Design for Electrical and Computer Engineers

Theory, Concepts, and Practice

Ralph M. Ford and Christopher S. Coulston

This document was prepared with LaTeX.

Design for Electrical and Computer Engineers © 2024 by Ralph Ford and Christopher Coulston is licensed under CC BY-NC-SA 4.0 For more infomation about the Create Commons license see: https://creativecommons.org/licenses/by-nc-sa/4.0/

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

0.1 About the Authors . . . iii
0.2 Preface . . . vi

1 System Design II: Behavior Models 1
1.1 Models . . . 2
1.2 State Diagrams . . . 4
1.3 Flowcharts . . . 6
1.4 Data Flow Diagrams . . . 8
1.5 Entity Relationship Diagrams . . . 11
1.6 The Unified Modeling Language . . . 14
1.6.1 Static View . . . 14
1.6.2 Use-Case View . . . 15
1.6.3 State Machine View . . . 18
1.6.4 Activity View . . . 18
1.6.5 Interaction View . . . 19
1.6.6 Physical View . . . 20
1.7 Project Application: Selecting Models . . . 21
1.8 Summary and Further Reading . . . 21
1.9 Problems . . . 24

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^{TM} 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

System Design II: Behavior Models

Genius is 1% inspiration and 99% perspiration.—Thomas Edison

The functional decomposition technique examined in Chapter 5 is a powerful modeling tool for system design that is applicable for describing input, output, and transform behavior. However, that approach by itself is limited in its descriptive ability. This was apparent in the digital stopwatch example that required the use of state diagrams, in addition to functional decomposition, to fully articulate the design. A state diagram is an example of a model, a standardized abstraction of a system. Models allow systems to be described without having to determine all of the implementation details. All models are not the same—they come in a variety of forms and each serves a different intention.

This chapter provides an overview of other design tools for describing system behavior, with an emphasis on computing systems. The first tool examined is the state diagram. This is followed by the flowchart, which describes algorithmic processes and logical behavior. Two modeling languages for information and data handling—data flow diagrams and entity relationship diagrams—are then examined. The final is the Unified Modeling Language, which is a collection of system views for describing behavior.

Learning Objectives

By the end of this chapter, the reader should:

- Be familiar with the following modeling tools for describing system behavior: state diagrams, flowcharts, data flow diagrams, entity relationship diagrams, and the Unified Modeling Language.
- Understand the intention and expressive power of the different models.
- Understand the domains in which the models apply.
- Be able to conduct analysis and design with the models.
- Understand what model types to choose for a given design problem.

1.1 Models

From the previous chapter we know that the top-down design process starts with an abstraction of the system to be built. This initial design is called an abstraction because it captures the essential characteristics of the system without specifying the underlying physical realization. An abstraction that is expressed in a standardized and accepted language is called a model. In other words a model is a standardized representation of a system, process, or object which captures its essential details without specifying the physical realization. A modeling language does not have to be formed from letters and words—often the words are graphical symbols. You are already familiar with many different models from everyday life such as blueprints, a diagram of a football play, knitting instructions, electrical schematics, and mathematical formulas to name a few. In order to be effective, a model should meet the following properties [Sat02]:

- *Be abstract.* This means that the model should be independent of final implementation and that there should be multiple ways of implementing the design based upon it.
- *Be unambiguous.* A model should have a single clear meaning in terms of describing the intended behavior.
- *Allow for innovation.* Models should encourage exploration of alternative system implementations and behaviors.

- *Be standardized.* Standardization provides a common language that can be understood by all. Designers should be wary of developing their own models that are ill-defined and not commonly understood.

- *Provide a means for communication.* A model should facilitate communication within the design team and with non-technical stakeholders.

- *Be modifiable.* A model should make design modifications relatively easy.

- *Remove unnecessary details and emphasize important features.* The intent is to simplify the design for ease of understanding. The most highly detailed information is typically identified in the detailed design.

- *Break the overall problem into sub-problems.* Most problems are too complex to be handled directly and must be decomposed into subsystems. This produces the design hierarchy.

- *Substitute a sequence of actions by a single action.* This allows understanding of the overall larger behavior, which can then be examined at other levels. This supports the ability to break a design into sub-problems.

- *Assist in verification.* A model should aid in demonstrating that the design meets the engineering requirements.

- *Assist in validation.* Validation is the process of demonstrating that the needs of the user are being met and the right system is being designed. The model should facilitate discussion with all stakeholders to ensure it meets everyone's expectations.

In order to meet these properties, most models have an ***object type,*** which is capable of encapsulating the actual components used to construct the target system. In order to capture the dependence of objects on one another models typically have a ***relationship type***. Finally, models have an ***intention,*** which is the intended class of behavior that it describes. For example, the intention of a circuit schematic and the schematic of a football play are entirely different. Since models are built with different intentions, it possible to choose the wrong model for a particular system—it would surprise a football team to see a play represented with resistors and capacitors!

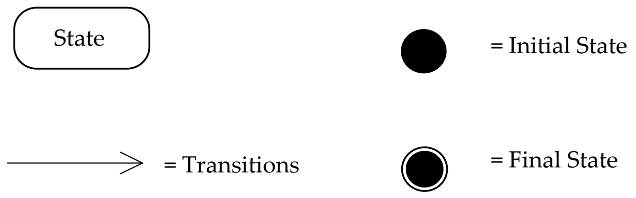


Figure 1.1: Symbols used in state diagrams.

Since models capture the essential details of a system in a standardized way then they are an ideal way to describe the functionality of a system at all levels of detail. In Chapter 5 the predominate method of describing functionality was with words. However, there are languages that describe system behavior. We start by examining state diagrams.

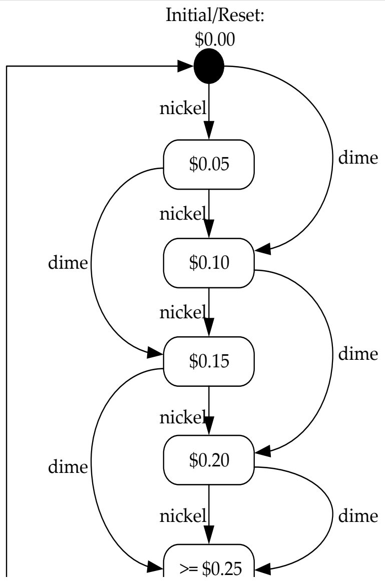
1.2 State Diagrams

State diagrams describe the behavior of systems with memory. A system with memory is able to modify its response to inputs based on the state of the system. The ***state*** of a system represents the net effect of all previous inputs to the system. Since the state characterizes the history of previous inputs, it is often synonymous with the word memory. Intuitively, a state corresponds to an operating mode of a system, and inputs are associated with transitions between states. To determine if a system has memory, ask the following question— "*Can the same input produce different outputs?*" If the answer to this question is yes, then the system has memory and can be modeled with a state diagram.

