

Patterns in Practice

The Unit Of Work Pattern And Persistence Ignorance

Jeremy Miller

Contents

[The Unit Of Work Pattern](#)
[Using the Unit of Work](#)
[Persistence Ignorance](#)
[Can the Business Logic Run Independently of the Database?](#)
[Can I Design My Domain Model Independently from the Database Model?](#)
[How Does My Persistence Strategy Affect My Business Logic?](#)
[More Units of Work](#)

In the April 2009 issue of *MSDN Magazine* ("[Persistence Patterns](#)") I presented some common patterns that you will encounter when using some sort of Object/Relational Mapping (O/RM) technology to persist business entity objects. I think it's unlikely that you or your team will be writing your own O/RM tooling from scratch, but these patterns are important to know to effectively use (or even just to choose) existing tooling.

In this article, I would like to continue the discussion of persistence patterns with the Unit of Work design pattern and examine the issues around persistence ignorance. Throughout most of this article, I will be using a generic invoicing system as the example problem domain.

The Unit of Work Pattern

One of the most common design patterns in enterprise software development is the Unit of Work. According to Martin Fowler, the Unit of Work pattern "maintains a list of objects affected by a business transaction and coordinates the writing out of changes and the resolution of concurrency problems."

The Unit of Work pattern isn't necessarily something that you will explicitly build yourself, but the pattern shows up in almost every persistence tool that I'm aware of. The *ITransaction* interface in NHibernate, the *DataContext* class in LINQ to SQL, and the *ObjectContext* class in the Entity Framework are all examples of a Unit of Work. For that matter, the venerable *DataSet* can be used as a Unit of Work.

Other times, you may want to write your own application-specific Unit of Work interface or class that wraps the inner Unit of Work from your persistence tool. You may do this for a number of reasons. You might want to add application-specific logging, tracing, or error handling to transaction management. Perhaps you want to encapsulate the specifics of your persistence tooling from the rest of the application. You might want this extra encapsulation to make it easier to swap out persistence technologies later. Or you might want to promote testability in your system. Many of the built-in Unit of Work implementations from common persistence tools are difficult to deal with in automated unit testing scenarios.

If you were to build your own Unit of Work implementation, it would probably look something like this interface:

```
public interface IUnitOfWork {
    void MarkDirty(object entity);

    void MarkNew(object entity);
    void MarkDeleted(object entity);
    void Commit();
    void Rollback();
}
```

Your Unit of Work class will have methods to mark entities as changed, new, or deleted. (In many implementations the explicit call to *MarkDirty* would be unnecessary because the Unit of Work itself has some way of automatically determining which entities have been changed.) The Unit of Work will also have methods to commit or roll back all of the changes.

In a way, you can think of the Unit of Work as a place to dump all transaction-handling code. The responsibilities of the Unit of Work are to:

- Manage transactions.
- Order the database inserts, deletes, and updates.
- Prevent duplicate updates. Inside a single usage of a Unit of Work object, different parts of the code may mark the same *Invoice* object as changed, but the Unit of Work class will only issue a single *UPDATE* command to the database.

The value of using a Unit of Work pattern is to free the rest of your code from these concerns so that you can otherwise concentrate on business logic.

Using the Unit of Work
MSDN Magazine Blog**MSDN Magazine June Issue Preview**

Monday morning the June issue of MSDN Magazine will go live on our Web site. Here's what you can expect to find in the magazine. Windows 8 figures pr... [More...](#)

Friday, May 31

MSDN Magazine May Issue Preview

Tomorrow afternoon the May issue of MSDN Magazine will go live on our Web site. Here's what you can look forward to in the upcoming issue. Craig Shoe... [More...](#)

Tuesday, Apr 30

[More MSDN Magazine Blog entries >](#)**Current Issue**[Browse All MSDN Magazines](#)[Subscribe to MSDN Flash newsletter](#)

Receive the MSDN Flash e-mail newsletter every other week, with news and information

One of the best ways to use the Unit of Work pattern is to allow disparate classes and services to take part in a single logical transaction. The key point here is that you want the disparate classes and services to remain ignorant of each other while being able to enlist in a single transaction. Traditionally, you've been able to do this by using transaction coordinators like MTS/COM+ or the newer System.Transactions namespace. Personally, I prefer using the Unit of Work pattern to allow unrelated classes and services to take part in a logical transaction because I think it makes the code more explicit, easier to understand, and simpler to unit test.

