

The practice of enterprise application development has benefited from the emergence of many new enabling technologies. Multi-tiered object-oriented platforms, such as Java and .NET, have become commonplace. These new tools and technologies are capable of building powerful applications, but they are not easily implemented. Common failures in enterprise applications often occur because their developers do not understand the architectural lessons that experienced object developers have learned.

Patterns of Enterprise Application Architecture is written in direct response to the stiff challenges that face enterprise application developers. The author, noted object-oriented designer Martin Fowler, noticed that despite changes in technology--from Smalltalk to CORBA to Java to .NET--the same basic design ideas can be adapted and applied to solve common problems. With the help of an expert group of contributors, Martin distills over forty recurring solutions into patterns. The result is an indispensable handbook of solutions that are applicable to any enterprise application platform.

The topics covered include:

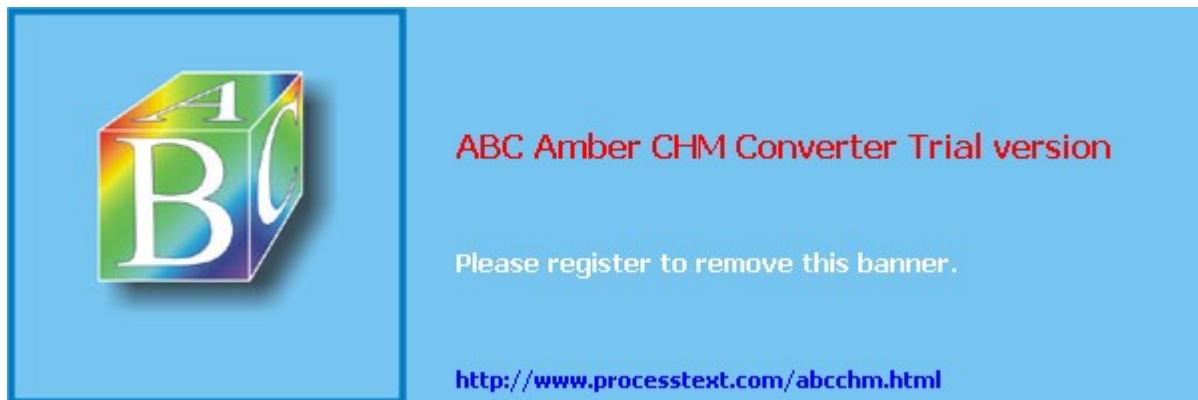
- Dividing an enterprise application into layers
- The major approaches to organizing business logic
- An in-depth treatment of mapping between objects and relational databases
- Using Model-View-Controller to organize a Web presentation
- Handling concurrency for data that spans multiple transactions
- Designing distributed object interfaces

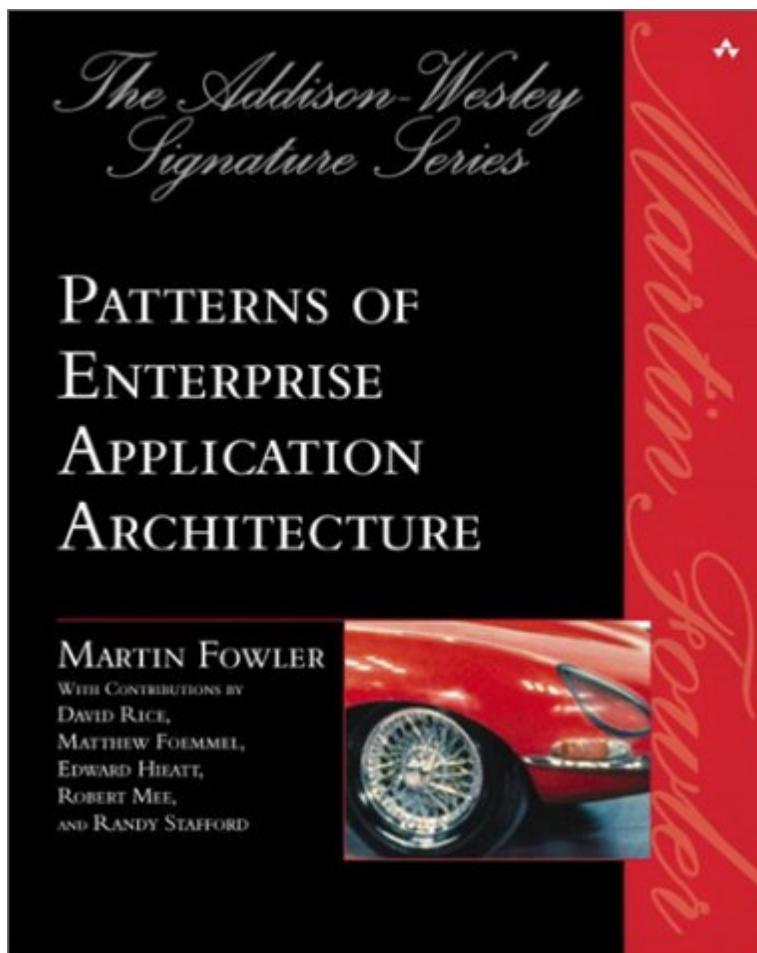
This book is actually two books in one. The first section is a short tutorial on developing enterprise applications, which you can read from start to finish to understand the scope of the book's lessons. The next section, the bulk of the book, is a detailed reference to the patterns themselves. Each pattern

