classes to represent every possible combination, you'll get a combinatorial explosion and thousands of classes. If you have more than a dozen or so classes, see if you can replace some with simple properties.

- Does a class have only a single subclass? If so, then you can probably remove it and move whatever it was trying to represent into the subclass.
- If there a class at the bottom of the hierarchy that is never instantiated? If the `Car` hierarchy has a `HalfTrack` class and the program never makes an instance of that class, then you probably don't need the `HalfTrack` class.
- Do the classes all make common sense? If the `Car` hierarchy contains a `Helicopter` class, there's probably something wrong. Either the class doesn't belong there or you should rename some classes so things make sense. (Perhaps you need a `Vehicle` class?)
- Do classes represent differences in a property's value rather than in behavior or the presence of properties? A simple sales program might not need separate classes to represent notebooks and three-hole punches because they're both simple products that you sell one at a time. You might want a separate class for more expensive objects like computers because they might have a `Warranty` property that notebooks and hole punches probably don't have.

Object Composition

Inheritance is one way you can reuse code. A child class inherits all of the code defined by its parent class, so you don't need to write it again. Another way to reuse code is *object composition*, a technique that uses existing classes to build more complex classes.

For example, suppose you define a `Person` class that has `FirstName`, `LastName`, `Address`, and `Phone` properties. Now you want to make a `Company` class that should include information about a contact person.

You could make the `Company` class inherit from the `Person` class so it would inherit the `FirstName`, `LastName`, `Address`, and `Phone` properties. That would give you places to store the contact person's information, but it doesn't make intuitive sense. A company is not a kind of person (despite certain Supreme Court rulings), so `Company` should not inherit from `Person`.

A better approach is to give the `Company` class a new property of type `Person` called `ContactPerson`. Now the `Company` class gets the benefit of the code defined by the `Person` class without the illogic and possible confusion of inheriting from `Person`.

This approach also lets you place more than one `Person` object inside the `Company` class. For example, if you decide the `Company` class also needs to store information about a billing contact and a shipping contact, you can add more `Person` objects to the class. You couldn't do that with inheritance.

DATABASE DESIGN

There are many different kinds of databases that you can use to build an application. For example, specialized kinds of databases store hierarchical data, documents, graphs and networks, key/value pairs, and objects. However, the most popular kind of databases are relational databases.

DATABASE RANKINGS

To see the top database engines ranked by popularity, go to `db-engines.com/en/ranking`. It's a pretty interesting list.

Relational databases are simple, easy to use, and provide a good set of tools for searching, combining data from different tables, sorting results, and otherwise rearranging data.

Like object-oriented design, database design is too big a topic to squeeze into a tiny portion of this book. However, there is room here to cover a few of the most important concepts of database design. You can find a book on database design for more complete information. (For example, see my book *Beginning Database Design Solutions*, Wrox, 2008.)

The following section briefly explains what a relational database is. The sections after that explain the first three forms of database normalization and why they are important.

Relational Databases

Before you learn about database normalization, you need to at least know the basics of relational databases.

A *relational database* stores related data in *tables*. Each table holds *records* that contain pieces of data that are related. Sometimes records are called *tuples* to emphasize that they contain a set of related values.

The pieces of data in each record are called *fields*. Each field has a name and a data type. All the values in different records for a particular field have that data type.

Figure 6-4 shows a small `Customer` table holding five records. The table's fields are `CustomerId`, `FirstName`, `LastName`, `Street`, `City`, `State`, and `Zip`. Because the representation shown in Figure 6-4 lays out the data in rows and columns, records are often called *rows* and fields are often called *columns*.

CustomerId	FirstName	LastName	Street	City	State	Zip
1028	Veronica	Jenson	176 Bradley Ave	Abend	AZ	87351
2918	Kirk	Wood	61 Beech St	Bugsville	CT	04514
7910	Lila	Rowe	8391 Cedar Ct	Cobblestone	SC	35245
3198	Deirdre	Lemon	2819 Dent Dr	Dove	DE	29183
5002	Alicia	Hayes	298 Elf Ln	Eagle	CO	83726

FIGURE 6-4: A table's records are often called rows and its fields are often called columns.