Let's say that your new invoicing system performs discrete actions upon existing invoices at various times in the invoice lifecycle. The business changes these actions fairly often and you'll frequently want to add or remove new invoice actions, so let's apply the Command pattern (see ["Simplify Distributed System Design Using the Command Pattern, MSMQ, and .NET"](#)) and create an interface called `IInvoiceCommand` that represents a single discrete action against an `Invoice`:

```
public interface IInvoiceCommand {
    void Execute(Invoice invoice, IUnitOfWork unitOfWork);
}
```

The `IInvoiceCommand` interface has one simple `Execute` method that is called to perform some type of action using an `Invoice` and an `IUnitOfWork` object. Any `IInvoiceCommand` object should use the `IUnitOfWork` argument to persist any changes back to the database within that logical transaction.

Simple enough, but this Command pattern plus Unit of Work pattern doesn't get interesting until you put several `IInvoiceCommand` objects together (see **Figure 1**).

Figure 1 Using IInvoiceCommand

```
public class InvoiceCommandProcessor {
    private readonly IInvoiceCommand[] _commands;
    private readonly IUnitOfWorkFactory _unitOfWorkFactory;

    public InvoiceCommandProcessor(IInvoiceCommand[] commands,
        IUnitOfWorkFactory unitOfWorkFactory) {
        _commands = commands;
        _unitOfWorkFactory = unitOfWorkFactory;
    }

    public void RunCommands(Invoice invoice) {
        IUnitOfWork unitOfWork = _unitOfWorkFactory.StartNew();

        try {
            // Each command will potentially add new objects
            // to the Unit of Work for insert, update, or delete
            foreach (IInvoiceCommand command in _commands) {
                command.Execute(invoice, unitOfWork);
            }

            unitOfWork.Commit();
        }
        catch (Exception) {
            unitOfWork.Rollback();
        }
    }
}
```

Using this approach with the Unit of Work, you can happily mix and match different implementations of the `IInvoiceCommand` to add or remove business rules in the invoicing system while still maintaining transactional integrity.

In my experience, business people seem to care quite a bit about late, unpaid invoices, so you're probably going to have to build a new `IInvoiceCommand` class that will alert the company's agents when an `Invoice` is determined to be late. Here is a possible implementation of this rule:

```
public class LateInvoiceAlertCommand : IInvoiceCommand {
    public void Execute(Invoice invoice, IUnitOfWork unitOfWork) {
        bool isLate = isTheInvoiceLate(invoice);
        if (!isLate) return;

        AgentAlert alert = createLateAlertFor(invoice);
        unitOfWork.MarkNew(alert);
    }
}
```

The beauty of this design to me is that the `LateInvoiceAlertCommand` can be completely developed and tested independently of the database or even the other `IInvoiceCommand` objects involved in the same transaction. First, to test the interaction of the `InvoiceCommand` objects with a Unit of Work, I might create a fake implementation of `IUnitOfWork` strictly for testing to be precise, I would call the `StubUnitOfWork` a recording stub:

```
public class StubUnitOfWork : IUnitOfWork {
```

```

public bool WasCommitted;
public bool WasRolledback;
public void MarkDirty(object entity) {
    throw new System.NotImplementedException();
}
public ArrayList NewObjects = new ArrayList();
public void MarkNew(object entity) {

    NewObjects.Add(entity);
}
}

```

Now that you've got a nice fake Unit of Work that runs independently of the database, the test fixture for the `LateInvoiceAlertCommand` might look something like the code in **Figure 2**.

Figure 2 Test Fixture for `LateInvoiceAlertCommand`

```

[TestFixture]
public class
    when_creating_an_alert_for_an_invoice_that_is_more_than_45_days_old {

    private StubUnitOfWork theUnitOfWork;
    private Invoice theLateInvoice;

    [SetUp]
    public void Setup() {
        // We're going to test against a "Fake" IUnitOfWork that
        // just records what is done to it
        theUnitOfWork = new StubUnitOfWork();

        // If we have an Invoice that is older than 45 days and NOT completed
        theLateInvoice = new Invoice {InvoiceDate =
            DateTime.Today.AddDays(-50), Completed = false};

        // Exercise the LateInvoiceAlertCommand against the test Invoice
        new LateInvoiceAlertCommand().Execute(theLateInvoice, theUnitOfWork);
    }

    [Test]
    public void
        the_command_should_create_a_new_AgentAlert_with_the_UnitOfWork() {

        // just verify that there is a new AgentAlert object
        // registered with the Unit of Work
        theUnitOfWork.NewObjects[0].ShouldBeOfType<AgentAlert>();
    }

    [Test]
    public void the_new_AgentAlert_should_have_XXXXXXXXXXXX() {
        var alert = theUnitOfWork.NewObjects[0].ShouldBeOfType<AgentAlert>();
        // verify the actual properties of the new AgentAlert object
        // for correctness
    }
}

