Composition

We started talking about **composition**, and compared it to inheritance.

Inheritance is a way to reuse functionality and attributes.

Composition is a way to make the combination of classes, act like a single coherent object.

Composition is creating a whole from different parts

We built this personal computer, by passing objects, to the constructor, like assembling the computer.

We can actually hide the functionality further.

In this case, we're not going to allow the **calling program**, to **access those objects**, **the parts**, **directly**.

We don't want anybody to access the Monitor, Motherboard, or ComputerCase directly.

Use Composition or Inheritance or Both?

As a general rule, when you're designing your programs in Java, you probably want to look at composition first.

Most of the experts will tell you, that as a rule, look at using composition before implementing inheritance.

You saw in this example, we actually used both.

All of our parts were able to inherit a set of attributes, like the manufacturer and model.

The calling code didn't have to know anything about these parts, to get Personal Computer to do something.

Why is Composition preferred over Inheritance in many designs?

The reasons composition is preferred over inheritance:

- Composition is more flexible. You can add parts in, or remove them, and these changes are less likely to have a downstream effect.
- Composition provides functional reuse outside of the class hierarchy, meaning classes can share attributes & behavior, by having similar components, instead of inheriting functionality from a parent or base class.
- Java's inheritance breaks encapsulation, because subclasses may need direct access to a parent's state or behavior.

Why is Inheritance less flexible?

Inheritance is less flexible.

Adding a class to, or removing a class from, a class hierarchy, may impact all subclasses from that point.

In addition, a new subclass may not need all the functionality or attributes of its parent class.