A state diagram is a drawing that consists of states and transition arcs as shown in Figure 1.1. Each state is represented as a rectangle with rounded edges with the name of the state written inside. Whenever possible, states should be given meaningful names. When there exists the possibility for ambiguity in the names, a table should be created that identifies the state names and their associated meanings. There are special circle symbols for both initial and final states. Transitions are drawn as arrows from a source state to a destination state. Since inputs cause the transitions between states, the arrows are labeled with their associated inputs. The outputs are listed directly in the state since it is assumed that the outputs are associated with states.

As an example, consider a vending machine that accepts nickels and dimes and dispenses a piece of candy when 25 cents has been deposited.

Table 1.1: A state diagram for a simple vending machine.

Module	Vending Machine Control Unit
Inputs	• Nickel: Signals that a nickel has been deposited. • Dime: Signals that a dime has been deposited.
Outputs	• Reset: Signals the FSM to return to the Initial/Reset state. • Vend: Signal to dispense candy.
Functionality	 Initial/Reset: $0.00 nickel dime $0.05 $0.10 $0.15 $0.20 >= $0.25

This vending machine can be modeled using a state diagram because the response of the machine to a coin depends on how much money has been deposited so far.

In order to give a more complete description of the vending machine the state diagram is embedded into the function table template introduce in Chapter 5as shown in Figure 1.1. This table lists the inputs and outputs of the vending machine along with the behavior represented by a state diagram.

The initialization/reset state is labeled $0.00, while the action of the machine (dispense or not dispense candy) is associated with the states—the only state that dispenses candy is $0.25. There are three types of transitions shown in this state diagram. The two labeled nickel and dime are associated with depositing those coins. The unlabeled transition from state \$0.25 to state \$0.00 is called an unconditional transition. A system with an unconditional transition between two states is assumed to remain in the first state for a defined time period before automatically moving to the second state. In this case it ensures that the machine dispenses a single candy before going on to the next transaction. It is common practice in state diagrams to assume that any unspecified input conditions cause the system to remain in the current state. For example, if no coin is inserted while in state \$0.20 the system remains in state \$0.20. Finally, note that since this machine does not dispense change, it can overcharge a customer for candy. This would not be a popular vending machine with users!

1.3 Flowcharts

The intention of a ***flowchart*** is to visually describe a process or algorithm, including its steps and control. Flowcharts are often scoffed at as being old-fashioned and overly simple—these criticisms are actually strengths. Since flowcharts have been around for a long time, they are easily recognized and understood. Furthermore, being simple makes them accessible to a wide audience. Due to their simplicity, they are applied in a great number of applications, including non-technical ones such as the description of business processes.

Some of the primary symbols used in flowcharts are shown in Figure 1.2. The names of the starting and ending steps of a flowchart are represented by ovals known as terminators. Individual processing steps are written inside of rectangles, while a process step that is elaborated by another flowchart is drawn as a rectangle with double sides. Elaborated processes allow the representation of hierarchy in the design. Certain points in a flowchart can lead to alternative destinations as represented by a decision or conditional symbol (diamond). The condition that determines the next step is written inside the diamond and the possible values of the condition are written on the arcs leaving the conditional step. As shown in Figure 1.2 there are multiple ways to indicate data stores for retrieving or saving data.

As an application, consider an embedded computer system that monitors the light level of its environment as described by the flowchart in Figure 1.2. The algorithmic process of the flowchart is easy and intuitive to understand. The system reads the current light value, stores it into an array, and then computes an average. In a complete system description, there would be a second flowchart describing how the system determines the average of the

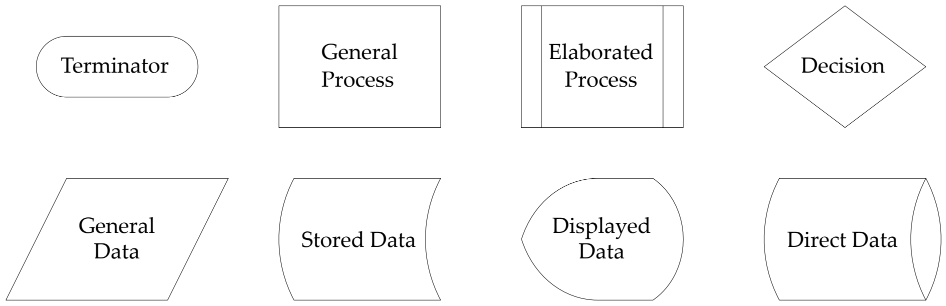
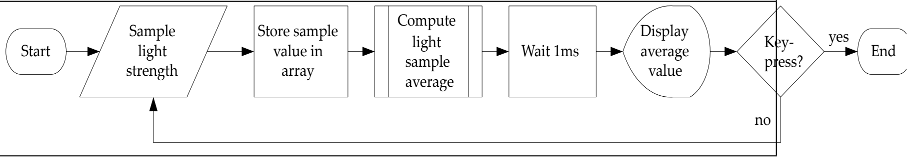


Figure 1.2: Basic flowchart symbols.

Table 1.2: A flowchart for an embedded system.

Module	Light level data logger
Inputs	• Light intensity: Ambient light intensity from environment. • Key-press: User request to abort data logging.
Outputs	• Terminal: Displays the average light intensity. • Disk: Stores a record of light intensities through time.
Functionality	 Start Sample light strength Store sample value in array Compute light sample average Wait 1ms Display average value Key-press? yes no End

values of the light samples, since this is identified as an elaborated process. The system then waits 1ms, writes the average to a terminal (display device), and then checks for a key-press. Light levels continue to be monitored in the absence of a key-press, otherwise the light monitoring process halts.

Figure 6.4

Flowcharts are an intuitive way to describe algorithmic processes. Limiting the complexity of a flowchart to between 10 and 20 steps enables the sequence of actions to be quickly comprehended while eliminating unnecessary details. Flowcharts are not able to represent the structure of data being manipulated and they are not particularly good at representing concurrent processes. For this we need data flow diagrams.

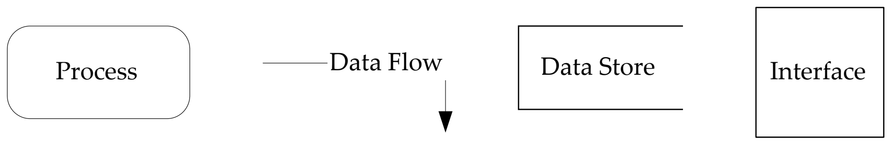


Figure 1.3: Data flow diagram symbols.

