Chapter 6

Preparing for Infrastructure

It's time to continue the exploration of applying DDD to our problem domain. There is no specific next step, but you may be wondering when I will start dealing with the infrastructure. For instance, we haven't yet made it possible to persist or depersist (materialize or reconstitute) the orders to or from a database. As a matter of fact, we haven't even thought about, let alone decided on, using a database.

This is intentional. We want to delay making this decision a bit longer, at least if we are feeling safe and familiar with existing infrastructural options. If this is our first DDD-ish project, this might not be the case, of course, but let's pretend.

The reason I want to delay the binding to a relational database is that in doing so we will focus on the Domain Model with as few distractions as possible. Hey, I'm an old database guy, so it's OK for me to talk about the database as a distraction. I would never dare to say that if I came from the OO side.

Seriously, I'm not saying that the database interaction isn't important; on the contrary, I *know* that it is important. But staying away from the database a bit longer makes trying out and exploring both small and large model changes more productive for us.

It's also much easier to write the tests, because changes done in tests won't be persistent between executions. The test executions themselves will also execute much faster, making it much more manageable for the developers to run the tests after each code change.

There are also problems with this approach. I have already said that we assume we have good control of dealing with the probable upcoming database interaction, so this is not where the problem lies. Instead, there's a good chance that you will want to write some UI prototypes early on, not only for your own sake as a way of challenging your Domain Model, but also as a way to dialogue with the customer. If these prototypes have "live" data that the user can interact with, add to, and so on, they will be more useful than what is common for early prototypes.

181

As you might recall, we ended Chapter 5, "A New Default Architecture," with a preliminary discussion about adding early UI examples. Users will find these examples even more interesting to try out if the data is also around after a restart. (Sure, this can easily be done if you start using the planned persistence solution directly. However, we expect that this will increase the overhead during early refactoring attempts, so what we want is to create an inexpensive "illusion.")

Note

Watch out so that your customer (or your boss) doesn't think that the whole application is done after only having seen one or two prototypes. Been there, done that.

The second thing I'm after at this point is to be able to write tests for save scenarios here, again without dealing with a real database. You might ask what the point of this is. Again, I'd like to focus on the model, the semantics "around" it, and so on.

I could certainly deal with this problem on an individual basis in the repositories, as I have done so far, but I see value in having consistency and just applying one solution to all the repositories. I also want a solution that scales up to early integration testing, and most importantly I want to write real repositories now and not stupid, temporary code that should be thrown away!

So even though I started out saying that it's too early for infrastructure, this chapter will deal with *preparation* for the infrastructure, and in such a way that we won't have to redo work when we add infrastructure. What we want is to write infrastructure-agnostic code.

POCO as a Lifestyle

What I also just said between the lines is that I'd really like to try to keep the main asset of my applications as free from infrastructure-related distractions as possible. The *Plain Old Java Object* (POJO) and *Plain Old CLR Object* (POCO) movement started out in Java land as a reaction against J2EE and its huge implications on applications, such as how it increased complexity in everything and made TDD close to impossible. Martin Fowler, Rebecca Parsons, and Josh MacKenzie coined the term POJO for describing a class that was free from

the "dumb" code that is only needed by the execution environment. The classes should focus on the business problem at hand. Nothing else should be in the classes in the Domain Model.

> ### *Note*
>
> This movement is one of the main inspirations for lightweight containers for Java, such as Spring [Johnson J2EE Development without EJB].

In .NET land it has taken a while for Plain Old… to receive any attention, but it is now known as POCO.

POCO is a somewhat established term, but it's not very specific regarding persistence-related infrastructure. When I discussed this with Martin Fowler he said that perhaps *Persistence Ignorance* (PI) is a better and clearer description. I agree, so I'll change to that from now on in this chapter.

PI for Our Entities and Value Objects

So let's assume we want to use PI. What's it all about? Well, PI means clean, ordinary classes where you focus on the business problem at hand without adding stuff for infrastructure-related reasons. OK, that didn't say all that much. It's easier if we take a look at what PI is *not*. First, a simple litmus test is to see if you have a reference to any external infrastructure-related DLLs in your Domain Model. For example, if you use NHibernate as your O/R Mapper and have a reference to nhibernate.dll, it's a good sign that you have added code to your Domain Model that isn't really core, but more of a distraction.

What are those distractions? For instance, if you use a PI-based approach for persistent objects, there's no *requirement* to do any of the following:

- Inherit from a certain base class (besides `object`)

- Only instantiate via a provided factory

- Use specially provided datatypes, such as for collections

- Implement a specific interface

- Provide specific constructors

- Provide mandatory specific fields

- Avoid certain constructs

There is at least one more, and one that is so obvious that I forgot. You shouldn't have to write database code such as calls to stored procedures in your Domain Model classes. But that was so obvious that I didn't write specifically about it.

Let's take a closer look at each of the other points.

Inherit from a Certain Base Class

With frameworks, a very common requirement for supporting persistence is that they require you to inherit from a certain base class provided by the framework.

The following might not look too bad:

```
public class Customer : PersistentObjectBase
{
    public string Name = string.Empty;
    …

    public decimal CalculateDepth()
    …
}
```

POCO as a Lifestyle

Well, it *wasn't* too bad, but it did carry some semantics and mechanisms that aren't optimal for you. For example, you have used the only inheritance possibility you have for your Customer class because .NET only has single inheritance. It's certainly arguable whether this is a big problem or not, though, because you can often design "around" it.

It's a worse problem if you have developed a Domain Model and now you would like to make it persistent. The inheritance requirement might very well require some changes to your Domain Model. It's pretty much the same when you start developing a Domain Model with TDD. You have the restriction from the beginning that you can't use inheritance and have to save that for the persistence requirement.

Something you should look out for is if the inheritance brings lots of public functionality to the subclass, which might make the consumer of the subclass have to wade through methods that aren't interesting to him.

It's also the case that it's not usually as clean as the previous example, but most of the time PersistentObjectBase forces you to provide some method implementations to methods in PersistentObjectBase, as in the Template Method pattern [GoF Design Patterns]. OK, this is still not a disaster, but it all adds up.

Note

This doesn't necessarily have to be a requirement, but can be seen as a convenience enabling you to get most, if not all, of the interface

---

implementation that is required by the framework if the framework is of that kind of style. We will discuss this common requirement in a later section.

This is how it was done in the Valhalla framework that Christoffer Skjoldborg and I developed. But to be honest, in that case there was so much work that was taken care of by the base class called `EntityBase` that implementing the interfaces with custom code instead of inheriting from `EntityBase` was really just a theoretical option.

Only Instantiate via a Provided Factory

Don't get me wrong, I'm not in any way against using factories. Nevertheless, I'm not ecstatic at being *forced* to use them when it's not my own sound decision. This means, for instance, that instead of writing code like this:

```
Customer c = new Customer();
```

I *have* to write code like this:

```
Customer c = (Customer)PersistentObjectFactory.CreateInstance
   (typeof(Customer));
```

Note

I know, you think I did my best to be unfair by using extremely long names, but this isn't really any better, is it?

```
Customer c = (Customer)POF.CI(typeof(Customer));
```

Again, it's not a disaster, but it's not optimal in most cases. This code just looks a lot weirder than the first instantiation code, doesn't it? And what often happens is that code like this increases testing complexity.

Often one of the reasons for the mandatory use of a provided factory is that you will consequently get help with dirty checking. So your Domain Model classes will get subclassed dynamically, and in the subclass, a dirty-flag (or several) is maintained in the properties. The factory makes this transparent to the consumer so that it instantiates the subclass instead of the class the factory consumer asks for. Unfortunately, for this to work you will also have to make your properties virtual, and public fields can't be used (two more small details that lessen the PI-ness a little). (Well, you *can use* public fields, but they can't be "overridden" in the generated subclass, and that's a problem if the purpose of the subclass is to take care of dirty tracking, for example.)

> ### *Note*
>
> There are several different techniques when using Aspect-Oriented Programming (AOP) in .NET, where runtime subclassing that we just discussed is probably the most commonly used. I've always seen having to declare your members as virtual for being able to intercept (or advice) as a drawback, but Roger Johansson pointed something out to me. Assume you want to make it impossible to override a member and thereby avoid the extra work and responsibility of supporting subclassing. Then that decision should affect both ordinary subclassing and subclassing that is used for reasons of AOP. And if you make the member virtual, you are prepared for having it redefined, again both by ordinary subclassing and AOP-ish subclassing.
>
> It makes sense, doesn't it?

Another common problem solved this way is the need for Lazy Load, but I'd like to use that as an example for the next section.

POCO as a Lifestyle

Use "Specially" Provided Datatypes, Such as Collections

It's not uncommon to have to use special datatypes for the collections in your Domain Model classes: special as in "not those you would have used if you could have chosen freely."

The most common reason for this requirement is probably for supporting Lazy Load, [Fowler PoEAA], or rather *implicit* Lazy Load, so that you don't have to write code on your own for making it happen. (Lazy Load means that data is fetched just in time from the database.)

But the specific datatypes could also bring you other functionality, such as special delete handling so that as soon as you delete an instance from a collection the instance will be registered with the Unit of Work [Fowler PoEAA] for deletion as well. (Unit of Work is used for keeping track of what actions should be taken against the database at the end of the current logical unit of work.)

> ### *Note*
>
> Did you notice that I said that the specific datatypes could *bring* you *functionality*? Yep, I don't want to sound overly negative about NPI (Not-PI).

You could get help with bi-directionality so that you don't have to code it on your own. This is yet another example of something an AOP solution can take care of for you.

Implement a Specific Interface

Yet another very regular requirement on Domain Model classes for being persistable is that they implement one or more infrastructure-provided interfaces.

This is naturally a smaller problem if there is very little code you have to write in order to implement the interface(s) and a bigger problem if the opposite is true.

One example of interface-based functionality could be to make it possible to fill the instance with values from the database without hitting setters (which might have specific code that you don't want to execute during reconstitution).

Another common example is to provide interfaces for optimized access to the state in the instances.

Provide Specific Constructors

Yet another way of providing values that reconstitute instances from the database is by requiring specific constructors, which are constructors that have nothing at all to do with the business problem at hand.

It might also be that a default constructor is needed so that the framework can instantiate Domain Model classes easily as the result of a Get operation against the database. Again, it's not a very dramatic problem, but a distraction nonetheless.

