test Equipment. Any special equipment or configurations required to run the test.
- Post-Delivery Tests. Tests performed on the deployed system.
- Test Identification. Enumeration of test cases and their unique identifiers.
- Corrective Action. What repairs must be made to the system in order to accept it.

It is not necessary for every test case to be passed in order for the system to be accepted. The acceptance test should stipulate the degree of importance surrounding each test. While it's easy to imagine writing the test cases for an acceptance test, the process can become a chicken-and-egg problem. That is you are trying to stipulate the test procedures and results for a system which has yet to be implemented of the system. This can often lead to revisions of the acceptance test plan later in the design cycle.

1.3 Case Study: Security Robot Design

In order to demonstrate the concepts involved in testing lets consider the design of a security system which monitors an office complex looking for intruders. The design team has decided to address the need by designing a mobile robot which autonomously navigates its way through the office space. The team, along with the customer, developed a number of requirements and from this we will focus on two that address a fundamental navigational problem.

- *The robot's center must stay within 12 to 18 centimeters of the wall over 90% of the course, while traveling parallel to a wall over a 3 meter course.*
- *The robot's heading should never deviate no more than 10 degrees from the wall's axis, while traveling parallel to a straight wall over a 3 meter course.*

This case study explores test cases for the acceptance, integration and unit testing related to these two requirements. The development of the test cases follows the proper order of test case development illustrated in Figure 7.1. That means acceptance tests are developing in conjunction with the requirements, integration tests are constructed during the system design, and unit tests during the system build.

Acceptance Testing

We start by constructing an acceptance test case to verify that the robot can achieve the stated requirements. A number of tests would need to be built and we create a test only for the first engineering requirement. A test could be performed by having someone observe the robot moving along a wall and mark (on the floor) whenever the robot strayed out of bounds. Such a test would not easily be repeatable because different people might judge what is meant by "out of bounds" differently. The accuracy of such a test is questionable because determining when and if a speeding robot crossed the boundary is difficult. Finally, there would be no permanent record of the test results making it difficult for the customer to actually verify the test was passed. A way to address these problems is to have the robot monitor its own distance from the wall. This is done in this case by a program written to monitor the position of the robot over time and store these values. From the specifications for a step-by-step acceptance in Table 1.3 it is clear what

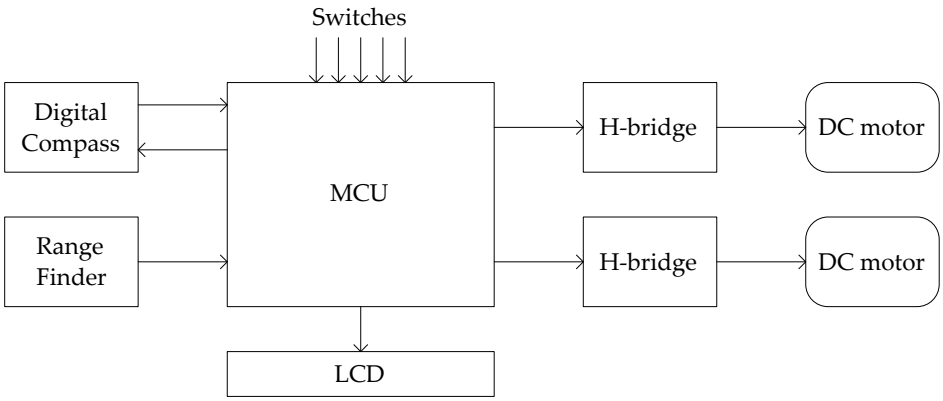

Figure 1.4: Level 1 design architecture for the mobile robot.

the test program must configure the robot to log the distance data while traversing the wall. After this data is downloaded from the robot it can be analyzed in a spreadsheet program to determine the needed metrics and archived for future reference.

Integration Testing

The team, in consultation with the customer, has gone through the requirements and created a complete set of acceptance tests in addition to the test in Table 7.3. They next turn to developing a high level design architecture that can meet the requirements. The design they create is shown in Figure 1.4, the Level 1 architecture of the autonomous robot.

The heart of the design is a microcontroller (MCU) which reads the sensor values, makes decisions, and controls the speed of the two drive motors. The robot moves and turns by adjusting the relative speed of each motor using a pulse-width modulated (PWM) signal from the MCU. The duty cycle of the PWM is directly proportional to the speed of the motor. The H-bridges then amplify the MCU output to power to the motors. The MCU also outputs a set of signals to send text to an LCD. The signal to the digital compass is bidirectional because the MCU must configure the operating mode of the compass before using it. The MCU receives an analog input from the range finder, where the voltage level of the signal is proportional to the distance to the obstacle. Finally, a set of switches are included that allow for manual input and testing of the robot.

