OODP 4: Minimize Dependencies

Today, we look at our fourth object-oriented design principle: **Minimize Dependencies**. We not only want to reduce the ***number*** of dependencies between our classes, but also desire to decrease the ***strength*** of the dependencies which we cannot remove without breaking functionality. Through several examples, we will see how and why to minimize dependencies.

Content Learning Targets

After completing this activity, you should be able to say:

- I can identify and categorize dependencies between classes in both UML and code.
- I can explain the design goals of "tell, don't ask" and "avoid message chains".
- I can define coupling and cohesion in my own words.
- I can distinguish cohesion from encapsulation.
- I can apply coupling and cohesion to analyze the level and quality of dependencies in a UML design or code.

Process Skill Goals

During the activity, you should make progress toward:

- Writing with technical detail using precise object-oriented design terminology. (Communication)
- Interpreting UML diagrams and imagining possible implementations. (Information Processing)

Facilitation Notes

First Hour: Model 1 introduces dependencies (in UML) and the "Tell, Don't Ask" design principle.

End the first hour with a 10-minute mini lecture that shows the difference between telling and asking for computing game revenue in the Vapor Sales Manager implementation assignment.

Second Hour: Model 2 TODO

Model 3 TODO

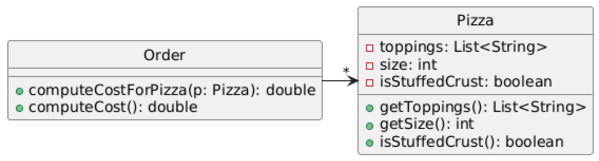
Key questions: #2, #4, #5,

Source files:

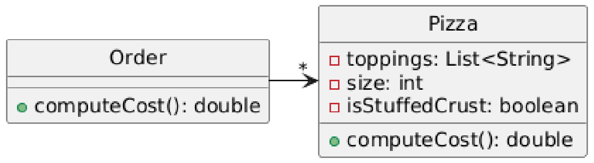
Model 1 Tell, Don't Ask

Recall the Pizza Restaurant design problem from a previous class, in which an Order consists of a list of Pizzas, and the cost of each Pizza depends on its toppings. Suppose that the restaurant adds different sizes (diameter, in inches) and a stuffed crust option, and both factors affect Pizza cost. Consider the two partial UML designs below. (Details not related to our current focus have been left out.)

Design A: (PlantUML source)



Design B: (PlantUML source)



Questions (30 min)

Start time:

1. The `computeCost()` method in Order will look very similar.

In Design A:

```java
public double computeCost() {
    double cost = 0;
    for (Pizza p : this.pizzas) {
        cost += this.computeCostForPizza(p);
    }
    return cost;
}
```

In Design B:

```java
public double computeCost() {
    double cost = 0;
    for (Pizza p : this.pizzas) {
        cost += p.computeCost();
    }
    return cost;
}
```

Where will the designs' differences appear in code, if not in Order's `computeCost()`?

In Design A's `Order.computeCostForPizza` method and Design B's `Pizza.computeCost` method.

2. Which design is better? Justify your choice.

Design B is better. In Design A, Order is asking the Pizza class for all of its internal data, then calculating the pizza's cost using that data. This violates encapsulation because Pizza doesn't have meaningful methods that operate on its data. It also violates the principle of "Tell, Don't Ask", which we will discuss today.

The design principle of “Tell, Don’t Ask” (see Fig. 1) encourages us to keep operations (methods) that operate on certain data with the class(es) containing that data. Instead of having Class A ask for Class B’s internal data and then do something with it, possibly updating Class B’s internal data afterward, we should have Class A ***tell*** Class B to perform that operation.

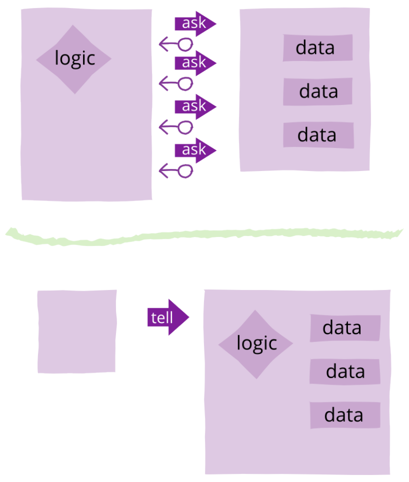


Figure 1: “Tell, Don’t Ask” diagram by Martin Fowler.

3. Asking is especially poor design when you return some internal class that the caller would otherwise not know exists. For example, suppose we have a class called `AppRunner` with a field called `framework` of type `LogFramework`, which supports logging errors and user actions to files. One of our methods:

```
public LogFramework getLogFramework() {
    return this.framework;
}
```

One of our client’s methods:

```
public void activateVerboseLogging() {
    LogFramework fw = this.appRunner.getLogFramework();
    fw.setLevel(5);
}
```

How could you refactor to “Tell, Don’t Ask” and avoid exposing the `LogFramework` class to the client?

(1) Remove the `getLogFramework()` method. (2) Change the client’s method to:

```
public void activateVerboseLogging() {
    this.appRunner.activateVerboseLogging();
}
```

(3) Add this method to `AppRunner`:

```
public void activateVerboseLogging() {
    this.framework.setLevel(5);
}
```

Now, our client doesn’t know that the `LogFramework` class exists.

Interlude: Discuss “Tell, Don’t Ask” example from the Vapor Sales Manager implementation assignment. See slides.

4. Based on the "Tell, Don't Ask" examples we've seen, list a few rules of thumb for avoiding asking, and/or red flags that suggest your design/code might be asking instead of telling.