1.4 Data Flow Diagrams

The intention of a ***data flow diagram*** (DFD) is to model the processing and flow of data inside a system. It is a function-oriented approach that is similar to functional decomposition—the processes inside a DFD accept data inputs, transform them in some way, and produce output data. A DFD is often used for the analysis of information systems due to its data emphasis, but can be broadly applied to ECE systems. It differs from functional decomposition in that functional decomposition is often closer to the implementation of the design, whereas the DFD models the system from a data point of view. A DFD is fundamentally different from a flowchart in that it does not encapsulate control and sequencing information, but allows multiple processes running concurrently. There are four symbols, shown in Figure 1.3, that are used in a DFD:

1. *Processes.* A rectangle with rounded corners that describes a useful task or function. They perform a transformation on the data.

2. *Data flows.* An arrow representing a data relationship between two processes.

3. *Data stores.* An open rectangle representing a data repository.

4. *Interfaces.* A square describing external agents or entities that use the system. They are also referred to as sources and sinks.

Like the general design process, DFDs are successively refined from the top-down. That is, there is a single top level (or Level 0) DFD describing the entire system and the interfaces and data stores that it interacts with. The rules for constructing a DFD are fairly intuitive. A process must have at least one input and one output. The refinement of a process at level N must have the same inputs and outputs as the process at level N-1. Data must be transformed in some way by a process. This process of refinement continues until a satisfactory level of detail is reached.

An example of a Level 1 DFD for a video browsing system is shown in Figure 1.3. Video databases are typically very large; due to their size, it is usually cumbersome to preview videos and extract important information. A solution to this problem is to apply image analysis techniques to identify shots (continuous scenes without a break) in a video and store both the location of the shot boundaries in the video and key frames that summarize each shot. The collection of key frames is known as a storyboard. The storyboards are stored in an annotation database that is much smaller in size than the original video database. Typically the user of a video browsing system has the ability to preview the storyboard and select shots from it to view. This example shows the data flow for such a system.

The data processing is readily apparent from the DFD. Videos in the database are processed to extract shot boundaries and key frames, which are stored in the annotation database. The user can submit requests to view a storyboard, which is retrieved from the annotation database. When a shot is selected from the storyboard, the users can preview the original shot, which is retrieved from the video database.

There are a few important points to note about DFDs. They are solution independent, specifying the behavior based upon data flow. Specific information on the data flows is defined in a formal language known as a ***data dictionary*** (it is not covered here). There can be concurrent processes represented in a DFD. In the video browsing example, multiple people can use the system simultaneously, and there is no implied sequencing between when the shot boundary detection process is run and the storyboards are previewed. It is clear that the video must undergo shot detection prior to viewing its storyboard. This information is listed in something known as an ***event table***. The event table for this example is shown in Table 6.1.

Table 6.1 Event table for the video browsing system.

An ***event*** is an occurrence at a specific time and place that needs to be remembered. Events can be classified into temporal, external, and state. A *temporal event* is one which happens because the system has reached some critical time. In this example, the generation of shot boundaries could occur for all new videos in the database at a specified time each day, but in this case it occurs whenever a new video is added to the database. An *external event* occurs outside the system boundary by a system user, in this case requesting either a storyboard or a video preview. A *state event* is the result of something changing within the system. Associated with each event is a *trigger*, the cause of the event. Each event has a process that is associated with it. Finally, associated with each event is a *source*, the entity responsible for triggering it.

Table 1.3: Level 1 data flow diagram for a video browsing system.

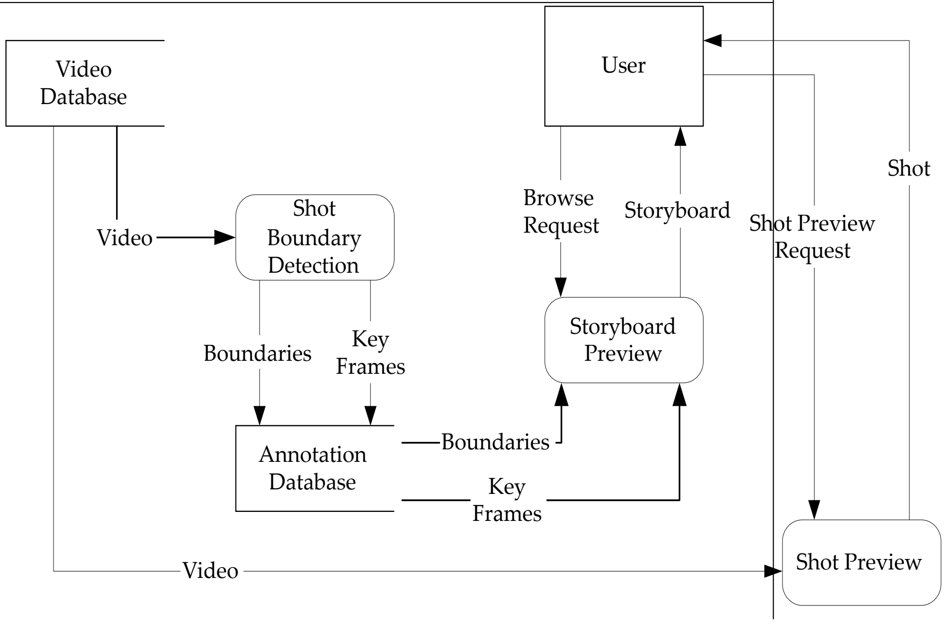
Module	Video Browsing System
Inputs	• Video: External video to the system that is entered into the video database. • Browse Request: User request to browse a particular video. • Shot Preview Request: User request to preview a particular short from a video.
Outputs	• Story Board: A sequence of frames summarizing the entire video. • Shot: The complete video corresponding to the still image in the story board.
Functionality	 Video Database User Video Shot Boundary Detection Browse Request Storyboard Shot Preview Request Shot Storyboard Preview Boundaries Key Frames Annotation Database Boundaries Key Frames Video Shot Preview

Table 1.4:

Event	Trigger	Process	Source
Annotate Video	New Video Arrival	Shot Boundary Detection	System
View Storyboard	Browse Request	Storyboard Preview	User
View Shot	Shot Preview Request	Shot Preview	User

1.5 Entity Relationship Diagrams

A database is a system that stores and retrieves data, and it is modeled by an ***entity relationship diagram*** (ERD). The intention of an ERD is to catalog a set of related objects (entities), their attributes, and the relationships between them. The entities and their relationships are real distinct things that have characteristics that need to be captured. The design of a database starts by describing the entities, their attributes, and the relationship between entities in an ERD. In order to ask meaningful questions about the data, the entities need to be related to one another. For example, a list of students and a list of courses by themselves have limited utility. However, by introducing a relationship between these two entities we can ask questions such as "*How many students are taking the Microelectronics course?*" The three elements used in the ERD modeling language are:

1. *Entities.* They are generally in the form of tangible objects, roles played, organizational units, devices, and locations. An instance is the manifestation of a particular entity. For example, an entity could be Student while an instance would be Kristen.

