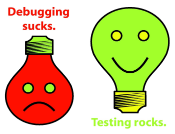

Loose Coupling with Demeter

by David Ackerman in Mountain View

The Law of Demeter, also known as the Principle of Least Knowledge, **is a strategy for keeping your code loosely coupled and easily testable.** It says **a class should interact directly with its collaborators and be shielded from understanding their internal structure**.

Have you ever seen code that looks like this?

```
public Emailer(Server server) {…} // taking a server in the constructor

public void sendSupportEmail(String message, String toAddress) {
  EmailSystem emailSystem = server.getEmailSystem();
  String fromAddress = emailSystem.getFromAddress();
  emailSystem.getSenderSubsystem().send(fromAddress, toAddress, message);
}
```

This design is faulty because it's:

1. **Unnecessarily complex.** Emailer interacts with multiple APIs that it doesn't really need (e.g., EmailSystem).
2. **Dependent on both Server's and EmailSystem's internal structure**. If any of them change, this class could break!
3. **Not reusable.** Any other Server implementation must also return an entire EmailSystem API.

Apart from the reasons above, is this testable? You might say yes, because this class uses dependency injection. However, you would have to mock out the Server, EmailSystem, and Sender! This will take at least 10 lines of unintelligible setup for each test. It's also very brittle, because changes to their API will break all of your tests!

The solution to these problems is to **take only your *direct* dependencies in from the constructor.** You shouldn't have to care that Server contains EmailSystem, which contains Sender.

One way to apply the Law of Demeter is to bypass the containing class entirely:

```
public Emailer(Sender sender, String fromAddress) {…}

public void sendSupportEmail(String message, String toAddress) {
  sender.send(fromAddress, toAddress, message);
}
```

This design is much healthier! Now Emailer doesn't have any dependency on Server or EmailSystem at all, because it requested exactly what was needed in the constructor. Emailer is now easier to understand because its **collaborators are known from its public API.** All of the objects Emailer interacts with are explicit, so it's much easier to reason about what methods it may invoke.

You can imagine later changing the structure of EmailSystem entirely. When Emailer takes the Sender dependency directly, it need not worry about that change. **Coupling is much lower.**

What if we wanted to use this Emailer class in a different context? Simple! Just use it elsewhere and pass a different implementation of Sender to it. No need to modify Server or EmailSystem. **It's now more reusable.**

When we only take our direct dependencies, we minimize coupling and maximize reusability. In a future episode, we will talk about how you can apply the Law of Demeter another way by increasing encapsulation.

More information, discussion, and archives:
http://googletesting.blogspot.com