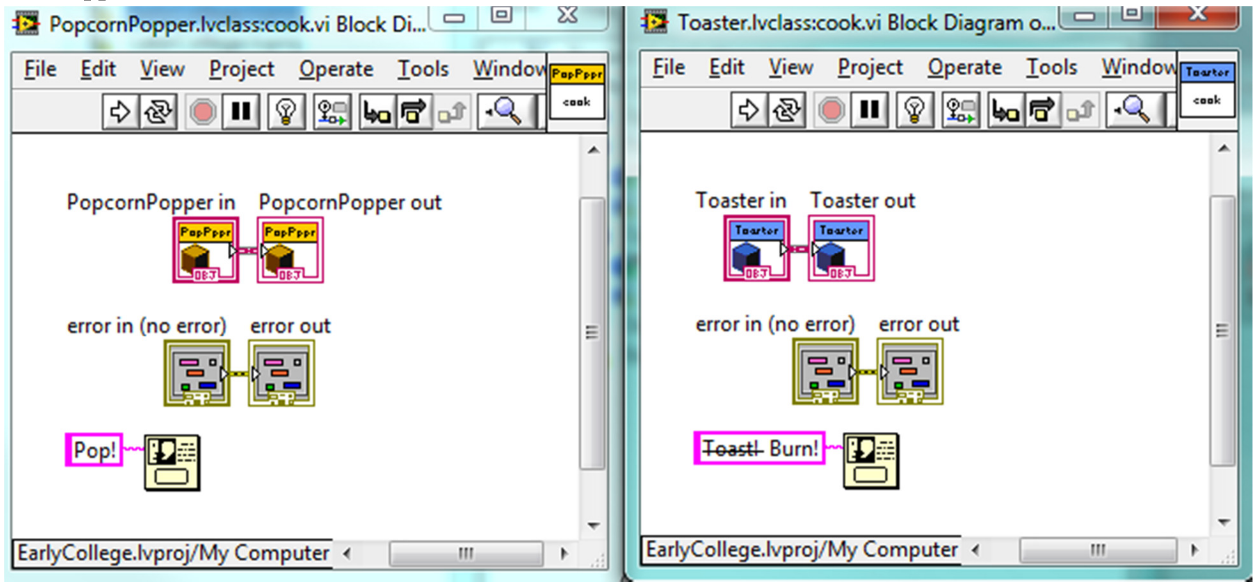
Strategy Pattern with LabVIEW

Paul J. Lotz
Lowell Observatory, 1400 W Mars Hill Road, Flagstaff AZ 86001

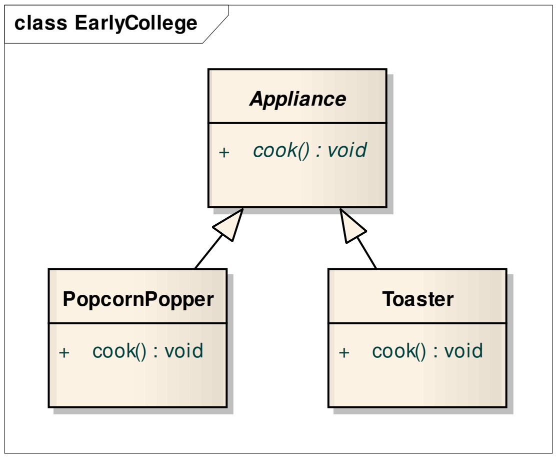
I decided to stick with the cooking example….

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.