provides usage and implementation information, as well as detailed code examples in Java or C#. The entire book is also richly illustrated with UML diagrams to further explain the concepts.

Armed with this book, you will have the knowledge necessary to make important architectural decisions about building an enterprise application and the proven patterns for use when building them.

*Released by Asmodeous*  
*<asmodeous77@hotmail.com>*





The practice of enterprise application development has benefited from the emergence of many new enabling technologies. Multi-tiered object-oriented platforms, such as Java and .NET, have become commonplace. These new tools and technologies are capable of building powerful applications, but they are not easily implemented. Common failures in enterprise applications often occur because their developers do not understand the architectural lessons that experienced object developers have learned.

Patterns of Enterprise Application Architecture is written in direct response to the stiff challenges that face enterprise application developers. The author, noted object-oriented designer Martin Fowler, noticed that despite changes in technology--from Smalltalk to CORBA to Java to .NET--the same basic design ideas can be adapted and applied to solve common problems. With the help of an expert group of contributors, Martin distills over forty recurring solutions into patterns. The result is an indispensable handbook of solutions that are applicable to any enterprise application platform.

The topics covered include:

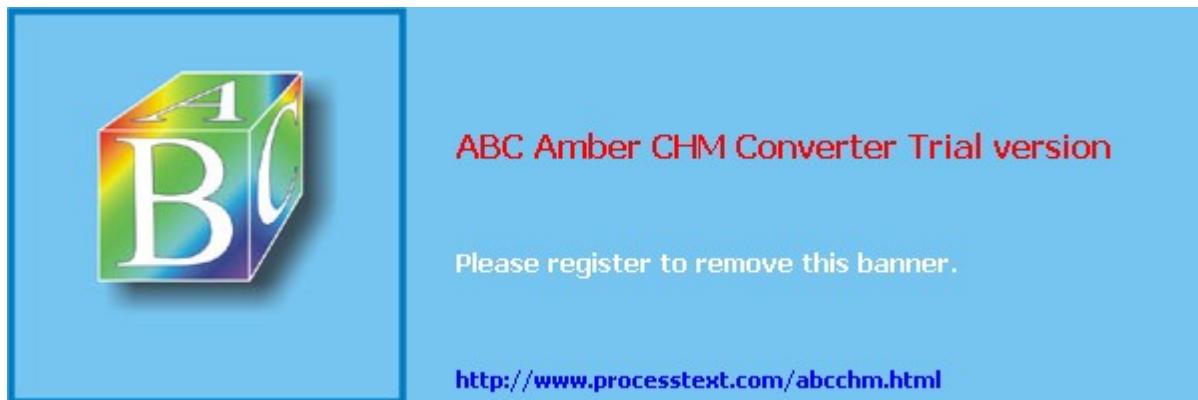
- Dividing an enterprise application into layers
- The major approaches to organizing business logic
- An in-depth treatment of mapping between objects and relational databases
- Using Model-View-Controller to organize a Web presentation
- Handling concurrency for data that spans multiple transactions
- Designing distributed object interfaces

This book is actually two books in one. The first section is a short tutorial on developing enterprise applications, which you can read from start to finish to understand the scope of the book's lessons. The next section, the bulk of the book, is a detailed reference to the patterns themselves. Each pattern

provides usage and implementation information, as well as detailed code examples in Java or C#. The entire book is also richly illustrated with UML diagrams to further explain the concepts.