2. *Relationships.* They are descriptors for the relationships between entities.

3. *Attributes.* They are features that are used to differentiate between instances of the entities.

Lets consider an ERDs describing academic life at a college . The process typically starts by interviewing the end-users and identifying the entities and their attributes. Assume that the result of this process is that the college wants to store data about three entities: Students, Courses, and Departments. The process of building an ERD starts by determining the relationships between entities. One way to do this is to build an ***entity relationship matrix*** as shown in Table 1.5. The entities constitute both the row and column headings, and the matrix entries represent the relationship, if any, that exists between entities. This is similar to the pairwise comparison matrix in Chapter 2 where user needs were systematically compared. Relationships are bi-directional because they have two participating entities.

From Table 1.5 we can see that a Student can take many courses and a Course has many students in the Student-Course relationship. A ***cardinality ratio***, associated with each relationship, describes the multiplicity of the

Table 1.5: Entity relationship matrix.

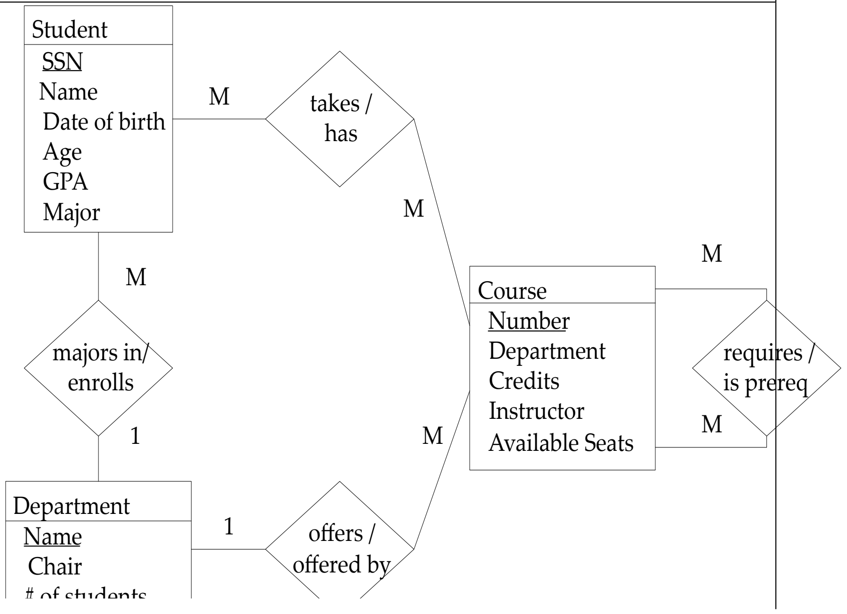
Student	**Course**	**Department**	
Student		takes many	majors
Course	has many	can require many / can be the prerequisite for many	is offere
Department	enrolls many	offers many	

entities in a relationship. For example, the Student-Course relationship is many-to-many, or M:M, because one student can take many courses and one course is taken by many students. The relationship between Student and Department is M:1 since a Department enrolls many students, but a student can major in only one Department. A recursive relationship, prerequisite, exists between the Course entity and itself because a course may have many other courses as prerequisites and may be the prerequisite for many other courses. Thus the prerequisite relationship has an M:M cardinality. It needs to be noted that in this example the needs of the college required relationships between all pairs of entities. If, for example, the college did not need to keep track of the students' majors, then the relationship between Student and Department would be left blank in the entity relationship matrix.

The resulting ERD is shown in Figure 1.6. The relationships are identified by the diamonds shaped symbols; the entities are denoted by the rectangles with their name at the top. The cardinality ratio is labeled on the links between the relationships and entities. Another piece of information shown in the ERD is the attributes associated with each entity. An attribute is a feature or characteristic of an entity that needs to be remembered. There are many different types of attributes; however, the most important are ***key attributes*** (which are underlined) which uniquely identify instances. For example, the identification number (ID) attribute of a Student is a key attribute.

The ERD allows for an easy interpretation of the relationships between entities as well as their attributes. Although beyond the scope of the discussion here, the ERD is a formal language that can be used to automatically generate the database structure. The relationships in the ERD allow queries to be asked of the resulting database. For example, the college could derive a student's course schedule from the database from the relationship between Student and Course.

Table 1.6: ERD for the college database system.

Module	Grade Database
Inputs	• Students: Data about students. • Departments: Data about departments. • Courses: Data about courses.
Outputs	• Relationship between departments and their enrolled students. • Relationship between students and the courses that they take. • Relationship between departments and the courses they offer. • Relationship between course and their prerequisite courses.
Functionality	 Student SSN Name Date of birth Age GPA Major M takes / has M M majors in/ enrolls 1 Course Number Department Credits Instructor Available Seats M requires / is prereq M M Department Name Chair 1 offers / offered by

1.6 The Unified Modeling Language

The ***Unified Modeling Language*** (UML) was created to capture the best practices of the object-oriented software development process. However, it has valuable modeling tools that can be applied more generally to ECE systems. The objective of this section is to provide an overview of UML and the six different system views that it encompasses. A caveat for the reader—UML is complex, and complete coverage is beyond the scope of this book. In order to fully understand its subtleties and nuances, the reader is advised to consult the references provided in Section 6.8.

In order to aid in the explanation of the different views we will create a UML model of a system called the Virtual Grocer, or v-Grocer for short. The v-Grocer enables a user to order groceries from home and have them delivered. The concept is to provide users with a barcode scanner that is connected to their home computer along with application software. When the user runs out of an item, he/she scans in the Universal Product Code (UPC). When users want to order groceries, they connect to the Internet, log into the grocery store web server, enter the quantity for the scanned items, and place their order. Once the order is completed, they are billed and the groceries are delivered to their houses at a prearranged time.