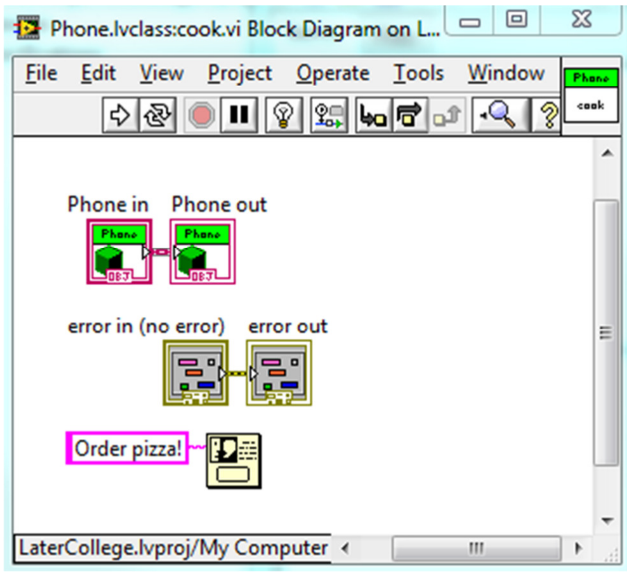

PopcornPopper.lvclass:cook.vi Block Di...
File Edit View Project Operate Tools Window
PopcornPopper in
PopcornPopper out
error in (no error)
error out
Pop!
EarlyCollege.lvproj/My Computer
Toaster.lvclass:cook.vi Block Diagram o...
File Edit View Project Operate Tools Window
Toaster in
Toaster out
error in (no error)
error out
~~Toast!~~ Burn!
EarlyCollege.lvproj/My Computer

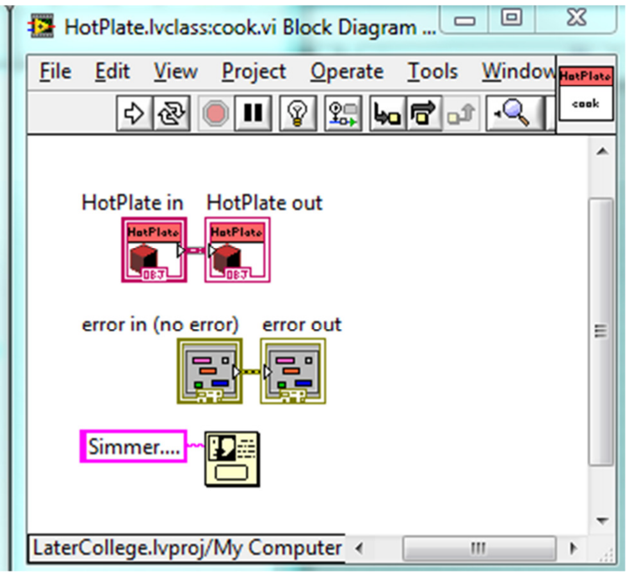
I can model this in the UML[1] simply enough.



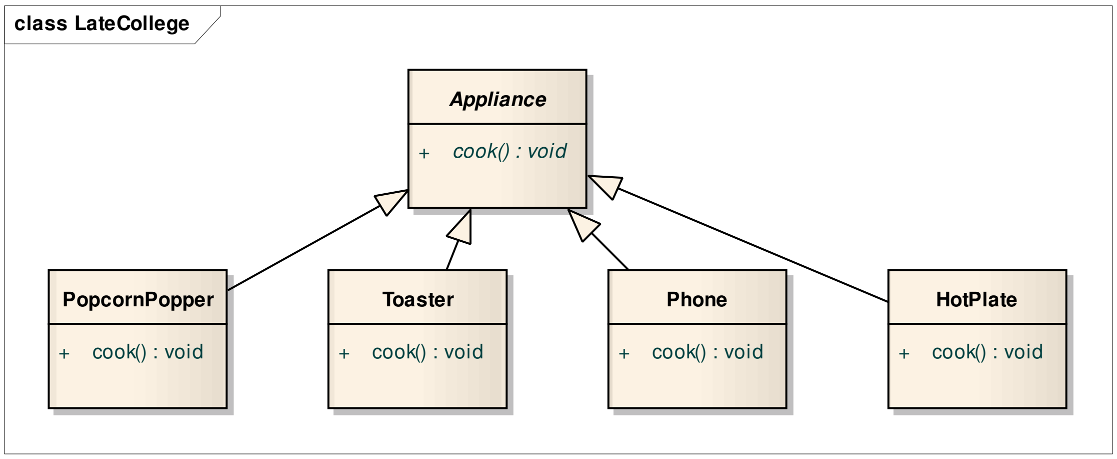
This approach works even as I add new college cooking appliances.

