Using `It.IsAny<>`, `It.Is<>`, or a variable all serve different purposes. They provide increasingly specific ways to match a parameter when setting up or verifying a method.

It.IsAny

The method set up with `It.IsAny<>` will match *any* parameter you give to the method. So, in your example, the following invocations would all return the same thing (`ReturnSomething`):

```
role.GetSomething(Guid.NewGuid(), Guid.NewGuid(), Guid.NewGuid());

Guid sameGuid = Guid.NewGuid();
role.GetSomething(sameGuid, sameGuid, sameGuid);

role.GetSomething(Guid.Empty, Guid.NewGuid(), sameGuid);
```

It doesn't matter the actual value of the `Guid` that was passed.

It.Is

The `It.Is<>` construct is useful for setup or verification of a method, letting you specify a function that will match the argument. For instance:

```
Guid expectedGuid = ...
mockIRole.Setup(r => r.GetSomething(
                 It.Is<Guid>(g => g.ToString().StartsWith("4")),
                 It.Is<Guid>(g => g != Guid.Empty),
                 It.Is<Guid>(g => g == expectedGuid)))
         .Returns(ReturnSomething);
```

This allows you to restrict the value more than just any value, but permits you to be lenient in what you accept.

Defining a Variable

When you set up (or verify) a method parameter with a variable, you're saying you want *exactly* that value. A method called with another value will never match your setup/verify.

```
Guid expectedGuids = new [] { Guid.NewGuid(), Guid.NewGuid(),
Guid.NewGuid() };
mockIRole.Setup(r => r.GetSomething(expectedGuids[0], expectedGuids[1],
expectedGuids[2]))
         .Returns(ReturnSomething);
```

Now there's exactly one case where `GetSomething` will return `ReturnSomething`: when all `Guid`s match the expected values that you set it up with.