Provide Mandatory Specific Fields

Some infrastructure solutions require your Domain Model classes to provide specific fields, such as Guid-based Id-fields or int-based Version-fields. (With Guid-based Id-fields, I mean that the Id-fields are using Guids as the datatype.) That simplifies the infrastructure, but it might make your life as a Domain Model-developer a bit harder. At least if it affects your classes in a way you didn't want to.

Avoid Certain Constructs/Forced Usage of Certain Constructs

I have already mentioned that you might be forced to use virtual properties even if you don't really want to. It might also be that you have to avoid certain constructs, and a typical example of this is read-only fields. Read-only (as when the keyword readonly is used) fields can't be set from the outside (except with constructors), something that is needed to create 100% PI-Domain Model classes.

Using a private field together with a get-only property is pretty close to a read-only field, but not exactly the same. It could be argued that a read-only field is the most intention-revealing solution.

> ### *Note*
>
> Something that has been discussed a lot is whether .NET attributes are a good or bad thing regarding decorating the Domain Model with information about how to persist the Domain Model.
>
> My opinion is that such attributes can be a good thing and that they don't really decrease the PI level if they are seen as default information that can be overridden. I think the main problem is if they get too verbose to distract the reader of the code.

PI or not PI?

POCO as a Lifestyle

PI or not PI—of course it's not totally binary. There are some gray areas as well, but for now let's be happy if we get a feeling for the intention of PI rather than how to get to 100%. Anything extreme incurs high costs. We'll get back to this in Chapter 9, "Putting NHibernate into Action," when we discuss an infrastructure solution.

> ### *Note*
>
> What is an example of something completely binary in real life? Oh, one that I often remind my wife about is when she says "that woman was *very* pregnant."

Something we haven't touched on yet is that it also depends on at what point in "time" we evaluate whether we use PI or not.

Runtime Versus Compile Time PI

So far I have talked about PI in a timeless context, but it's probably most important at compile time and not as important at runtime. "What does that mean?" I hear you say? Well, assume that code is created for you, infrastructure-related code that you never have to deal with or even see yourself. This solution is probably better than if you have to maintain similar code by hand.

This whole subject is charged with feelings because it's controversial to execute something other than what you wrote yourself. The debugging experience might turn into a nightmare!

Note

Mark Burhop commented as follows:

Hmmm… This was the original argument against C++ from C programmers in the early 90s. "C++ sticks in new code I didn't write." "C++ hides what is really going on." I don't know that this argument holds much water anymore.

It's also harder to inject code at the byte level for .NET classes compared to Java. It's not supported by the framework, so you're on your own, which makes it a showstopper in most cases.

What is most often done instead is to use some alternative techniques, such as those I mentioned with runtime-subclassing in combination with a provided factory, but it's not a big difference compared to injected code. Let's summarize with calling it emitting code.

The Cost for PI Entitites/Value Objects

I guess one possible reaction to all this is "PI seems great—why not use it all the time?" It's a law of nature (or at least software) that when everything seems neat and clean and great and without fault, then come the drawbacks. In this case, I think one such is overhead.

I did mention earlier in this chapter that speed is something you will sacrifice for a high level of PI-ness, at least for runtime PI, because you are then directed to use reflection, which is quite expensive. (If you think compile-time PI is good enough, you don't need to use reflection, but can go for an AOP solution instead and you can get a better performance story.)

You can easily prove with some operation in a tight loop that it is magnitudes slower for reading from/writing to fields/properties with reflection compared to calling them in the ordinary way. Yet, is the cost *too* high? It obviously depends on the situation. You'll have to run tests to see how it applies to your own case. Don't forget that a jump to the database is very expensive compared to a lot you're doing in your Domain Model, yet at the same time, you aren't comparing apples and apples here. For instance, the comparison might not be between an ordinary read and a reflection-based read.

POCO as a Lifestyle

A Typical Example Regarding Speed

Let's take an example to give you a better understanding of the whole thing. One common operation in a persistence framework is deciding whether or not an instance should be stored to the database at the end of a scenario. A common solution to this is to let the instance be responsible for signaling IsDirty if it is to be stored. Or better still, the instance could also signal itself to the Unit of Work when it gets dirty so that the Unit of Work will remember that when it's time to store changes.

But (you know there had to be a "but," right?) that requires some abuse of PI, unless you have paid with AOP.

> **Note**
>
> There are other drawbacks with this solution, such as it won't notice the change if it's done via reflection and therefore the instance changes won't get stored. This drawback was a bit twisted, though.

POCO as a Lifestyle

An alternative solution is not to signal anything at all, but let the infrastructure remember how the instances looked when fetched from the database. Then at store time compare how the instances look now to how they looked when read from the database.

Do you see that it's not just a comparison of one ordinary read to one reflection-based read, but they are totally different approaches, with totally different performance characteristics? To get a real feeling for it, you can set up a comparison yourself. Fetch one million instances from the database, modify one instance, and then measure the time difference for the store operation in both cases. I know, it was another twisted situation, but still something to think about.

Other Examples

That was something about the speed cost, but that's not all there is to it. Another cost I pointed out before was that you might get less functionality automatically if you try hard to use a high level of PI. I've already gone through many possible features you could get for free if you abandon some PI-ness, such as automatic bi-directional support and automatic implicit Lazy Load.

It's also the case that the dirty tracking isn't just about performance. The consumer might be very interested as well in using that information when painting the forms—for example, to know what buttons to enable.

So as usual, there's a tradeoff. In the case of PI versus non-PI, the tradeoff is overhead and less functionality versus distracting code in the core of your application that couples you to a certain infrastructure and also makes it harder to do TDD. There are pros and cons. That's reasonable, isn't it?

The Cost Conclusion

So the conclusion to all this is to be aware of the tradeoffs and choose carefully. For instance, if you get something you need alongside a drawback you can live with, don't be too religious about it!

That said, I'm currently in the pro-PI camp, mostly because of how nice it is for TDD and how clean and clear I can get my Entities and Value Objects.

I also think there's a huge difference when it comes to your preferred approach. If you like starting from code, you'll probably like PI a great deal. If you work in an integrated tool where you start with detailed design in UML, for example, and from there generate your Domain Model, PI is probably not that important for you at all.

But there's more to the Domain Model than Entities and Value Objects. What I'm thinking about are the Repositories. Strangely enough, very little has been said as far as PI for the Repositories goes.

PI for Our Repositories

I admit it: saying you use PI for Repositories as well is pushing it. This is because the purpose of Repositories is pretty much to give the consumer the illusion that the complete set of Domain Model instances is around, as long as you adhere to the protocol to go to the Repository to get the instances. The illusion is achieved by the Repositories talking to infrastructure in specific situations, and talking to infrastructure is not a very PI-ish thing to do.

For example, the Repositories need something to pull in order to get the infrastructure to work. This means that the assembly with the Repositories needs a reference to an infrastructure DLL. And this in its turn means that you have to choose between whether you want the Repositories in a separate DLL, separate from the Domain Model, or whether you want the Domain Model to reference an infrastructure DLL (but we will discuss a solution soon that will give you flexibility regarding this).

Problems Testing Repositories

It's also the case that when you want to test your Repositories, they are connected to the O/R Mapper and the database.

> ### *Note*
>
> Let's for the moment assume that we will use an O/R Mapper. We'll get back to a more thorough discussion about different options within a few chapters.

Suddenly this provides you with a pretty tough testing experience compared to when you test the Entities and Value Objects in isolation.

Of course, what you could do is mock your O/R Mapper. I haven't done that myself, but it feels a bit bad on the "bang for the bucks" rating. It's probably quite a lot of work compared to the return.

Problems Doing Small Scale Integration Testing

In previous chapters I haven't really shown any test code that focused on the Repositories at all. Most of the interesting tests should use the Domain Model. If not, it might be a sign that your Domain Model isn't as rich as it should be if you are going to get the most out of it.

That said, I did use Repositories in some tests, but really more as small integration tests to see that the cooperation between the consumer, the Entities, and the Repositories worked out as planned. As a matter of fact, that's one of the advantages Repositories have compared to other approaches for giving persistence capabilities to Domain Models, because it was easy to write Fake versions of the Repositories. The problem was that I wrote quite a lot of dumb code that has to be tossed away later on, or at least rewritten in another assembly where the Repositories aren't just Fake versions.

What also happened was that the semantics I got from the Fake versions wasn't really "correct." For instance, don't you think the following seems strange?

```
[Test]
public void FakeRepositoryHaveIncorrectSemantics()
{
    OrderRepository r1 = new OrderRepository();
    OrderRepository r2 = new OrderRepository();

    Order o = new Order();

    r1.Add(o);
    x.PersistAll();

    //This is fine:
    Assert.IsNotNull(r1.GetOrder(o.Id));
```

POCO as a Lifestyle

```
    //This is unexpected I think:
    Assert.IsNull(r2.GetOrder(o.Id));
}
```

Note

As the hawk-eyed reader saw, I decided to change `AddOrder()` to `Add()` since the last chapter.

I'm getting a bit ahead of myself in the previous code because we are going to discuss save scenarios shortly. Anyway, what I wanted to show was that the Fake versions of Repositories used so far don't work as expected. Even though I thought I had made all changes so far persistent with `PersistAll()`, only the first Repository instance could find the order, not the second Repository instance. You might wonder why I would like to write code like that, and it's a good question, but it's a pretty big misbehavior in my opinion.

What we could do instead is mock each of the Repositories, to test out the cooperation with the Entities, Repositories, and consumer. This is pretty cheaply done, and it's also a good way of testing out the consumer and the Entities. However, the test value for the Repositories themselves isn't big, of course. We are kind of back to square one again, because what we want then is to mock out one step further, the O/R Mapper (if that's what is used for dealing with persistence), and we have already talked about that.

