# 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 classes, but also desire to decrease the ***strength*** of dependencies which we cannot remove without breaking functionality.

## 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 apply coupling and cohesion to analyze the level and quality of dependencies in a UML design or code.
- I can distinguish cohesion from encapsulation.

## Process Skill Goals

*During the activity, you should make progress toward:*

- Writing with technical detail using precise OODP terms. (Communication)
- Interpreting UML diagrams and envisioning possible code. (Information Processing)

## Facilitation Notes

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

**Second Hour:** Model 2 introduces message chains with some examples. The solar system example includes a code refactoring activity.

For Model 3, some potentially useful analogies:

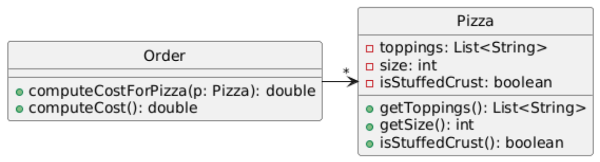
- High Cohesion: a well-organized closet where all related items are stored together.
- Low Cohesion: a messy closet where clothes, shoes, etc. are scattered everywhere.
- High Coupling: welding together all the parts of a car so they can't be separated, can't be independently repaired/replaced, and are forced to move as one unit.
- Low Coupling: using a trailer hitch to connect a car and a trailer, allowing them to be connected or disconnected easily, and enabling them to function independently.

Key questions: #2, #4, #6, #15

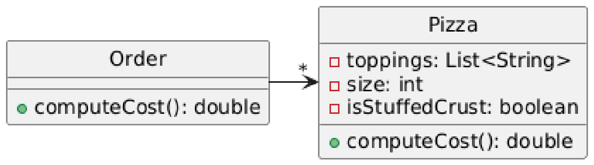
## Model 1 Tell, Don't Ask

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 these two partial UML designs. (Irrelevant details omitted.)

Design A: (PlantUML source)



Design B: (PlantUML source)



### Questions (40 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 A's `Order`, `computeCostForPizza` and B's `Pizza`, `computeCost` method.

**2.** Which design is better? Why?

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

The design principle of "Tell, Don't Ask" (see Fig. 1) encourages us to keep 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, we should have Class A ***tell*** Class B to perform that operation. For example, Pizza has all the data it needs to compute its cost, so Order should ***tell*** Pizza to compute its own cost.

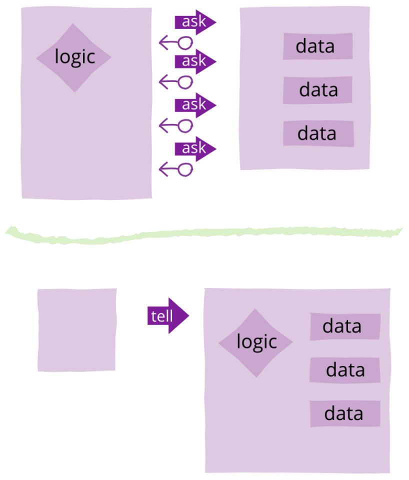


Figure 1: "Tell, Don't Ask" diagram by Martin Fowler.

**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 not expose `LogFramework` 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.

**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.

**5**. See the "Tell Do Not Ask" slides for an example from the Vapor Sales Manager.

## Model 2 Message Chains

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

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

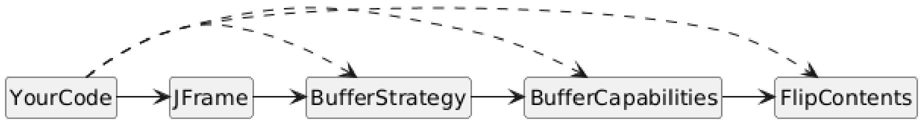
For example (this is taken from Java AWT/Swing; the details of these classes are not important), your code might have a line like

```
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:

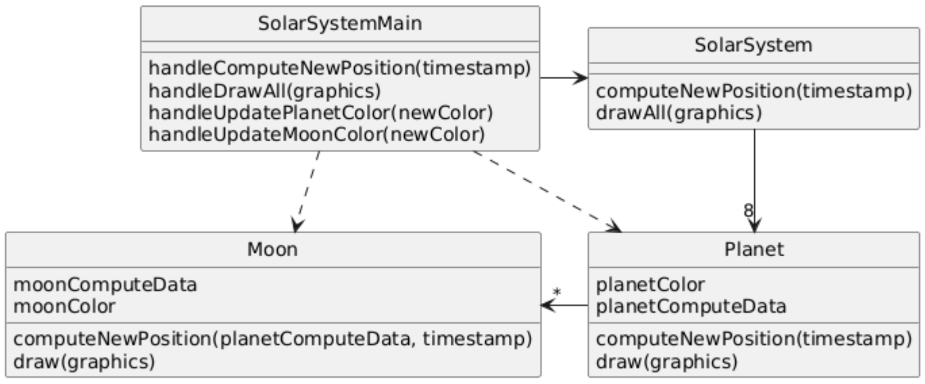
**6**. If you could modify these classes however you wanted, how could you eliminate this message chain to reduce dependencies?

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.

**7**. 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.

**8**. 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.

**9**. 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

**10**. 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.

**11**. 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

Our discussion of dependencies in object-oriented software designs leads us to two key concepts: ***cohesion*** and ***coupling***.

### Questions (20 min) Start time:

**12**. In object-oriented design, the term cohesion refers to how closely related the parts of a class (or more generally, a module) are to each other. The name comes from the physical property of cohesion, which describes how well the parts of a substance stick together.

**13**. The term coupling refers to how strongly classes in a design depend on each other. The name comes from mechanical couplings, which connect shafts together at their ends so they move together.

**14**. Which of cohesion and coupling is generally considered good, and which is considered bad? Why?

High cohesion is good because it means the parts of a class are closely related, making the class easier to understand, maintain, and reuse. Low coupling is good because it means classes are less dependent on each other, making the overall system more modular and easier to change without affecting other parts.

**15**. What mnemonic can we use to help us remember which of cohesion and coupling is good and which is bad?

COHESION contains the letter "H", so we want ***H***igh co***H***esion. COUPLING contains the letter "L", so we want ***L***ow coup***L***ing.

**16**. How does coupling affect our ability to maintain and extend a software system?

High coupling makes it harder to maintain and extend a system because changes in one class can have ripple effects on other classes that depend on it. Low coupling allows for easier maintenance and extension since classes can be modified independently.

**17**. If we do our object-oriented design work carefully, we will:

a) ***divide and conquer***, breaking our larger system into several classes.

b) give each class a clear purpose (***high cohesion***), allowing it to do its one job well.

c) include only necessary dependencies between classes (***low coupling***), making it easier to change one class without affecting others.

**18**. Imagine you're writing code to manage a school's students. Your design should:

- Handle adding or removing students from the school
- Storing each student's name, phone number, and grades for specific courses (can use a courseId String)
- Setting the individual course grades for a particular student
- Compute the average GPA of all the students in the school
- Sort the students by last name to print out a report of students and GPA.

Discuss and come up with a design with your team. How many classes does your system need?

See the "Coupling and Cohesion" slides for an analysis of the coupling–cohesion tradeoff as you add more classes to this design.

**19**. Compare/contrast cohesion and encapsulation.

Both cohesion and encapsulation can be summarized as "keeping related things together". Cohesion is often used to refer to the relatedness of elements of a ***module*** (collection of classes), while encapsulation is often used to refer to the relatedness of fields and methods within a ***class***. Both support maintainability and reusability of code.