11.1 The Problem Domain: Food Delivery Networks

We do not write software in a vacuum; we study computer science to learn how to use vast computational power to solve real-world problems. As professionals in industry and academia, the programs we create serve a purpose, whether to satisfy a need from a client, to improve individual lives and society, or to advance human knowledge and technology. In the previous chapters of these course notes, we have learned about the fundamentals of programming, mathematical proof, and running-time analysis. We have focused on developed the knowledge and skills required to create and analyse programs.

In this final chapter of the course, we will take what we've learned and apply it to design and implement a large program to solve a real-world problem. As a first step to this, we'll learn about how to approach a new domain to understand how we can apply computer science techniques to both represent and solve problems in that domain.

What is a problem domain?

A **problem domain** is collection of knowledge[1] about a specific field, phenomenon, or discipline, and an understanding of the goals, problems, deficiencies, and/or desired improvements within that area. Each problem domains encompass many different kinds of knowledge, including terminology and definitions, concepts and skills, and context and history. Through your lectures, tutorials, and assignments, you've touched on a wide array of problem domains, such as tracking marriage records, subway delays, and hypertension rates in the City of Toronto, the game of Wordle, and cryptography.

[1] We use the term *domain-specific* knowledge to refer to knowledge about a particular domain. Society often uses the term *domain experts* to refer to people who have a great deal of knowledge in a particular domain.

Let's unpack how we explored the domain of cryptography in Chapter 8. We first introduced the key scenario of two people communicating securely so that their messages could not be deciphered by an eavesdropper. As we dove into cryptography, we learned about:

- *terminology and definitions*: symmetric-key and public-key cryptosystems, encryption and decryption, various existing cryptosystems
- *concepts and skills*: proving that a cryptosystem is correct; justifying the security of a cryptosystem based on the presumed hardness of mathematical problems like Integer Factorization
- *context and history*: ancient cryptosystems, how cryptography is applied to Internet communciations

Our previous study of programming enabled us to write programs, but we had to learn all about the domain of cryptography before being able to implement cryptographic algorithms ourselves. Our knowledge of Python programming alone might have been sufficient to explain what operations are performed on what data in, for example, `rsa_generate_key`, `rsa_encrypt`, and `rsa_decrypt`. But it was the domain-specific knowledge we learned that explained *how* we came up with these algorithms and why they are correct.

Introducing a food delivery system

Now, we'll introduce a new problem domain that we will spend the rest of this chapter studying. Seeing the proliferation of various food delivery apps, you have decided to create a food and grocery delivery app that focuses on students. Your app will allow student users to order groceries and meals from local grocery stores and restaurants. The deliveries will be made by couriers to deliver these groceries and meals—and you'll need to pay the couriers, of course! Your working name for the app is *SchoolEats*, but hey, you can always pick a better name before launching your app.

You think this is a great idea and are incredibly excited. Your friend is a bit more cautious, and wonders how many couriers will be needed to make grocery and meal deliveries in a timely manner, which of course will depend on how many people use the app. You and your friend decide to put the computational skills you've learned in this course to help answer this question.

This problem domain is likely a familiar one; the idea of having food delivered to your doorstep has existed for a long time. The preceding paragraph uses some familiar terminology, such as couriers and deliveries. You may even be familiar with existing apps that already do this, such as *UberEats*, *Skip the Dishes*, or *DoorDash*. When thinking about designing and implementing this app, you are probably considering:

- how restaurants will register with the app and post menus
- how customers will register with the app to browse restaurants and place orders
- how couriers will register with the app to claim orders and deliver them from restaurants to customers
- …and more

Food delivery as a system

We can view food delivery in Toronto as a **system**, which is a group of entities (or agents) that interact with each other over time. Systems modeling is frequently used to conceptualize how an organization operates. The first part of creating a computational model for such a system is to design and implement the various entities in the system—in the case of *SchoolEats*, these are entities like couriers and the users placing orders.

The entities in a system are not static; they change over time. New users sign up and place food orders; couriers pick up meals from restaurants and deliver them to users. For a live app, these events are driven by real humans interacting with the app in real-time. In this chapter, however, we're going to look at another way of driving change in our food delivery system over time. The second part of our computational model is a *simulation* that uses randomness to generate events that cause the system to change over time. For example, our food delivery simulation will specify how often users place an order, taking into account that some times of day are busier than others.

Computational simulations are a powerful tool because they harness the speed and reliability of your computer to perform complex calculations and produce results that can be analysed and visualized. But simulations are reliant on the accuracy of their underlying mathematical models, and are ultimately approximations of the real world. A well-designed simulation allows the programmer to start with a simple model and extend and tweak it in response to new domain-specific knowledge.

Chapter roadmap

Over the course of this chapter, we'll study how to design and implement both of these parts of a computational model for our food delivery platform, *SchoolEats*. This case study will also give us an opportunity to explore the design of a relatively complex software system. We'll use what we've learned about classes to model the entities in a food delivery network, and study a specific kind of simulation known as the *discrete-event simulation*. We hope you're excited to get started!