[1] Unified Modeling Language


Phone.lvclass:cook.vi Block Diagram on L...
File Edit View Project Operate Tools Window
Phone in
Phone out
error in (no error)
error out
Order pizza!
LaterCollege.lvproj/My Computer


HotPlate.lvclass:cook.vi Block Diagram ...
File Edit View Project Operate Tools Window
HotPlate in
HotPlate out
error in (no error)
error out
Simmer....
LaterCollege.lvproj/My Computer

I update the UML model accordingly.

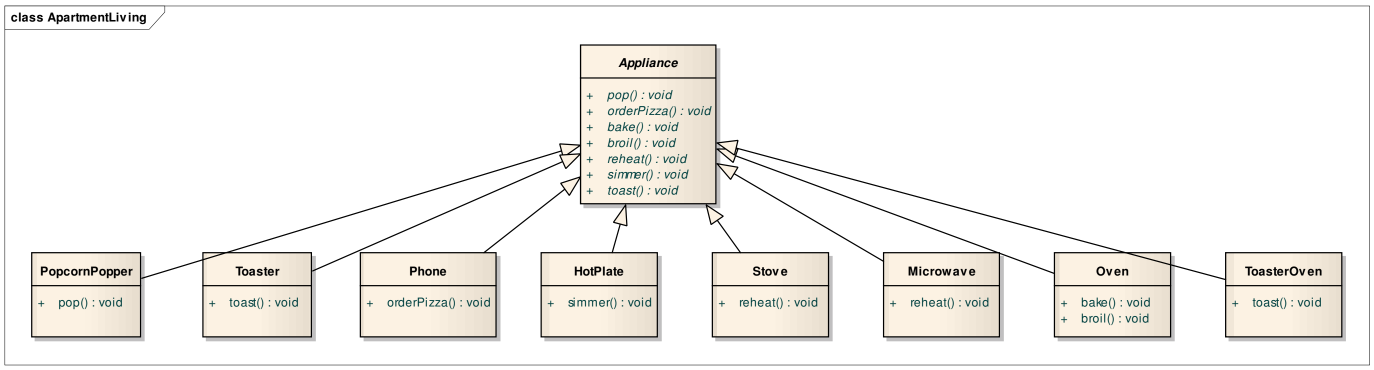


The model works! It seems I have mastered the culinary arts! Cooking behavior simply depends on the type of cooking appliance! My senior thesis is complete! Life is good! Bring on the final exams!

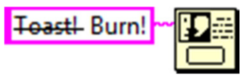
Ah, but college doesn't last forever! Suddenly I find myself a graduate, with a job(!) and an apartment. Before long I find that I have a kitchen(!) with some mysterious new tools for creating meals, including a Stove and a Microwave. Fortunately, I still remember how to do research, so I learn from 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.

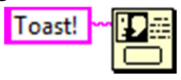
How should I handle all this? I realize I can replace my single *Applicance:cook* method with a series of more specific methods, i.e., *Appliance:pop*, *Appliance:toast*, *Appliance:orderPizza*, *Appliance:simmer*, *Appliance:bake*, *Appliance:broil*, *Appliance:reheat*, and create override methods in each concrete appliance class.[2] 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.



becomes


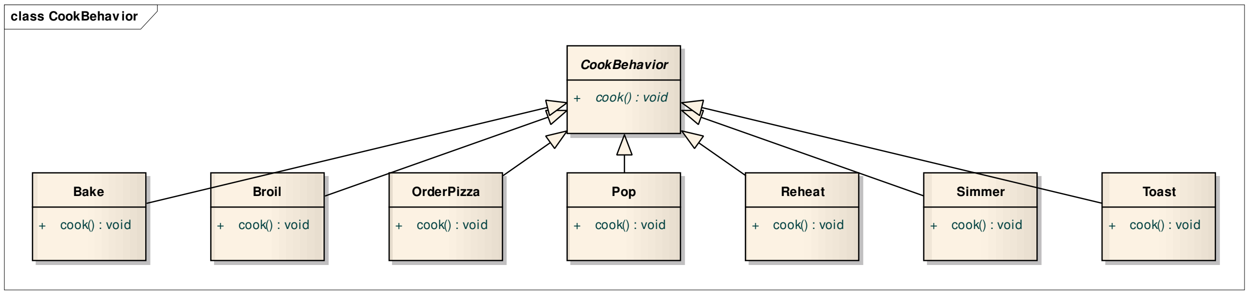
in Toaster:toast and again in ToasterOven:toast.