Clearly, the interaction of the MCU with each of the compass, rangefinder,

Table 1.3: A step-by-step acceptance test case for the autonomous robot.

Test Writer: Sue L. Engineer					
Test Case Name:	Robot Acceptance Test #1				
Description:	The robot's center must stay within 12 to 18 centimeters of the wall over 90% of t				
Tester Information					
Name of Tester:					
Hardware Ver:	Robot 1.0				
Setup:	Completed robot should be fully charged and placed on 3 meter test track.				
Step	**Action**	**Expected Result**	**Pass**	**Fail**	**Comments**
1	Write a program to monitor the robots position from the wall.	Program should be statically tested to verify accuracy. Should sample wall at a sufficient rate depending on speed.			
2	Put robot on test track, run test, and download data.	The robot should travel down the entire length of the test track and then stop.			
3	Plot test data in a spreadsheet program.	Plot of position vs. time should be within 12 – 18 cm 90% of the time.			
Overall test result:					

LCD, switches, and H-bridges should be examined. However, this will be left to the unit test because the MCU makes a great test harness which can be used to provide stimulus to and read outputs from these I/O. In addition, many of the routines from these tests can be reused later in the development process.

There are many interactions between subsystems that could be tested during integration testing. A careful examination of the system must be done to determine which combinations of subsystems are most likely to create problems. Experience and component selection play a large part in molding expectations. For example, the magnetic field created by the windings in the DC motor can affect the reading generated by the compass. This interaction could potentially affect the headings read by the MCU and cause the robot to go off course by more than the allowable 10 degrees. Thus a step-by-step integration test is created in Table 1.4 to test the operation of the motors with the magnetic compass.

It is clear from this test case that a testing program needs to be written in order to prompt the user to align the robot, capture compass readings, and then to spool them back to the user. The requirement that the compass readings deviate no more than 10 degrees is based on the engineering requirement that the robot deviate no more than 10 degrees from the walls axis while navigating down the hallway.

Unit Testing

Once the Level 1 architecture is developed and the test cases written to ensure that the architecture is capable of meeting the design requirements, the design team moves on to selecting components to use in the design. The design team must select the units so that the resulting system can meet the engineering requirements. Each of the individual components Figure 1.4 needs to be considered as a candidate for unit testing. In general each functional unit might have several test cases comprising its unit test. A unit test for the digital compass component will illustrate this. Prior to presenting the test case, the functional design requirements for the unit are given in Table 1.5.

In order to be useful in the overall design the compass module must be able to accurately report the robot's heading. The requirements place an upper bound of 10 degrees on the error in the heading of the robot. Thus, the matrix test case is Table 1.6 is constructed to configure the compass and then reads heading data from it. This unit test looks for heading errors greater than 10 degrees.

Table 1.4: A step-by-step integration test case for the compass and motors.

Test Writer: Sue L. Engineer						
Test Case Name:	Robot Integration Test #1				**Test ID #:**	
Description:	Checks interaction of DC motors on the magnetic compass.				**Type:**	
Tester Information						
Name of Tester:					**Date:**	
Hardware Ver:	Robot 1.0				**Time:**	
Setup:	A wooden turn-table should be placed on top of the cardinal direction map. This r					
Step	**Action**	**Expected Result**	**Pass**	**Fail**	**Comments**	
1	Write program to spool compass readings while simultaneously driving motors.	Program should be statically tested to verify accuracy. Should sample compass at a sufficient rate depending on speed.				
2	Run acceptance test	Test program should prompt user to turn the robot to an orientation and then spin the motors will then spin up and down.				
3	Plot spooled data in spreadsheet program.	Plots should be analyzed to see if compass deviated any more than 10 degrees from				

Table 1.5: The functional requirements for the digital compass.