1.6.1 Static View

Object-oriented design (OOD) is fundamentally different from functional software design in that it emphasizes objects instead of functions. ***Objects*** represent both data (attributes) and the methods (functions) that can act upon the data. An object represents a particular instance of something known as a ***class***, which defines the attributes and methods. An object encapsulates all of this information. The data and methods that an object encapsulates are available only to that object by default. Thus, other objects cannot change that state of an object unless given specific permission to do so. This improves the reliability and maintainability of software systems, because changes made to the internal representation of a class are not seen by methods outside of the class.

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*. A very simple class diagram with a single class is shown in Figure 1.4. The specification of a class has three parts, a name, a list of attributes, and a set of methods. The name of the class for this example is Customer and it has three attributes: Name, Address, and CustId. It has one method associated

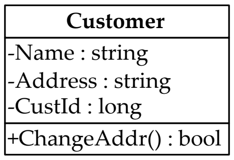


Figure 1.4: Class diagram notation. The class has a label (Customer), attributes (Name, Address, and CustId), and methods (ChangeAddr()).

with it, ChangeAddr(), which can change the Address attribute. A class, just like an entity in an ERD, is a generalization of a set of a particular thing or instance. For example, the Customer class could have a particular instance called Ms. Robinson.

Classes are related to one another by relationships that define how classes interact with each other. For example, if a class is a subset of another class (a kind of relation) then the subclass inherits all the attributes and methods of the *superclass*. In UML, there are a host of relationships; among these are generalization, composition, and associations. Two classes are related by a *generalization relationship* when one is a subset of the other. Two classes are related by a *composition relationship* when one is composed of members of the other. Two classes are related by an *association relationship* whenever they need to send messages to one another. Relationships are drawn as lines connecting the two participant classes. The terminals of the line have different shapes depending on the type of relationship. Just like an ERD, the relationships between classes have cardinality defined by the rule base. In order to illustrate these points, Figure 1.5 shows the class diagram for a portion of the v-Grocer system.

The class diagram has five classes—Customer, Delivery, Order, Item, and GroceryCart. The GroceryCart class is derived from the many-to-many relationship which exists between the Order and Item classes. Just like the rules for an ERD, the cardinality of a relationship is read at the end of the association in which it is involved. The only difference is that the many cardinality in a class diagram is denoted by a * symbol. For example, a delivery is sent to one customer, while a customer may receive many deliveries.

1.6.2 Use-Case View

The intention of the ***use-case view*** is to capture the overall behavior of the system from the user's point of view and to describe cases in which

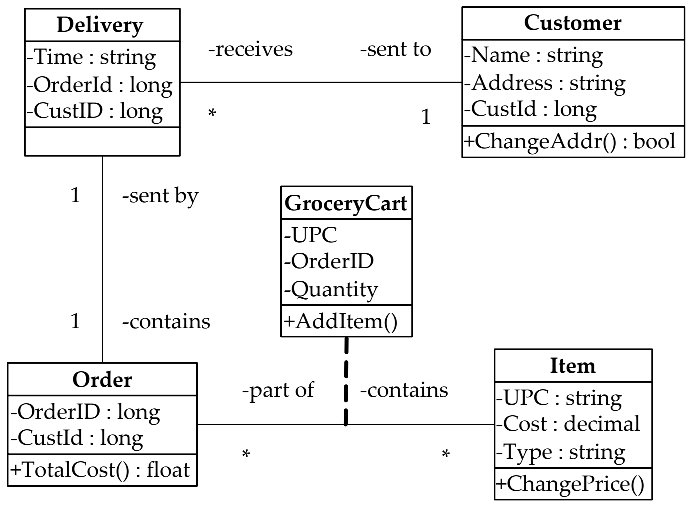


Figure 1.5: Class diagram for the v-Grocer.

the system will be used. It is characterized by a *use-case diagram*, and an example for the v-Grocer is shown in Figure 1.6. There are only two symbols employed in a use-case diagram, actors and use-cases. Actors, drawn as stick figure people, are idealized people or systems that interact with the system. For this example the actors are customers, delivery people, the database, clerks, and the web server. Use-cases are drawn as ovals in the diagram, and in this example, they are WebOrder, DeliverOrder, and AssembleOrder. A use-case is a particular situation when actors use the system and is usually represented by a sequence of actions that will be performed by the system. This sequence of actions typically represents a high level of functionality. Every actor that interacts with a particular use-case is connected to it with a line. Finally, the entire collection of use-cases is enclosed in a rectangle with the actors outside. This rectangle represents the system boundary and consequently is labeled with the name of the system.

From Figure 1.6 it is apparent that a Customer, the WebServer, and Database interact when creating a WebOrder. Use-cases are often described in a table as shown for this particular example in Table 1.7.

Table 6.3

Use-cases focus on a very high level of functionality and describe who will interact with the system and how they will interact. Given their high level of abstraction, they can easily be incorporated into a variety of ECE projects. They are also simple and easy to understand and thus can be incorporated

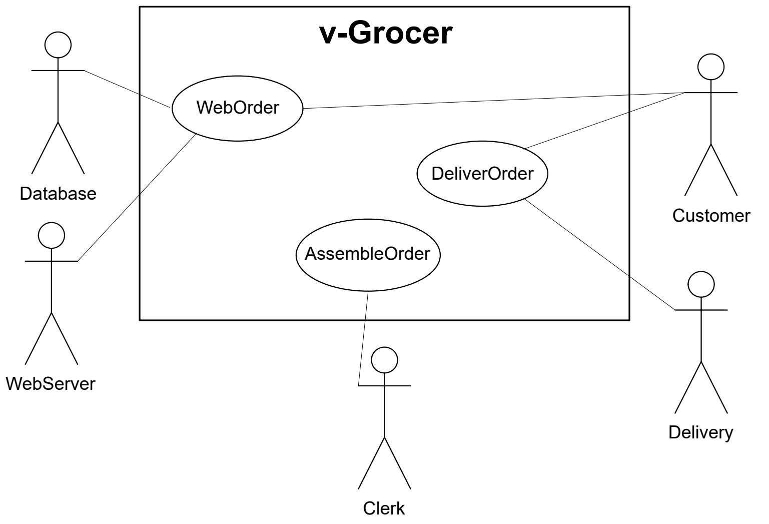


Figure 1.6: Use-case diagram for the v-Grocer.

Table 1.7: WebOrder use-case description.

Use-Case	WebOrder
Actors	Customer, Database, and WebServer
Description	This use-case occurs when a customer submits an order via the WebServer. If it is a new customer, the WebServer prompts them to establish an account and their customer information is stored in the Database as a new entry. If they are an existing customer, they have the opportunity to update their personal information.
Stimulus	Customer order via the GroceryCart.
Response	Verify payment, availability of order items, and if successful trigger the AssembleOrder use-case.

