Master and Student...

**Master**: Grasshopper, it has been some time since our last meeting. Have you been deep in meditation on inheritance?

**Student**: Yes, Master. While inheritance is powerful, I have learned that it doesn’t always lead to the most flexible or maintainable designs.

**Master**: Ah yes, you have made some progress. So, tell me my student, how then will you achieve reuse if not through inheritance?

**Student**: Master, I have learned there are ways of “inheriting” behavior at runtime through composition and delegation.

**Master**: Please, go on...

**Student**: When I inherit behavior by subclassing, that behavior is set statically at compile time. In addition, all subclasses must inherit the same behavior. If however, I can extend an object’s behavior through composition, then I can do this dynamically at runtime.

**Master**: Very good, Grasshopper, you are beginning to see the power of composition.

**Student**: Yes, it is possible for me to add multiple new responsibilities to objects through this technique, including responsibilities that were not even thought of by the designer of the superclass. And, I don’t have to touch their code!

**Master**: What have you learned about the effect of composition on maintaining your code?

**Student**: Well, that is what I was getting at. By dynamically composing objects, I can add new functionality by writing new code rather than altering existing code. Because I’m not changing existing code, the chances of introducing bugs or causing unintended side effects in pre-existing code are much reduced.

------------------------------------------------------------------------------------------------------------------------

Most important Design Principle: (the Open-Closed principle)

Classes should be open for extension, but closed for modification.

Come on in; we’re open. Feel free to extend our classes with any new behavior you like. If your needs or requirements change (and we know they will), just go ahead and make your own extensions.

Sorry, we’re closed. That’s right, we spent a lot of time getting this code correct and bug free, so we can’t let you alter the existing code. It must remain closed to modification. If you don’t like it, you can speak to the manager.

------------------------------------------------------------------------------------------------------------------------

Q: How can I make every part of my design follow the Open-Closed Principle?

A: Usually, you can’t. Making OO design flexible and open to extension without the modification of existing code takes time and effort. In general, we don’t have the luxury of tying

down every part of our designs (and it would probably be wastefu). Following the Open-Closed Principle usually introduces new levels of abstraction, which adds complexity to our code. You want to concentrate on those areas that are most likely to change in your designs and apply the principles there.

Q: How do I know which areas of change are more important?

A: That is partly a matter of experience in designing OO systems and also a matter of the knowing the domain you are working in. Looking at other examples will help you learn to identify areas of change in your own designs.

------------------------------------------------------------------------------------------------------------------------

A hint: think of decorator objects as “wrappers.”

1. Decorators have the same supertype as the objects they decorate.
2. You can use one or more decorators to wrap an object.
3. Given that the decorator has the same supertype as the object it decorates, we can pass

around a decorated object in place of the original (wrapped) object.

1. The decorator adds its own behavior either before and/or after delegating to the object it

decorates to do the rest of the job.

1. Objects can be decorated at any time, so we can decorate objects dynamically at runtime

with as many decorators as we like.

The Decorator Pattern attaches additional responsibilities to an object dynamically. Decorators provide a flexible alternative to subclassing for extending functionality.

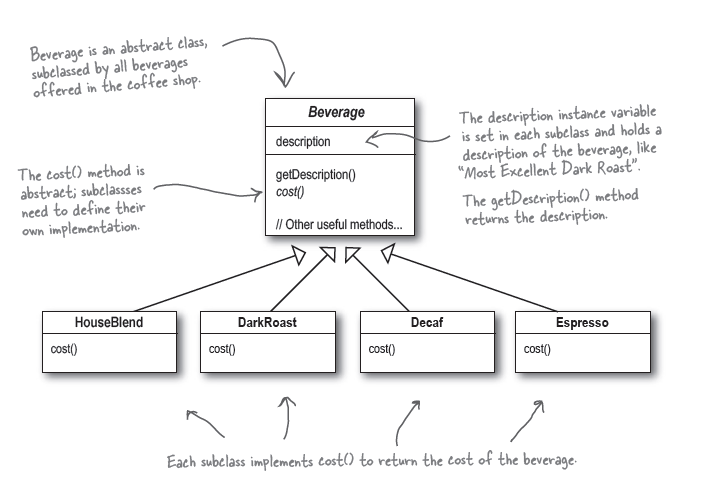
------------------------------------------------------------------------------------------------------------------------

Problem statement:

Welcome to Starbuzz Coffee

Starbuzz Coffee has made a name for itself as the fastest growing coffee shop around. If you’ve seen one on your local corner, look across the street; you’ll see another one. Because they’ve grown so quickly, they’re scrambling to update their ordering systems to match their beverage offerings.

When they first went into business they designed their classes like this...



In addition to your coffee, you can also ask for several condiments like steamed milk, soy, and mocha (otherwise known as chocolate), and have it all topped off with whipped milk. Starbuzz charges a bit for each of these, so they really need to get them built into their order system. Here’s their first attempt...