Module	Digital Compass – Geosensor version 2.3
Inputs	• Earth's magnetic field: An orientated field of magnetic force beginning and ending at the earth's magnetic poles. • SClk – Clock signal to clock data through the module. Maximum Frequency is 10Mhz. • SDIn – Serial data input to send data into the compass module. Date is valid on positive SClk edges.
Outputs	• SDOut – Serial data output from the compass module. Data is valid on negative clock edges.
Functionality	Senses the earth's magnetic field and determines the orientation of the compass with respect to the field. This orientation is stored in an internal register and can be retrieved through the SPI interface.
Test	Comp-UT-01

Table 7.6

1.4 Guidance

Tests have a lifetime beyond the obvious need to check proper operation of the subsystems, their integration, and the overall performance of the system. Test cases describe how the system operates in plain English. Test cases can be used to develop diagnostics, assist in writing technical documentation, and aid marketing and sales staff in understanding system performance. Testing is a value-added process in design. Beyond attempting to remove bugs from the system, Burke and Coyner [Bur03] suggest the following are good reasons to perform testing:

- *Testing reduces the number of bugs in existing and new features.* Testing does not eliminate all the bugs, but rather reduces the probability of a bug making it to production.
- *Tests are good documentation.* Tests provide insight to others on the operation of the unit under test and how to interface to it.

Table 1.6: Matrix unit test for the digital compass.

Test Writer: Sue L. Engineer	
Test Case Name:	Compass Unit Test #1
Description:	Checks that the compass returns correct angular measurements to the MCU. Test
Tester Information	
Name of Tester:	
Hardware Ver:	Compass Module - Geosensor version 2.3
Setup:	Compass module should be wired to the MCU through the SPI interface pins. The

Step	Action	Expected Result	Pass	Fail	Comments
1	Compile compass.c in /test directory	IDE should generate no warnings or errors.			
2	Download	MCU should report "download successful"			
3	Execute	MCU should display compass splash screen on terminal interface.			
4	Orientate compass to 0 degrees.	Terminal interface should display 0 degrees +/- 10 degrees.			
5	Orientate compass to 30 degrees.	Terminal interface should display 30 degrees +/- 10 degrees.			
6	Orientate compass to 45 degrees.	Terminal interface should display 45 degrees +/- 10 degrees.			

- *Tests reduce the costs of change.* A change to a complex design with no tests can produce bugs that are difficult to track down. A good set of regression tests can help localize the effect of bugs introduced by changes.
- *Tests improve design.* In order to create a testable design, you need to create highly cohesive, loosely coupled units.
- *Tests allow you to refactor.* Subcomponents of a testable design can be changed and optimized with less chance of introducing new errors. This is because tests exist that can verify the redesigned (refactored) module functions correctly.
- *Tests constrain features.* When a test is written before building the associated module, the exact requirements are defined. Hence, when a unit passes its test, there is confidence that the requirements have been met.
- *Tests defend against other designers.* Often a design needs to have circuitry to deal with special cases. Tests that check these special cases can make sure that future modification do not remove them.
- *Testing is fun.* Writing tests requires creative solutions to complex design problems.
- *Testing forces you to slow down and think.* When writing a test before incorporating a feature into a design, you are forced to see how the new feature fits into the existing design framework.
- *Testing makes development faster.* On a component level, testing slows development. However, as the design becomes larger and more complex, modules can be more easily integrated into the design without causing malfunctions in existing components.
- *Tests reduce fear.* Would you rather improve a unit with a test suite or one without?

1.5 Summary and Further Reading

Testing is an important part of the design process that helps to ensure systems will operate properly. This chapter examined basic principles of testing including black box testing, white box testing, controllability, and

observability. They address the manner in which tests can be conducted, controlled, and states of the system observed. The use of stubs, which are employed to simulate system inputs and outputs were examined, as well as the properties of test cases. The different phases of testing from unit tests through integration tests to acceptance tests were examined. Testing proceeds from the most detailed level of the system to the most general, and the tests performed in each phase are traceable to their corresponding phases in the design development process.

The field of testing has been well developed by the software engineering community. Software Engineering: An Engineering Approach [Pet00] provides a good overview of testing principles such as black box and white box testing. It also includes a number of test strategies beyond those considered here. The Glossary of Vulnerability Testing Terminology from the University of Oulu's Electrical and Information Engineering department [Oul04] provides an extensive list of terms related to testing in the software domain. Gray's 1985 article *Why Do Computers Stop and What Can Be Done About It?* [Gra85] coined the terms Heisenbug and Bohrbug. This article introduces many interesting facts about how supercomputers fail. It provides a rare chance to look at the inner world of a supercomputer company. Many of the topics in the unit test section were influenced by Dianne L. Runnel's article *How to Write Better Test Cases* [Run99]. In this article, she defines precisely what is meant by unit test, and gives a clear picture of how to construct a unit test. This article along with many other scholarly articles on testing can be found at www.stickyminds.com. An exceptional set of documents and templates are available from the Systems Engineering Processing Group of the United States Air Force. While intended for software development, the checklists for unit and integration testing contain many insightful points. They are accessible at

https://ossg.gunter.af.mil/applications/sep/menus/Main.aspx. The list of acceptance test items was due in part to the information fond at: http://www.tbs-sct.gc.ca/emf-cag/acceptance/outline/atpo-vper_e.asp

1.6 Problems

1. Explain the differences between black box and white box testing.

2. Identify a circuit simulator (analog or digital) that you are familiar with. Explain the features of this simulator, which increase the observability and controllability of the circuit being simulated.