The "relational" part of the term "relational database" comes from relationships defined between the database's tables. For example, consider the `Orders` table shown in Figure 6-5. The `Customers` table's `CustomerId` field and the `Orders` table's `CustomerId` field form a relationship between the two tables. To find a particular customer's orders, you can look up that customer's `CustomerId` in the `Customers` table in Figure 6-4, and then find the corresponding `Orders` records.

CustomerId	OrderId	DateOrdered	DateFilled	DateShipped
1028	1298	4/1/2015	4/4/2015	4/4/2015
2918	1982	4/1/2015	4/3/2015	4/4/2015
3198	2917	4/2/2015	4/7/2015	4/9/2015
1028	9201	4/5/2015	4/6/2015	4/9/2015
1028	3010	4/9/2015	4/13/2015	4/14/2015

FIGURE 6-5: The Customers table's CustomerId column provides a link to the Orders table's CustomerID column.

One particularly useful kind of relationship is a foreign key relationship. A *foreign key* is a set of one or more fields in one table with values that uniquely define a record in another table.

For example, in the `Orders` table shown in Figure 6-5, the `CustomerId` field uniquely identifies a record in the `Customers` table. In other words, it tells you which customer placed the order. There may be multiple records in the `Orders` table with the same `CustomerId` (a single customer can place multiple orders), but there can be only one record in the `Customers` table that has a particular `CustomerId` value.

The table containing the foreign key is often called the *child table*, and the table that contains the uniquely identified record is often called the *parent table*. In this example, the `Orders` table is the child table, and the `Customers` table is the parent table.

LOOKUP TABLES

A *lookup table* is a table that contains values just to use as foreign keys.

For example, you could make a `States` table that lists the states that are allowed by the application. If your company has customers only in New England, the table might contain the values Maine, New Hampshire, Vermont, Massachusetts, Connecticut, and Rhode Island.

The `Customers` table would be a child table connected to the `States` table with a foreign key. That would prevent a user from adding a new customer in a state that wasn't allowed.

In addition to validating user inputs, lookup tables allow the users to configure the application. If you let users modify the `States` table, they can add new records when they decide to work with customers in new states.

Building a relational database is easy, but unless you design the database properly, you may encounter unexpected problems. Those problems may be that:

- Duplicate data can waste space and make updating values slow.
- You may be unable to delete one piece of data without also deleting another unrelated piece of data.

- An otherwise unnecessary piece of data may need to exist so that you can represent some other data.
- The database may not allow multiple values when you need them.

The database-speak euphemism for these kinds of problems is *anomalies.*

Database normalization is a process of rearranging a database to put it into a standard (normal) form that prevents these kinds of anomalies. There are seven levels of database normalization that deal with increasingly obscure kinds of anomalies. The following sections describe the first three levels of normalization, which handle the worst kinds of database problems.

First Normal Form

First normal form (*1NF*) basically says the table can be placed meaningfully in a relational database. It means the table has a sensible, down-to-earth structure like the kind your grandma used to make.

Relational database products tend to enforce most of the 1NF rules automatically, so if you don't do anything too weird, your database will be in 1NF with little extra work.

The official requirements for a table to be in 1NF are:

1. Each column must have a unique name.
2. The order of the rows and columns doesn't matter.
3. Each column must have a single data type.
4. No two rows can contain identical values.
5. Each column must contain a single value.
6. Columns cannot contain repeating groups.

To see how you might be tricked into breaking these rules, suppose you're a weapons instructor at a fantasy adventure camp. You teach kids how to whack each other safely with foam swords and the like. Now consider the signup sheet shown in Table 6-1.

TABLE 6-1: Weapons Training Signup Sheet

NAME	WEAPON	WEAPON
Shelly Silva	Broadsword	
Louis Christenson	Bow	
Lee Hall	Katana	
Sharon Simmons	Broadsword	Bow
Felipe Vega	Broadsword	Katana
Louis Christenson	Bow	
Kate Ballard	Everything	

Here campers list their names and weapons for which they want training. You'll call them in for instruction on a first-come-first-served basis.

This signup sheet violates the 1NF rules in several ways.

It violates Rule 1 because it contains two columns named `Weapon`. The idea is that a camper might want help with more than one weapon. That makes sense on a signup sheet but won't work in a relational database.

