Learn Computer Science with Swift

Computation Concepts, Programming Paradigms, Data Management, and Modern Component Architectures with Swift and Playgrounds

Jesse Feiler

Apress®

Learn Computer Science with Swift

Computation Concepts, Programming Paradigms, Data Management, and Modern Component Architectures with Swift and Playgrounds

Jesse Feiler

Apress®

Learn Computer Science with Swift: Computation Concepts, Programming Paradigms, Data Management, and Modern Component Architectures with Swift and Playgrounds

Jesse Feiler
Plattsburgh, New York, USA

ISBN-13 (pbk): 978-1-4842-3065-7 ISBN-13 (electronic): 978-1-4842-3066-4
https://doi.org/10.1007/978-1-4842-3066-4

Library of Congress Control Number: 2017962300

Cover image designed by Freepik

Managing Director: Welmoed Spahr
Editorial Director: Todd Green
Acquisitions Editor: Aaron Black
Development Editor: James Markham
Technical Reviewer: Aaron Crabtree
Coordinating Editor: Jessica Vakili
Copy Editor: Karen Jameson
Compositor: SPi Global
Indexer: SPi Global
Artist: SPi Global

Table of Contents

About the Author

Jesse Feiler is an author and developer focusing on nonprofits and small businesses using innovative tools and technologies. Active in the community, he has served on the boards of Mid-Hudson Library System (including three years as president), Philmont Main Street Committee, Philmont and Plattsburgh Public Libraries, HB Studio and Playwrights Foundation, Plattsburgh Planning Board, Friends of Saranac River Trail, Saranac River Trail Greenway, and Spectra Arts.

His apps include *NP Risk — The Nonprofit Risk App* (with Gail B. Nayowith), *Saranac River Trail*, *Minutes Machine*, and *Utility Smart*. They are available through Champlain Arts on the App Store at http://bit.ly/ChamplainArts.

His large-scale projects have included contingency planning and support for open market monetary policy and bank supervision operations for the Federal Reserve Bank of New York's Systems Development and Data Processing functions as chief of the Special Projects Staff and the System Components Division; implementation of the Natural Sales Projection Model at Young & Rubicam (the first computer-based new product projection model); development of the Mac client for Prodigy to implement their first web browser; management information systems and interfaces for legal offices, Apple, and The Johnson Company; as well as consulting, writing, and speaking about the Year 2000 problem.

Smaller-scale projects for businesses and nonprofits have included design and development of the first digital version of *Josef Albers's Interaction of Color* (for Josef and Anni Albers Foundation and Yale University Press), database and website development for Archipenko Foundation, along with rescue missions for individuals and organizations who found out about contingency planning when they least expected to learn about it. Together with Curt Gervich, Associate Professor at State University of New York College at Plattsburgh, he created Utility Smart, an app to help people monitor their use of shared natural resources.

Jesse is founder of Friends of Saranac River Trail and of Philmont Main Street Committee. He is heard regularly on The Roundtable from WAMC Public Radio for the Northeast where he discusses the intersection of society and technology. He is a speaker and guest lecturer as well as a teacher and trainer specializing in the business and technology of iOS app development. He also provides consulting services for organizations that need help focusing on their objectives and the means to achieve them with modern technology. He is co-author with Gail B. Nayowith of *The Nonprofit Risk Book* as well as *The Nonprofit Risk App* — NP Risk).

About the Technical Reviewer

A passionate developer and experience enthusiast, **Aaron Crabtree** has been involved in mobile development since the dawn of the mobile device. He has written and provided technical editing for a variety of books on the topic, as well as taken the lead on some very cool, cutting-edge projects over the years. His latest endeavor, building apps for augmented reality devices, has flung him back where he wants to be: as an early adopter in an environment that changes day by day as new innovation hits the market. Hit him up on Twitter where he tweets about all things mobile and AR: @ aaron_crabtree

Introduction

Computer Science is the study of computers and their operations. It includes concepts of computability and how software is designed that are now being taught to students as young as six years old. It also includes complex concepts of the largest, latest, and most advanced computers and systems. This book provides an introduction to people who want to learn the basics for practical reasons: they want to understand the principles of computer science that will help them to become developers (or better developers). The focus is on practical applications of computer science. Along those lines, Swift, the modern language developed originally at Apple, is used for many examples that are shown in Swift playgrounds. You will find practical discussions of issues as varied as debugging techniques and user-interface design that are essential to know in order to build apps today. Note that Swift playgrounds are used to demonstrate a number of computer science concepts, but this is not a book solely about Swift. Not all of the language constructs are demonstrated in the book.