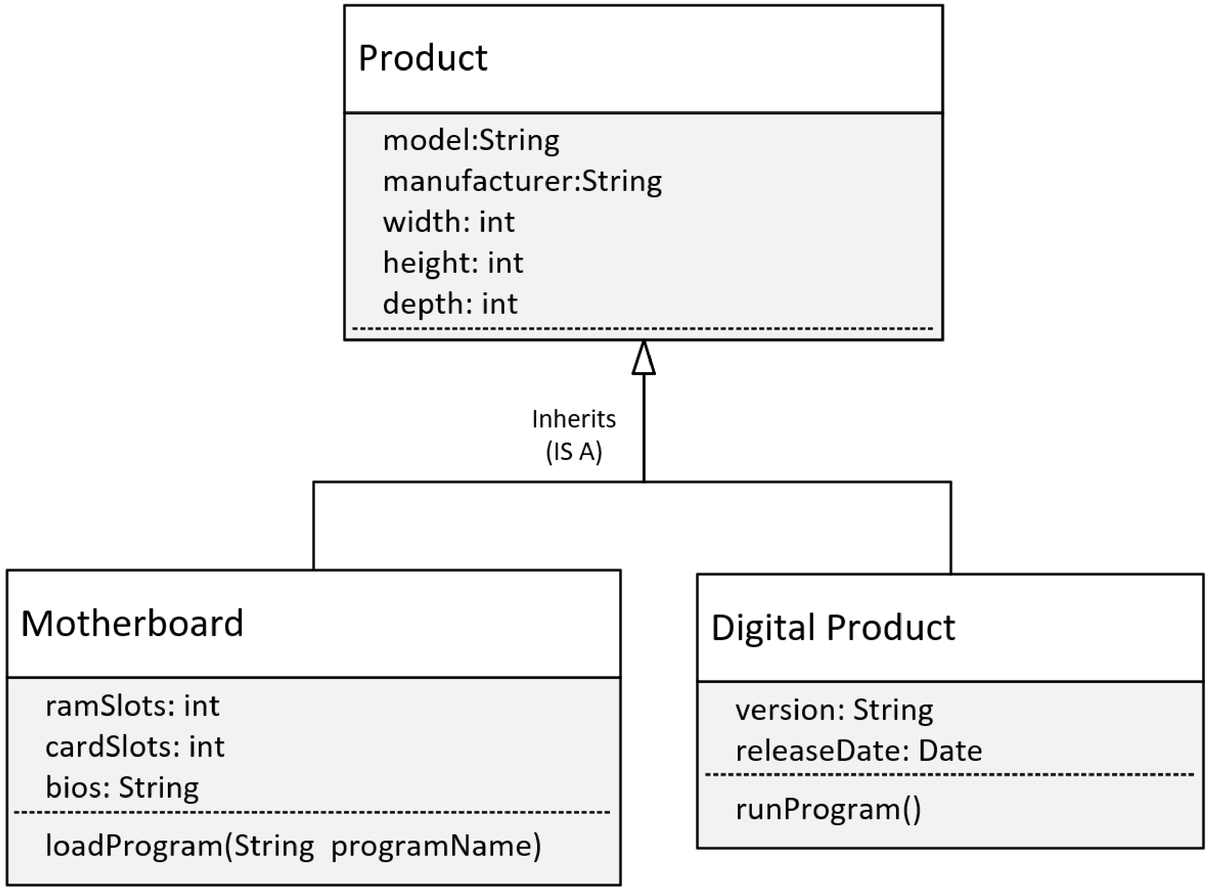
Adding a Digital Product

Let's say we want to include digital products, like software products, in our product inventory.

Should Digital Product inherit from Product?

Here we show the model with Digital Product, inheriting from our current definition of Product.

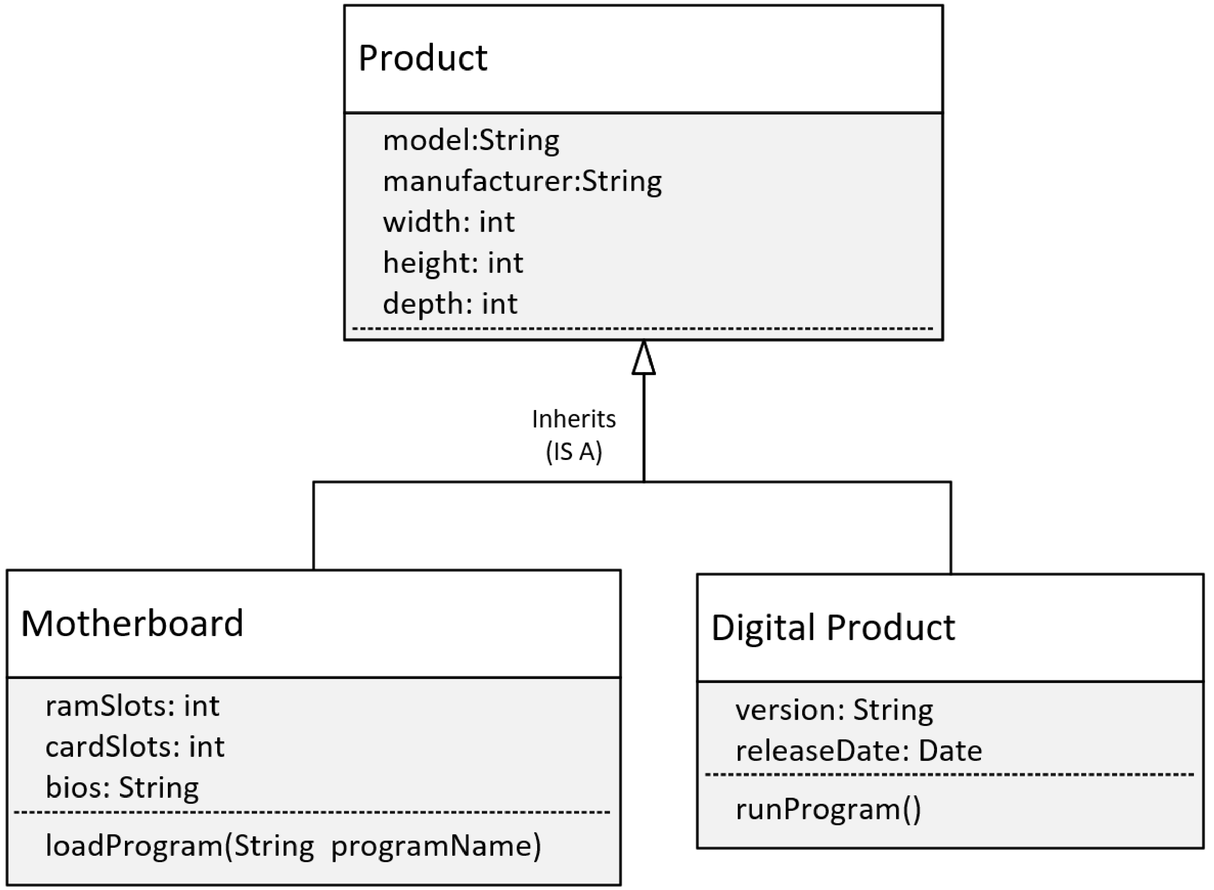
If we do this, this should mean Digital Product has Product's attributes, but this isn't true now.