It violates Rule 2 because the order of the rows indicates the order in which the campers signed up and the order in which you'll tutor them. In other words, the ordering of the rows is important. (The order of the columns might also be important if you assume the first `Weapon` column holds the camper's primary weapon.)

It violates Rule 3 because Kate Ballard didn't enter the name of a weapon in the first weapon column. Ideally, that column's data type would be `Weapon` and campers would just enter a weapon's name, not a general comment such as "Everything."

It violates Rule 4 because Louis Christenson signed up twice for tutoring with the bow. (I guess he wants to get *really* good with the bow.)

The signup sheet doesn't violate Rule 5, but that's mostly due to luck. There's nothing (except common sense) to stop campers from entering multiple weapons in each `Weapon` column, and that would violate Rule 5.

Here's how you can put this signup sheet into 1NF.

Rule 1—The signup sheet has two columns named `Weapon`. You can fix that by changing their names to `Weapon1` and `Weapon2`. (That violates Rule 6, but we'll fix that later.)

Rule 2—The order of the rows in the signup sheet determines the order in which you'll call campers for their tutorials, so the ordering of rows is important. To fix this problem, add a new field that stores the ordering data explicitly. One way to do that would be to add an `Order` field, as shown in Table 6-2.

TABLE 6-2: Ordered Signup Sheet

ORDER	NAME	WEAPON1	WEAPON2
1	Shelly Silva	Broadsword	
2	Louis Christenson	Bow	
3	Lee Hall	Katana	
4	Sharon Simmons	Broadsword	Bow
5	Felipe Vega	Broadsword	Katana
6	Louis Christenson	Bow	
7	Kate Ballard	Everything	

An alternative that might be more useful would be to add a `Time` field instead of an `Order` field, as shown in Table 6-3. That preserves the original ordering and gives extra information that the campers can use to schedule their days.

TABLE 6-3: Signup Sheet with Times

TIME	NAME	WEAPON1	WEAPON2
9:00	Shelly Silva	Broadsword	
9:30	Louis Christenson	Bow	
10:00	Lee Hall	Katana	
10:30	Sharon Simmons	Broadsword	Bow
11:00	Felipe Vega	Broadsword	Katana
11:30	Louis Christenson	Bow	
12:00	Kate Ballard	Everything	

Rule 3—In Table 6-3, the `Weapon1` column holds two kinds of values: the name of a weapon or "Everything" (for Kate Ballard).

Depending on the application, there are several approaches you could take to fix this kind of problem. You could split a column into two columns, each containing a single data type. Alternatively, you could move the data into separate tables linked to the original record by a key.

In this example, I'll replace the value "Everything" with multiple records that list all the possible weapon values. The result is shown in Table 6-4.

TABLE 6-4: Signup Sheet with Explicitly Listed Weapons

TIME	NAME	WEAPON1	WEAPON2
9:00	Shelly Silva	Broadsword	
9:30	Louis Christenson	Bow	
10:00	Lee Hall	Katana	
10:30	Sharon Simmons	Broadsword	Bow
11:00	Felipe Vega	Broadsword	Katana
11:30	Louis Christenson	Bow	
12:00	Kate Ballard	Broadsword	
12:00	Kate Ballard	Bow	
12:00	Kate Ballard	Katana	

Rule 4—The current design doesn't contain any duplicate rows, so it satisfies Rule 4.

Rule 5—Right now each column contains a single value, so the current design satisfies Rule 5. (The original signup sheet would have broken this rule if it had used a single `Weapons` column instead of using two separate columns and people had written in lists of the weapons they wanted to study.)

Rule 6—This rule says a table cannot contain repeating groups. That means you can't have two columns that represent the same thing. This means a bit more than two columns don't have the same *data type*. Tables often have multiple columns with the same data types but with different meanings. For example, the `Camper` table might have `HomePhone` and `CellPhone` fields. Both of them would hold phone numbers, but they represent different *kinds* of phone numbers.

In the current design, the `Weapon1` and `Weapon2` columns hold the same type and kind of data, so they form a repeating group.