Earlier Approach

So it's good to have Repositories in the first place, especially when it comes to testability. Therefore I used to swallow the bitter pill and deal with this problem by creating an interface for each Repository and then creating two implementing classes, one for Fake and one for real infrastructure. It could look like this. First, an interface in the Domain Model assembly:

```
public interface ICustomerRepository
{
    Customer GetById(int id);
    IList GetByNamePattern(string namePattern);
    void Add(Customer c);
}
```

Then two classes (for example, `FakeCustomerRepository` and `MyInfrastructureCustomerRepository`) will be located in two different assemblies (but all in one namespace, that of the Domain Model, unless of course there are several partitions of the Domain Model). See Figure 6-1.
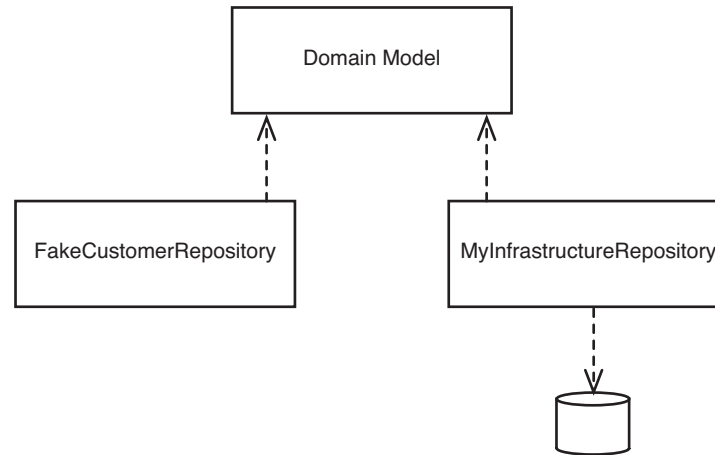
Figure 6-1   Two Repository assemblies

That means that the Domain Model itself won't be affected by the chosen infrastructure when it comes to the Repositories, which is nice if it doesn't cost anything.

But it does cost. It also means that I have to write two Repositories for each Aggregate root, and with totally different Repository code in each case.

Further on, it means that the production version of the Repositories lives in another assembly (and so do the Fake Repositories), even though I think Repositories are part of the Domain Model itself. "Two extra assemblies," you say, "That's no big deal." But for a large application where the Domain Model is partitioned into several different assemblies, you'll learn that typically it doesn't mean two extra assemblies for the Repositories, but rather the amount of Domain Model assemblies multiplied by three. That is because each Domain Model assembly will have its own Repository assemblies.

Even though I think it's a negative aspect, it's not nearly as bad as my having the silly code in the Fake versions of the Repositories. That feels just bad.

A Better Solution?

The solution I decided to try out was creating an abstraction layer that I call NWorkspace [Nilsson NWorkspace]. It's a set of adapter interfaces, which I have written implementations for in the form of a Fake. The Fake is just two levels of hashtables, one set of hashtables for the persistent Entities (simulating a database) and one set of hashtables for the Unit of Work and the Identity Map. (The Identity Map keeps track of what identities, typically primary keys, are currently loaded.)

POCO as a Lifestyle

The other implementation I have written is for a specific O/R Mapper.

> ### *Note*
>
> When I use the name NWorkspace from now on, you should think about it as a "persistence abstraction layer." NWorkspace is just an example and not important in itself.

Thanks to that abstraction layer, I can move the Repositories back to the Domain Model, and I only need one Repository implementation per Aggregate root. The same Repository can work both against an O/R Mapper and against a Fake that won't persist to a database but only hold in memory hashtables of the instances, but with similar semantics as in the O/R Mapper-case. See Figure 6-2.
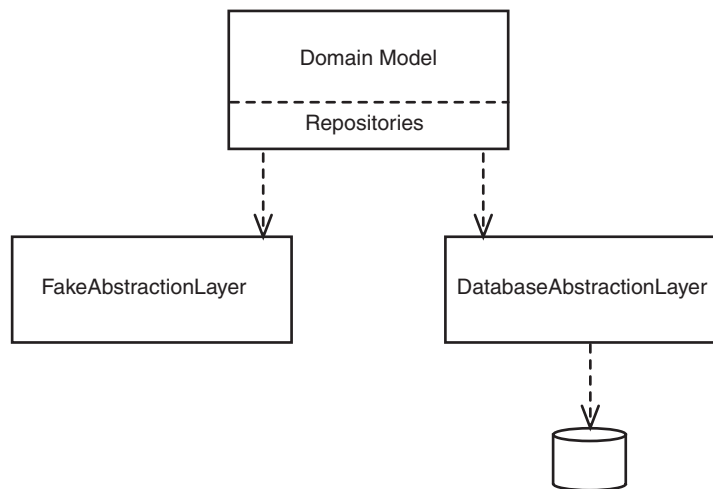

Figure 6-2   A single set of Repositories thanks to an abstraction layer

The Fake can also be serialized to/deserialized from files, which is great for creating very competent, realistic, and at the same time extremely refactoring-friendly early versions of your applications.

Another possibility that suddenly feels like it could be achieved easily (for a small abstraction layer API at least) could be to Mock the infrastructure instead of each of the Repositories. As a matter of fact, it won't be a matter of Mocking one infrastructure-product, but all infrastructure products that at one time will have adapter implementations for the abstraction layer (if that happens, that there will be other implementations than those two I wrote—it's probably not

that likely). So more to the point, what is then being Mocked is the abstraction layer.

It's still a stretch to talk about PI Repositories, but with this solution I can avoid a reference to the infrastructure in the Domain Model. That said, in real-world applications I have kept the Repositories in a separate assembly anyway. I think it clarifies the coupling, and it also makes some hacks easier to achieve and then letting some Repository methods use raw SQL where that proves necessary (by using connection strings as markers for whether optimized code should be used or not).

However, instead of referring to the Persistence Framework, I have to refer to the NWorkspace DLL with the adapter interfaces, but that seems to be a big step in the right direction. It's also the case that there are little or no distractions in the Repositories; they are pretty "direct" (that is, if you find the NWorkspace API in any way decent).

So instead of writing a set of Repositories with code against an infrastructure vendor's API and another set of Repositories with dummy code, you write one set of Repositories against a (naïve) attempt for a standard API.

<table>
<tr><td>

POCO as a Lifestyle

</td><td>

Note

I'm sorry for nagging, but I must say it again: It's the concept I'm after! My own implementation isn't important at all.

</td></tr>
</table>

Let's find another term for describing those Repositories instead of calling the PI Repositories. What about *single-set Repositories*? OK, we have a term for now for describing when we build a single set of Repositories that can be used both in Fake scenarios and in scenarios with a database. What's probably more interesting than naming those Repositories is seeing them in action.

Some Code in a Single-Set Repository

To remind you what the code in a Fake version of a Repository could look like, here's a method from Chapter 5:

```
//OrderRepository, a Fake version
public Order GetOrder(int orderNumber)
{
    foreach (Order o in _theOrders)
    {
        if (o.OrderNumber == orderNumber)
            return o;
    }
    return null;
}
```

OK, that's not especially complex, but rather silly, code.

If we assume that the `OrderNumber` is an Identity Field [Fowler PoEAA] (Identity Field means a field that binds the row in the database to the instance in the Domain Model) of the `Order`, the code could look like this when we use the abstraction layer (`_ws` in the following code is an instance of `IWorkspace`, which in its turn is the main interface of the abstraction layer):

```
//OrderRepository, a single-set version
public Order GetOrder(int orderNumber)
{
    return (Order)_ws.GetById(typeof(Order), orderNumber);
}
```

Pretty simple and direct I think. And—again—that method is done now, both for Fake and for when real infrastructure is used!

The Cost for Single-Set Repositories

So I have yet another abstraction. Phew, there's getting to be quite a lot of them, don't you think? On the other hand, I believe each of them adds value.

Still, there's a cost, of course. The most obvious cost for the added abstraction layer is probably the translation at runtime that has to be done for the O/R Mapper you're using. In theory, the O/R Mapper could have a native implementation of the abstraction layer, but for that to happen some really popular such abstraction layer must be created.

Then there's a cost for building the abstraction layer and the adapter for your specific O/R Mapper. That's the typical framework-related problem. It costs a lot for building the framework, but it can be used many times, if the framework *ever* becomes useful.

With some luck, there will be an adapter implementation for the infrastructure you are using and then the cost isn't yours, at least not the framework-building cost. There's more, though. You have to learn not only the infrastructure of your choice, but also the abstraction layer, and that can't be neglected.

POCO as a Lifestyle

Note

It was easier in the past as you only had to know a little about Cobol and files. Now you have to be an expert on C# or Java, Relational Databases, SQL, O/R Mappers, and so on, and so forth. If someone tries to make the whole thing simpler by adding yet another layer, that will tip the scales, especially for newcomers.

Yet another cost is, of course, that the abstraction layer will be kind of the least common denominator. You won't find all the power there that you can find in your infrastructure of choice. Sure, you can always bypass the abstraction layer, but that comes with a cost of complexity and external Repository code, and so on. So it's important to investigate whether your needs could be fulfilled with the abstraction layer to 30%, 60%, or 90%. If it's not a high percentage, it's questionable whether it's interesting at all.

Ok, let's return to the consumer for a while and focus on save functionality for a change.

Dealing with Save Scenarios

As I said at the end of the previous chapter, I will move faster from now on and not discuss all the steps in my thought process. Instead it will be more like going directly to the decided solution. Not decided as in "done," but decided as in "for now."

With that said, I'd still like to discuss tests, especially as a way of clarification.

Now I'd like to discuss save scenarios from the consumer's point of view. So here's a test for showing how to save two new Customers:

```
[Test]
Public void CanSaveTwoCustomers()
{
    int noOfCustomersBefore =
        _GetNumberOfStoredCustomers();

    Customer c = new Customer();
    c.Name = "Volvo";
    _customerRepository.Add(c);

    Customer c2 = new Customer();
    c2.Name = "Saab";
    _customerRepository.Add(c2);

    Assert.AreEqual(noOfCustomersBefore,
        _GetNumberOfStoredCustomers());

    _ws.PersistAll();

    Assert.AreEqual(noOfCustomersBefore + 2,
        _GetNumberOfStoredCustomers());
}
```

At first glance, the code just shown is pretty simple, but it "hides" lots of things we haven't discussed before. First, it reveals what kind of consumer code I want to be able to write. I want to be able to do a lot of stuff to several different instances, and then persist all the work with a single call such as `PersistAll()`. (The call to `_GetNumberOfStoredCustomers()` goes to the persistence engine to check the number of persistent customers. It's not until after `PersistAll()` that the number of persistent customers has increased.)