Armed with this book, you will have the knowledge necessary to make important architectural decisions about building an enterprise application and the proven patterns for use when building them.

*Released by Asmodeous*  
*<asmodeous77@hotmail.com>*



# Patterns of Enterprise Application Architecture

---

(formerly known as Information System Architecture)

- [Background](#)
- [The Content](#)
- [Why the rename?](#)
- [Revision History](#)

Last Significant Update: [June 2002](#)

## Background

One of the recurring themes of my career is how to structure sophisticated enterprise applications, particularly those that operate across multiple processing tiers. It's a subject I discussed briefly in a chapter in Analysis Patterns, and one that I looked at in more depth in many consulting engagements. In the last few years I've been doing a lot of work with Enterprise Java: but there's no surprise that many of the ideas and patterns involved are exactly the same as those that I've run into earlier with CORBA, DCE, and Smalltalk based systems.

This notion of common lessons has been amplified by my collaborations with developers at ThoughtWorks. A lot of old time developers at ThoughtWorks did a lot of work with Forte - which was a sophisticated distributed environment. Others have background in CORBA and DCOM. Again we see common lessons and patterns that are relevant whatever the technological platform.

So I decided to take advantage of this knowledge to capture and write down some of these ideas. Most of the writing here is mine, but the ideas come from many ThoughtWorkers, although Dave Rice and Matt Foemmel have contributed a particularly large amount of information. This material is the content of my next book: [Patterns of Enterprise Application Architecture](#), which is due to appear at [OOPSLA 2002](#).

As I've been writing this book, I've been putting information on these web pages, both to give readers an idea of what I'm working on, and to solicit feedback on the evolving book. In June 2002 I sent my final draft to Addison-Wesley and it was "released to production". That means the hard intellectual work is done, we are now in the process of copy editing, book design, indexing, and the other tasks of production - most of which is handled by Addison-Wesley.

AW have graciously allowed me to keep these pages up until the book is released in November. Most of the text here corresponds to the second draft that was sent out for review in April, not the final draft for the book. (It seems that if you write a book and make it available on the web, people complain that the book is only a copy of the free web pages!) So the final pages will be much neater in the book itself.

However we decided to include in this version some of the newer material on offline concurrency - most of which was contributed by my colleague David Rice. The book will include code examples for the offline concurrency patterns. David has played a big part in putting together this section and he has turned into a significant contributor - contributing about 10% of the book.

The final book will also include two further guest pattern descriptions:

- *Repository* by Edward Hieatt and Robert Mee
- *Service Layer* by Randy Stafford

When the book appears in print I'll bring up a new site with abbreviated versions of the patterns, similar to the way I did for the refactoring book. I haven't yet decided how that will be done, but I will certainly change the URLs at that point.

Thanks to the many of you that have given me comments on the book as it was in the works. Keep your eyes open for my next writing projects, whatever they turn out to be.

## The Content

The book is organized in two parts. The first part is a set of narrative chapters that can be read from start to finish. In the preliminary book designs these come to around 100 pages. The second part is a reference section of the patterns themselves. The idea is that the chapters discuss the general issues and summarize the patterns that are relevant - then the patterns chapters capture the detail.