```

Persistence Ignorance

When I choose or design a persistence solution for one of my projects, I'm mindful of the impact of the persistence infrastructure on my business logic code—at least in a system with a great deal of business domain logic. Ideally, I'd like to design, build, and test my business logic relatively independently of the database and the persistence infrastructure code. Specifically, I want a solution that supports the idea of persistence ignorance or Plain Old CLR Objects (POCOs).

First, what is persistence ignorance, and where did it come from? In his book *Applying Domain-Driven Design and Patterns: With Examples in C# and .NET* (Pearson Education, Inc., 2006), Jimmy Nilsson defines POCO as "...ordinary classes where you focus on the business problem at hand without adding stuff for infrastructure-related reasons. ... The classes should focus on the business problem at hand. Nothing else should be in the classes in the Domain Model."

Now, why should you care? Let's start the discussion by being very clear that persistence ignorance is only a means to various design goals and not an end goal in and of itself. When I evaluate a new persistence tool, I'm generally asking myself the following questions in descending order of importance. Persistence ignorance is not completely required to satisfy these questions, but a solution based on persistence ignorance generally does better than solutions that require infrastructure concerns to be embedded in business objects.

Can the Business Logic Run Independently of the Database?

This is the most important question to me. I think that one of the most important factors in the success of any software project is the ability to use rapid feedback cycles. In other words, I want to shorten the time and effort in between "I just coded something new" and "I have proved that the new code is working so I will go on to something else" or "the new code has a problem, so I will fix it right now."

In my career, I have consistently observed teams to be much more productive in architectures where it is easy to unit test or even just exercise little pieces of new code without running through the entire application. Conversely, I've found architectures that have a tight run time coupling between business

application. Conversely, I've found architectures that have a tight run time coupling between business logic and infrastructure to be very difficult to develop in.

Let's reconsider the case of the business rule that says you should create a new Agent Alert when you encounter an open invoice older than 45 days. At some later point, the business may decide that you should create alerts at 30 days instead. In order to verify changes to the invoicing alert logic, you have to first put the code into a state that exercises this rule against open and closed invoices and invoices older or newer than the 30-day threshold. This is the point where the architecture starts to matter. Can I walk right up to the new invoicing alert logic and unit test a new business rule, or do I have to jump through technical hoops first?

In terms of a rapid feedback cycle, the worst extreme is a model where the business logic cannot function without the presence of infrastructure. For example, the first homegrown O/RM I created forced you to write Entity classes like this:

```
public class Invoice : MySpecialEntityType {
    private long _id;
    public Invoice(long id) {
        _id = id;
        // Read in the rest of the Invoice information
        // from the database
        loadDataFromDatabase(id);
    }
    public Invoice() {
        // Hit the database
        _id = fetchNextId();
    }
}
```

With this design, you literally cannot create an Invoice class without being connected to a database. In my original design, there was no easy way to use or exercise my business entity classes without having the database engine configured correctly and available.

It is certainly possible to write automated tests against code that is accessing the database, but it generally requires significantly more effort in developer time to set up test data than to write automated tests against objects that do not depend on a database. To set up a simple invoice in the database, you would have to contend with non-null fields and data to satisfy referential integrity requirements that do not pertain to the business rule in question.

Instead, wouldn't it be nicer just to say "if I have an open invoice that is 31 days old, then..." in code and be done with it? To that end, how about moving the Invoice class to more of a persistence-ignorant approach like this:

```
// The Invoice class does NOT depend on
// any sort of persistence infrastructure
public class Invoice {
    private long _id;

    public Invoice() { }

    public bool IsOpen { get; set; }
    public DateTime? InvoiceDate { get; set; }
}
```

Now, when you want to test the late invoicing alert rule, you can quickly create an open invoice that is 31 days old for the testing scenario with simple code in the test fixture like this:

```
[SetUp]
public void SetUp() {
    Invoice theLateInvoice = new Invoice() {
        InvoiceDate = DateTime.Today.AddDays(-31),
        IsOpen = true
    };
}
```

In this approach, I've separated the business logic code for the late invoice rule and the Invoice class itself away from the persistence code. As long as you can easily create invoice data in memory, you can rapidly unit test the business rules.