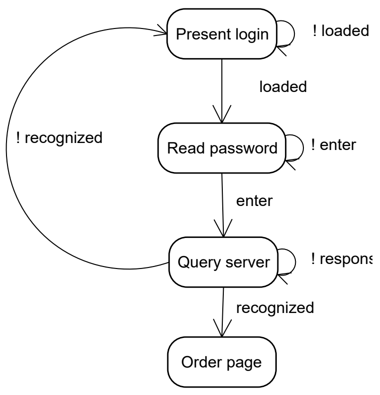


Figure 1.7: State diagram for the v-Grocer customer login.

into presentations to non-technical audiences. Finally, use-cases are simply not busy work—they are fundamental to the development of other UML models.

1.6.3 State Machine View

The ***state machine view*** is characterized by a *state diagram* as was discussed in Section 6.2. Again, the intention of a state diagram is to describe systems with memory. Figure 1.7 shows the state view of a customer logging into the v-Grocer web server.

1.6.4 Activity View

The intention of the ***activity view***, characterized by an *activity diagram*, is to describe the sequencing of processes required to complete a task. In UML, the tasks elaborated are the individual use-cases identified in a use-case diagram. Since more than one actor may be involved in completing a task, an activity diagram can express the concurrent nature of tasks. An activity diagram is composed of states, transitions, forks, and joins. Figure 1.8 contains an activity diagram for the v-Grocer order delivery system. The diagram gives a clear visual picture for the activities that need to be completed for an order to be packed and delivered. After a complete order is placed, the flow is forked into two concurrent processing branches. One of these branches is for completion of the order, while the other addresses its scheduling, delivery, and coordination with other deliveries. When both of those processes are completed, they are joined together and then the delivery

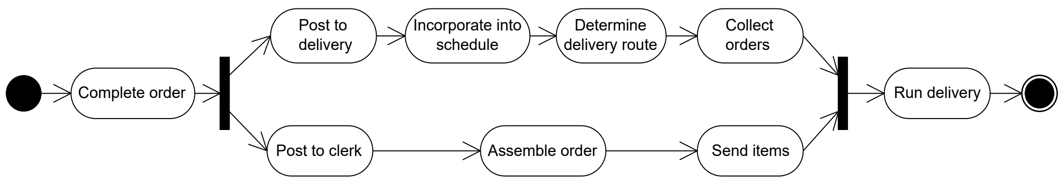


Figure 1.8: Activity diagram for order processing and delivery for the v-Grocer.

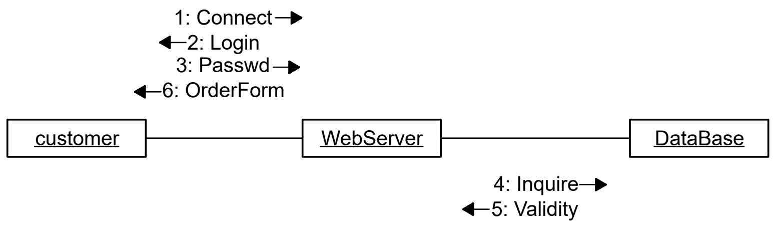


Figure 1.9: Collaboration diagram for the WebOrder use-case.

run is made.

1.6.5 Interaction View

The intention of the ***interaction view*** is to show the interaction between objects. It is characterized by collaboration and sequence diagrams. In an object-oriented system, tasks are completed by passing messages between objects. The interaction view shows how messages are exchanged in order to accomplish a task. These tasks are usually the use-cases. Since messages are sent through time, the interaction view must be able to express the concept of order. We start by examining collaboration diagrams.

Two objects collaborate together in order to produce some meaningful result. A *collaboration diagram* shows the sequencing of messages that are exchanged between classes in order to complete a task. It consists of the classes that participate in the realization of the task and the messages exchanged. The messages are drawn as arrows from the sending object to the receiving object. The message arrows are labeled with the name of the message and its numerical position in the sequence of messages exchanged to realize the task. For example, Figure 1.9 shows how the WebOrder use-case is realized using the Customer, WebServer, and Database classes. Note, WebServer and Database are introduced as new classes and are not shown as part of the class diagram in Figure 6.9.

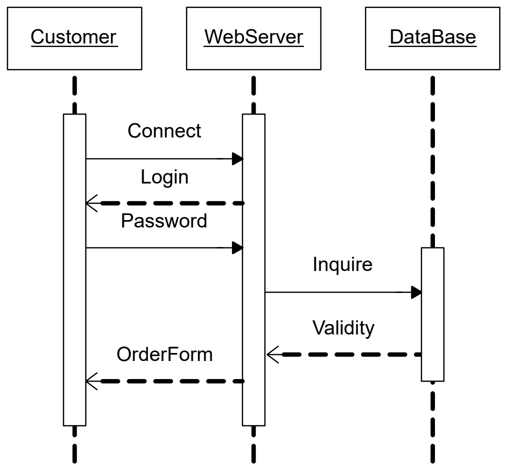


Figure 1.10: Sequence diagram for the WebOrder use-case.

The collaboration diagram is similar in form to the class diagram, the difference being that the relationships are annotated with the messages that are exchanged. Consequently, the collaboration diagram aides the developers in understanding and implementing the methods used between classes. In order to emphasize the order in which messages are exchanged, a developer can also use a sequence diagram.

A *sequence diagram* contains the same information as the corresponding collaboration diagram. Where the collaboration diagram emphasizes the objects that interact to produce a behavior, a sequence diagram emphasizes the message order that produces a behavior. As shown in Figure 1.10, the classes that participate in the behavior are listed in a row at the top of the diagram. From each class a dotted vertical line is drawn downward that represents the lifeline of its class (the vertical axis represents time). When an object is actively requesting or waiting for information from another object, a rectangle is drawn over the dotted lifeline of the object. The message is drawn as an arrow from the sending object to the receiving object and labeled with the name of the message. The activity diagrams can be applied generally to ECE systems to show the interaction between entities, particularly the sequencing of messages.