A missing piece of the puzzle is that the Repository was fed with the `_ws` at instantiation time. In this way, I can control the Repositories that should participate in the same Unit of Work and those that should be isolated in another Unit of Work.

Yet another thing that might be interesting is that I ask the Repositories for help (the `Add()` call) in notifying the Unit of Work that there is a new instance for persisting at next `PersistAll()` call. I'm referring to the life cycle I want to have for a persistent instance (I touched on this in Chapter 5, so I won't repeat it here).

What I think is worth pointing out is that if I expect it to be enough to associate the Aggregate root with the Unit of Work, the instances that are part of the Aggregate and that the Aggregate root reaches will get persisted to the database as well when we say `PersistAll()`.

Again, Aggregates assist us well; they provide a tool for knowing the size of graph that will be persisted because the Aggregate root is marked for being persisted. Again, Aggregates make for simplification.

Note

O/R Mappers are often able to be configured for how far the reach of persist by reachability should go. But even when they are configured that way, the Aggregates are a very good guide in my opinion. What I mean is that I use the Aggregates for determining how far the reachability should reach, when I do the configuration.

Let's take a closer look at the reasoning behind the decisions discussed so far.

Reasons for the Decisions

Why did I choose as I did? First, I want to use the Unit of Work pattern. I want its characteristics: to create a logical Unit of Work.

So you can make lots of changes to the Domain Model, collect the information in the Unit of Work, and then ask the Unit of Work to save the collected changes to the database.

There are several styles of Unit of Work to use. The one I prefer is to make it as transparent as possible for the consumer and therefore the only message needed is to say Add() to the Repository (which in turn will talk to the Unit of Work).

If the reconstitution is done via the Repository, the Unit of Work-implementation can inject some object that can collect information about changes. Otherwise, there can be a snapshot taken at read time that will be used to control the changes by the Unit of Work at persist time.

I also chose to control save or not save (PersistAll()) outside of the Repositories. In this particular example, I could just as well have had PersistAll() directly on the CustomerRepository, but I chose not to. Why? Why not let Repositories hide Unit of Work completely? Well, I could, but I often find that I want to synchronize changes to several Aggregates (and therefore also to several different Repositories) in a single logical unit, and that's the reason. So code like this is not only possible to write, but also very typical:

Dealing with Save Scenarios

```
Customer c = new Customer();
_customerRepository.Add(c);

Order o = new Order();
_orderRepository.Add(o);

_ws.PersistAll();
```

One alternative might be the following:

```
Customer c = new Customer();
_customerRepository.Add(c);
_customerRepository.PersistAll();

Order o = new Order();
_orderRepository.Add(o);
_orderRepository.PersistAll();
```

But then I have two different Unit of Work instances and two Identity Maps (it doesn't *have* to be that way, but let's assume it for the sake of the discussion), which can give pretty strange effects if we aren't very careful. After all, all five lines in the first example were one scenario, and because of that I find it most intuitive and appropriate to treat it like one scenario regarding how I deal with the Unit of Work and Identity Map as well. I mean, the scenario should just have one Unit of Work and one Identity Map.

Another thing that might be a problem is that when the Repository hid the Unit of Work it probably meant that there were two database transactions. That in turn means that you might have to prepare to add compensating operations when the outcome of a scenario isn't as expected. In the previous case, it's probably not too disastrous if the Customer is added to the database but not the Order. However, it can be a problem, depending upon your Aggregate design.

That said, Aggregates "should" be designed so that they are in a consistent state at PersistAll() time. But the loose relationship between Aggregates doesn't typically live under such strict requirements. That might make you like the second solution. On the other hand, the second solution would store two totally unrelated customers in the same PersistAll() if both of those customers were associated to the Repository. That is actually less important than grouping a customer and its orders together. Aggregates are about objects, not classes.

What speaks for the solution in the second example is if one Aggregate comes from one database and the other Aggregate is stored in another database at another database server. Then it's probably easiest to have two Unit of Work-instances anyway, one for each database. So, solution two is slightly less coupled.

Note

I could even let Add() fulfill the transaction, but then I have different semantics from those I have discussed and expressed so far. It would be crucial to call Add() at the right point in time. This is less important with the solution I have chosen, as long as the call is done before PersistAll().

With Add() fulfilling the transaction, it would also mean that it's certainly not a matter of "gather all changes and persist them all at PersistAll()," which again is very different from my current solution.

While we're at it, why not then encapsulate the whole thing in the Entity instead so that you can write the following code?

```
Customer c = new Customer()
c.Name = "Saab";
c.Save();
```

I think it's moving away from the style I like. I think it breaks the *Single Responsibility Principle* (SRP) [Martin PPP], and it's low PI-level. I think I'm also moving into "matter of taste" territory.

I also have a problem with inconsistency if one save goes well and others do not. (You could argue that physical transaction and Unit of Work don't *have* to be the same, but that increases complexity in my opinion. The way I see it is if you don't have to do something, you shouldn't.)

However, by using my favorite technique, there's nothing to stop me from getting the same effects of storing one Aggregate at a time if I really want to by letting the Repositories hide the Unit of Work and Identity Map. It could then look like this:

```
Customer c = new Customer();
_customerRepository.Add(c);
_ws1.PersistAll();

Order o = new Order();
_orderRepository.Add(o);
_ws2.PersistAll();
```

Note

For the previous scenario, the end result would be the same with a single _ws, but that depended on the specific example.

Dealing with Save Scenarios

What I mean is that I *can* have one Unit of Work/Identity Map when I so wish, and several when I so wish. I think this is slightly more flexible, which I like a lot, and this is one more reason for my choice; namely that I currently prefer to see the Unit of Work as something belonging to the consumer of the Domain Model (that is the Application layer or the presentation layer) rather than the Domain Model itself.

If we assume that each Repository has its own Identity Map, it can get a bit messy if the same order is referenced from two different Repositories, at least if you make changes to the same logical order (but two different instances) in both Repositories.

As far as risks are concerned, what it boils down to is which risk you prefer. The risk of committing instances that you weren't done with because a PersistAll() call will deal with more instances than you expected? Or the risk of forgetting to commit a change because you'll have to remember what Repositories to ask to do PersistAll().

I'm not saying that it's a solution without problems, but again, I prefer the Identity Map *and* the Unit of Work to belong to the scenario.

> ### *Note*
>
> Deciding on what programming model you want is up to you, as usual. There are pros and cons to each.

I have mentioned Unit of Work and Identity Map together over and over again. It's such a common combination, not only in my text, but in products as well. For example, there is Persistence Manager in JDO [Jordan/Russell JDO] and Session in Hibernate [Bauer/King HiA].

I thought it might deserve a pattern, and I was thinking about writing it up, but when I discussed it with Martin Fowler he notified me that he discusses that in [Fowler PoEAA] when he talks about Identity Map and Unit of Work. That's more than enough, so I decided not to repeat more about that.

OK, now there's been a lot of talk and no action. Let's start building the Fake mechanism and see where we end up.

Let's Build the Fake Mechanism

Let's move on in an interface-based way for a while, or at least for a start. I have already touched on a couple of the methods, but let's start from the beginning. A reduced version of the interface of the abstraction layer (which I earlier in the chapter already called IWorkspace) could look like this:

```
public interface IWorkspace
{
    object GetById(Type typeToGet, object idValue);
    void MakePersistent(object o);
    void PersistAll();
}
```

So far the whole interface is pretty straightforward. The first method is called GetById() and is used for reconstituting an object from the database. You say what type you expect and the identity value of the object.

The second method, called MakePersistent(), is used for associating new instances with the IWorkspace instance so that they will be persisted at next PersistAll(). Finally, PersistAll() is for persisting what is found in the Unit of Work into the database.

MakePersistent() isn't needed if you have read the instance from the database with GetById(), because then the instance is already associated with the Unit of Work.

So far I think you'll agree that the API is extremely simple, and I think it is very important in order to keep complexity down in this abstraction layer. OK, it's not all that competent yet, so we need to add more.

More Features of the Fake Mechanism

The first thing that springs to mind is that we need to deal with transactions as a very important concept, at least from a correctness standpoint. On the other hand, it's not something that is important for most UI-programmers. (We could swap "UI-programmers" for "UI-code" just as well.) What I mean is that I don't want to put the responsibility for transaction management on the UI-programmer because it's too much of a distraction for him—and too important to be seen as a distraction. Still, I want adaptable transaction scope so that there isn't only a predefined set of possible transactions.

So my goals are pretty similar to those of declarative transactions in COM+, but I have chosen a pretty different API. Instead of setting attributes on the classes for describing whether they require transactions or not, I will just say that `PersistAll()` internally does all its work in an explicit transaction, even though you explicitly didn't ask for it

I know that on the face of it this feels overly simple to many old-timers. That goes for me as well, because I believe transaction handling is so important that I like to deal with it manually. If the goal is to be able to deal with something like 90% of the situations, however, I think `PersistAll()` could very well use an explicit transaction, and it's as simple as that.

Again, it sounds way too simplistic, and of course there are problems. One typical problem is logging. Assume that you log to the database server; you don't always want the logging operation to fail if the ordinary work fails. However, that's simple to deal with; you just use a separate workspace instance for the logging. If you want it to use the abstraction layer at all, the logging will probably just be implemented as a Service instead, which probably has nothing to do with the abstraction layer. As a matter of fact, there's a good chance that you will use a third-party product for logging, or perhaps something like log4net [Log4Net]. It's not something that will interfere with or be disturbed by the transaction API of the abstraction layer.

Another problem is that there might well be a need for the `GetById()` method to live in the same transaction as the upcoming `PersistAll()`. That won't happen by default, but if you want to force that, you can call the following method before `GetById()`:

```
void BeginReadTransaction(TransactionIsolationLevel til)
```

To emphasize this even more, there is also an overload to `GetById()` to ask for an exclusive lock, but this comes with a warning tag. Make sure you know what

you're doing when you use this! For example, there should be no user interaction whatsoever after `BeginReadTransaction()` or read with exclusive lock and before `PersistAll()`.

But I digress—what is important for the Fake? Because the Fake only targets single-user scenarios, the transactional semantics aren't very important, and for reasons of simplicity those will probably not be dealt with at all. Still, the consumer code can be written with transaction-handling in mind when the Fake is used, of course, so you don't have to change the code when it comes to swapping the Fake for the real infrastructure.