Answers will vary. One red flag: lots of getter method usage, especially on the same object in quick succession. One rule of thumb: try to push a method as far "down" as possible, toward the class(es) containing the involved data. Add "tell" methods along the way to transmit data for the command.

Model 2 Message Chains

A ***message chain*** is code in the form

```
someObject.someMethod().otherMethod().stillOtherMethod();
```

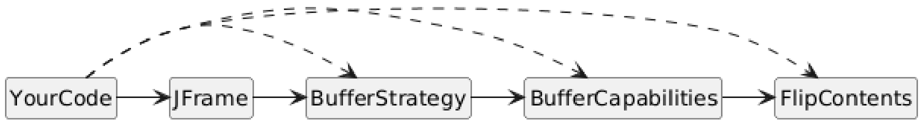
For example (this is taken from Java AWT/Swing; the details of these classes are not important), your code might have a line like

```
myFrame.getBufferStrategy().getCapabilities().getFlip().wait(17);
```

A rational programmer might split up this message chain by introducing intermediate variables:

```
BufferStrategy strategy = myFrame.getBufferStrategy();
BufferCapabilities capabilities = strategy.getCapabilities();
FlipContents flip = capabilities.getFlipContents();
flip.wait(17);
```

By splitting the message chain like this, the dependencies become even clearer, as we see in the UML diagram below. Your code knows details ***four levels deep*** in the called operations!



(PlantUML source)

Questions (30 min)

Start time:

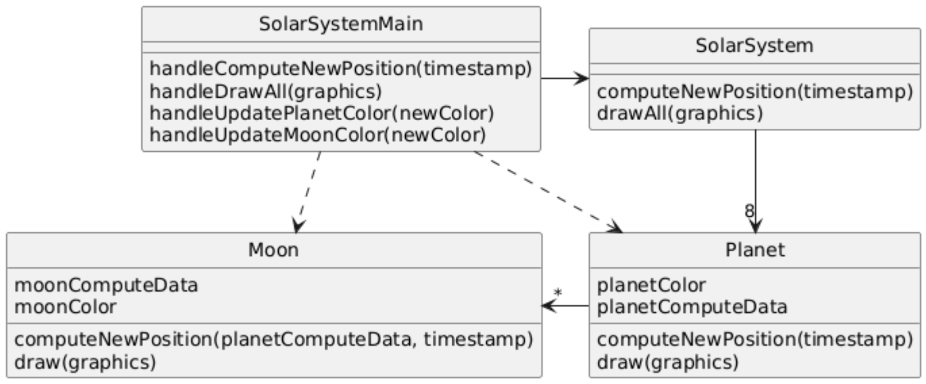
5. If you could modify these classes however you wanted, how could you eliminate this message chain to reduce dependencies? Draw your updated design on a whiteboard, paper, or in PlantUML, and explain below.

The solution is usually to embed the required feature in the first class in the chain. This insulates the caller from the inner classes. Then the first class might implement the feature itself, or if it still needs to rely on its internals, repeat the message chain removal. For this example, this process gives this design.

Solar System Problem

System description: A Java program draws a minute-by-minute updated diagram of the solar system including all planets and moons. To update the moon's position, the moon's calculations must have the updated position of the planet it is orbiting. The diagram is colored—all planets are drawn the same color and all moons are drawn the same color. However, it needs to be possible to modify the planet color or the moon color.

6. Examine the UML diagram below of a possible design for this system. Which dependencies seem essential? Which seem like we might be able to weaken or eliminate them?


SolarSystemMain
handleComputeNewPosition(timestamp)
handleDrawAll(graphics)
handleUpdatePlanetColor(newColor)
handleUpdateMoonColor(newColor)
SolarSystem
computeNewPosition(timestamp)
drawAll(graphics)
8
Moon
moonComputeData
moonColor
computeNewPosition(planetComputeData, timestamp)
draw(graphics)
*
Planet
planetColor
planetComputeData
computeNewPosition(timestamp)
draw(graphics)

(PlantUML source)

The SolarSystemMain dependencies on Moon and Planet seem unnecessary, but the rest are essential.

7. Investigate the implementation of this design provided in the `src/solarSystem` package. For the seemingly unnecessary dependencies, where do they appear in the code? What principle(s) do these dependencies violate?

SolarSystemMain depends on Moon and Planet via the message chains in the `handleUpdatePlanetColor` and `handleUpdateMoonColor` methods. The use of non-enhanced for loops is intentional to highlight the message chains.

8. Improve the UML design by applying our standard procedure for handling the type of dependency you found. Use the PlantUML source linked above as a starting point, and paste your updated PlantUML link below. (Remember to click the "Decode URL" button after making changes.)

Improved design, derived from mechanically following the advice to propagate the method into the first class of the chain: PlantUML link

9. Revisit the system description. What do you notice about the color specifications that suggests a further design improvement?

The system description has all planets sharing the same color and all moons sharing the same color, so we only need one `planetColor` and one `moonColor` for the SolarSystem, which can be stored as fields. The current design duplicates data by storing the same moon color in all Moon objects and the same planet color in all Planet objects.

10. Refactor the solar system project to match this design, which includes the improvement from the previous question. You may get help from GenAI tools. Looking at the methods where you previously saw evidence of unnecessary dependencies, what does the new implementation do? How does it avoid the bad dependencies of the original version?

Each of SolarSystemMain's `handleUpdatePlanetColor` and `handleUpdateMoonColor` methods should now be a one-liner that calls the appropriate setter on `this.solarSystem`. This prevents SolarSystemMain from needing to know that the Planet and Moon classes exist.

Model 3 Coupling and Cohesion