ROTTEN REPETITION

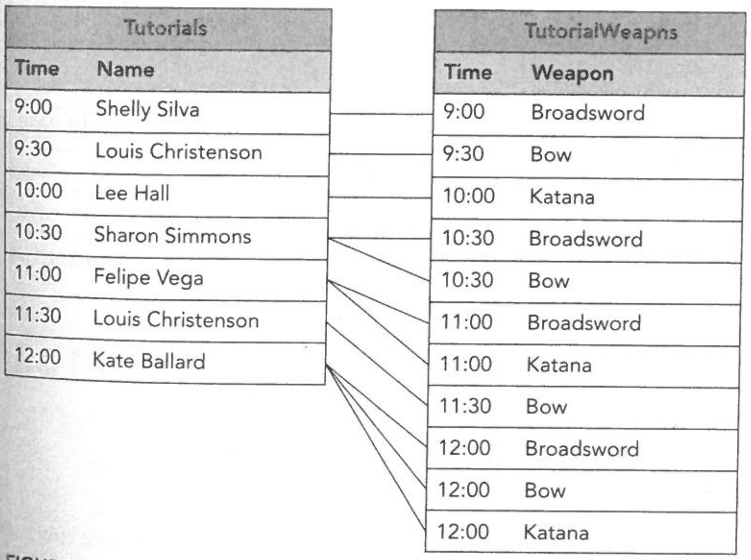
In general, adding a number to field names to differentiate them is a bad idea. If the program doesn't need to differentiate between the two values, then adding a number to their names just creates a repeating group.

The only time this makes sense is if the two fields contain similar items that truly have different meanings to the application. For example, suppose a space shuttle requires two pilots: one to be the primary pilot and one to be the backup in case the primary pilot is abducted by aliens. In that case, you could name the fields that store their names `Pilot1` and `Pilot2` because there really is a difference between them.

Usually in cases like this, you can give the fields more descriptive names such as `Pilot` and `Copilot`.

Another way to look at this is to ask yourself whether the record "Sharon Simmons, Broadsword, Bow" and the rearranged record "Sharon Simmons, Bow, Broadsword" would have the same meaning. If the two have the same meaning even if you switch the values of the two fields, then those fields form a repeating group.

The way to fix this problem is to pull the repeated data out into a new table. Use fields in the original table to link to the new one. Figure 6-6 shows the new design. Here the `Tutorials` and `TutorialWeapons` tables are linked by their `Time` fields.



Tutorials

Time	Name
9:00	Shelly Silva
9:30	Louis Christenson
10:00	Lee Hall
10:30	Sharon Simmons
11:00	Felipe Vega
11:30	Louis Christenson
12:00	Kate Ballard

TutorialWeapns

Time	Weapon
9:00	Broadsword
9:30	Bow
10:00	Katana
10:30	Broadsword
10:30	Bow
11:00	Broadsword
11:00	Katana
11:30	Bow
12:00	Broadsword
12:00	Bow
12:00	Katana

FIGURE 6-6: This design is in first 1NF. Lines connect related records.

Second Normal Form

A table is in *second normal form* (*2NF*) if it satisfies these rules:

1. It is in 1NF.
2. All non-key fields depend on all key fields.

Without getting two technical, a *key* is a set of one or more fields that uniquely identifies a record. Any table in 1NF must have a key because 1NF Rule 4 says, "No two rows can contain identical values." That means there must be a way to pick fields to guarantee uniqueness, even if the key must include every field.

For an example of a table that is not in 2NF, suppose you want to schedule games for campers at the fantasy adventure camp. Table 6-5 lists the scheduled games.

TABLE 6-5: Camp Games Schedule

TIME	GAME	DURATION	MAXIMUMPLAYERS
1:00	*Goblin Launch*	60 mins	8
1:00	*Water Wizzards*	120 mins	6
2:00	*Panic at the Picnic*	90 mins	12
2:00	*Goblin Launch*	60 mins	8
3:00	*Capture the Castle*	120 mins	100
3:00	*Water Wizzards*	120 mins	6
4:00	*Middle Earth Hold'em Poker*	90 mins	10
5:00	*Capture the Castle*	120 mins	100

The table's primary key is `Time+Game`. It cannot have two instances of the same game at the same time (because you don't have enough equipment or counselors), so the combination of `Time+Game` uniquely identifies the rows.