- [Preface](#)
- [Introduction](#)
- **Narrative Chapters**
  - [Layering](#)
  - [Organizing Domain Logic](#)
  - [Web Presentation](#)
  - [Mapping to a Relational Database](#)
  - [Concurrency](#)
  - [Distribution Strategies](#)
  - [Putting it all Together](#)
- **Patterns Chapters**
  - Domain Layer Patterns [Transaction Script](#), [Domain Model](#), [Table Module](#)
  - Web Server Patterns [Model View Controller](#), [Page Controller](#), [Front Controller](#), [Template View](#), [Transform View](#), [Two Step View](#), [Application Controller](#)
  - Object / Relational Mapping Patterns [Active Record](#), [Row Data Gateway](#), [Table Data Gateway](#), [Data Mapper](#), [Metadata Mapping](#), [Identity Map](#), [Identity Field](#), [Serialized LOB](#), [Foreign Key Mapping](#), [Dependent Mapping](#), [Association Table Mapping](#), [Embedded Value](#), [Lazy Load](#), [Concrete Table Inheritance](#), [Single Table Inheritance](#), [Class Table Inheritance](#), [Inheritance Mappers](#), [Query Object](#)
  - Concurrency Patterns [Unit of Work](#), [Optimistic Offline Lock](#), [Pessimistic Offline Lock](#), [Implicit Lock](#), [Coarse-Grained Lock](#), [Client Session State](#), [Database Session State](#), [Server Session State](#)
  - Distribution Patterns [Remote Facade](#), [Data Transfer Object](#)
  - Base Patterns [Gateway](#), [Record Set](#), [Value Object](#), [Money](#), [Plugin](#), [Service Stub](#), [Layer Supertype](#), [Special Case](#), [Registry](#)

# Why the rename?

As I've been working on this during 2001 I was doing it under the title of "Information Systems Architecture". As we were working on the book proposal for Addison-Wesley, the marketing folks there said that "information systems" was a rather old fashioned term, so instead we chose the term "enterprise application". (I have no qualms about selling out, providing the price is right.)

## Revision History

Here's a list of the major updates to this paper

- *June 2002*: Updated the pattern names to match the final draft and included updates to the offline concurrency material from the final draft: this includes the [concurrency narrative](#) and the patterns [Optimistic Offline Lock](#), [Pessimistic Offline Lock](#), [Coarse-Grained Lock](#), and [Implicit Lock](#).
- *April 2002*: More work on preparing for the second draft. I've added [Page Controller](#) and made lots of minor changes. I gave a particular washing to the domain logic chapter.
- *Mar 27 2002*: A second update this month, again prompted by changes due to the official review comments. I've added a new chapter on putting the patterns together and added patterns for [Metadata Mapping](#), [Query Object](#) and [Record Set](#)
- *Mar 2002*: My official reviewers, as usual, gave me a lot to chew on. This version reflects much of this mastication. The various aspects touching on business vs systems transactions have all been rolled into a general chapter on concurrency. I've split database gateway into separate patterns for its two main forms, and I've hunted down some stupid coding styles that would hurt multi-threading. I've also added a .NET example for [Page Controller](#) and [Template View](#) and added the [Money](#) pattern. This isn't quite the second draft yet, as I've still got more comments to work on.
- *Jan 2002*: I added an introduction and updated some examples on the web server patterns. Dave added both a chapter and patterns on concurrency. We renamed the material to Enterprise Application Architecture. This revision is the first draft of the book and corresponds to the copy sent out for technical review.
- *Nov 2001*: I did a general walk-through of many of the patterns tidying up on examples and sketches. I broke apart [Identity Field](#) into several patterns, including [Foreign Key Mapping](#), [Dependent Mapping](#) and [Association Table Mapping](#). Added content and examples for [Registry](#), and [Special Case](#). Dave Rice added [Service Stub](#)
- *Oct 2001*: Added write-ups for the inheritance OR Mappings. Dave Rice and Matt Foemmel added [Plugin](#). Added example for [Table Module](#). Many of these new examples use C#.
- *Sep 2001*: Dave Rice added a code example for [Unit of Work](#). I re-did the discussion of views in web servers, renamed distributed facade to session facade, added [Table Module](#) and did more rejigging of the layering section.
- *August 2001*: Rewrote narratives on layers and O/R mapping. Added code example for [Gateway](#)
- *July 2001*: Major update to web server patterns. Added [Lazy Load](#)
- *May 2001*: Updated and added code samples to [Transaction Script](#) and [Domain Model](#)
- *April 2001*: Updated most of the O/R mapping patterns. Added code examples to most of

them

- *March 2001:* Initial Upload

---

© Copyright [Martin Fowler](#), all rights reserved



# Introduction

---

In case you haven't realized it, building computer systems is hard. As the complexity of the system gets greater the task of building the software gets exponentially harder. Like any profession we can only progress by learning, both from our mistakes and our successes. This book represents some of this learning written in a form that I hope will help you to learn these lessons quicker than I did, or communicate to others more effectively than I did before I boiled these patterns down.