3. A mobile robot is being built. It uses a two DC motors in a differential drive configuration, a microcontroller to control movement and an ultrasonic sensor to detect obstacles. The robot is built to wander around without bumping into objects. Explain how stubs could be used in testing to take the place of incomplete subsystems. Be specific.

4. Consider that you have an op amp integrated circuit package, such as the LM741 in Appendix C. What type of testing would be appropriate for testing this device? Write a short test plan for doing so.

5. Explain under what situations a matrix test is appropriate.

6. Explain under what situations a step-by-step is test appropriate.

7. Consider the stages of unit testing, integration testing, and acceptance testing. For each of these stages, identify the corresponding requirements that each test should be traceable to.

8. Consider the case study robot design in Section 7.3, which presents an acceptance test for the first system requirement. Develop an acceptance test for the second system requirement.

9. Consider the case study robot design in Section 7.3. Develop an integration test that demonstrates the combined operation of the DC motors, MCU, and range finder.

10. Consider the case study robot design in Section 7.3. Develop an integration test that demonstrates the combined operation of the digital compass, MCU, and LCD.

11. Consider the case study robot design in Section 7.3. Develop unit tests for range finder, the DC motors, the H-bridges, and the LCD.

12. **Project Application.** Develop an acceptance test suite for your project. The acceptance tests should apply to the engineering requirements developed for the system.

13. **Project Application.** Develop an integration test suite for your project. The integration tests should apply to the higher levels of the design architecture and address the interaction between functional units.

14. **Project Application.** Develop a unit test quite for your project. The unit tests should apply to the lowest level units in the design.

Appendix A Glossary

Term	Definition
acceptance test	An acceptance test verifies that the system meets the ***Requirements Specification*** and stipulates the conditions under which the customer will accept the system (Chapter 7).
activity on node	A form of a ***network diagram*** used in a project plan. In the Activity on Node (AON) form, activities are represented by nodes and the dependencies by arrows (Chapter 10).
activity	An activity is a combination of a ***task*** and its associated ***deliverables*** that is part of a project plan (Chapter 10).
activity view	The activity view is part of the ***Unified Modeling Language***. It is characterized by an activity diagram; its ***intention*** is to describe the sequencing of processes required to complete a task (Chapter 6).
Analytical Hierarchy Process (AHP)	A decision-making process that combines both quantitative and qualitative inputs. It is characterized by weighted criteria against which the decision is made, a numeric ranking of alternatives, and computation of a numerical score for each alternative (Appendix B and Chapters 2 and 4).
artifact	System, component, or process that is the end-result of a design (Chapter 2).
automated script test	An automated script test is a sequence of commands given to a unit under test. For example, a test may consist of a sequence of inputs that are provided to the unit, where the outputs for each input are then verified against pre-specified values (Chapter 7).
baseline requirements	The original set of requirements that are developed for a system (Chapter 3).
black box test	A test that is performed without any knowledge of internal workings of the unit under test (Chapter 7).