There is one critical piece of advice I give to people who want to learn how to develop apps, and that is to use them. Download and try to use every app that you possibly can. Read reviews of apps. Talk to people about their experiences. Too often, people jump into trying to write apps without knowing what the state of the art (and of the marketplace) is today.

Many people have helped in the development of this book. Carole Jelen of Waterside Productions has once again been instrumental in bringing the book into being. At Apress, Jessica Valiki and Aaron Black have been essential guides and partners in helping to shape the book and its content.

In the course of writing this book, I've been lucky enough to be involved in several app development projects that have provided case studies and examples of the process of app development. Thanks are due particularly to Curt Gervich, Maeve Sherry, and Michael Otton at Center for Earth and Environmental Science at State University of New York College at Plattsburgh as well as Sonal Patel-Dame of Plattsburgh High School.

Downloading Playgrounds for the Book

You can download playgrounds from the book from the author's website at champlainarts.com.

CHAPTER 1

Thinking Computationally

Computer science is the term that applies to the basic principles involved in developing computer software and systems that incorporate that software. It is abstract and theoretical in the sense that it typically is considered outside the syntax and structure of specific computer languages and hardware.

That is the definition that we use in this book. If you explore other books and articles on the Web (including descriptions of computer science courses at all levels and types of education), you will find a wide array of other definitions.

This chapter provides an overview of the topic and focuses on key elements of computer science. This book provides a practical approach to computer science, so you'll see how the elements can fit into your work rather than looking at a theoretical view of computer science. The focus is on how you will use the concepts and principles of computer science in building real apps.

The key elements of computer science are divided into two groups in this book. The first is the pair of concepts that developers use as part of their work every day:

- Recognizing patterns
- Using abstractions

J. Feiler, *Learn Computer Science with Swift*, https://doi.org/10.1007/978-1-4842-3066-4_1

Then you'll see the four tasks that are used in every aspect of software development from the largest system to the smallest component of a tiny system. These tasks are the following:

- Formulating a computational problem
- Modeling the problem or process
- Practicing decomposition
- Validating abstractions

In the remaining chapters, you'll find descriptions of syntax elements and structures, but in this chapter, the focus is on the concepts you use to carry out the basic software development tasks that are over and above syntax and structure.

Computer Science Today

Computer science principles and techniques are implemented in computer hardware and software using various programming languages and devices. Even users get into the picture as they learn to enter data, share it with others, convert data from one format to another (think spreadsheet to email) and a host of other tasks that demonstrate computer science in action.

One of the challenges in teaching and learning computer science is that in order to learn the principles, you have to have enough knowledge and experience of computer hardware and software to understand how they interact with computer science principles.

This has been a tremendous challenge for decades. If you want to learn how to be a builder, you can start by building a doll house or a bird house. Your materials might consist of paper and (if you want a permanent structure) some glue or even staples. The basic principles of home construction can be simply demonstrated and described.

