Determining If a Class Has a Single Responsibility

How can you determine if the `Gear` class contains behavior that belongs somewhere else? One way is to pretend that it's sentient and to interrogate it. If you rephrase every one of its methods as a question, asking the question ought to make sense. For example, "*Please Mr. Gear, what is your ratio?"* seems perfectly reasonable, while *"Please Mr. Gear, what are your gear_inches?"* is on shaky ground, and "*Please Mr. Gear, what is your tire (size)?"* is just downright ridiculous.

Don't resist the idea that "*what is your tire?"* is a question that can legitimately be asked. From inside the `Gear` class, `tire` may feel like a different kind of thing than `ratio` or `gear_inches`, but that means nothing. From the point of view of every other object, anything that `Gear` can respond to is just another message. If `Gear` responds to it, someone will send it, and that sender may be in for a rude surprise when `Gear` changes.

Another way to hone in on what a class is actually doing is to attempt to describe it in one sentence. Remember that a class should do the smallest possible useful thing. That thing ought to be simple to describe. If the simplest description you can devise uses the word "and," the class likely has more than one responsibility. If it uses the word "or," then the class has more than one responsibility and they aren't even very related.

OO designers use the word *cohesion* to describe this concept. When everything in a class is related to its central purpose, the class is said to be *highly cohesive* or to have a single responsibility. The Single Responsibility Principle (SRP) has its roots in Rebecca Wirfs-Brock and Brian Wilkerson's idea of Responsibility-Driven Design (RDD). They say "A class has responsibilities that fulfill its purpose." SRP doesn't require that a class do only one very narrow thing or that it change for only a single nitpicky reason, instead SRP requires that a class be cohesive—that everything the class does be highly related to its purpose.

How would you describe the responsibility of the `Gear` class? How about *"Calculate the ratio between two toothed sprockets"*? If this is true, the class, as it currently exists, does too much. Perhaps *"Calculate the effect that a gear has on a bicycle"*? Put this way, `gear_inches` is back on solid ground, but `tire` size is still quite shaky.

The class doesn't feel right. `Gear` has more than one responsibility but it's not obvious what should be done.

Determining When to Make Design Decisions

It's common to find yourself in a situation where you know something isn't quite right with a class. Is this class really a *Gear*? It has rims and tires, for goodness sake! Perhaps *Gear* should be *Bicycle*? Or maybe there's a *Wheel* in here somewhere?

If you only knew what feature requests would arrive in the future you could make perfect design decisions today. Unfortunately, you do not. Anything might happen. You can waste a lot of time being torn between equally plausible alternatives before rolling the dice and choosing the wrong one.

Do not feel compelled to make design decisions prematurely. Resist, even if you fear your code would dismay the design gurus. When faced with an imperfect and muddled class like `Gear`, ask yourself: *"What is the future cost of doing nothing today?"*

This is a (very) small application. It has one developer. You are intimately familiar with the `Gear` class. The future is uncertain and you will never know less than you know right now. The most cost-effective course of action may be to wait for more information.

The code in the `Gear` class is both *transparent* and *reasonable,* but this does not reflect excellent design, merely that the class has no dependencies so changes to it have no consequences. If it were to acquire dependencies it would suddenly be in violation of both of those goals and should be reorganized *at that time*. Conveniently, the new dependencies will supply the exact information you need to make good design decisions.

When the future cost of doing nothing is the same as the current cost, postpone the decision. Make the decision only when you must with the information you have at that time.

Even though there's a good argument for leaving `Gear` as is for the time being, you could also make a defensible argument that it should be changed. The structure of every class is a message to future maintainers of the application. It reveals your design intentions. For better or for worse, the patterns you establish today will be replicated forever.

`Gear` *lies* about your intentions. It is neither *usable* nor *exemplary*. It has multiple responsibilities and so should not be reused. It is not a pattern that should be replicated.

There is a chance that someone else will reuse `Gear`, or create new code that follows its pattern while you are waiting for better information. Other developers believe that your intentions are reflected in the code; when the code lies you must be alert to programmers believing and then propagating that lie.

This "improve it now" versus "improve it later" tension always exists. Applications are never perfectly designed. Every choice has a price. A good designer understands this tension and minimizes costs by making informed tradeoffs between the needs of the present and the possibilities of the future.