You should quickly review the 1NF rules and convince yourself that this table is in 1NF. In case you haven't memorized them yet, the 1NF rules are:

1. Each column must have a unique name.
2. The order of the rows and columns doesn't matter.
3. Each column must have a single data type.
4. No two rows can contain identical values.
5. Each column must contain a single value.
6. Columns cannot contain repeating groups.

Even though this table is in 1NF, it suffers from the following anomalies:

- **Update anomalies**—If you modify the `Duration` or `MaximumPlayers` value in one row, other rows containing the same game will be out of sync.

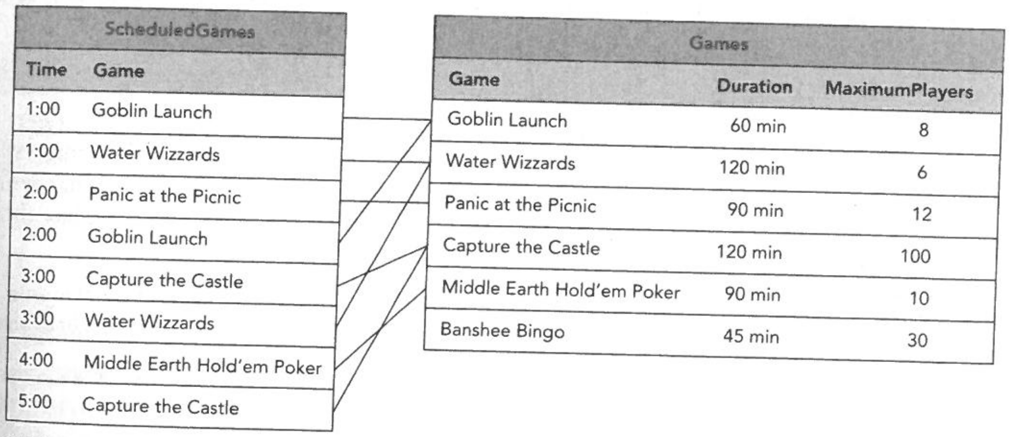
- **Deletion anomalies**—Suppose you want to cancel the *Middle Earth Hold'em Poker* game at 4:00, so you delete that record. Then you've lost all the information about that game. You no longer know that it takes 90 minutes and has a maximum of 10 players.
- **Insertion anomalies**—You cannot add information about a new game without scheduling it for play. For example, suppose *Banshee Bingo* takes 45 minutes and has a maximum of 30 players. You can't add that information to the database without scheduling a game.

The problem with this table is that it's trying to do too much. It's trying to store information about both games (duration and maximum players) and the schedule.

The reason it breaks the 2NF rules is that some non-key fields do not depend on *all* the key fields. Recall that this table's key fields are `Time` and `Game`. A game's duration and maximum number of players depends only on the `Game` and not on the `Time`. For example, *Water Wizzards* lasts for 120 minutes whether you play at 1:00, 4:00, or midnight.

To fix this table, move the data that doesn't depend on the *entire* key into a new table. Use the key fields that the data does depend on to link to the original table.

Figure 6-7 shows the new design. Here the `ScheduledGames` table holds schedule information and the `Games` table holds information specific to the games.



ScheduledGames

Time	Game
1:00	Goblin Launch
1:00	Water Wizzards
2:00	Panic at the Picnic
2:00	Goblin Launch
3:00	Capture the Castle
3:00	Water Wizzards
4:00	Middle Earth Hold'em Poker
5:00	Capture the Castle

Games

Game	Duration	MaximumPlayers
Goblin Launch	60 min	8
Water Wizzards	120 min	6
Panic at the Picnic	90 min	12
Capture the Castle	120 min	100
Middle Earth Hold'em Poker	90 min	10
Banshee Bingo	45 min	30

FIGURE 6-7: Moving the data that doesn't depend on *all* the table's key fields puts this table in 2NF.

Third Normal Form

A table is in *third normal form (3NF)* if:

1. It is in 2NF.
2. It contains no transitive dependencies.

A *transitive dependency* is when a non-key field's value depends on another non-key field's value.

