I decided to stick with the cooking example….

Suppose I am a college student. My understanding of what it means to cook might reflect the appliances in my dorm room.

Each appliance has a different “cook” behavior.

I can model this in the UML\(^1\) simply enough.

This approach works even as I add new college cooking appliances.

---

\(^1\) Unified Modeling Language
I update the UML model accordingly.

```
class LateCollege

Appliance
  + cook() : void

PopcornPopper
  + cook() : void

Toaster
  + cook() : void

Phone
  + cook() : void

HotPlate
  + cook() : void
```

The model works! It seems I have mastered the culinary arts! Cooking behavior simply depends on the type of cooking appliance! My senior thesis is complete! Life is good! Bring on the final exams!

Ah, but college doesn’t last forever! Suddenly I find myself a graduate, with a job(!) and an apartment. Before long I find that I have a kitchen(!) with some mysterious new tools for creating meals, including a Stove and a Microwave. Fortunately, I still remember how to do research, so I learn from the best sources (Help! My Apartment Has a Kitchen) that with my Oven I can bake and broil! (Amazing!) Not only that, but when I want to reheat I can choose either the Stove or the Microwave! (Who knew?) And my new ToasterOven can burn a slice of bread just as well as my venerable Toaster.

How should I handle all this? I realize I can replace my single Appliance:cook method with a series of more specific methods, i.e., Appliance:pop, Appliance:toast, Appliance:orderPizza, Appliance:simmer,
Appliance:bake, Appliance:broil, Appliance:reheat, and create override methods in each concrete appliance class. This sounds complicated, but I try it.

So I implement Oven:bake and Oven:broil. I also have Stove:reheat and Microwave:reheat. This means repeating code but it works. Similarly, I implement Appliance:toast as Toaster:toast and ToasterOven:toast.

Then a culinary revelation occurs! My roommate shows me there is a setting to control the desired darkness of the toast, and he advocates a setting that doesn’t trigger the smoke alarm! After much consideration, I decide his way really might be preferable (albeit less exciting, and arguably less efficient since we have to remember to test the alarms separately), so I agree to change the implementation of the toast method to reflect my newly acquired expertise. I find I have to modify both Toaster:toast and ToasterOven:toast, even though these have identical implementations.

I hadn’t realized cooking was going to be so hard! I have a sense that my model will become more and more difficult to maintain as I learn more about my kitchen and how to cook! Was subclassing Appliance at each step really the answer?

These are dark days. (Possibly this is in part be because we didn’t pay the electric bill on time.) At a candlelight dinner with my buddies I muse on the complexities of surviving in the modern kitchen, and my best friend looks me in the eye and says, “Dude! You need a Strategy to cook!”

I reflect on his advice that night and realize that yes, I indeed need just that. What is changing is the algorithm for cooking. I decide I should encapsulate what changes, that is, the CookBehavior. So I create an abstract class, CookBehavior, and then child classes representing different types of cooking behavior.

---

2 I presume that the methods on the top-level class implement some default behavior (e.g., do nothing) so that we do not have to create an override implementation in every child class, but just where we actually need to implement different behavior.
Then I can create an Appliance:cook method that delegates the actual work to a method defined on the CookBehavior interface. The particular algorithm that executes depends on the type of CookBehavior we have selected.

This is the heart of the strategy pattern. We create an abstraction (CookBehavior) of a family of algorithms as an abstract class, implement the concrete algorithms (CookBehaviors) in subclasses, and then in our context method (Appliance:cook) program to the interface (CookBehavior), selecting the concrete object corresponding to the algorithm (one of the CookBehaviors) that we need for actual execution.

My full Appliance:cook method lets me use any of the CookBehaviors interchangeably.

Note that I have added an attribute of type CookBehavior to the Appliance class.
To prepare a dish, I need to set up an appropriate *Appliance*\(^3\), equip it with a suitable CookBehavior (algorithm), and invoke *Appliance*:cook.

Further, to make dinner, I set up the Kitchen, select an *Appliance* and *CookBehavior* for each dish, and I am on my way!