For this introduction I want to set the scope of the book, and provide some of the background that'll underpin the ideas in the book.

## Architecture

The software industry has a delight in taking words and stretching them into a myriad of subtly contradictory meanings. One of the biggest sufferers is "architecture". I tend to look at architecture as one of those impressive sounding words, used primarily to indicate that we are talking about something that's important. But I'm pragmatic enough to not let my cynicism get in the way of attracting people to my book :-)

In this book I want to concentrate of the major parts of an enterprise application and how these parts fit together. Some of these patterns can reasonably be called architectural patterns, in that they represent significant decisions about these parts, others are more design patterns that help you to realize that architecture.

The dominant architectural pattern is that of layers, which I describe more in chapter 2. So this book is about how you decompose an enterprise application into layers and how these layers work together. Most non-trivial enterprise applications use a layered architecture of some form, but there are some situations where other approaches, such as pipes and filters are valuable. I don't go into those situations, focusing instead on the context of a layered architecture because its the most widely useful.

## Enterprise Applications

Lots of people write computer software, and we all call it all software development. Yet there are distinct kinds of software out there, each of which has its own challenges and complexities. This comes out when I talk with some of my friends in the telecoms field. In some ways enterprise applications are

much easier than telecoms software - we don't have very hard multi-threading problems, we don't have hardware and software integration. But in other ways its much tougher. Enterprise applications often have large and complex data to work on, together with business rules that fail all tests of logical reasoning. Although some techniques and patterns are relevant for all kinds of software, many are relevant for only one particular branch.

In my career I've concentrated on enterprise applications, so my patterns here are all about that. But what do I mean by the term "enterprise application"? I can't give a precise definition for them, but I can give some indications.

I'll start by example. Things that are enterprise applications include: payroll, patient records, shipping tracking, cost analysis, credit scoring, insurance, accounting, foreign-exchange trading. Things that are not enterprise applications include: automobile fuel injection, word processors, elevator controllers, chemical plant controllers, telephone switches, operating systems, compilers, games.

Enterprise applications usually involve **persistent data**. The data is persistent because it needs to be around between multiple runs of the program - indeed it usually needs to persist for several years. During this time there will be many changes in the programs that use it. It will often outlast the hardware that originally created much of the data, and outlast operating systems and compilers. During that time there'll be many changes to the structure of the data in order to store new pieces of information without disturbing the old. Even if there is a fundamental change and the company installs a completely new application to handle a job, the data has to be migrated to the new application.

There's usually **a lot of data**, a moderate system will have over 1GB of data organized in tens of millions of records, so much data that managing it is a major part of the system. Older systems used indexed file structure such as IBM's VSAM and ISAM. Modern systems usually use databases, mostly relational databases. The design and feeding of these databases has turned into a sub-profession of its own.

Usually many people **access data concurrently**. For many systems this may be less than a hundred people, but for web based systems that talk over the Internet, this goes up by orders of magnitude. With so many people there are definite issues in ensuring that they can access the system properly. But even without that many people there are still problems in making sure that two people don't access the same data at the same time in such a way that would cause errors. Transaction manager tools handle some of the burden of this, but often it's impossible to hide this from application developers.

With so much data, there's usually **a lot of user interface screens** to handle this data. It's not unusual to have hundreds of distinct screens. Users of enterprise applications vary from habitual to regular users. Usually they will have little technical expertise. The data has to be presented lots of different ways for different purposes.

Enterprise applications rarely live in an island. Usually they need to **integrate with other enterprise applications** scattered around the enterprise. The various systems are usually built at different times with different technologies. Even the collaboration mechanisms will be different: COBOL data files, CORBA, messaging systems. Every so often the enterprise will try to integrate its different systems using a common communication technology. Of course it hardly ever finishes the job, so there are several different unified integration schemes in place at once.