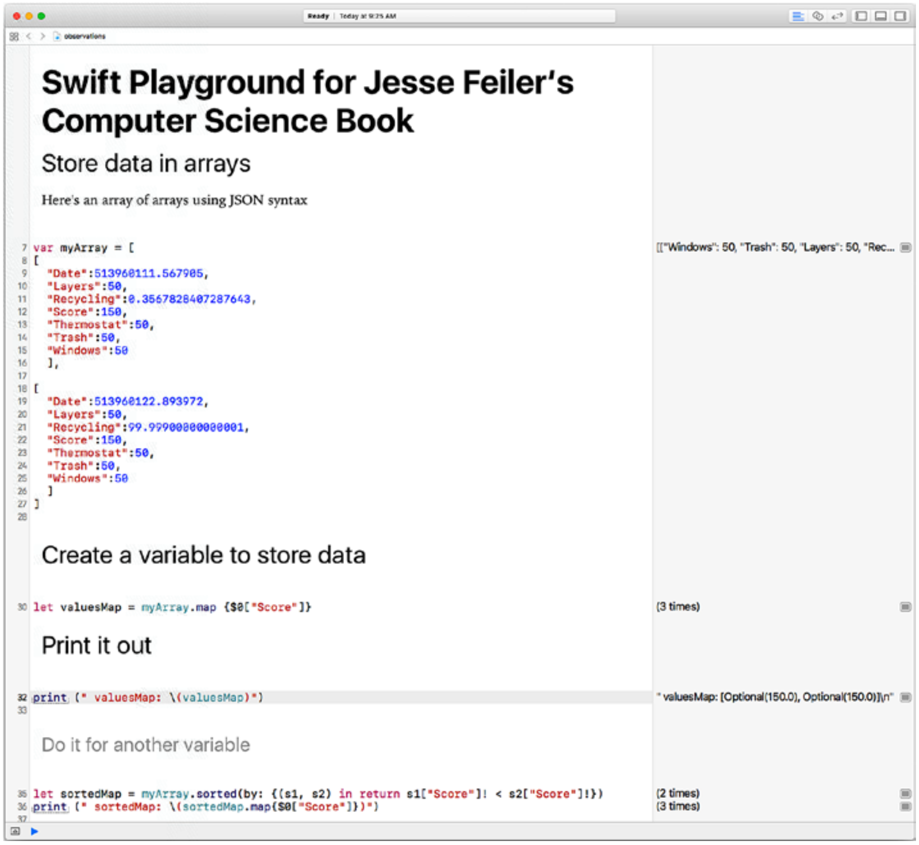
The challenge with computer science is that to build a small project, you may be able to write a single line of code, but, in order for it to run and do something – anything – you need a computer, and it needs an operating system. (This was true going back to the earliest days of computers.) The computer today will consist of electronic components, and the operating system today of even the most minimal computer is incredibly complex. The steps you take to get to a "simple" computer science app are enormous.

Using Swift Playgrounds

If you have an iPad or Mac, you have access to Apple's free Swift Playgrounds tool. Together with the device itself, you have all of the components you need to start to build simple apps with even a single line of code.

Swift Playgrounds provides a massive infrastructure on top of which you can write a few lines of code to start to explore computer science as well as specific languages and techniques. You can run this code in the playground and watch the results. (You can also modify the results and the code as it is running if you want to experiment).

For your own use or for others, you can easily annotate the code. Figure 1-1 shows a playground with annotations. It is running, and you can see the result of the print statements at the bottom of the window. At the right, you see a sidebar that monitors the code as it runs.



Swift Playground for Jesse Feiler's Computer Science Book

Store data in arrays

Here's an array of arrays using JSON syntax

```
var myArray = [
[
  "Date":513960111.567985,
  "Layers":50,
  "Recycling":0.3567828407287643,
  "Score":150,
  "Thermostat":50,
  "Trash":50,
  "Windows":50
  ],

[
  "Date":513960122.893972,
  "Layers":50,
  "Recycling":99.99900000000001,
  "Score":150,
  "Thermostat":50,
  "Trash":50,
  "Windows":50
  ]
]
```

```
[["Windows": 50, "Trash": 50, "Layers": 50, "Rec...
```

Create a variable to store data

```
let valuesMap = myArray.map {$0["Score"]}
```

```
(3 times)
```

Print it out

```
print (" valuesMap: \(valuesMap)")
```

```
" valuesMap: [Optional(150.0), Optional(150.0)]\n"
```

Do it for another variable

```
let sortedMap = myArray.sorted(by: {(s1, s2) in return s1["Score"]! < s2["Score"]!})
print (" sortedMap: \(sortedMap.map{$0["Score"]})")
```

```
(2 times)
(3 times)
```

Figure 1-1. *Swift Playgrounds in action*

If you see Figure 1-1 in color, you can see that the elements of code syntax are colored automatically to help you understand what is going on in the code. This coloring and indentation happens automatically as you type.