I hear the now very frightened experienced developer exclaim, "Hey, what happened to `Rollback()`?"

Well, the way I see it, it's not important to have rollback in the API. If `PersistAll()` is responsible for commit or rollback internally, what will happen then is that when the consumer gets the control back from `PersistAll()`, all changes or none have been persisted. (The consumer is notified about a rollback by an exception.)

The exception to this is when you are after `BeginReadTransaction()` and you then want to cancel. Then you call the following method:

```
void Clean()
```

It will roll back the ongoing transaction and will also clean the Unit of Work and the Identity Map. It's a good idea to use `Clean()` after a failed transaction because there will be no attempt at all in NWorkspace to roll back the changes in Domain Model instances. Sure, it depends upon what the problem with the failed transaction was, but the simple answer is to restart.

Some problems can lead to a retry within `PersistAll()`. In the case of a deadlock, for example, `PersistAll()` can retry a couple of times before deciding it was a failure. This is yet another thing that simplifies life for the consumer programmer so that she can focus on what is important to her, namely to create a good user experience, not following lots and lots of protocols.

Now we have talked a lot about the functionality that is important for NWorkspace, but not for the Fake version of NWorkspace. Let's get back on track and focus for a while on the Fake and its implementation instead.

The Implementation of the Fake

I'm not going to drag you through the details of the Fake implementation—I'm just going to talk conceptually about how it's built, mostly in order to get a feeling for the basic idea and how it can be used.

The fake implementation uses two layers of Identity Maps. The first layer is pretty similar to ordinary Identity Maps in persistence frameworks, and it

keeps track of all Entities that you have read within the current scenario. The second layer of Identity Maps is for simulating the persistent engine, so here the instances aren't kept on a scenario level, but on a global level (that is, the same set of Identity Maps for all scenarios).

So when you issue a call to GetById(), if the ID is found in the Identity Map for the requested type, there won't be a roundtrip to the database (or in case of the Fake, there won't be a jump to the second layer of Identity Maps). On the other hand, if the ID isn't found in the first layer of Identity Maps, it's fetched from the second layer, copied to the first layer, and then returned to the consumer.

The MakePersistent() is pretty simple; the instance is just associated with the first layer of Identity Maps. And when it's time for PersistAll(), all instances in the first layer are copied to the second layer. Simple and clean.

This describes the basic functionality. Still, it might be interesting to say a bit about what's troublesome, also. One example is that I don't want the Fake to influence the Domain Model in any way at all. If it does, we're back to square one, adding infrastructure-related distractions to the Domain Model, or even worse, Fake-related distractions.

One example of a problem is that I don't know which is the Identity field(s) of a class. In the case of the real infrastructure, it will probably know that by some metadata. I could read that same metadata in the Fake to find out, but then the Fake must know how to deal with (theoretically) several different metadata formats, and I definitely don't like that.

The simplistic solution I've adopted is to assume a property (or field) called Id. If the developer of the Domain Model has used another convention, it could be described to the Fake at instantiation of the FakeWorkspace.

Again, this was more information than you probably wanted now, but it leads us to the important fact that there are additional things that you can/need to do in the instantiation phase of the Fake compared to the infrastructure implementations of NWorkspace.

To take another example, you can read from file/save to file like this:

```
//Some early consumer
IWorkspace ws = new
    NWorkspaceFake.FakeWorkspace("c:/temp/x.nworkspace");

//Do stuff...

((NWorkspaceFake.FakeWorkspace)ws).
    PersistToFile("c:/temp/x.nworkspace");
```

We talked quite a lot about PI and the Fake mechanism in a way that might lead you to believe that you must go for a PI-supporting infrastructure later on if you choose to use something like the Fake mechanism now. This is not

the case at all. It's not even true that non-PI-supporting infrastructure makes it harder for you to use TDD. It's traditionally the case, but not a must.

Speaking of TDD, has the Fake affected our unit tests much yet?

Affecting the Unit Tests

Nope, certainly not all tests will be affected. Most tests should be written with classes in the Domain Model in as isolated a way as possible, without a single call to Repositories. For instance, they should be written during development of all the logic that should typically be around in the Domain Model classes. Those tests aren't affected at all.

The unit tests that should deal with Repositories are affected, and in a positive way. It might be argued that these are more about integration testing, but it doesn't have to be that way. Repositories are units, too, and therefore tests on them are unit tests.

And even when you do integration testing with Repositories involved, it's nice to be able to write the tests early and to write them (and the Repositories) in a way so that it is possible to use them when you have infrastructure in place as well.

I think a nice goal is to get all the tests in good shape so that they can run both with the Fake mechanism and the infrastructure. That way you can execute with the Fake mechanism in daily work (for reasons of execution time) and execute with the infrastructure a couple of times a day and at the automatic builds at check in.

You can also work quite a long way without infrastructure in place. You must, of course, also think a bit about persistence, and especially for your first DDD projects, it's important to work iteratively from the beginning. But when you get more experience, delaying the addition of the persistence will give you the shortest development time in total and the cleanest code.

This also gives you the possibility of another refactoring rhythm, with more instant feedback whether you like the result or not. First, you get everything to work with the Fake (which is easier and faster than getting the whole thing, including the database, to the right level), and if you're happy then you proceed to get it all to work with the infrastructure. I believe the big win is that this will encourage you to be keener to do refactorings that would normally just be a pain, especially when you are unsure about the outcome. Now you can give them a try pretty easily.

Of course, trying to avoid code duplication is as important for unit tests as it is for the "real" code. (Well, at least close to as important. There's also a competing strive to "show it all" inline.) Therefore I only want to write the tests once, but be able to execute them both for the Fake and the real infrastructure

when in place. (Please note that this only goes for some of the tests of course. Most of the tests aren't about the Repositories at all so it's important that you partition your tests for this aspect.)

Structure of the Repository-Related Tests

One way of approaching this is to write a base class for each set of tests that are Repository-affecting. Then I use the Template Method pattern for setting up an IWorkspace the way I want. It could look like this, taking the base class first:

```
[TestFixture]
public abstract class CustomerRepositoryTestsBase
{
    private IWorkspace _ws;
    private CustomerRepository _repository;

    protected abstract IWorkspace _CreateWorkspace();

    [SetUp]
    public void SetUp()
    {
        _ws = _CreateWorkspace();
        _repository = new CustomerRepository(_ws);
    }

    [TearDown]
    public void TearDown()
    {
        _ws.Clean();
    }
}
```

Let's Build the Fake Mechanism

Then the subclass, which looks like this:

```
public class CustomerRepositoryTestsFake :
        CustomerRepositoryTestsBase
{
    protected override IWorkspace _CreateWorkspace()
    {
        return new FakeWorkspace("");
    }
}
```

OK, that was the plumbing for the tests related to Repositories, but what about the tests themselves? Well, there are several different styles from which to choose, but the one I prefer is to define as much as possible in the base class, while at the same time making it possible to decide in the subclass if a certain test should be implemented or not at the moment. I also want it to be impossible to forget to implement a test in the subclass. With those requirements in place, my favorite style looks like this (first, how a simplified test looks in the base class):

```
[Test]
public virtual void CanAddCustomer()
{
    Customer c = new Customer();
    c.Name = "Volvo";
    c.Id = 42;
    _repository.Add(c);
    _ws.PersistAll();
    _ws.Clean();

    //Check
    Customer c2 = _repository.GetById(c.Id);
    Assert.AreEqual(c.Name, c2.Name);

    //Clean up
    _repository.Delete(c2);
    _ws.PersistAll();
}
```

Note that the second level of Identity Maps isn't cleared when `new FakeWorkspace("")` is done because the second level Identity Maps of the Fake are `static` and therefore not affected when the Fake *instance* is recreated. That's just how it is with a database, of course. Just because you open a new connection doesn't mean the `Customers` table is cleared.

So it's a good thing that the Fake works in this way, because then I will need to clean up after the tests with the Fake just as I will when I'm using the tests with a real database, *if* that's the approach I'm choosing for my database testing.

Of course, `IWorkspace` must have `Delete()` functionality, which I haven't discussed yet, otherwise it won't be possible to do the cleaning up. As a matter of fact, in all its simplicity the `Delete()` is quite interesting because it requires an Identity Map of its own for the Fake in the Unit of Work. Instances that have been registered for deletion will be held there until `PersistAll()`, when the deletion is done permanently.

To support this, `IWorkspace` will get a method like this:

```
void Delete(object o);
```

Unfortunately, it also introduces a new problem. What should happen to the relationships for the deleted object? That's not simple. Again, more metadata is needed to determine how far the delete should cascade—metadata that is around for the real infrastructure, but not useful here. (The convention currently used for the Fake is to not cascade.)

OK, back to the tests. In the subclass, I typically have one of three choices when it comes to the `CanAddCustomer()` test. The first alternative is to do nothing, in which case I'm going to run the test as it's defined in the base class. This is hopefully what I want.

The second option should be used if you aren't supporting a specific test for the specific subclass for the time being. Then it looks like this in the subclass:

```
[Test, Ignore("Not supported yet...")]
public override void CanAddCustomer() {}
```

This way, during test execution it will be clearly signaled that it's just a temporary ignore.

Finally, if you "never" plan to support the test in the subclass, you can write it like this in the subclass:

```
[Test]
public override void CanAddCustomer()
{
    Console.WriteLine
        ("CanAddCustomer() isn't supported by Fake.");
}
```

OK, you still *can* forget a test if you do your best. You'd have to write code like this, skipping the Test-attribute:

```
public override void CanAddCustomer() {}
```

There's a solution to this, but I find the style I have shown you to be a good balance of amount of code and the risk of "forgetting" tests.

I know, this wasn't YAGNI, because right now we don't have any implementation other than the Fake implementation, but see this just as a quick indicator for what will happen later on.

Let's Build the Fake Mechanism

Note

This style could just as well be used in the case of multiple implementations for each Repository.

For many applications, it might be close to impossible to deal with the whole system in this way, especially later on in the life cycle. For many applications, it's perhaps only useful for early development testing and early demos, but even so, if it helps with this, it's very nice. If it works all the way, it's even nicer.