For example, suppose the fantasy adventure camp has a library. (So campers have something to read after they get injured playing the games.) Posted in the library is the following list of the counselors' favorite books, as shown in Table 6-6.

TABLE 6-6: Counselors' Favorite Books

COUNSELOR	FAVORITEBOOK	AUTHOR	PAGES
Becky	*Dealing with Dragons*	Patricia Wrede	240
Charlotte	*The Last Dragonslayer*	Jasper Fforde	306
J.C.	*Gil's All Fright Diner*	A. Lee Martinez	288
Jon	*The Last Dragonslayer*	Jasper Fforde	306
Luke	*The Color of Magic*	Terry Pratchett	288
Noah	*Dealing with Dragons*	Patricia Wrede	240
Rod	*Equal Rites*	Terry Pratchett	272
Wendy	*The Lord of the Rings Trilogy*	J.R.R. Tolkein	1178

This table's key is the `Counselor` field.

If you run through the 1NF rules, you'll see that this table is in 1NF.

The table has only a single key field, so a non-key field cannot depend on only *some* of the key fields. That means the table is also in 2NF.

When posted on the wall of the library, this list is fine. Inside a database, however, it suffers from the following anomalies:

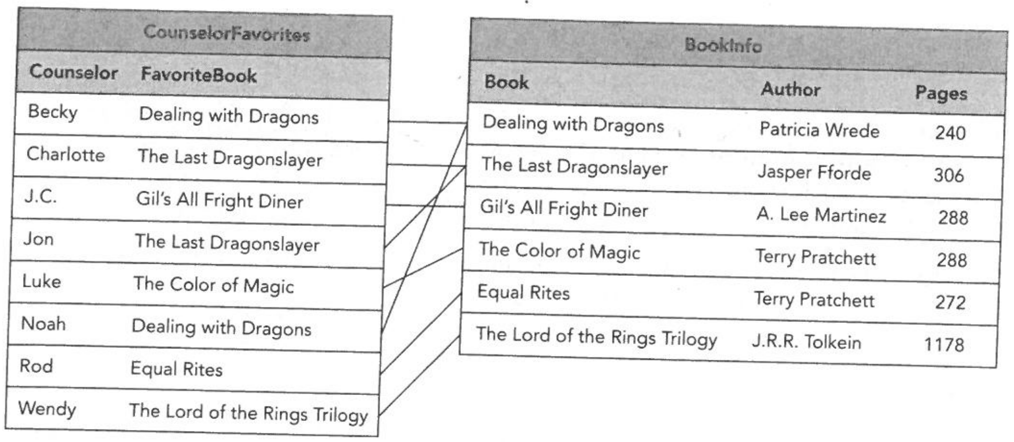
- **Update anomalies**—If you change the `Pages` value for Becky's row (*Dealing with Dragons*), it will be inconsistent with Noah's row (also *Dealing with Dragons*). Also if Luke changes his favorite book to *Majestrum: A Tale of Hengis Hapthorn*, the table loses the data it has about *The Color of Magic*.
- **Deletion anomalies**—If J.C. quits being a counselor to become a professional wrestler and you remove his record from the table, you lose the information about *Gil's All Fright Diner*.
- **Insertion anomalies**—You cannot add information about a new book unless it's someone's favorite. Conversely, you can't add information about a person unless he declares a favorite book.

The problem is that some non-key fields depend on other non-key fields. In this example, the `Author` and `Pages` fields depend on the `FavoriteBook` field. For example, any record with `FavoriteBook` *The Last Dragonslayer* has `Author` Jasper Fforde and `Pages` 306 no matter whose favorite it is.

DIAGNOSING DEPENDENCIES

A major hint that there is a transitive dependency in this table is that there are lots of duplicate values in different columns. Another way to think about this is that there are "tuples" of data (`FavoriteBook+Author+Pages`) that go together.

You can fix this problem by keeping only enough information to identify the dependent data and moving the rest of those fields into a new table. In this example, you would keep the `FavoriteBook` field in the original table and move its dependent values `Author` and `Pages` into a new table. Figure 6-8 shows the new design.



