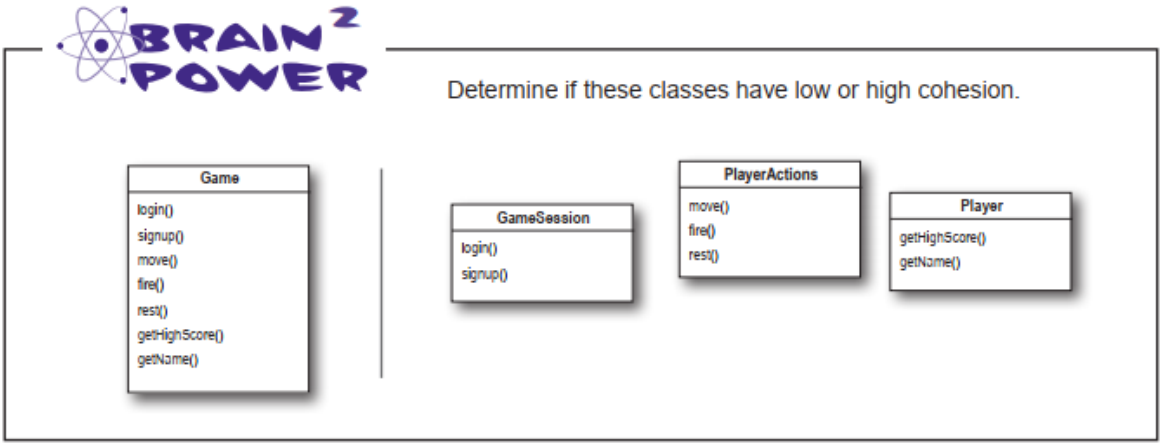
Cohesion

Cohesion is "a measure of how closely a class or module supports a single purpose or responsibility[1]."

Given that a class "determine[s] what you can do with objects…but also how it is done[2]," ideally, the classes that we make should be *highly* cohesive, meaning that classes should typically have a tightly bound scope of actions and responsibilities that it makes use of when they are called on the objects that we have. Cohesion makes it easy to understand the purpose of the class that we have created and when to use (or not to use) a class. It also helps in aiding in creating more robust and flexible code that can be more easily changed to meet ever-evolving demands.

The following image provides an example for high versus low cohesion. It is taken from the *Head First Design Patterns* textbook[3].



The Game() class on the left is a class that is of low cohesion. It has many different actions that it performs. More specifically, it isn't quite that it has many actions but rather *all the actions under this class are not related to each other*. In other words, the actions under Game() do not share the same responsibility. This can be contrasted with the three classes on the right. While each of these classes have multiple *actions*, the actions therein are all related to each other and therefore each class on the right have one and only one *responsibility*. If we were to imagine the Game() class to be implemented in code, then if any one of the actions within Game() were to change, then any method or subclass that depends on Game() has the possibility to function in unintended ways. On the other hand, if GameSession(), PlayerActions(), and Player() were implemented in code, then a change will be less likely to cause unintended consequences down the road. Additionally, as a nice side effect, when classes have high cohesiveness, debugging becomes a simpler task as it is easier to know where to go in the code to fix an issue with classes defined in the example above on the right than on the left.

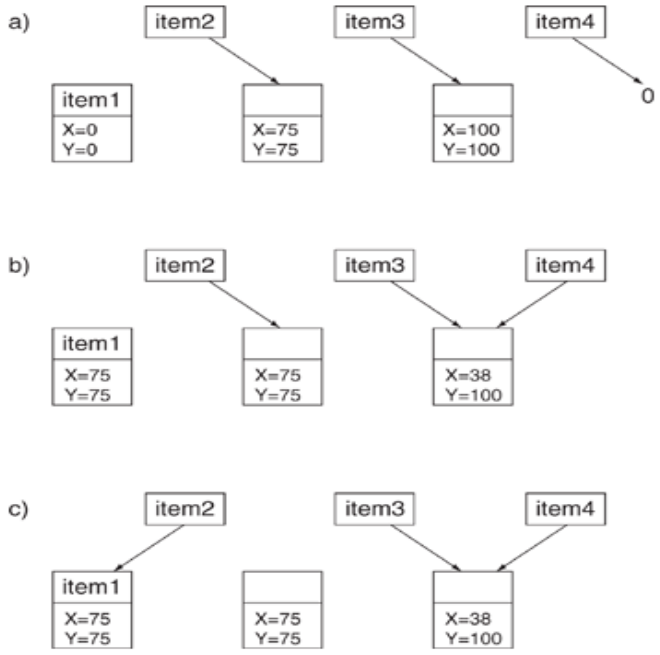
---

[1] Freeman, Eric, et.al, *Head First Design Patterns, 2nd Edition (2021)* p. 340.
[2] Mailund, Thomas, *Advanced Object-Oriented Programming in R* (2017): Chapter 2: "Class Hierarchies", DOI: 10.1007/978-1-4842-2919-4_2
[3] Freeman, Eric, et.al, *Head First Design Patterns, 2nd Edition (2021)* p. 341.

Identity

Identity is "the nature of an object that distinguishes it from all other objects[4]." Within a class we want to make sure that the objects that we call or instantiate are indeed the objects that we are referring to. Furthermore, we want to make sure that the objects that we make should be objects that we want to call in our methods (versus, data that we can store somewhere else). The objects that we put into our classes should have an identity—a purpose—and shouldn't be there without being used or called. If two objects share the same identity, then a change to one of the two objects will lead to the same change in the other object. This has the potential to be problematic as if we aren't sure of or careful about the changes that we make on objects within a class, then we can't guarantee that there won't be unintended side effects, bugs, or errors in our code later down the line. Take for example the following diagram from Booch's *Object-Oriented Analysis and Design with Applications*[5]:

Figure 3-4 Object Identity



With each change to how item1, item2, item3, and item4 refer to data, eventually, we could end up in outcome C where X=75 and Y=75 doesn't refer to anything (which can result in a memory leak[6]) and item3 and item4 refer to the same thing. If we keep in mind that objects should have their own unique identity and role within a class (which is similar, but distinct, to how classes should be highly cohesive), this will help to more easily identify where errors and bugs are within our code thereby allowing us to have cleaner design and readability of our code.

---

[4] Booch, Grady, et al., *Object-Oriented Analysis and Design with Applications, Third Edition* (2007), Chapter 3: Classes and Objects

[5] *Ibid.*

[6] *Ibid.*