"You can do that?" they ask.

"You can in Java," you say.

The Prototype pattern says that when it takes a lot of resources, or a lot of code, to create an object, you should consider simply copying an existing object and customizing it instead. In Java, you can copy objects if they have copy constructors, or you can use the clone method. Figure 11-2 illustrates how you can represent the Prototype design pattern — repeat this process as needed to create as many objects as you need.

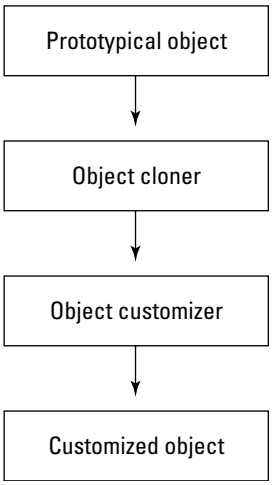

Figure 11-2: The Prototype pattern at work.

In code, all you have to do is set up a prototypical cheesecake and keep calling the `clone` method on it — no need to create a cheesecake from scratch in your code every time — and then add some customization to the new cheesecake, as needed.

The GoF book says the Prototype pattern should: "Specify the kinds of objects to create using a prototypical instance, and create new objects by copying this prototype."

Decoupling Abstractions from Implementations with the Bridge Pattern

There you are, designing car remotes for various types of cars. But it's getting confusing. You have an abstract class that's extended to create various types

of car remotes: those that just control the car alarm, those that start the car remotely, and so on. But you need to deal with various different car types, such as Toyota, Honda, and so on. And to support new remotes that are planned, your abstract `Remote` class has to change as needed.

This could get pretty messy. Your abstract `Remote` class can change, and it also needs to know what type of car it's dealing with before you can extend it to create various types of remotes — in other words, the car that the remote has to work with can also change. So you've got two things that can change: your abstract `Remote` class and the `Car` implementation the remote is supposed to work with.

As you'd expect, where there are two things that can change, and they're tied together, there's a pattern that can help out. The Bridge pattern comes to the rescue by saying that you should separate out the Car type into its own class. The remote will contain a car using a "has-a" relationship so that it knows what kind of car it's dealing with. This relationship looks like the one shown in Figure 11-3 — the "has-a" connection between the remote and the car type is called the *bridge*.

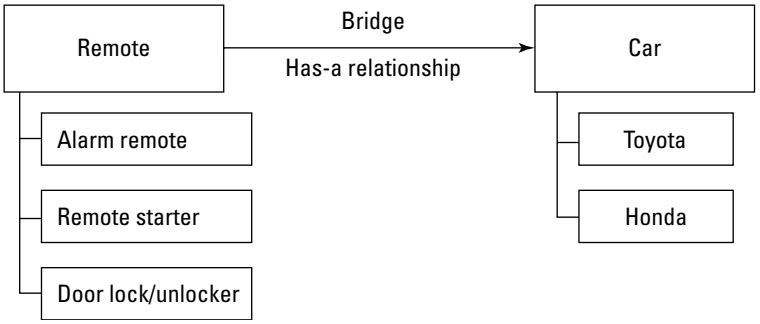

Figure 11-3: Using the Bridge pattern.

The inspiration here is that when you have an abstraction that can vary, and that's tied to an implementation that can also vary, you should decouple the two.

The GoF book says the Bridge design pattern should, "Decouple an abstraction from its implementation so that the two can vary independently."