As an aside, when you're choosing an existing persistence tool, be cautious of the way that lazy loading is implemented. Some tools will implement transparent lazy loading with the Virtual Proxy pattern to make lazy loading effectively invisible to the business entities themselves. Other tools rely on code generation techniques that embed the lazy-loading support directly into the business entities and effectively create a tight run time coupling from the entities to the persistence infrastructure.

Almost as important as the time it takes for developers to author automated tests is the speed with which these tests execute. Automated tests involving data access or Web service access can easily run an order of magnitude or more slowly than tests that run completely within a single AppDomain. This may not seem like much of an issue, but as the size of the project begins to grow over time, slow-running automated tests can easily detract from a team's productivity and even eliminate the usefulness of the automated tests in the first place.

Can I Design My Domain Model Independently from the Database Model?

Now that I've decoupled my business layer from the database at run time, I'm faced with the issue of

Now that I've decoupled my business logic from the database schema, I'm faced with the issue of whether I can design my object structure independently of the database schema (and vice versa). I should be able to design my object model based on the necessary behavior of the business logic. The database, on the other hand, should be designed with efficiency of reading and writing data as well as enforcing referential integrity. (By the way, referential integrity is essential for using an O/RM tool.)

To demonstrate a typical scenario where the domain model can diverge from the database structure, let's switch the example domain to an energy trading system. This system is responsible for tracking and pricing quantities of petroleum sold, bought, and delivered. In a simplistic system, a trade may include several quantities purchased and a record of the quantities delivered. Here's the rub, though: quantity means unit of measure plus the number of those units. For this trading system, let's say that you track quantities in three different units:

```
public enum UnitOfMeasure {
    Barrels,

    Tons,
    MetricTonnes
}
```

If you were to lock the domain model to the flat structure of the database schema, you might get classes like this:

```
public class FlatTradeDetail {
    public UnitOfMeasure PurchasedUnitOfMeasure {get;set;}
    public double PurchasedAmount {get;set;}
    public UnitOfMeasure DeliveredUnitOfMeasure { get; set; }
    public double DeliveredAmount { get; set; }
}
```

While the structure is suitable for the database structure and convenient for solutions where you generate the code structure from an existing database, this structure does not lend itself to carrying out the business logic.

A large part of the business logic in the energy trading system requires quantities to be compared, subtracted, and added—but you must always assume that the units of measure are different and do the conversions from one to another before making any comparison. From painful experience I can state that using the flat structure generated directly out of the database schema makes this logic laborious to implement.

Instead of the flat structure, let's implement the Money pattern and create a class that models the behavior of a Quantity, as shown in **Figure 3**.

Figure 3 Quantity

```
public class Quantity {
    private readonly UnitOfMeasure _uom;
    private readonly double _amount;

    public Quantity(UnitOfMeasure uom, double amount) {
        _uom = uom;
        _amount = amount;
    }

    public Quantity ConvertTo(UnitOfMeasure uom) {
        // return a new Quantity that represents
        // the equivalent amount in the new
        // Unit of Measure
    }

    public Quantity Subtract(Quantity other) {
        double newAmount = _amount - other.ConvertTo(_uom).Amount;
        return new Quantity(_uom, newAmount);
    }

    public UnitOfMeasure Uom {
        get { return _uom; }
    }

    public double Amount {
        get { return _amount; }
    }
}
```

When you use the Quantity class to model and reuse common behavior around unit of measure quantities, the TradeDetail class might look more like this:

```
public class TradeDetail {
    private Quantity _purchasedQuantity;
    private Quantity _deliveredQuantity;
    public Quantity Available() {
        return _purchasedQuantity.Subtract(_deliveredQuantity);
    }
}
```

```

    }
    public bool CanDeliver(Quantity requested) {
        return Available().IsGreaterThan(requested);
    }
}

```

The Quantity class will definitely make the TradeDetail logic easier to implement, but now the object model has varied from the database structure. Ideally, your persistence tooling should support this sort of mapping.