In real life, the basics (there are always exceptions) are that I'm focusing on writing the core of the tests against the Fake implementation. I also write CRUD tests against the Fake implementation, but in those cases I also inherit to tests for using the database. That way, I test out the mapping details.

That said, no matter if you use something like the ideas of the abstraction layer or not, you will sooner or later run into the problems of database testing. I asked my friend Philip Nelson to write a section about it. Here goes.

◆ Database Testing
By Philip Nelson

At some point when you are working with DDD and all the various flavors of automated testing, you will probably run into a desire to run a test that includes access to a database. When you first encounter this, it seems like no problem at all. Create a test database, point your application at the test database, and start testing.

Let's focus more on automated testing, so let's assume you wrote your first test with JUnit or NUnit. This test just loaded a `User` domain object from the database and verified that all the properties were set correctly. You can run this test over and over again and it works every time. Now you write a test to verify that you can update the fields, and because you know you have a particular object in the database, you update that.

The test works, but you have introduced a problem. The read test no longer works because you have just changed the underlying data, and it no longer matches the expectations of your first test. OK, that's no problem; these tests shouldn't share the same data, so you create a different `User` for the update test. Now both tests are independent of each other, and this is important: at all times you need to ensure that there are no residual effects between tests. Database-backed tests always have preconditions that you must meet. Somewhere in the process of writing your tests, you must account for the fact that you have to reset the database to meet the preconditions the test expects. More details on that to come.

At some point, you will have written tests for all the basic `User` object life cycle states, let's say *create, read, update,* and *delete* (CRUD). In your application, there may very well be other database-backed operations. For example, the `User` object may help enforce a policy of maximum failed login attempts. As you test the updating of the failed login attempts and last login time fields, you realize that your update actually isn't working. The update test didn't catch the problem because the database already had the field set to the expected value. As the smart person you are, you figured this out very quickly. However, young Joe down the hall has just spent four hours working on the problem. His coding skills aren't so much the problem as much as his understanding of how the code connects to the database and how to isolate the data in this test from all the other test data he now has. He just doesn't notice that the `LastUpdateDate` field is not reset between tests. After all, the code is designed to hide the database as an implementation detail, right?

Database
Testing

At this point, you start to realize that just having separate test data for each test is going to be more complicated than you want it to be. You may or may not have understood as clearly as necessary just how important it was to reset the data between tests, but now you do. Fortunately, your xUnit test framework has just the thing for you. There is a setup and teardown code block that is just made for this sort of thing. But like most things in programming, there is more than one way to do it, and each has its strengths and weaknesses. You must understand the tradeoffs and determine the balance that suits you best.

I would categorize the techniques available to you in four ways:

- Reset the database before each test.

- Maintain the state of the database during the run.

- Reset the data for just the test or set of tests you are running before the run.

- Separate the testing of the unit from the testing of the call to the database.

Reset the Database Before Each Test

Database Testing

At first glance, this might seem the most desirable, but possibly the most time-consuming, option. The plus to this approach is that the whole system is in a known state at the start of the test. You don't have to worry about strange interactions during test runs because they all start at the same place. The down-side is time. After you have gotten past the initial parts of your project, you will find yourself waiting while your test suite runs. If you are doing unit testing and running your tests after each change, this can very quickly become a significant part of your time. There are some options that can help. How they apply to you will depend on many things, from the type of architecture you use to the type of database system you use.

One simple but slow approach is to restore the database from backup before each test. With many database systems, this is not practical because of the amount of time it takes. However, there are some systems where it is possible. If you are programming against a file-based database, for example, you may only need to close your connections and copy a file to get back to the original state. Another possibility is an in-memory database, such as HSQL for Java, that can be restored very quickly. Even if you are using a more standard system like Oracle or SQL Server, if your design for data access is flexible enough, you may be able to switch to an in-memory or file-based database for your tests. This is especially true if you are using an O/R Mapper that takes the responsibility of building the actual SQL calls for you and knows multiple dialects.

The project DbUnit offers another way to reset a database before the test runs. Essentially, it's a framework for JUnit that allows you to define "data sets" that are loaded on clean tables before each test run. Ruby on Rails has a similar system that allows you to describe your test class data in the open YAML (YAML Ain't Markup Language) format and apply it during test setup. These tools can work well, but you may start running into problems as these database inserts may be logged operations that can be too slow. Another approach that could work for you is to use bulk load utilities that may not be as extensively logged. Here is how this might work for SQL Server. First, use a database recovery option on your test database of Simple Recovery. This eliminates most logging and improves performance. Then, during test design, do the following:

- Insert data into a test database that will only act as a source of clean data for your tests

- Call transact sql commands that export the test data to files

- Write transact sql commands to do bulk insert of the data in these files

Then during the test setup method, do the following:

- Truncate all your tables. This is not a logged operation and can be very fast.

- Issue the bulk copy commands created earlier to load the test data into the test database.

A variation of this technique is to set up your test database and load the initial test data using normal data management techniques. Then, provided your database supports such an operation, detach the underlying data files from the server. Make a copy of these files as these will be the source of your clean test data. Then, write code that allows you to

- Detach the server from its data files

- Copy the clean test data over the actual server files

- Attach the database to the copy

In many cases, this operation is very fast and can be run in the test fixture setup or, if the amount of data isn't too large, in the test setup itself.

Yet another variation supported by some of the O/R Mapping tools is to build the database schema from mapping information. Then you can use your Domain Model in test setup to populate data as needed for each test suite or possibly for each test fixture.

Maintain the State of the Database During the Run

You are probably thinking "But that is a data management exercise!", and you are correct. There is another technique that is even simpler to execute. Essentially, what you do is to run each test in a transaction and then roll back the transaction at the end of the test. This could be a little challenging because transactions are not often exposed on public interfaces, but again, this all depends on your architecture. For MS .NET environments, Roy Osherove came up with a very simple solution that draws on ADO.NET's support for COM+ transaction enlistment. What this tool does is allow you to put a `[Rollback]` attribute on specific test methods, allowing those methods to be run in their own COM+ transaction. When the test method finishes, the transaction is then rolled back automatically for you with no code on your part and independent of the teardown functionality of the class.

What's really great about this technique is that your tests can be blissfully unaware of what's happening underneath them. This functionality has been packaged up in a project called XtUnit [XtUnit] and is an extension to the NUnit testing framework. This is by far the simplest approach to keeping your database in pristine shape. It does come with a price, though. Transactions are by their nature logged, which increases the execution time. COM+ transactions use the Distributed Transaction Coordinator, and distributed transactions are slower than local transactions. The combination can be brutal to the execution speed of your test suite.

Naturally, if you are testing transaction semantics themselves, you may have some additional problems with this technique. So depending on the specifics of your project, this can be a really great solution or a solution in need of a replacement as the number of tests grows. Fortunately, you would not necessarily have to rewrite much code should test speed become a problem because the solution is transparent to the test. You will either be able to live with it, or you will have to adopt other techniques as your project grows.

Reset the Data Used by a Test Before the Test

This approach relieves you and your system of having to reset the entire database before the test call. Instead, you issue a set of commands to the database before or after the test, again typically in the setup and/or teardown methods. The good news is that you have much less to do for a single class of tests. The previous techniques can still be used, but now you have the additional option of issuing simple insert, update, or delete statements as well. This is now less painful, even if logged, because the impact is less. Again, reducing the logging effort with database setup options is still a good idea if your system supports it.

Database Testing

There is a downside, though. As soon as you make the leap to assuming you know exactly what data will be affected by your test, you have tied yourself into understanding what not just the code under test is doing with data, but also what additional code the code under test calls out to. It may get changed without you noticing it. Tests may break that are not an indication of broken code. A bug will get filed that may not be a bug at all but is maintenance work just the same.

It is possible to use this technique successfully, but after many years of test writing, I have found this approach to be the most likely to break without good reason. You can minimize this, though. If you mock out the classes your code under test calls out to, that code won't affect the database. This is in spirit much closer to what is meant by a unit test. Mocking is not useful if you are automating system tests where the interactions between real classes are exactly what you want to test. At any rate, doing this requires an architecture decision to allow for easy substitution of external classes by the testing framework. If you are going to do that, you have naturally found yourself working toward the final option I am covering here.

Don't Forget Your Evolving Schema!

You will no doubt find yourself making changes to the database over time. While in the initial stages you may want to just modify the database directly and adjust your code as needed, after you have a released system, it's time to think about how these changes should become part of your process. Working with a test system actually makes this easier.

I prefer the approach of creating and running alter scripts against your database. You could run these scripts immediately after your database reset, but because all that database state is contained in your source controlled development environment, it probably makes sense to develop the scripts and use them to modify your test environment. When your tests pass, you check it all in and then have the scripts automatically set up to run against your QA environment as needed.

It's even better if this happens on your build server because the running of the alter scripts is then tested often before it's applied to your live systems. Of course, that assumes you regularly reset your QA environment to match your live environment. I'm sure there are many variations of this process, but the most important thing is to plan ahead to ensure a reliable release of both code and database changes.

Database Testing

Separate the Testing of the Unit from the Testing of the Call to the Database

Few things in the Agile community's mailing lists and forums generate more discussion than the testing of code that interacts with databases. Among those who are practicing (and still defining) the techniques of TDD, it is a highly held value that all code can be tested independently as units as they are written. In the spirit of TDD, it wouldn't make sense to write a test for a class and start out your implementation with a direct call to a database. Instead, the call is delegated out to a different object that abstracts the database, which is replaced with a stub or a mock as you write the test. Then you write the code that actually implements the database abstraction, and test that *it* calls the underlying data access objects correctly. Finally, you would write a few tests that test that your infrastructure that connects real database calls to your code works correctly. As the old saying goes, most problems in coding can be solved with another layer of indirection. As always, the devil's in the details.

First, let's clarify a few definitions. A stub is a replacement piece of code that does just enough to not cause problems for the caller and no more. A stub for a database call, such as a call to a JDBC `Statement`, would be accomplished by having a bare-bones class that implements the `Statement`'s interface and simply returns a `ResultSet` when called, possibly ignoring the parameters passed and certainly not executing a database call.

A mock `Statement` would do all that and also allow the setting of expectations. I'll say more on that in a moment. Like a stub, the test would use the mock command instead of the real command, but when the test was complete, it would "ask" the mock command to "verify" that it was called correctly. The *expectations* for a mock `Statement` would be values for any parameters the call needed, the number of times that `executeQuery` was called, the correct SQL to be passed to the statement and so on. In other words, you tell the mock what to expect. Then you ask it to *verify* that it did receive these expectations after the test is complete.