Adding a Digital Product

A digital product wouldn't really have width, height, and depth, so this model isn't a good representation of what we want to build.

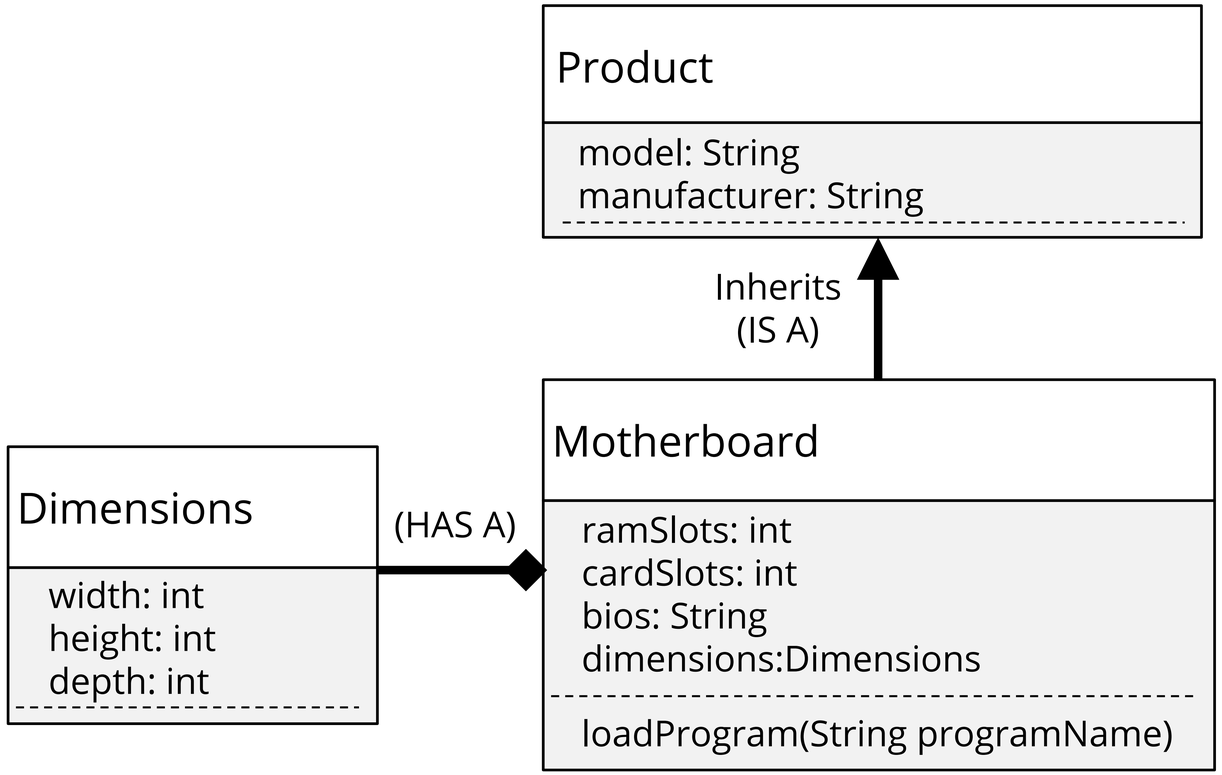
It would be better if we didn't have those three attributes on Product, but instead used composition to include them on certain products, but not all products.