The issue of having variance between an object model and a database model generally does not arise on systems with simple business logic. In those systems, an Active Record architecture that simply generates the entity classes from the database structure may be the easiest solution. Alternatively, you may use a database mapper tool that allows you to generate a database schema from the entity objects.

How Does My Persistence Strategy Affect My Business Logic?

It's not a perfect world. Any persistence tooling will have some sort of impact on how you shape your entity classes. For example, my team uses NHibernate for persistence. Due to the way that NHibernate implements lazy-loaded property relationships, many of our properties have to be marked as virtual just to enable lazy loading like this new Customer property on Invoice:

```

public class Invoice {
    public virtual Customer Customer { get; set; }
}

```

The Customer property is marked as Virtual for no other reason than to enable NHibernate to create a dynamic proxy for the Invoice object that provides a lazy-loaded Customer property

Marking members as virtual strictly to support lazy loading is an annoyance, but there are worse potential problems. Increasing numbers of teams are using Domain Driven Design (DDD). One of the common strategies in DDD is to create domain model classes that cannot be placed into an invalid state. The domain model classes themselves would also jealously guard their internal data to enforce business rules internally. To enact this design philosophy, an Invoice class might look more like **Figure 4**.

Figure 4 Domain Model Invoice Class

```

public class Invoice {
    private readonly DateTime _invoiceDate;
    private readonly Customer _customer;
    private bool _isOpen;

    // An Invoice must always have an Invoice Date and
    // a Customer, so there is NO default constructor
    // with no arguments
    public Invoice(DateTime invoiceDate, Customer customer) {
        _invoiceDate = invoiceDate;

        _customer = customer;
    }

    public void AddDetail(InvoiceDetailMessage detail) {
        // determines whether or not, and how, a
        // new Invoice Detail should be created
        // and added to this Invoice
    }

    public CloseInvoiceResponse Close(CloseInvoiceRequest request) {
        // Invoice itself will determine if the correct
        // conditions are met to close itself.
        // The _isOpen field can only be set by
        // Invoice itself
    }

    public bool IsOpen {
        get {
            return _isOpen;
        }
    }

    public DateTime InvoiceDate {
        get { return _invoiceDate; }
    }

    public Customer Customer {
        get { return _customer; }
    }
}

```

By no means is this approach to crafting a domain model applicable to every type of application, but if you do choose this style of architecture it will affect your choice of persistence tooling.

The version of Invoice in **Figure 4** has only one constructor function that enforces the rule that no Invoice can be created without an Invoice Date and a Customer. Many, if not most, persistence technologies require a no-argument constructor function

require a no-argument constructor function.

Also, this version of the Invoice class jealously guards internal fields like `_isOpen` without public setter properties. Again, many persistence tools can work only against public properties. If you want to use the more strict DDD style of entities, you will need to research whether your persistence tooling can support mapping fields, private properties, or constructor functions with arguments.

More Units of Work


There are some other important issues with using the Unit of Work pattern that are worth considering. If you are interested in using the Unit of Work pattern explicitly in your application, these issues should be researched in the context of your application:

- The division of responsibility between your Repositories and the Unit of Work. You may say that all reads happen through Repositories and writes through the Unit of Work. You may also use a Unit of Work implementation that forces you to do reads and queries through the Unit of Work to make state change tracking easier.
- How to deliver the correct Unit of Work and the underlying Identity Map helpers to the various classes taking part in a logical transaction. Many people use an Inversion of Control container to correctly attach the Unit of Work for the current HttpContext, thread, or other scoping strategy.

Persistence ignorance is somewhat controversial in the .NET community and its importance is heavily disputed. At this time, persistence ignorance is not well supported in the persistence tooling available in the Microsoft .NET Framework itself or in the extended .NET ecosystem. NHibernate is the one tool that comes closest, but even with NHibernate you will have to compromise your domain model somewhat to enable persistence.

Send your questions and comments to mmpatt@microsoft.com.

Jeremy Miller, a Microsoft MVP for C#, is also the author of the open-source [StructureMap](#) tool for Dependency Injection with .NET and the forthcoming [StoryTeller](#) tool for supercharged FIT testing in .NET. Visit his blog, "[The Shade Tree Developer](#)," part of the CodeBetter site.

**NEVRON
CHART for .NET**
DOWNLOAD YOUR
FREE TRIAL

The industry leading
charting and gauge
components for WinForm,
WPF, ASP.NET and MVC.