When you think about it, I think you have to agree that a unit test for a `User` class should not have to concern itself with what the database does. So long as the class interacts with the database abstraction correctly, we can assume the database will do its part correctly or at least that tests to your data access code will verify that fact for you. You just have to verify that the code correctly passes all the fields to the database access code, and you have done all you need to do. If only it could always be that simple!

To take this to its logical conclusion, you might end up writing mock implementations of many of the classes in your code. You will have to populate the members and properties of those mocks with at least enough data to satisfy

the requirements of your tests. That's a lot of code, and an argument could be made that just having to support that much code will make your code harder to maintain. Consider this alternative. There are some new frameworks available in a variety of languages that allow you to create mock objects from the definitions of real classes or interfaces. These dynamically created mocks allow you to set expectations, set expected return values and do verification without writing much code to mock the class itself. Known collectively as Dynamic Mocks, the technique allows you to simply pass in the class to mock to a framework and get back an object that will implement its interface.

There are many other sources of information on how to mock code, but much less on how to effectively mock database access code. Data access code tends to have all these moving parts to deal with: connections, command objects, transactions, results, and parameters. The data access libraries themselves have driven the creation of a wide variety of data access helpers whose aim it is to simplify this code. It seems that none of these tools, the helpers, or the underlying data access libraries like JDBC or ADO.NET were written with testing in mind. While many of these tools offer abstractions on the data access, it turns out to be fairly tricky work to mock all those moving parts. There is also that issue of having to test the mapping of all those fields between the data access code and rest of your classes. So here are some pieces of advice to help you through it.

Test everything you can without actually hitting your database access code. The data access code should do as little as you can get away with. In most cases, this should be CRUD. If you are able to test all the functionality of your classes without hitting the database, you can write database tests that only exercise these simpler CRUD calls. With the aid of helper methods you may be able to verify that the before and after set of fields match your expectations by using reflection to compare the objects rather than hand coding all the property tests. This can be especially helpful if you use an O/R Mapper, such as Hibernate, where the data access is very nicely hidden, but the mapping file itself needs to be tested. If all the other functionality of the class is verified without the database hit, you often only need to verify that the class's CRUD methods and the mapping are working correctly.

Test that you have called the data access code correctly separately from testing the database code itself. For example, if you have a `User` class that saves via a `UserRepository`, or perhaps a Data Access Layer in nTier terminology, all you need to test is that the `UserRepository` is called correctly by your upper-level classes. Tests for the `UserRepository` would test the CRUD functionality with the database.

To test certain types of data access, these simple CRUD tests may not be adequate. For example, calls to stored procedures that aren't directly related

to class persistence fall in this category. At some point, you may need to test that these procedures are called correctly, or in your test you may need data back from one of these calls for the rest of your test. Here are some general techniques to consider in those cases where you really are going to mock JDBC, ADO.NET, or some other data access library directly.

You must use factories to create the data access objects and program against interfaces rather than concrete types. Frameworks may provide factories for you, as the Microsoft Data Access Application Block does in its more recent versions. However, you also need to be able to use these factories to create mock implementations of the data access classes, something not supported out of the box by many factory frameworks. If you have a factory that can be configured by your test code to provide mocks, you can substitute a mock version of these data access classes. Then you can verify virtually any type of database call. You still may need mock implementations of these classes, though. For ADO.NET, these can be obtained from a project called .NET Mock Objects [MockObjects]. Versions for other environments may also be available. The only combination of a factory-based framework that can work with mocks directly that I am aware of is the SnapDAL framework [SnapDAL] that builds on .NET Mock Objects to supply mock implementations of the ADO.NET generic classes and was built to fully support mock objects for the ADO.NET generic interfaces.

Whether you can use these frameworks or not depends on many factors, but one way or another, you will need your application to support a factory that can return a real or a mock of your data access classes. When you get to a position where you can create mock instances of your data access classes, you can now use some or all of the following techniques to test:

- The mock returns result sets, acting as a stub for your code. The result set gets its data from the test code in standard mock object style.

- The mock can return test data from files, such as XML files or spreadsheets of acceptance test data provided by your business units.

- The mock can return test data from an alternate database or a more refined query than the real query under test.

- If you can substitute a data access command, you can "record" the data access and later "play it back" from a mock object in your tests because you would have access to all the parameters and the returned results.

- Mocks can have expectations verified of important things like the connection being opened and closed, transactions committed or rolled back, exceptions generated and caught, and data readers read, and so on.

Database Testing

I hope these ideas can get you started with an appreciation for the many possibilities there are for testing database-connected code. The text was written basically in order of complexity, with database resetting or the rollback techniques being the easiest to implement. Fully separating out the data access code from the rest of your code will always be a good move and will improve your testing success and test speed. At the most fine-grained level, allowing for mock versions of your actual database calls offers great flexibility in how you gather, provide, and organize test data. If you are working with legacy code that uses the lower-level data access libraries of your language, you would probably move toward this sort of approach. If you are starting a new application, you can probably start with the idea of simply resetting your database between tests and enhance your test structure as test speed begins to affect your productivity.

◆

Thanks, Phil! Now we are armed to deal with the problem of database testing no matter what approach we choose.

Note

Neeraj Gupta commented on this by saying, "You can use the Flashback feature of Oracle database to bring the database back to the previous state for the testing."

We have come quite a long way, but there is one very big piece of the preparing for infrastructure puzzle that we haven't dealt with at all. How did I solve `_GetNumberOfStoredCustomers`?

What I'm missing is obviously querying!

Querying

Querying is extremely different in different infrastructure solutions. It's also something that greatly risks affecting the consumer. It might not be apparent at first when you start out simply, but after a while you'll often find quite a lot of querying requirements, and while you are fulfilling those, you're normally tying yourself to the chosen infrastructure.

Let's take a step back and take another solution that is not query-based. Earlier, I showed you how to fetch by identity in a Repository with the following code:

```
//OrderRepository
public Order GetOrder(int orderNumber)
```

```
{
    return (Order)_ws.GetById(typeof(Order), orderNumber);
}
```

However, if `OrderNumber` isn't an identity, the interface of the Repository method must clearly change to return a list instead, because several orders can have ordernumber 0 before they have reached a certain state. But then what? `GetById()` is useless now, because `OrderNumber` isn't an Identity (and let's assume it's not unique because I said the answer could be a list). I need a way to get to the second layer of Identity Maps of the Fake for the `Orders`. Let's assume I could do that with a `GetAll(Type typeOfResult)` like this:

```
//OrderRepository
public IList GetOrders(int orderNumber)
{
    IList result = new ArrayList();
    IList allOrders = _ws.GetAll(typeof(Order));

    foreach (Order o in allOrders)
    {
        if (o.OrderNumber == orderNumber)
            result.Add(o);
    }

    return result;
}
```

It's still pretty silly code, and it's definitely not what you want to write when you have infrastructure in place, at least not for real-world applications.

Single-Set of Query Objects

Just as with the Repositories, it would be nice to write the queries "correctly" from day one in a single implementation which could be used both with the Fake and with the real infrastructure. How can we deal with that?

Let's assume that we want to work with Query Objects [Fowler PoEAA] (encapsulate queries as objects, providing object-oriented syntax for working with the queries) and we also change the signature from `GetAll()` to call it `GetByQuery()` and to let it take an `IQuery` (as defined in NWorkspace) instead. The code could now look like this:

```
//OrderRepository
public IList GetOrders(int orderNumber)
{
    IQuery q = new Query(typeof(Order));
    q.AddCriterion("OrderNumber", orderNumber);
    return _ws.GetByQuery(q);
}
```

OK, that was pretty straightforward. You just create an `IQuery` instance by saying which type you expect in the result. Then you set the criteria you want for holding down the size of the result set as much as possible, typically by processing the query in the database (or in the Fake code, in the case of when you're using the Fake implementation).

Note

We could pretty easily make the query interface more fluent, but let's stay with the most basic we can come up with for now.

That was what to do when you want to instantiate part of the Domain Model. Let's get back to the `_GetNumberOfStoredCustomers()` that we talked about earlier. How could that code look with our newly added querying tool? Let's assume it calls the Repository and a method like the following:

```
//CustomerRepository
public int GetNumberOfStoredCustomers()
{
     return _ws.GetByQuery(new Query(typeof(Customer))).Count;
}
```

It works, but for production scenarios that solution would reduce every DBA to tears. At least it will if the result is a SELECT that fetches all rows just so you can count how many rows there are. It's just not acceptable for most situations.

We need to add some more capabilities in the querying API. Here's an example where we have the possibility of returning simple types, such as an `int`, combined with an aggregate query (and this time aggregate isn't referring to the DDD pattern Aggregate, but, for example, to a SUM or AVG query in SQL):

```
//CustomerRepository
public int GetNumberOfStoredCustomers()
{
    IQuery q = new Query(typeof(Customer),
        new ResultField("CustomerNumber", Aggregate.Count));
    return (int)_ws.GetByQuery(q)[0];
}
```

A bit raw and immature, but I think that this should give you an idea of how the basic querying API in NWorkspace is designed.

It would be nice to have a standard querying language, wouldn't it? Perhaps the absence of one was what made Object Databases not really take off. Sure, there was Object Query Language (OQL), but I think it came in pretty late, and it was also a pretty complex standard to implement. It's competent, but

complex. (Well, it was probably a combination of things that hindered Object Databases from becoming mainstream; isn't it always?) I'll talk more about Object Databases in Chapter 8, "Infrastructure for Persistence."

What I now want, though, is a querying standard for persistence frameworks—something as widespread as SQL, but for Domain Models. Until we have such a standard, the NWorkspace version could bridge the gap, for me at least. Is there a cost for that? Is there such a thing as a free lunch?

The Cost for Single-Set of Query Objects

Of course there's a cost for a transformation, and that goes for this case, too. First, there's a cost in performance for going from an `IWorkspace` implementation to the infrastructure solution. However, the performance cost will probably be pretty low in the context of end-to-end scenarios.