Term	Definition
bottom-up design	An approach to system design where the designer starts with basic components and synthesizes them to achieve the design objectives. This is contrasted to ***top-down*** design (Chapter 5).
Bohrbug	Bohrbugs are reliable ***bugs***, in which the error is always in the same place. This is analogous to the electrons in the Bohr atomic model which assume a definite orbit (Chapter 7).
brainstorming	A freeform approach to concept generation that is often done in groups. This process employs five basic rules: 1) no criticism of ideas, 2) wild ideas are encouraged, 3) quantity is stressed over quality, 4) build upon the ideas of others, and 5) all ideas are recorded (Chapter 4).
Brainwriting	A variation of ***brainstorming*** where a group of people systematically generate ideas and write them down. Ideas are then passed to other team members who must build upon them.
break-even point	The break-even point is the point where the number of units sold is such that there is no profit or loss. It is determined from the total costs and revenue (Chapter 10).
bug	A problem or error in a system that causes it to operate incorrectly (Chapter 7).
cardinality ratio	The cardinality ratio describes the multiplicity of the entities in a relationship. It is applied to ***entity relationship diagrams*** and Unified Modeling Language ***static view diagrams*** (Chapter 6).
class	Classes are used in object-oriented system design. A class defines the attributes and methods (functions) of an ***object*** (Chapter 6).
cohesion	Refers to how focused a module is—highly cohesive systems do one or a few things very well. Also see ***coupling*** (Chapter 5).
component design specification	See ***subsystem design specification*** (Chapter 3).
concept fan	A graphical tree representation of design decisions and potential solutions to a problem. Also see ***concept table*** (Chapters 1 and 4).
concept generation	A phase in the ***design process*** where many potential solutions to solve the problem are identified (Chapter 1).
concept table	A tool for generating concepts to solve a problem. It allows systematic examination of different combinations, arrangements, and substitutions of different elements for a system. Also see ***concept fan*** (Chapter 4).

Term	Definition
conditional rule-based ethics	An ethics system in which there are certain conditions under which an individual can break a rule. This is generally because it is believed that the moral good of the situation outweighs the rule. Also see ***rule-based ethics*** (Chapter 11).
constraint	A special type of requirement that encapsulates a design decision imposed by the environment or a stakeholder. Constraints often violate the abstractness property of engineering requirements (Chapter 3).
controllability	A principle that applies to testing. Controllability is the ability to set any node of the system to a prescribed value (Chapter 7).
copyright	Copyrights protect published works such as books, articles, music, and software. A copyright means that others cannot distribute copyrighted material without permission of the owner (Chapter 11).
coupling	Modules are coupled if they depend upon each other in some way to operate properly. Coupling is the extent to which modules or subsystems are connected. See also ***cohesion*** (Chapter 5).
creative design	A formal categorization of design projects. Creative designs represent new and innovative designs (Chapter 2).
critical path	The path with the longest duration in a project plan. It represents the minimum time required to complete the project (Chapter 10).
cross-functional team	Cross-functional teams are those that are composed of people from different organizational functions, such as engineering, marketing, and manufacturing. Also see ***multi-disciplinary team*** (Chapter 9).
data dictionary	A dictionary of data contained in a ***data flow diagram***. It contains specific information on the data flows and is defined using a formal language (Chapter 6).
data flow diagram	The ***intention*** of a data flow diagram (DFD) is to model the processing and flow of data inside a system (Chapter 6).
decision matrix	A matrix that is used to evaluate and rank concepts. It integrates both the user-needs and the technical merits of different concepts (Chapter 4).
derating	A decrease in the maximum amount of power that can be dissipated by a device. The amount of derating is based upon operating conditions, notably increases in temperature (Chapter 8).
deliverable	Deliverables are entities that are delivered to the project based upon completion of ***tasks.*** Also see ***activity*** (Chapter 10).

Term	Definition
descriptive design process	Describes typical activities involved in realizing designs with less emphasis on exact sequencing than a ***prescriptive design process*** (Chapter 1).
design architecture	The main (Level 1) organization and interconnection of modules in a system (Chapter 5).
design phase	Phase in the ***design process*** where the technical solution is developed, ultimately producing a detailed system design. Upon its completion, all major systems and subsystems are identified and described using an appropriate model (Chapter 1).
design process	The steps required to take an idea from concept to realization of the final system. It is a problem-solving methodology that aims to develop a system that best meets the customer's need within given constraints (Chapter 1).
design space	The space, or collection, of all possible solutions to a design problem (Chapter 2).
detailed design	A phase in the technical design where the problem can be decomposed no further and the identification of elements such as circuit components, logic gates, or software code takes place (Chapter 5).
engineering requirement	A requirement of the system that applies to the technical aspects of the design. An engineering requirement should be abstract, unambiguous, verifiable, traceable, and realistic (Chapter 3).
entity relationship diagram (ERD)	An ERD is used to model database systems. The ***intention*** of an ERD is to catalog a set of related objects (entities), their attributes, and the relationships between them (Chapter 6).
entity relationship matrix	A matrix that is used to identify relationships between entities in a database system (Chapter 6).
ethics	Philosophy that studies ***morality***, the nature of good and bad, and choices to be made (Chapter 11).
event	An event is an occurrence at a specific time and place that needs to be remembered and taken into consideration in the system design (Chapter 6).
event table	A table that is used to store information about ***events*** in the system. It includes information regarding the event trigger, the source of the event, and process triggered by the event (Chapter 6).
failure function	The failure function, $F(t)$, is a mathematical function that provides the probability that a device has failed at time t (Chapter 8).