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?

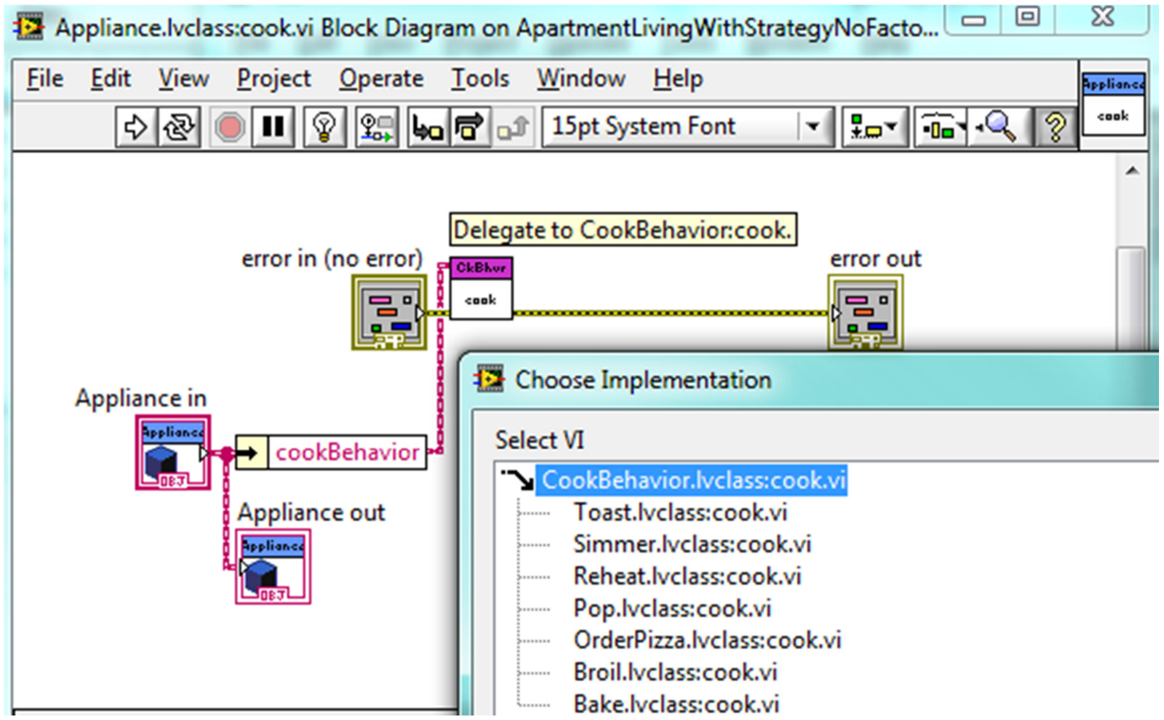
These are dark days. (Possibly this is in part be because we didn't pay the electric bill on time.) At a candlelight dinner with my buddies I muse on the complexities of surviving in the modern kitchen, and my best friend looks me in the eye and says, "Dude! You need a Strategy to cook!"

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.



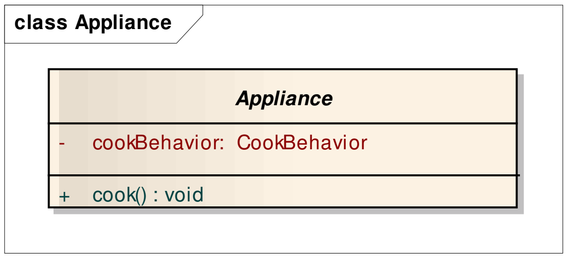
Then I can create an *Appliance*:cook method that delegates the actual work to a method defined on the *CookBehavior* interface. The particular algorithm that executes depends on the type of *CookBehavior* we have selected.