---

\(^3\) If there is no other reason to create different types of appliances, we could just use an *Appliance* object. For the purposes of this exercise we will presume we want to define unique types of appliances in order to implement other methods necessary to satisfy other requirements.
This changes my life! Shortly thereafter I start dating a lovely woman and invite her over to dinner. I make the bold decision to Sauté a dish (sounds romantic!), which I can now confidently do simply by adding a Sauté class under *CookBehavior* and equipping Stove with a Sauté object. My girlfriend loves it!

After we marry, have a couple beautiful kids, and move to a new home, we invite friends over to the new place for a cookout on the day of the big game. I adeptly add a Grill class under *Appliance* and a Barbecue class under *CookBehavior*. I can devote more time to the kids. Life is indeed wonderful!

**Strategy Pattern Summary**

Use the Strategy Pattern to encapsulate a family of algorithms. This is most useful if the implementation is likely to change over time (new or modified algorithms), or if it can vary depending on circumstances (examples: a desktop and a real-time application may do the same thing but in different ways; we may have multiple ways to parse a file depending on the interfacing application that will use the data).
Consequences
Use of the Strategy Pattern benefits us by allowing us to separate the implementations of an algorithm from the context that invokes them, thereby allowing us to change either the algorithms or the context independently of the other. It helps us avoid having to implement an algorithm multiple times and reduces the need for case structures. It allows us to equip a context object with our choice of suitable algorithm implementations very easily.

There are some possibly negative consequences, too. First, we may have to create more classes than we might have to otherwise. Second, it is incumbent on the application developer to know the strategies a particular context class can support (not necessarily a bad thing). For instance, it is possible to direct a Phone to Broil, but probably this is not very productive. If the different algorithms require widely varying input parameters, each concrete context object might have to store all possible sets of parameters, which could be unwieldy. Finally, it takes some time to implement the Strategy Pattern, so we want to use it where it will be of most benefit to us—where the algorithm implementations currently vary or are likely to vary.
Appendix I: Interfaces

The *CookBehavior* class functions as an *interface*. Strictly speaking, of course, LabVIEW does not have interfaces (at least not yet), but we can implement something that allows most of the benefits of interfaces.

First, let’s talk about a few terms:

**Abstract Class**: "A class that can be used only as a superclass of some other class; no objects of an abstract class may be created except as instances of a subclass."\(^4\) We will not have an *Appliance* object constant in our applications, only constants for objects that represent particular types of appliances. Similarly, we won’t have a *CookBehavior* object constant, only constants for objects that represent the various concrete implementations. LabVIEW does not allow us to declare that a class is abstract, but we can easily define that a class is abstract in our design.

**Pure Virtual Method (Abstract Method)**: A method that has no implementation. This means that if I have a pure virtual method in a parent class (e.g., *CookBehavior:cook* was a pure virtual method), every child of *CookBehavior* **must** implement cook. LabVIEW does not allow the creation of a method with no implementation, but it does provide an option (under Class Properties… Item Settings) to require overrides and we can use this to accomplish the same goal.

---

Note that in this case we don’t require overrides to call the parent method. In fact, since we want to treat the parent method as a pure virtual method, the overrides in this situation do not (and really must not) call the parent method.  

**Interface:** "A class can have abstract (or pure virtual) methods and nonabstract methods. A class in which all methods were declared as abstract (or pure virtual) would correspond to the Java idea of an interface."  

We can approximate an interface in LabVIEW by defining an abstract class with pure virtual methods. 

Now that we know what an interface is, we can ask why we would want to use one. Well, interfaces offer several advantages.

---

5 There are situations, on the other hand, when it is quite useful to include a default implementation (often, do nothing, or trap a fault) in a method in an abstract class. What we have then is an abstract class that is not functioning strictly as an interface. 