Term	Definition
failure rate	The failure rate, $\lambda(t)$, for a device is the expected number of device failures that will occur per unit time (Chapter 8).
fixed costs	Fixed costs are those that are constant regardless of the number of units produced and cannot be directly charged to a process or activity (Chapter 10).
float	The amount of ***slippage*** that an activity in a project plan can experience without it becoming part of a new ***critical path*** (Chapter 10).
flowchart	A modeling diagram whose intention is to visually describe a process or algorithm, including its steps and control (Chapter 6).
functional decomposition	A design technique in which a system is designed by determining its overall functionality and then iteratively decomposing it into component subsystems, each with its own functionality (Chapter 5).
functional requirement	A ***subsystem design specification*** that describes the inputs, outputs, and functionality of a system or component (Chapters 3 and 5).
Gantt chart	Gantt charts are a bar graph representation of a project plan where the activities are shown on a timeline (Chapter 10).
Heisenbugs	Heisenbugs are ***bugs*** that are not always reproducible with the same input. This is analogous to the Heisenberg Uncertainty Principle, in which the position of an electron is uncertain (Chapter 7).
high-performance team	A team that significantly outperforms all similar teams. Part of the Katzenbach and Smith team model (Chapter 9).
integration test	An integration test is performed after the units of a system have been constructed and tested. The integration test verifies the operation of the integrated system behavior (Chapter 7).
intention	The intention of a model is the target behavior that it aims to describe (Chapter 6).
interaction view	The interaction view is part of the ***Unified Modeling Language***. Its ***intention*** is to show the interaction between objects. It is characterized by collaboration and sequence diagrams (Chapter 6).
key attribute	An attribute for an entity in a database system that uniquely identifies an instance of the entity (Chapter 6).

Term	Definition
lateral thinking	A thought process that attempts to identify creative solutions to a problem. It is not concerned with developing the solution for the problem, or right or wrong solutions. It encourages jumping around between ideas. It is contrasted to ***vertical thinking*** (Chapter 4).
liable	Required to pay monetary damages according to law (Chapter 11).
marketing requirement (specifications)	A statement that describe the needs of the customer or end-user of a system. They are typically stated in language that the customer would use (Chapters 2 and 3).
maintenance phase	Phase in the ***design process*** where the system is maintained, upgraded to add new functionality, or design problems are corrected (Chapter 1).
matrix test	A matrix test is a test that is suited to cases where the inputs submitted are structurally the same and differ only in their values (Chapter 7).
mean time to failure	The mean time to failure (MTTF) is a mathematical quantity which answers the question, "*On average how long does it take for a device to fail?*" (Chapter 8).
module	A block, or subsystem, in a design that performs a function (Chapter 5).
morals	The ***principles*** of right and wrong and the decisions that derive from those principles (Chapter 11).
multi-disciplinary team	In general, a multi-disciplinary team is one in which the members have complementary skills and the team may have representation from multiple technical disciplines. Also see ***cross-functional team*** (Chapter 9).
negligence	Failure to exercise caution, which in the case of design could be in not following reasonable standards and rules that apply to the situation (Chapter 11).
network diagram	A network diagram is a directed graph representation of the activities and dependencies between them for a project (Chapter 10).
Nominal Group Technique (NGT)	A formal approach to brainstorming and meeting facilitation. In NGT, each team member silently generates ideas that are reported out in a round-robin fashion so that all members have an opportunity to present their ideas. Concepts are selected by a multi-voting scheme with each member casting a predetermined number of votes for the ideas. The ideas are then ranked and discussed (Chapters 4 and 9).