Revised Class Diagram

Consider this revised class diagram.

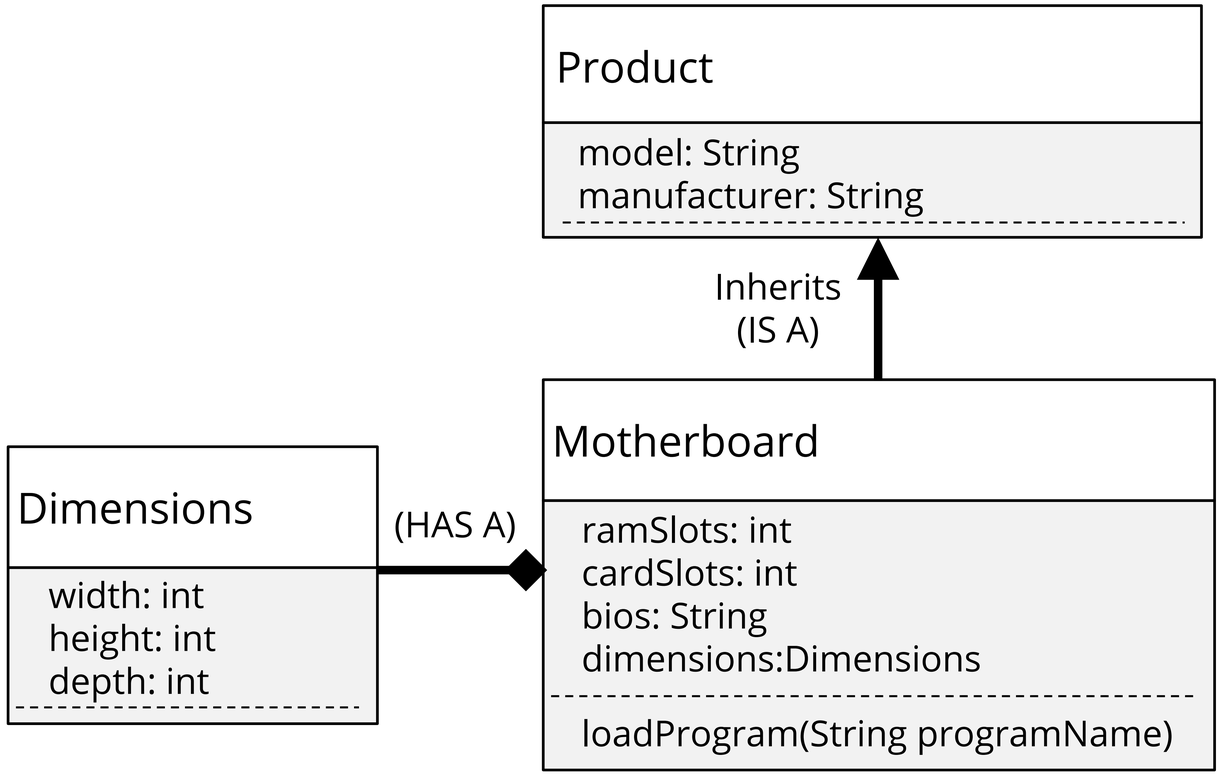
We haven't completely removed the class hierarchy, but we've made the base class, Product more generic.



Revised Class Diagram

We've removed the width, height, and depth attributes from Product, and made a new class, Dimensions, with those attributes.

And we've added an attribute to Motherboard, which is dimensions, which has those attributes.



Revised Class Diagram

Is this a better model?

Well, it's more flexible.

This design allows for future enhancements to be made, like the addition of the subclass Digital Product, without causing problems for existing code, that may already be extending Product.

By pulling width, height, and depth, into a dimension class, we can use composition, to apply those attributes to any product.