7 This is not quite the same as a Java interface, especially since a Java class can implement more than one interface. To do this in LabVIEW using the approach we have described would require the use of multiple inheritance, which LabVIEW (fortunately, I think) does not support. My experience has been that the presented approximation of an interface is the nonetheless the most useful (and comprehensible) approach currently available in LabVIEW. Others have created more complex interface concepts in LabVIEW that endeavor to do more (see, for example, http://lavag.org/topic/10621-interface/page_p_63208__hl__exterface__fromsearch__1#entry63208).
We achieve some measure of *loose coupling* and *high cohesion*: We can separate the implementation of an interface from the context for which it serves an interface. For instance, the interface in the Strategy Pattern allows us to separate the implementation of the various *CookBehavior* algorithms from the *Appliance* Context. In our example, we can copy *CookBehavior* and its children, and *Appliance* and its children, to separate projects, work on them independently, and then reintegrate the results into a larger project.

Note that the project with *Appliance* only has the *CookBehavior* interface in its dependencies, not the concrete CookBehaviors. Hence I can work on *Appliance* and its children without concerning myself with the details of any of the CookBehaviors, while a colleague works on implementing the various CookBehaviors without concerning herself with the details of *Appliance* and its children. This allows us to do design by contract. My colleague can even create a whole new set of CookBehaviors without requiring modifications to *Appliance* or its children. Or—if appropriate—we can reuse *CookBehavior* in another application. We can create algorithms that we can reuse in many applications. Moreover, we can modify the implementations of the various CookBehaviors and test these independently of the *Appliance* family.
Appendix II: Factory Method Pattern

A more sophisticated implementation of this example actually uses a second Gang of Four design pattern, the Factory Method Pattern, in combination with the Strategy Pattern. Let’s revisit the DinnerSimulator:main method using this approach.

We are using Enumerated values (strict typedefs) to specify the appliance or behavior to use.\(^8\) We create the corresponding objects using object “factories” in Kitchen:prepareDish and Appliance:cook.

---

\(^8\) This isn’t the only option but it is the best one we’ve found.
Note that the Appliance class now has an attribute of type ICookBehaviorFactory.

```
class Appliance

  cookBehaviorFactory: ICookBehaviorFactory

  cook(CookBehaviorValue) : void
```

Why do we need IApplianceFactory::createAppliance and ICookBehaviorFactory::createCookBehavior? Why not just use an object constant of the desired type in the first place, as we did in the first implementation?

Well, we can, actually, and it does work (as we have seen), and it is still be an implementation of the Strategy Pattern, but there are reasons to create the objects in a factory.

When we implement the Strategy Pattern, we often select the algorithm to use at compile time (as we do in this example). It is not uncommon, however, to make this selection at run-time (dynamically). To take this a step further, in the State Pattern, which uses the same architecture but for a different purpose, we routinely select an object (representing the next state) dynamically, typically within the implementation of a method of another concrete state.9 If we use a NextState object directly, then ThisState has a reference to NextState. Suppose NextState may transition to AnotherState or EndState. It has references to these. Pretty quickly we end up with a fairly complex web of references and every concrete state class links somehow to every other state class.

Now replace the object constants with an enumerated value. Each state class has a reference only to the typedef (and the context only has references to the typedef, the interface factory class, and the abstract top-level state class). We have reduced the linking dramatically. This might not immediately seem important, but projects open much faster, build faster, and deploy faster—and more reliably—if we reduce the linking. This is especially important in LabVIEW Real-Time, where there is a finite (and small) level of layers available.

Moreover, since the Factory Method Pattern uses an interface, we have the same source code decoupling benefits noted above. Again, we can focus our development or maintenance efforts on one thing at a time, without fear of breaking something else as long as we satisfy the interface requirements.

For these reasons, limiting linking in this way results in applications that are much, much more robust than they would otherwise be.

The Factory Method Pattern is straightforward to implement. We create an interface for the factory (ICookBehaviorFactory) and the concrete factory itself (CookBehaviorFactory).

---

9 Sometimes this is in the Context, but the point is the same.
ICookBehaviorFactory:createCookBehavior has (for all intents and purposes) no implementation. (We never call this code.)

CookBehaviorFactory:createCookBehavior returns the specified object.
Yes, we do have to create a couple more classes but the result is a more robust application easier to pull apart and maintain.