1.6.6 Physical View

The intention of the ***physical view*** is to demonstrate the physical components of the system and how the logical views map to them. The physical

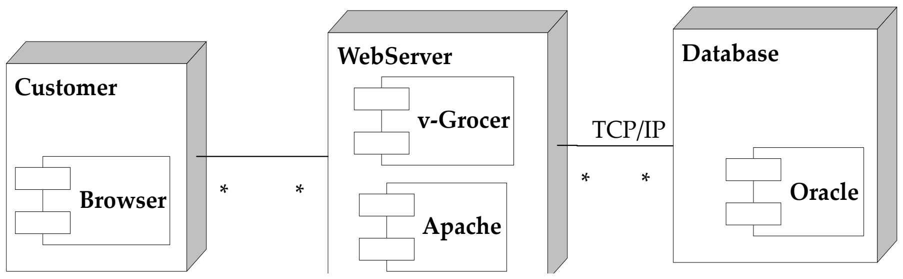


Figure 1.11: The combined component and deployment diagram for the v-Grocer. The software files are the Browser, v-Grocer, Apache, and Oracle, while the hardware components are the Customer, WebServer, and Database.

view is characterized by a component and deployment diagram. A *component diagram* shows the software files and the interrelationships that make up the system. Software files are shown in rectangles. Lines connect together files that need to communicate. A *deployment diagram* shows the hardware and communications components that will be used to realize the system. The hardware components are drawn as cubes and labeled with their names. Figure 1.11 shows a combined component and deployment diagram for the v-Grocer system. The software files are the Browser, v-Grocer, Apache, and Oracle, while the hardware components are the Customer, WebServer, and Database.

1.7 Project Application: Selecting Models

Chapters 5 and 6 have presented a variety of models for representing the behavior of systems. The design team needs to select the correct combination of tools to properly describe a design for its eventual implementation. Table 6.4 provides a summary of the models and their different intentions.

Table 6.4

1.8 Summary and Further Reading

Models provide a convenient method of describing a system at a high level of abstraction. This allows a system to be described without having to determine all of the implementation details. All models are not the same—they come in a variety of forms and each is designed to serve a different intention.

Table 1.8: Guidance for model selection.

Model	**Intention**
Functional Decomposition	To describe the input, output, and functional transfo tions applied to information (electrical signals, bits, en etc.) in a system. It is broad in application. This provides a view that is close to the actual system imple tation. See Chapter 5.
State Diagram	To describe the behavior of systems with memory. The very flexible when it comes to application. They are applied to digital design, but state diagrams can also be to describe the high level behavior of complex systems. only prerequisite on their use is that the system has me See Section 6.2.
Flowchart	To describe a process or algorithm, including its steps control. They are applicable to many problem domains software to describing business practices. See Section
Data Flow Diagram	To model the processing, transformation, and flow of inside a system. They are typically supplemented b event table describing all possible events and the resu actions. Creating a DFD requires the designer to car think about the uses of the system and how the system react to external users and events. See Section 6.4.
Entity Relationship Diagram	To catalog a set of related objects (entities), their attrib and the relationships between them. ERDs capture the ties and relationships of a portion of the world into a fo data model. The graphical language describes the attrib of the entities and the cardinality of the relationships. E are formal enough to be unambiguously translated in complete description of a database system. See Section
The Unified Modeling Language.	The intention of UML is to describe complex software tems. However, certain views are well suited to descri systems at a high level and are applicable to many dom The process of viewing a system from the six perspec listed below decreases the chances that crucial details o design will be overlooked.
Class Diagram	To describe classes and their relationship in an ob oriented software system. Class diagrams are prim for software design and are not easily accessible to a technical audience. See Section 6.6.1.
Use-Case Diagram	To capture the overall behavior of the system from the u point of view and describe cases in which the system w used. See Section 6.6.2.
State Machine	This is essentially the same as the state diagram, but is a formal UML view. See Sections 6.2 and 6.6.3.
Activity Diagram	To describe the sequencing of processes required to com a task. Composed of states, transitions, forks, and j Can show concurrent processes. See Section 6.6.4.
	To show the interaction and passing of messages bet

The properties of effective models were presented, and we saw that they are similar to those of an engineering requirement. Models should encourage innovation by allowing the exploration of alternative implementations. Since models are built from abstractions of the actual system components, they are an effective means for communicating with non-technical stakeholders. Finally, models provide an excellent means of documenting the development of a design from the highest level down to the detailed design.

This chapter presented a variety of models for describing system behavior that included state diagrams, flowcharts, data flow diagrams, entity relationship diagrams, and the Unified Modeling Language. Table 6.4 provides a quick reference that describes the intention and application of the models examined in both Chapters 5 and 6.

Flowcharts have been around for quite some time and an early work describing them is Flowcharts by Chapin [Cha71]. The book Programming Logic and Design [Far02] covers flowcharts extensively and their application in programming. State diagrams are fundamental to the ECE field and further information on them is available in virtually any introductory digital design textbook. Data flow diagrams and entity relationship diagrams are common tools used in information systems analysis and design and can be further explored in Systems Analysis and Design [Sat02] and Fundamentals of Database Systems [Elm94]. Two references for UML are The Unified Modeling Language Reference Manual [Rum98] and Schaum's Outline of UML [Ben01].

1.9 Problems

1. Why is it important for a model to separate the design of a system from its realization?

2. Classify each of the following as either a model, not a model, or sometimes a model. Justify your answer based on the definition and properties of a model.

    - Adiagram of a subway system,
    - a computer program,
    - a football play,
    - drivers license,
    - a floor plan of the local shopping mall (a "you are here" diagram).,
    - an equation,
    - A scratch n' sniff perfume advertisement in a fashion magazine,
    - The 1812 overture,
    - a braille sign reading "second floor",
    - sheet music for the Brandenburg Concerto,
    - the United States Constitution,
    - a set of car keys,
    - the ASCII encoding of an email message.

3. Which of the following systems has memory? Justify your answer using the concepts of input, output, and state.

    - an ink pen,
    - a resistor,
    - a capacitor,
    - a motorized garage door,
    - an analog wrist watch,
    - the air pressure in an air compressor,
    - the thermostat which controls the furnace in a house,
    - a light switch,
    - a political system,

- the temperature of a large lake,
- a book,
- a computer's hard drive.

4. A can of soda has memory. Your objective is to figure out what characteristic of the can is the state variable and what input causes it to change. Based upon this, draw a state diagram for a can of soda. Label the transition arcs with the input responsible for the transition. Hint: no special equipment is needed to elicit the change of state.

5. Consider the state diagram for the vending machine shown in Figure 6.2. Now assume that the system accepts nickels, dimes, and quarters. Also assume that it is capable of returning change to the user after a purchase. Create a state diagram that represents this new system. Make sure to define the output signals and their value for each state.

