When it comes to marking their design out of 10, assess **UML Correctness** and **Requirements Coverage**. Please also include some **Design Feedback**, even if the feedback ends up being that the design is fine.

UML Correctness covers correct use of the UML format. If they make an error here, typically penalize them only once for any one type of error, and **don't subtract more than -5 total** for UML notation.

-The diagram must be a UML class diagram.
-It should include the following elements:
-**Classes**, each with their own boxes split into three parts.
-Interfaces, typically split into two parts unless they add some constant variables
-**Methods**, which don't have to include the constructors. Typically we would also re-list the methods from an interface in the class which implements them, so if they haven't done that **apply a -0.5 penalty**.
-**Return types**, **-0.5 if forgotten**
-**Parameters**, **-0.5 if forgotten**
-**Variables** included in the classes.
-**Data types** listed for those variables. **-0.5 if forgotten**
-**Access levels** on methods and variables, with – for private, + for public, and # for protected. Any public variable is a mistake, as is forgetting to use them at all, **for -0.5.**
-**Relationships**, using the dotted line with clear arrowhead for interfaces, solid line with clear arrowhead for inheritance, and solid line with black arrowhead for "has-a" relationships. **-0.5** if you find any issues there (max of -0.5 even for multiple issues)
-**Multiplicities** on "has-a" relationships, and not on interface or inheritance relationships. **-0.5 if forgotten**.
-The submission should be a pdf, but that's the only format restriction. If they submit a phone picture of a hand-drawn diagram... well, if it's legible then that's okay (though feel free to mention I said not to do that!) If it's not legible, that's their problem!

Requirements Coverage concerns whether everything described in the requirements for the model actually shows up in the diagram, but not an assessment of the quality or validity of the design. You'll need to use a little more discretion when deciding if a requirement is missing and how much to dock for it, but I'll at least list what to look out for As with UML Correctness, **don't subtract more than -5 total** for requirements errors.

-Their diagram must cover the **Model** component of their soccer team program.
-If they include the **other packages** that's **okay**, but **not required**. If they seem to have **muddled up** the **Controller** and **View** with the Model, though (like intermixing their methods and attributes) that's a **-2** sort of problem.
-They don't need to cover the **Driver**.
-They need to have included the following elements:
-**Player**
-**Name** (First and last, though just one full name field is okay)
-**Date of birth**, any data type is acceptable. Age is also acceptable but may cause issues later, covered in Design Feedback.
-**Preferred Position,** which might be a String or an enum. Could even be a whole Position object, though with some reservations explained below.
-**Starting lineup position**, for players who are a part of the starting lineup. This is different than their preferred position, though it might not be stored directly

with the Player. It depends on how the Team tracks the Starting lineup. It certainly should be somewhere!

-**Skill**, a number from 1 to 5. Might also be an enum. Probably no point in it being a floating point number, but not wrong.

-**Team**

-**Team Size**, which could either be a stored value or derived from the data structure storing the team members

-**Jersey Number**, which might be an attribute of the players or something tracked by the Team. It could even potentially be implicit, like using the array index of the Player object in a Team array, but there will still need to be methods to retrieve a Player's jersey number later.

-**Starting Lineup**, which is almost certain to be a data structure storing player objects, or else a designation that Players have. It's hard for this to be completely implicit, though, so there ought to be something you can see in the design which marks which players are part of the lineup.

-**Benched Players**. They may have an explicit class for these, or a data structure to store them, or they might make "bench" a position. On the other hand, they might track this implicitly, like using a null position, or having one list tracking the whole team and another tracking just the starting lineup. If you don't see an explicit way to track this, there's no penalty, but remind them they'll need a way to distinguish players on the lineup and the bench.

-There are also **two required methods** the Model must offer toward the Controller (for example, through a Team interface). One returns a String of the whole Team, while the other returns a String of just the starting lineup. These Strings need to include other information that will probably require other supporting methods to put together, like players' names, jersey numbers, positions, etc.

-They **don't need to cover exceptions** in their design just yet, so certain special requirements like checking that everyone in the team is under 10 years old don't need to be listed yet, but it's okay if they include a method for that as well.

While unlikely to be a source of lost marks unless the case is exceptionally bad, you should also include **Design Feedback**.

-**Most classes should probably have an interface** that other classes then use to refer to them, like a Team having an array of Player interface objects implemented by PlayerImpl concrete objects. This is one design occasion where you can apply a **-0.5** if there's no interface use going on.

-**Position** should probably be an **Enum**, not just a String. If it's given its own whole class, caution them that there isn't much difference between positions and this might add unnecessary complexity.

-If they chose to track players' **ages** instead of **birthdates**, point out that it will be harder to track players' ages as they grow up.

-Any method which **returns the entire team data structure** is probably overkill and should be called as such. Tell them the Model shouldn't be handing that over without good reason when you can offer methods that return specific things about the team like the players' names or the team size without exposing the entire team object to tampering. At bare minimum it should only hand over a copy and not the original object. I'm willing to call this a **-0.5 penalty** unless it's explicitly returning a copy.

-Feel free to provide **feedback on names**, like if someone uses the same name for the class and the interface, or uses vague attributes and method names. If you can't tell what their design is doing, you can't know if it's going to cover the requirements!
-If they have **a lot of public methods**, especially for getting (or worse, setting) a bunch of attributes without any clear reason to, definitely suggest they review their methods and try to cut back or at least make some private helper methods.
-Designs with **more than 5 or 6 classes/interfaces** probably have **too many**. In particular, caution them about making objects for things that could be data structures, like the starting lineup, or simple enums, like the positions.
-It's not wrong to have **a full-on Player interface / Abstract Player class / Benched and Starting Player Concrete classes**, but it's a very heavy-duty implementation and will create some challenges when for example a Benched player needs to be "promoted" to a Starter.