Term	Definition
non-disclosure agreement	An agreement that prevents the signer from disseminating information about a company's products, services, and trade secrets (Chapter 11).
object	Objects represent both data (attributes) and the methods (functions) that can act upon data. An object represents a particular instance of a ***class***, which defines the attributes and methods (Chapter 6).
object type	Characteristic of a model used in design. The object type is capable of encapsulating the actual components used to construct the system (Chapter 6).
objective tree	A hierarchical tree representation of the customer's needs. The branches of the tree are organized based upon functional similarity of the needs (Chapter 2).
observability	This principle applies to testing. Observability is the ability to observe any node of a system (Chapter 7).
over-specificity	This refers to applying targets for ***engineering requirements*** that go beyond what is necessary for the system. Over-specificity limits the size of the ***design space*** (Chapter 3).
pairwise comparison	A method of systematically comparing all customer needs against each other. A comparison matrix is used for the comparison and the output is a scoring of each of the needs (Appendix B, Chapter 2, and Chapter 4).
parallel system	A system that contains multiple modules performing the same function where a single module would suffice. The overall system functions correctly when any one of the submodules is functioning (Chapter 8).
patent	A patent is a legal device for protecting a design or invention. If a patent is held for a technology, others cannot use it without permission of the owner (Chapter 11).
path-complete coverage	Path-complete coverage is where every possible ***processing path*** is tested (Chapter 7).
performance requirement	A particular type of ***engineering requirement*** that specifies performance related measures (Chapter 3).
physical view	The physical view is part of the ***Unified Modeling Language***. Its ***intention*** is to demonstrate the physical components of a system and how the logical views map to them. It is characterized by a component and deployment diagram (Chapter 6).

Term	Definition
potential team	A team where the sum effort of the team equals that of the individuals working in isolation. Part of the Katzenbach and Smith team model (Chapter 9).
prescriptive design process	An exact process, or systematic recipe, for realizing a system. Prescriptive design processes are often algorithmic in nature and expressed using flowcharts with decision logic (Chapter 1).
principle	Fundamental rules or beliefs that govern behavior, such as the Golden Rule (Chapter 11).
problem identification	The first phase in the design process where the problem is identified, the customer needs identified, and the project feasibility determined (Chapter 1).
processing path	A processing path is a sequence of consecutive instructions or states encountered while performing a computation. They are used to develop test cases (Chapter 7).
prototyping and construction phase	Phase in the ***design process*** in which different elements of the system are constructed and tested. The objective is to model some aspect of the system, demonstrating functionality to be employed in the final realization (Chapter 1).
pseudo-team	An under-performing team where the sum effort of the team is below that of the individuals working in isolation. Part of the Katzenbach and Smith team model (Chapter 9).
Pugh Concept Selection	A technique for comparing design concepts to the user needs. It is an iterative process where concepts are scored relative to the needs. Each concept is combined, improved, or removed from consideration in each iteration of the process (Chapter 4).
real team	A team where the sum effort of the team exceeds that of the individuals working in isolation. Part of the Katzenbach and Smith team model (Chapter 9).
redundancy	A design has redundancy if it contains multiple modules performing the same function where a single module would suffice. Redundancy is used to increase ***reliability*** (Chapter 8).
reliability	Reliability, *R(t)*, is the probability that a device is functioning properly (has not failed) at time t (Chapter 8).
research phase	Phase in the ***design process*** where research on the basic engineering and scientific principles, related technologies, and existing solutions for the problem are explored (Chapter 1).

Term	Definition
Requirements Specification	A collection of engineering and marketing requirements that a system must satisfy in order for it to meet the needs of the customer or end-user. Alternate terms that are used for the Requirements Specification are the *Product Design Specification* and the *Systems Requirements Specification* (Chapter 1 and 3).
reverse-engineering	Process where a device or process is taken apart to understand how it works (Chapter 11).
routine design	A formal categorization of design projects. They represent the design of artifacts for which theory and practice are well-developed (Chapter 2).
rule-based ethics	Rule-based ethics are based upon a set of rules that can be applied to make decisions. In the strictest form, they are considered to be absolute in terms of governing behavior (Chapter 11).
satisfice	Satisfice means that a solution may meet the design requirements, but not be the optimal solution (Chapter 11).
series system	A system in which the failure of a single component (or subsystem) leads to failure of the overall system (Chapter 8).
situational ethics	Situational ethics are where decisions are made based on whether they produce the highest good for the person (Chapter 11).
slippage	Refers to an activity in a project plan taking longer than its planned time to complete. See also ***critical path*** and ***float*** (Chapter 10).
standards	A standard or established way of doing things. Standards ensure that products work together, from home plumbing fixtures to the modules in a modern computer. They ensure the health and safety of products (Chapter 3).
state	The state of a system represents the net effect of all the previous inputs to the system. Since the state characterizes the history of previous inputs, it is often synonymous with the word memory (Chapter 6).
state diagram (machine)	Diagram used to describe systems with memory. It consists of states and transitions between states (Chapter 6).
static view	The static view is part of the ***Unified Modeling Language***. The ***intention*** of the static view is to show the classes in a system and their relationships. The static view is characterized by a class diagram (Chapter 6).
step-by-step test	A step-by-step test case is a prescription for generating a test and checking the results. It is most effective when the test consists of a complex sequence of steps (Chapter 7).