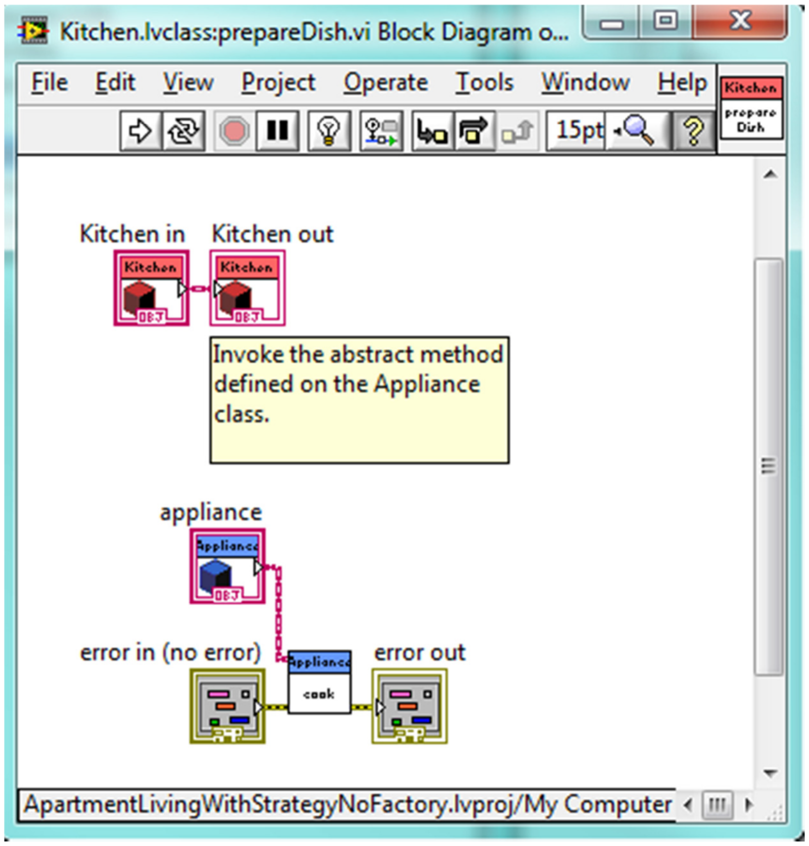
This is the heart of the strategy pattern. We create an abstraction (*CookBehavior*) of a family of algorithms as an abstract class, implement the concrete algorithms (CookBehaviors) in subclasses, and then in our context method (*Appliance*:cook) program to the interface (*CookBehavior*), selecting 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.