Then there's the cost of loss of power because the NWorkspace-API is simplified, and competent infrastructure solutions have more to offer and that is probably much worse. Yet another cost is that the API of NWorkspace itself is pretty raw and perhaps not as nice as the querying API of your infrastructure solution. OK, all those costs sound fair, and if there's a lot to be gained, I can live with them.

I left out one very important detail before about querying and the Identity Map: bypassing the cache when querying or not.

Querying and the Cache

I mentioned earlier that when you do a `GetById()`, if that operation must be fulfilled by going to persistence, the fetched instance will be added to the Identity Map before being returned to the consumer. That goes for `GetByQuery()` as well; that is, the instances will be added to the Identity Map.

However, there's a big difference in that the `GetByQuery()` won't investigate the Identity Map before hitting persistence. The reason is partly that we want to use the power of the backend, but above all that we don't know if we have all the necessary information in the Identity Map (or cache if you will) for fulfilling the query. To find out if we have that, we need to hit the database. This brings us to another problem. `GetById()` starts with the Identity Map; `GetByQuery()` does not. This is totally different behavior, and it actually means that `GetByQuery()` bypasses the cache, which is problematic. If you ask for the new `Customer` Volvo that has been added to the Identity Map/Unit of Work, but has not been persisted, it will be found if you ask by ID but not when you ask by name with a query. Weird.

To tell you the truth, it was a painful decision, but I decided to let `GetByQuery()` do an implicit `PersistAll()` by *default* before going to the database. (There's an override to `GetByQuery()` to avoid this, but again, it's the default to implicitly call `PersistAll()`.) I came to the conclusion that this default style is probably most in line with the rest of NWorkspace and its goal of simplicity and the lessened risk of errors. This is why I made some sacrifices with transactional semantics. Some might argue that this violates the principle of least amount of surprise. But I think it depends on your background, what is surprising in this case.

The biggest drawback is definitely that when doing `GetByQuery()`, you might get save-related exceptions. What a painful decision—but I need to decide something for now to move on.

Do you remember the simple and unoptimized version of the `_GetNumberOf-StoredCustomers()`? It's not just slow—it might not work as expected when it looks like this (which goes for the optimized version as well:

```
public int GetNumberOfStoredCustomers()
{
    return _ws.GetByQuery(new Query(typeof(Customer))).Count;
}
```

The reason it won't work for my purpose is that `GetByQuery()` will do that implicit `PersistAll()`. Instead, an overload must be used like this, where `false` is for the `implicitPersistAll` parameter:

```
public int GetNumberOfStoredCustomers()
{
    return _ws.GetByQuery(new Query(typeof(Customer)
        , false)).Count;
}
```

Querying

Note

And of course we could (and should) use the aggregate version instead. The focus here was how to deal with implicit `PersistAll()`.

All this affects the programming model to a certain degree. First of all, you should definitely try to adopt a style of working in blocks when it comes to querying and changing for the same workspace instance. So when you have made changes, you should be happy about them before querying because querying will make the changes persistent.

Note

You might be right. I might be responsible for fostering a sloppy style of consumer programmers. They just code and it works, even though they forget about saving explicitly and so on.

An unexpected and unwanted side effect is that you can get totally different exceptions from `GetByQuery()` from what you expect because the exception might really come from `PersistAll()`. Therefore, it's *definitely* a good idea to do the `PersistAll()` explicitly in the consumer code anyway.

And again, if you hate this behavior, there's nothing to stop you from using the overload. (Actually, with `GetById()` you could do it the other way around, so that an overload goes to the database regardless, without checking the Identity Map.) "*I don't care about my own transient work; I want to know what's in the database.*"

That was a bit about how querying works in relation to the Identity Map. Next up is where to host the queries.

Where to Locate Queries

As I see it, we have at least the following three places in which to host query instances:

- In Repositories

- In consumers of Repositories

- In the Domain Model

Let's discuss them one by one.

In Repositories

Probably the most common place to set up queries is in the Repositories. Then the queries become the tool for fulfilling method requests, such as `GetCustomersByName()` and `GetUndeliveredOrders()`. That is, the consumer might send parameters to the methods, but those are just ordinary types and not Query Objects. The Query Objects are then set up in accordance with the method and possible parameter values.

In Consumers of Repositories

In the second case, the queries are set up in the consumers of the Repositories and sent to the Repositories as parameters. This is typically used in cases of

highly flexible queries, such as when the user can choose to fill in any fields in a large filtering form. One such typical method on a Repository could be named GetCustomersByFilter(), and it takes an IQuery as parameter.

In the Domain Model

Finally, it might be interesting to set up typed Query Objects in the Domain Model (still queries that implement IQuery of NWorkspace). The consumer still gets the power of queries to be used for sending to Repositories, for example, but with a highly intuitive and typesafe API. How the API looks is, of course, totally up to the Domain Model developer.

Instead of the following simple typeless query:

```
//Consumer code
IQuery q = new Query(typeof(Customer));
q.AddCriterion("Name",
"Volvo");
```

the consumer could set up the same query with the following code:

```
//Consumer code
CustomerQuery q = new CustomerQuery();
q.Name.Eq("Volvo");
```

In addition to getting simpler consumer code, this also further encapsulates the Domain Model.

It's also about lessening the flexibility, and that is very good. Don't make everything possible.

Assuming you like the idea of Domain Model-hosted queries, which queries do we need and how many?

Querying

Aggregates as a Tool Again

That last question may have sounded quite open, but I actually have an opinion on that. Guess what? Yes, I think we should use Aggregates.

Each of your Aggregates is a typical candidate for having Query Object in the Domain Model. Of course, you could also go the XP route of creating them when needed for the first time, which is probably better.

Speaking of Aggregates, I'd like to point out again that I see Aggregates as the *default* mechanism for determining how big the loadgraphs should be.

Note

With loadgraph, I mean how far from the target instance we should load instances. For example, when we ask for a certain order, should we also load its customer or not? What about its orderLines?

And when the default isn't good enough performance-wise, we have lots of room for performance optimizations. A typical example is to not load complete graphs when you need to list instances, such as Orders. Instead, you create a cut down type, perhaps called OrderSnapshot, with just the fields you need in your typical list situations. It's also the case that those lists won't be integrated with the Unit of Work and Identity Map, which probably is exactly what you want, again because of performance reasons (or it might create problems—as always, it depends).

An abstraction layer could support creating such lists so that your code will be agnostic about the implementation of the abstraction layer at work. It could look like this:

```
//For example OrderRepository.GetSnapshots()
IQuery q = new Query(typeof(Order), typeof(OrderSnapshot)
    , new string[]{"Id", "OrderNumber", "Customer.Id"
    , "Customer.Name"});
q.AddCriterion("Year",
year);
result _ws.GetByQuery(q);
```

For this to work, OrderSnapshot must have suitable constructor, like this (assuming here that the Ids are implemented as Guids):

```
//OrderSnapshot
public OrderSnapshot(Guid id, int orderNumber
    , Guid customerId, string customerName)
```

Note

Getting type explosion is commonly referred to as the Achilles heel of O/R Mappers, which I provided an example of earlier. In my experience, the problem is there, but not anything we can't deal with. First, you do this only when you really have to, so not all Aggregate roots in your model will have a snapshot class. Second, you can often get away with a single snapshot class for several different list scenarios. The optimization effect of doing it at all is often significant enough that you don't need to go further.

Another common approach is to use Lazy Load for tuning by loading some of the data just in time. (We'll talk more about this in a later chapter.)

And if that isn't powerful enough, you can write manual SQL code and instantiate the snapshot type on your own. Just be very clear that you are starting to actively use the database model as well at that point in time.

One or Two Repository Assemblies?

With all this in place, you understand that you have many different possibilities for how to structure the Repositories, but you might wonder how it's done in real-world projects.

In the most recent large project of mine (which is in production—it's not a toy project), I use a single set of Repositories, both for Fake execution and execution against the database. There are a few optimizations that use native SQL, but I used a little hack there so that if the optimized method finds an injected connection string, it calls out to another method where I'm totally on my own.

Otherwise, the un-optimized code will be used instead. That way, the Fake will use the un-optimized version, and the database-related code will typically use the optimized version.

Again, this is only used in a handful of cases. Not extremely nice and clean, but it works fine for now.

Querying

Specifications as Queries

Yet another approach for querying is to use the Specification pattern [Evans DDD] (encapsulate conceptual specifications for describing something, such as what a customer that isn't allowed to buy more "looks like"). The concept gets a describing name and can be used again and again.

Using this approach is similar to creating type safe Domain Model-specific queries, but it goes a step further because it's not generic querying that we are after, but very specific querying, based on domain-specific concepts. This is one level higher than the query definitions that live in the Domain Model, such as `CustomerQuery`. Instead of exposing generic properties to filter by, a specification is a concept with a very specific domain meaning. A common example is the concept of a gold customer. The specification describes what is needed for a customer to be a gold customer.

The code gets very clear and purposeful because the concepts are caught and described as Specification classes.

Even those Specification classes could very well spit out `IQuery` so that you can use them with `GetByQuery()` of `IWorkspace` (or equivalent) if you like the idea of an abstraction layer.

Other Querying Alternatives

So far we have just talked about Query Objects as being the query language we need (sometimes embedded in other constructs, though). As a matter of fact, when the queries get complex, it's often more powerful to be able to write the queries in, for example, a string-based language similar to SQL.

I haven't thought about any more query languages for NWorkspace. But perhaps I can someday talk some people into adding support for NWorkspace queries as output from their queries. That way, their queries (such as something like SQL, but for Domain Model) would be useful against the infrastructure solutions that have adapter implementations for NWorkspace.

I can't let go of the idea that what querying language you want to use is as much of a lifestyle choice as is the choice of programming language. Of course, it might depend even more on the situation as to which query language we want to use, if we can choose. That's a nice way to end this subject for now, I think (more to come in Chapter 8 and 9). It's time to move on.

Summary

So we have discussed what Persistence Ignorance is and some ideas of how to keep the infrastructure out of the Domain Model classes, while at the same time prepare for infrastructure. We also discussed database testing, and a lot of attention was spent on an idea of introducing an abstraction layer.

In the next chapter, we will change focus to the core of the Domain Model again, this time focusing on rules. You will find that lots of the requirements in the list we set up in Chapter 4 are about rules, so this is a near and dear topic when Domain Models are discussed.