6. Use a state diagram to describe the high level operation of the ChipMunk Recorder (CMR). The CMR records sounds and then plays them back at a variety of speeds, making a recorded voice sound like a high pitched chipmunk. The CMR receives user input from a keyboard and an audio source. The behavior of the system is described as follows:

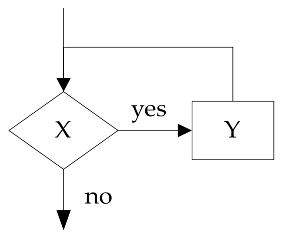
   - When powered up, the CMR enters a wait state.
   - If 'R' is pressed the recorder begins recording.
   - Any key-press will put the CMR back into the wait state.
   - If 'S' is pressed, the CMR is ready to change the playback speed. A subsequent numerical input between 1 and 5 will cause the playback speed to be changed to that value.
   - Pressing the 'R' key when in the adjust playback speed mode will cause the CMR to go to the wait state.
   - Pressing a 'P' key will cause the CMR to playback the recorded sounds. When done playing the entire recording, the CMR will loop back and start playing at the beginning.
   - Any key-press while in the playback mode will cause the CMR to go back into the wait state.

   Draw a state diagram describing the behavior of the CMR. Create a table which lists every state and its associated output.

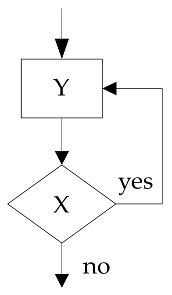
7. Build a state diagram to describe the state of the tape cartridges used to backup a company's network drives. When new **unformatted** cartridges are received they are immediately labeled with a unique ID. Before a tape is used it is formatted turning it into a **blank** tape. On the first day of the week a complete backup is made of the network drives, transforming blank tapes into **active** tapes. The active tapes, made every fourth week, are moved off site making them **archival** tapes. Active tapes older than 3 months are assumed to have out of date information and are reformatted into blank tapes. Archival tapes more than 2 years old are reformatted and put back into circulation.

8. Build a flowchart to describe the operation of a microcontroller (MCU) based temperature regulating system. The system monitors the temperature of a heated environment using thermistors and regulates the temperature by turning fans on to cool the environment. The MCU periodically reads the temperature from each of the 64 different thermistors (each is driven by its own constant current source) by selecting each through an analog multiplexor. The voltage is converted into an 8-bit digital value using the MCU's analog-to-digital converter. If any of the 64 thermistors exceeds a high temperature threshold, the MCU uses a complex algorithm to determine the number of fans to turn on, otherwise all the fans are turned off.

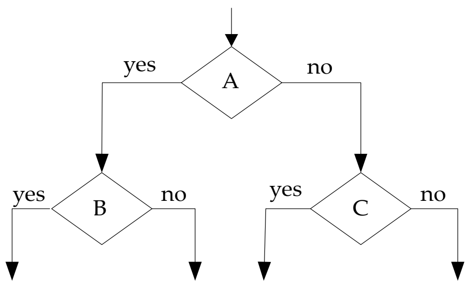
9. Write an algorithmic description for each of the flow charts below using while, if, or do statements.



(a)



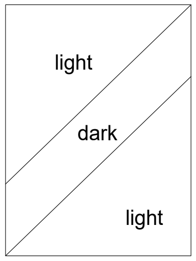
(b)



(c)

10. Create a flowchart that outlines how to crochet a two-tone blanket with a diagonal stripe across it as shown below. A blanket is crocheted by linking together a sequence of basic stitches. For the purposes of the flowchart assume that a basic stitch is an elaborate process. Basic stitches are made from either dark or light yarn. The blanket should

be 100 stitches wide by 150 stitches high. The diagonal stripe runs at a 45 degree angle from the horizontal.



11. Build a data flow diagram and event table to represent an image archiving system for an art museum. The art museum maintains a database of digital images of paintings from museums all over the world. The following information is known about the image database system:

    - Images are shared among museums in a participating network. Whenever a participating museum posts a new image, it sends a broadcast email to all participating museums in the network with the image attached as an email. It provides the name of the painting and artist in the body of the email. All new images received are added directly to the museum's own image database.
    - When inserted into the museum's database, each image is provided with a tag identifying the name of the painting and the artist. Furthermore, this triggers an image analysis routine that classifies the image based into predefined categories such as portrait, natural scene, and modern. Furthermore, it stores key features that are extracted from the image.
    - The key features are used to identify and retrieve visually similar images from the museum's database. Another image processing algorithm is run that compares the visual similarity of the new image to all images in the database. This process produces a matching score of 0 to 100 that is stored.
    - The museum's image database is available to visitors via computer kiosks placed throughout the museum. Kiosk users can retrieve and view images in one of three ways. First, they can specify the name of the artist or painting. Second, they can retrieve a class of images, such as modern. Third, once they have received a painting, they can submit a request to view visually

similar images. The visually similar images are retrieved for viewing based upon the matching score.

12. Build an ERD to keep track of the bicycle frames manufactured at a local company. The following are notes from an interview with the owner.

    We custom build bike frames to the dimension of each individual customer. When a customer comes in we take measurements in their height, leg length, arm length, torso length, weight, and waist measurement. Since we have high customer satisfaction our customers order new frames every several year. Hence we would like to date these measurements in order to track how a customer's body changes through time .Each frame is built on one set of measurements. Clearly, we need to keep track of our customer's contact information like name, address, phone number, and email address. We would like to know which employee built which frame. We would like to store basic information like name, address, phone, and SSN for each employee. Each frame is built by one employee using a variety of different titanium tubing. We have strict inventory control on all of our tubing and need to keep track of its grade, lot #, Outer Diameter (OD), Inner Diameter (ID), and manufacturer. Tubing is uniquely identified by its lot #. Finally we need to keep information on the frame. Each frame is given a unique serial number, and has a color, type, and dimensions.

13. Extend Problem 6.7 to create an ERD that captures data about the tape cartridges used in the backup system. Every Sunday night a full backup is made of all network drives. A full backup creates an identical copy of the network drives on the tape cartridges. Due to the large amount of information, a full backup requires many tapes. On the other nights of the week an incremental backup is made. An incremental backup stores only files modified since the last backup (either full or incremental). Incremental backups are much smaller than a full backup, and consequently many incremental backups fit on a single tape. A tape contains only full or incremental backup information – the unused portion of the last tape used for a full backup is never used to store incremental backups. Your company wants to

keep track of tapes, full backups, and incremental backups. An ID and state should be tracked for each tape. For full backups, the system needs to track the creation date and the number of tapes used. For an incremental backup it should track the date it was made. The relationships between the backup type and the tape will capture which tapes participated in which backup. (Hint: the state of a tape should be an attribute of the tape entity - unformatted, blank, etc. They are not attributes and are possible values for the state attribute.)

14. **Project Application.** Develop behavior models that are applicable for describing your system. Table 6.4 is provided to help in making the determination as to which models are applicable

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).