In this book, you will find a number of examples of computer science principles that are demonstrated with Swift Playgrounds. You can download them as described in the Introduction.

One very important point to know about Swift Playgrounds is that the code you are writing is real. It is real in the sense that it is actual code written in the Swift programming language (the language for most iOS, tvOS, and watchOS apps today as well as a number of macOS apps). You can copy some code from an app you're working on and paste it into a playground so you can experiment with it. (There are some details on how to do this in Chapter 7).

Note The playground shown in Figure 1-1 is a real-life example of using production code in a playground. This code is part of an app that was not doing exactly what it should. It was isolated into a playground where we could fiddle with the syntax until it worked properly. Once that was done, the revised code was pasted back into the app, and it's now part of Utility Smart that you can download for free in the App Store. If some of the code in Figure 1-1 seems complex, you are right. It was line 35 that caused the confusion. You'll find out about the `map` function in Chapter 3.

Basic Concepts and Practices of Computer Science Today

These are the basic concepts and practices that developers use in their everyday work whether it is designing complex systems or writing very simple apps. They apply to software that is used for games, for accounting, for managing assets (real estate or digital media), or just about anything else people want their computers to do. If you want details of the history of computer science and the major steps to today's world, you can find a great deal of information on the Web and in your local library. This section is based on actual developers' work.

You can learn these over time as you develop apps, and you can find them in many books and articles. These concepts and practices are not specific to computer science: they are part and parcel of many design and development disciplines. (Don't worry, the following section is devoted specifically to software development).

Both of these concepts and practices stem from a very basic truth: writing code is a complex and expensive process. Not only does the code have to be written, but it also needs to be tested and revised over time. Computer code can have a very long life. (When the Year 2000 problem was addressed in the late 1990s, code from the 1950s and 1960s was found in many production systems. The authors of the code in many cases were retired or deceased, and what documentation that might have existed was lost. Much of the cost of mitigating the Year 2000 problems derived from rewriting existing code).

Because writing code is expensive, it is wise to minimize the amount of code to be written and rewritten and tested. Both of these concepts help to minimize the amount of code to be written. The overall theme is that to write the best code possible (that is, well-written, well-tested, and well-documented code) as quickly as possible, follow one simple rule: Don't Write Code. Failing that, write as little code as possible. And, to put it in a more traditional way, use as much existing code as possible.

Recognizing Patterns

If you recognize patterns, you may be able to reduce the amount of work you have to do by seeing a pattern and realizing that you can implement the pattern itself rather than each particular variation of it from scratch.

A classic example of patterns is shown in Figure 1-2, the west front of Notre Dame in Paris. Your first reaction may be personal (perhaps you have been to Paris) or it may be general – along the lines of how beautiful the facade is. An architect, designer, or software developer might go beyond the personal and the general to notice that this facade consists of three doorways at the street level and two towers at the top level.