Term	Definition
strengths and weakness analysis	A technique for the evaluation of potential solutions to a design problem where the strengths and weaknesses are identified (Chapter 4).
structure charts	Specialized block diagrams for visualizing functional software designs. They employ input, output, transform, coordinate, and composite modules (Chapter 5).
strict liability	A form of ***liability*** that focuses only on the product itself—if the product contains a defect that caused harm, the manufacturer is liable (Chapter 11).
stub	A stub is a device that is used to simulate a subcomponent of a system during testing. Stubs simulate inputs or monitor outputs from the unit under test (Chapter 7).
subsystem design specification	Engineering requirements for subsystems that are constituents of a larger, more complex system (Chapter 3).
system integration	Phase in the ***design process*** where all of the subsystems are brought together to produce a complete working system (Chapter 1).
task	Tasks are actions that accomplish a job as part of a project plan. Also see ***activity*** and ***deliverable*** (Chapter 10).
Team Process Guidelines	Guidelines developed by a team that govern their behavior and identify expectations for performance (Chapter 9).
technical specification	A list of the technical details for a given system, such as operating voltages, processor architecture, and types of memory. The technical specification is fundamentally different from a requirement in that it indicates what was achieved in the end versus what a system needs to achieve from the outset. (Chapter 3).
test coverage	Test coverage is the extent to which the test cases cover all possible ***processing paths*** (Chapter 7).
test phase	Phase in the design process where the system is tested to demonstrate that it meets the requirements (Chapters 1 and 7).
testable	A design is testable when a failure of a component or subsystem can be quickly located. A testable design is easier to debug, manufacture, and service in the field (Chapter 7).
top-down design	An approach to design in which the designer has an overall vision of what the final system must do, and the problem is partitioned into components, or subsystems that work together to achieve the overall goal. Then each subsystem is successively refined and partitioned as necessary. This is contrasted to ***bottom-up*** design (Chapter 5).

Term	Definition
tort	The basis for which a lawsuit is brought forth (Chapter 11).
trade secret	An approach to protecting intellectual property where the information is held secretly, without ***patent*** protection, so that a competitor cannot access it (Chapter 11).
under-specificity	This refers to a state of the ***Requirements Specification***. When it is under-specified, requirements do not meet the needs of the user and/or embody all of the requirements needed to implement the system (Chapter 3).
Unified Modeling Language (UML)	A modeling language that captures the best practices of object-oriented system design. It encompasses six different system views that can be used to model electrical and computer systems (Chapter 6).
unit test	A unit test is a test of the functionality of a system module in isolation. It establishes that a subsystem performs a single unit of functionality to some specification (Chapter 7).
use-case view	The use-case view is part of the ***Unified Modeling Language***. Its ***intention*** is to capture the overall behavior of the system from the user's point of view and to describe cases in which the system will be used (Chapter 6).
utilitarian ethics	In utilitarian ethics, decisions are made based upon the decision that brings about the highest good for all, relative to all other decisions (Chapter 11).
validation	The process of determining whether the requirements meet the needs of the user (Chapter 3).
value	A value is something that a person or group believes to be valuable or worthwhile. Also see ***principles*** and ***morals*** (Chapter 11).
variable costs	Variable costs vary depending upon the process or items being produced, and fluctuate directly with the number of units produced (Chapter 10).
variant design	A formal categorization of design projects. They represent the design of existing systems, where the intent is to improve performance or add features (Chapter 2).
verifiable	Refers to a property of an engineering requirement. It means that there should be a way to measure or demonstrate that the requirement is met in the final system realization (Chapter 3).
vertical thinking	A linear, or sequential, thought process that proceeds logically towards the solution of a problem. It seeks to eliminate incorrect solutions. It is contrasted to ***lateral thinking*** (Chapter 4).

Term	Definition
whistleblower	A person who goes outside of their company or organization to report an ethical or safety problem (Chapter 11).
white box test	White box tests are those that are conducted with knowledge of the internal working of the unit under test (Chapter 7).
work breakdown structure	The work breakdown structure (WBS) is a hierarchical breakdown of the tasks and deliverables that need to be completed in order to accomplish a project (Chapter 10).
working group	A group of individuals working in isolation, who come together occasionally to share information. Part of the Katzenbach and Smith team model (Chapter 9).