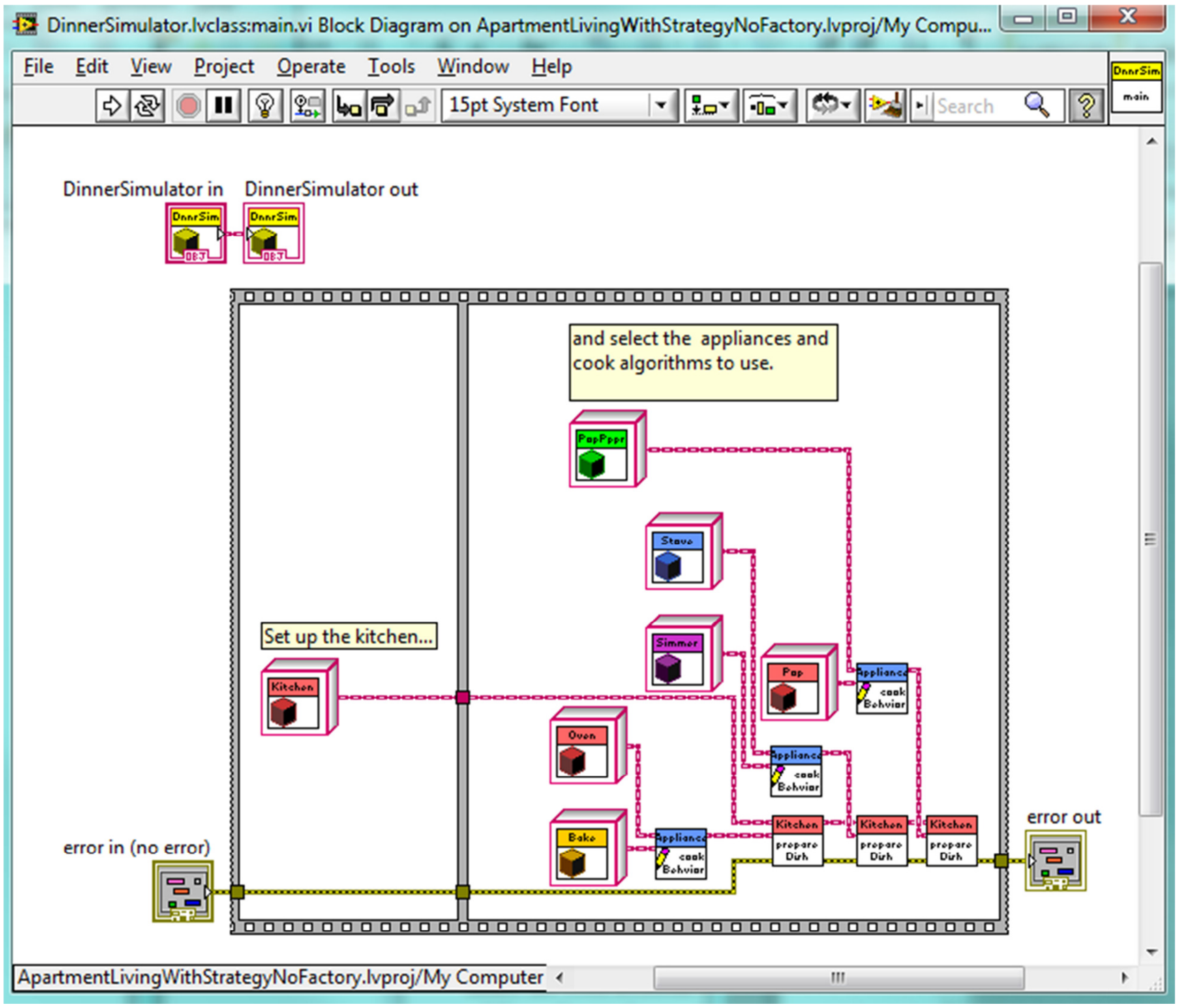
To prepare a dish, I need to set up an appropriate *Appliance*[3], equip it with a suitable CookBehavior (algorithm), and invoke *Appliance*:cook.



Invoke the abstract method defined on the Appliance class.

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.


DinnerSimulator.lvclass:main.vi Block Diagram on ApartmentLivingWithStrategyNoFactory.lvproj/My Compu...
File Edit View Project Operate Tools Window Help
15pt System Font
Search
DnnrSim
main
DinnerSimulator in
DinnerSimulator out
and select the appliances and cook algorithms to use.
Set up the kitchen...
Kitchen
PapPppr
Stove
Simmer
Pop
Oven
Bake
Appliance cook Behavior
Kitchen prepare Dish
error in (no error)
error out
ApartmentLivingWithStrategyNoFactory.lvproj/My Computer

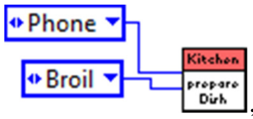
This changes my life! Shortly thereafter I start dating a lovely woman and invite her over to dinner. I make the bold decision to Sauté a dish (sounds romantic!), which I can now confidently do simply by adding a Sauté class under *CookBehavior* and equipping Stove with a Sauté object. My girlfriend 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.