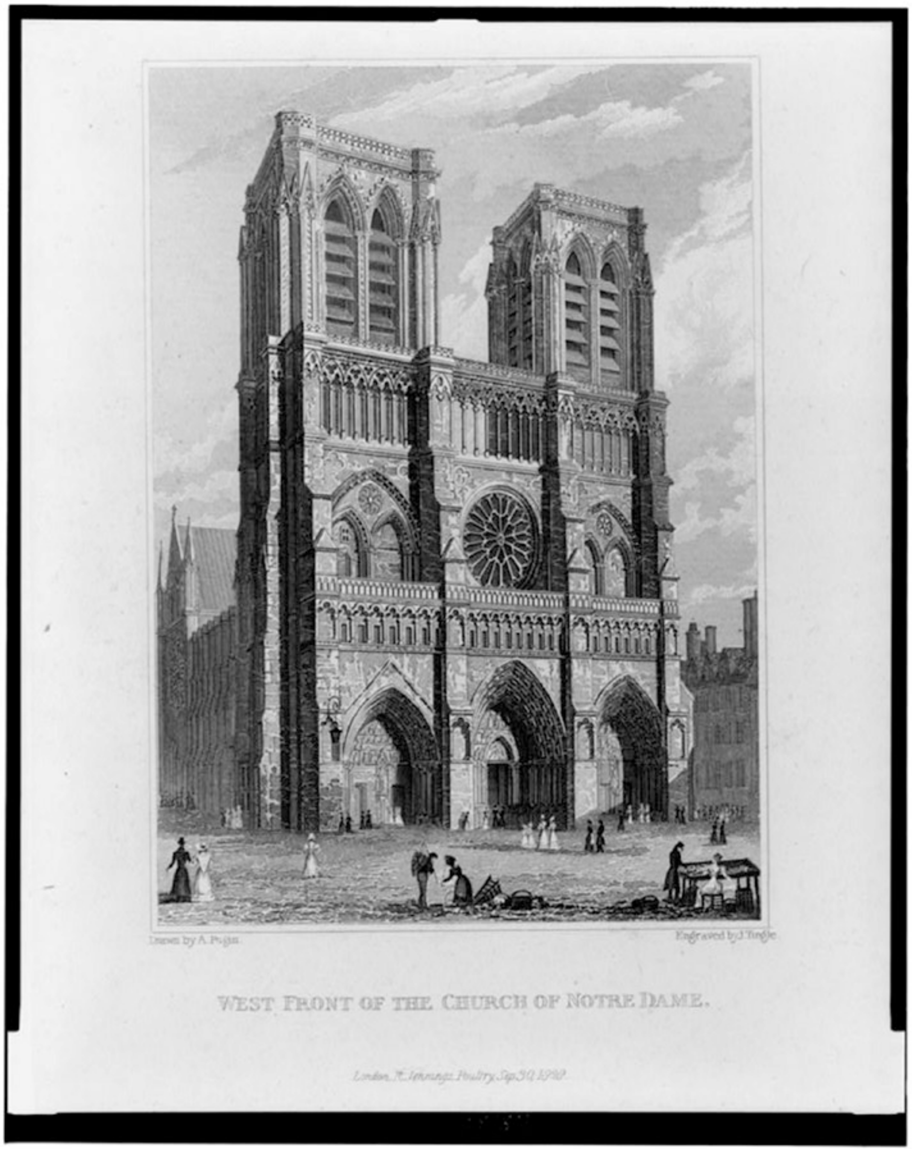

Drawn by A. Pugin
Engraved by J. Tingle
WEST FRONT OF THE CHURCH OF NOTRE DAME.
London, R. Jennings, Poultry, Sep. 30, 1829.

Figure 1-2. *West front of Notre Dame, Paris*

The west front of Notre Dame presents a multitude of patterns that repeat with slight variations. The three doorways at the first level are similar in overall width and height, but if you look closely, they are not copies of one another. Likewise, the two towers are fundamentally the same, but they, too, have variations. Almost every other element of the facade is part of a repeating pattern of one sort or another. (The most obvious exception to this is the large rose window in the center of the second level: it is unique, and its uniqueness reflects its religious importance).

The importance of recognizing patterns is that once you do so, your job in describing or implementing a concept (be it an app or a cathedral) may be made easier. You no longer have to describe or build each detail or component: you can describe the pattern that is replicated.

Using Abstractions

Often, as is the case on the west front of Notre Dame, patterns are repeated with variations. (The dimensions of the doorways are the same but the decoration and meaning of the statues differ.) The part of the pattern that repeats can be considered an abstraction – the essence of the pattern. In computer terms, the abstraction can be what you need to implement to support multiple uses of the pattern.

For example, if you need code to ask the user of an app for an address, that can become part of a pattern that also allows you to ask the user for a name. (The term *design pattern* is sometimes used to describe the reusable code).

