1 Deliverable Code Smells

1.1 The problems you all saw with the original code

The original code experienced a myriad of so called "internal" problems. Although it behaved as expected, it was plagued with bad encapsulation and a faulty interface. For starters, dynamic naming for instance variables makes for less predictable code. In the original version, the use of "instance_variable_set" to set arbitrary key-value pairs from "params" is an anti-pattern, leaving the user to try and decipher what parameters are implemented in the class. In fact, the Ruby API described "instance_variable_set" to be a thing that "sets the instance variable names by symbol to object, thereby frustrating the efforts of the class's author to attempt to provide proper encapsulation. The variable did not have to exist prior to this call". A frustrating functionality, because a class should not have parameters that someone does not know how to call.

Instead, in an OOP language, one should use inheritance to have classes with specific parameters that other instances wouldn't need to share. In this case, a super class "MountainBike" will have the parameters that are shared amongst the two bikes. In order to implement extra parts which only one bike might have we can include an embedded data structure (such as a hash) that is passed all the parts and can iterate over those parts to figure out the cost. This will accomplish the same task without having classes with unspecified instance methods.

Each of these properties (rigid, front suspension, full_suspension) can also be classes (not subclasses) with their own parameters. Giving those objects behavior would limit hard coding of these methods to calculate value, and lessen the "godness" of this class. What I do think is good about this code is the global parameters, "TIRE_WIDTH_FACTOR" and others.

1.2 What new Ruby you had to learn to accomplish this

This assignment required finding about more about instance variables, and how state works in Ruby classes. Other than that, it required experimenting more with the #{} syntax, which I did not use in my own programs. The most interesting thing I learned in observing the code was the "instance_variable_set" method for assigning instance variables of various names to an object. I also learned more about symbols in Ruby.

1.3 What does your improved code looks like?

The "fixed" code does not trust the user to provide the names of attribute variables. The constructor checks its input to ensure that the required fields are provided and that their values are within proper bounds. If values are missing or out of bounds, errors are properly handled and propogated up the stack.

The "fixed" code also includes more than one class, for the different types of bikes. While this erases the semantics of the heirarchy of the bike, treating the different types of bikes totally independetly clarifies the class methods siginificantly. In a system that say, sold bikes, this would clarify the inventory system because each type of bike could be treated seperately.

Further, the code is now much easier to document because each function only has to handle a single "type" of bike. Previously, the method documentation would've had to describe each case of the calcultion. Now, each method is *almost* self-documenting.

1.4 In what ways do you feel it's a solution?

Each class in our solution has a specific role. It is possible to ask them specific questions. For instance, it's possible to ask the Rigid, FrontSuspension, and FullSuspension what their price is, and for each one it's a small simple method that does not need to check what is passed to it and have multiple cases. It's also easier to explain what each class does in a sentence without using "and" or "or". Another benefit of this solution is that no code is repeated anywhere else in the solution. The parent MountainBike class has the code for to_s and owner, while the child classes have all the class specific information, like price and off road ability. This also makes adding another type of bike relatively easy, if there's some new type of suspension "mixed_suspension" then none of the other classes need to change to handle it. Since each class only does one thing, and it's relatively easy to add to this solution I feel that it's a good solution.