CounselorFavorites

Counselor	FavoriteBook
Becky	Dealing with Dragons
Charlotte	The Last Dragonslayer
J.C.	Gil's All Fright Diner
Jon	The Last Dragonslayer
Luke	The Color of Magic
Noah	Dealing with Dragons
Rod	Equal Rites
Wendy	The Lord of the Rings Trilogy

BookInfo

Book	Author	Pages
Dealing with Dragons	Patricia Wrede	240
The Last Dragonslayer	Jasper Fforde	306
Gil's All Fright Diner	A. Lee Martinez	288
The Color of Magic	Terry Pratchett	288
Equal Rites	Terry Pratchett	272
The Lord of the Rings Trilogy	J.R.R. Tolkein	1178

FIGURE 6-8: Moving non-key fields that depend on other non-key fields into a separate table puts this table in 3NF.

Higher Levels of Normalization

Higher levels of normalization include Boyce-Codd normal form (BCNF), fourth normal form (4NF), fifth normal form (5NF), and Domain/Key Normal Form (DKNF). Some of these later levels of normalization are fairly technical and confusing, so I won't cover them here. See a book on database design for details.

Many database designs stop at 3NF because it handles most kinds of database anomalies without a huge amount of effort. In fact, with a little practice, you can design database tables in 3NF from the beginning, so you don't need to spend several steps normalizing them.

More complete levels of normalization can also lead to confusing database designs that may make using the database harder and less intuitive, possibly giving rise to extra bugs and sometimes reduced performance.

One particular compromise that is often useful is to intentionally leave some data denormalized for performance reasons. A classic example is in ZIP codes. ZIP codes and street addresses are related, so if you know a street address, you can look up the corresponding ZIP code. For example, the ZIP code for 1 Main St., Boston, MA is 02129-3786.

Ideally, normalization would tell you to store only the street address and then use it to look up the ZIP code as needed. Unfortunately, these relationships aren't as simple as, "All Main St. addresses in Boston have the ZIP code 02129-3786." ZIP codes depend on which part of the street contains the address and sometimes even which side of the street the address is on. That means you can't build a table to perform a simple lookup.

You could build a much more complicated table to find an address's ZIP code, perhaps with some confusing code. Or you might use some sort of web service provided by the United States Postal Service.

Usually, however, developers just include the ZIP code as a separate field in the address. That means there's a lot of "unnecessary" duplication, but it doesn't take up much extra room and it makes looking up addresses much easier.

LOADS OF CODES

Addresses and postal codes are also related outside of the United States. For example, the postal code for 1 Main St., Dungiven, Londonderry England is BT47 4PG, and the postal code for 1 Main St., Vancouver, BC, Canada is V6A 3Y5. You can use various postal websites to look up codes for different addresses in different countries.

In theory, you could look up the postal codes for any address. In practice, it's a lot easier to just include them in the address data.

SUMMARY

Low-level design fills in some of the gaps left by high-level design to provide extra guidance to developers before they start writing code. It provides the level of detail necessary for programmers to start writing code, or at least for them to start building classes and to finish defining interfaces. Low-level design moves the high-level focus from *what* to a lower level focus on *how*.

Like most of the topics covered in this book, low-level design is a huge subject. There's no way to cover every possible approach to low-level design in a single chapter. However, this chapter does provide an introduction to two important facets of low-level design: object-oriented design and database design.

Object-oriented design determines what classes the application uses. Database design determines what tables the database contains and how they are related. Object-oriented design and database design aren't all you need to do to ensure success, but poor designs almost always lead to failure.

The boundary between high-level and low-level design is rather arbitrary. Low-level design tasks are similar to high-level design tasks but with a greater level of detail. In fact, the same kinds of tasks can slip into the next step in software engineering: development.

The next chapter provides an introduction to software development. It explains some general methods you can use to organize development. It also describes a few useful techniques you can use to reduce the number of bugs that are introduced during development.

EXERCISES

1. Consider the `ClassyDraw` classes `Line`, `Rectangle`, `Ellipse`, `Star`, and `Text`. What properties do these classes all share? What properties do they not share? Are there any properties shared by some classes and not others? Where should the shared and nonshared properties be implemented?
2. Draw an inheritance diagram showing the properties you identified for Exercise 1. (Create parent classes as needed, and don't forget the `Drawable` class at the top.)