Combining Patterns and Abstractions for Development

In practice, developers often work with patterns and abstractions at the same time because they are really two sides of the same coin. In designing an app (or a part of an app), developers look to patterns that they can implement with the same basic code. This reduces the amount of code that needs to be written.

As the design process continues, developers also look for near-patterns. If parts of the project can be modified slightly, a pattern may emerge. This is an iterative, creative, and judgmental process. Frequently, the extreme of pattern-building may make the app more complex for people to use. As a project evolves with input from users and developers, refinements can be made on both sides (user and developer) so that a good balance is made between repetitive patterns and customization for the user.

As part of this process, you frequently find yourself looking at the suggested process to see not only if there is some pattern to reuse but also if there is an abstraction that can be created so that the user sees extreme customization (that is, ease of use) and the developer works on a generic abstraction).

A lot of the coding techniques you'll find in modern software development help you to implement patterns and abstractions.

Fundamental Tasks for Developers

Building on the basic principles of patterns and abstractions, you can actually start to plan your project. There are four basic tasks for developers. Once you're familiar with them, the rest of the book explores specifics of implementation.

- Formulating a computational problem
- Modeling the problem or process

- Practicing decomposition
- Validating abstractions

Formulating a Computational Problem

The first step is formulating your project as a computational problem. This is more than just saying, “Let’s build an app.” It means deciding not only *what* your goal is but also *why* it is amenable to computation (that is, why computer science comes into play). Computer science isn’t the answer to everything: if you want to paint the dining room, it’s not going to be of much help.

In theoretical computer science, there are at least five types of computational problems. In deciding whether or not a specific project is amenable to computerization, classic computer science suggests that you find if it falls into one of these categories:

- Choice or decision. Find a yes/no answer to a specific question. Typically, the question is phrased in terms of numbers and values (is person X greater than 21 years of age?, is value x odd or even?)
- Search. In this problem, a body of data is searched and the choice/decision true values are returned. (Of all students enrolled in a school, how many will be eligible to vote in the next election?)
- Count. This variation asks merely how many values would be returned from a search. Note that the operations involved in a search can be more complex than in a count – you don’t care who the students are in this case so you don’t need to find out names or addresses.

- Optimization. Of all results of a search, which is the best? If the search is for all eligible voters near a specific address, you can use the results to optimize the result to find the voters near a specific address who voted in the last election and have a car (so might be willing to provide a ride to the polling place).
- Function. In effect, this is a search problem (which in turn is built on a choice problem). It is further refined with the optimizable results that can be further narrowed down. A simplified description of a function problem is one that returns a more complicated answer than yes/no or a count. (Remember, this is a simplification.)

If a problem is not one of these five, it is not computational. This may sound bizarre because it's hard to see where something like Pages or Excel or even Swift Playgrounds fits into this list. Never fear: a project can be broken down into computational pieces. In fact, if you really want to delve deeply into the project, you'll find that each line of code can often be considered to consist of a number (often many) of computational pieces.

Recognizing and Describing the Problem

Once you have formulated the problem, your task isn't over. There are still two very important aspects involved in formulating an idea for an app. In fact, these are steps that you take at the beginning and, repeatedly, at many stages through the development process. You may be chomping at the bit wanting to get into code and technology, but you have to start with the idea: what is the purpose of your project? If it's to build an app, what does the app do?

Perhaps the best guidance in formulating what your app does can be found on websites like Kickstarter or any other resource that helps people describe a not-yet-built project. You can answer any number of specific questions, but you must somehow know what your project or app will accomplish.

Many developers are happy to leave the marketing to other people, but you must be able to describe the project in clear and specific terms for many purposes beyond marketing. In the case of an app, one critical step in the development process is getting an icon for the app. Icon design is a very special area of design and graphic arts. Few developers produce final icons (many provide rough sketches for development). You are likely to need to sit with a graphic designer to discuss what the icon will look like. That conversation starts out with the designer's question: what does this app do?

Tip The conversation between app developer and designer can be particularly useful in the development process because it can clarify the project. This applies to any discussion with a non-developer. Describing the app to a friend or relative can be very productive: they tend to ask basic questions that can help you refine your design.

Defining a Project and Goal

With a computational problem that you want to focus on and a description of the problem in hand, you can move to defining a project and your goal. The project in general is to refine the computational problem at the core of your project and to make sure you can define it in appropriate terms for anyone who needs to know about it (friends, relatives, colleagues, investors, potential users, and the like).

Specifically, you need to start thinking about the scope of your project. Part of computer science is learning to define projects and split them into component parts if necessary. For a specific project, you may want to think about how to break it into manageable components even if you intend to do it in one process. Knowing how to split it apart if necessary can be a helpful backup plan in case you need to do it in the future.

What Isn't a Computational Problem

The most common non-computational problems you run across tend to involve people and data. (Note that this is an entirely subjective point of view based on personal experiences. But it is shared by many developers).

Sometimes an app is envisioned as something almost magical – it will provide the answer to a question posed by the user. If you cannot break down the problem into computational components, you can't answer the question. In thinking and talking about a problem, you may want to pose the question: how will we do this? You don't need to look for an answer in code at this point; rather, you need to know how the problem under discussion can be resolved. If it involves a person's judgment and that judgment cannot be quantified, it's hard to see how it can be computational. If it involves referring to data and the data is not available, you also have a non-computational problem. You may be able to break a judgment down into computational components, but, ultimately, if you are left with judgment that cannot be computed ("gut feeling" is a term some people use for this), you need some tool other than computer science.

Tip Although not all problems are computational, you can frequently use a computational formulation to crunch numbers and display data so that a judgmental kernel is left. Users can use your app to clear away every computational issue and then use their own judgment on that non-computational kernel.

Modeling the Problem or Process

As soon as you can formulate the problem and the part(s) of the problem that your project encompasses, you can start to model the problem. At this stage you can use any tools that you want to - pencil and paper, smart board, iPad, or anything else. You might want to draw boxes that perform parts of the task you want to build. Don't worry about code - just think about something (whatever it turns out to be) that, for example, computes a person's telephone number (yes, that is a computational problem - a search).

This model might turn out to be how your app is structured, but at this stage, it is just how parts of your app will do things that together make up the entire app. What you want to do at this point is to decide if this collection of tasks or operations (the terms are interchangeable in this context) can produce the results you need. Once you have a rough model, try to break it. What happens if the phone number lookup fails or returns the wrong number? What other components will be impacted?

Don't worry about every loose end in a high-level model, but many people keep a list of these loose ends and assumptions. It's very easy to start assuming that they are all dealt with later on and, without that list of assumptions, you can wind up with an almost-ready app that misses a critical component. (Any developer can recount many examples of this).

Practicing Decomposition

Once you have a conceptual model, it's time to drill down into it: take each component apart and look at its components. (This process is known as *decomposition*.) As you decompose the entire project into smaller and smaller parts, you are often going to be specifying components that will be implemented in code.

As you decompose the model, you may start to realize that this or that component is something that you know how to implement already or that can be implemented using known resources. If you are very lucky, your decomposed project can be implemented with very little additional work.

Rearranging and Recomposing the Project Pieces

But "lucky" doesn't happen very often. In the real world, what developers often find is that if they make some adjustments to the model, the decomposed pieces may become easier to implement. Perhaps the most important point to make about the entire design process is that until it is actually being implemented, everything should be considered changeable.

Take the project apart and put it back together again as you rethink each component. The goal is to make a project that does what you want it to do and to gradually refine the components into manageable and implementable pieces.

There's not a word about code yet. All of the modeling and decomposition is theoretical. Many developers (including the author) think that the longer you work hypothetically, the more robust your implementation will be. Somehow, moving into the code implementation can be a distraction from the design and planning process. Not everyone agrees with this, but many developers do agree.

Validating Abstractions

One of the most important aspects of computer science is that it gives us a way to talk about the development process and about not-yet-built software. Concepts such as decomposition are formalized ways of working in this realm of not-yet-built software. Of course, when a project is actually implemented, the proof of the pudding is revealed: either it works or it doesn't.