Even if a company unifies the technology for integration, they run into problems with differences in business process and **conceptual dissonance** with the data. One division of the company may think a customer is someone who has a current agreement with the company, another department also counts those that had a contract but don't any longer, another one counts product sales but not service sales.

That may sound like it's easy to sort out but when have hundreds of records where every field can have subtly different meanings, the sheer size of the problem becomes a challenge - even if the only person who knows what this field really means is still with the company. (And of course all of this changes without warning.) As a result data has to be constantly read, munged, and written in all sorts of different syntactic and semantic formats.

Then there is the matter of what goes on under the term "business logic". I find this a curious term because there are few things more less logical than business logic. When you build an operating system you strive to keep the whole thing logical. But business rules are just given to you and without major political effort there's nothing you can do to change them. The rules are haphazard often with no seeming logic. Of course they got that way for a reason: some salesman negotiated to have a certain yearly payment two days later than usual because that fitted with his customer's accounting cycle and thus won half a million dollars of business. A few thousand of these one off special cases is what leads to the **complex business illogic** that makes business software so difficult.

For some people the term "Enterprise Application" implies a large system. But it's important to remember that not all enterprise applications are large, even though they can provide a lot of value to the enterprise. Many people assume that since small systems aren't large, they aren't worth bothering with. To some degree there's merit here, if a small system fails it usually makes less of a noise than a big system. However I think such thinking tends to short-change the cumulative effect of many small projects. If you can do things that improve small projects, then that cumulative effect can be very significant on an enterprise, particularly since small projects often have disproportionate value. Indeed one of the best things you can do is turn a large project into a small project by simplifying its architecture and process.

## Kinds of enterprise application

When we discuss how to design enterprise applications, and what patterns to use, it's important to realize that enterprise applications are different, and that different problems lead to different ways of doing things. I have a set of alarm bells that go off when people say "always do this". For me, much of the challenge (and interest) in design is in knowing about alternatives and judging the trade offs of when to use one alternative over another. There is a large space of alternatives to choose from, but here I'll pick three points on this very big plane.

Consider a B2C (business to customer) online retailer: people browse and, with luck and shopping cart, buy. For such a system we need to be able handle a very high volume of users, so our solution needs to be both reasonably efficient in terms of resources used, but also scalable so that you can increase the load you can handle by adding more hardware. The domain logic for such an application is pretty straightforward: capturing orders, some relatively simple pricing and shipping calculations, and notification of shipments. We want anyone to be able access the system easily, so that implies a pretty generic web presentation which can be used with the widest possible range of browsers. Data source includes a database for holding orders and perhaps some communication with an inventory system to help with availability and delivery information.

Contrast this with a system for automating the processing of leasing agreements. In some ways this is a much simpler system than the retailer because there are much fewer users, no more than a hundred or so at any one time. Where it's more complicated is in the business logic. Calculating monthly bills on a lease, handling events such as early returns and late payments., validating data as a lease is booked - all of

these are complicated tasks since much of the leasing industry competition comes in the form of yet more little variations over deals done in the past. A complex business domain such as this is challenging because the rules are so arbitrary. If you design an operating system or a telephone switch at least are trying to develop the processing in a logical way. Business rules often defy all logic, and are subject to regular changes over time.

Such a system also has more complexity in the UI. At the least this means a much more involved HTML interface with more screens and more complex screens. Often these kinds of systems have UI demands which lead users to want a more sophisticated presentation than a HTML front end allows - so a more conventional rich client interface is needed. The more complex user interaction also leads to more complicated transaction behavior: booking a lease may take an hour or two during which the user is in a logical transaction. We also see a complex database schema with a couple of hundred tables, and connections to packages for asset valuation and pricing.

A third example point would be a simple expense tracking system for a small company. Such a system has few users and simple logic. It can easily be made accessible across the company with an HTML presentation. The only data source is a few tables in a database. Yet even a simple system like this is not devoid of a challenge. You have to build it very quickly. You also have to bear in mind that it may grow later as people want to calculate the reimbursement checks, feed them into the payroll system, understand tax implications, provide reports for the CFO, tie into airline reservation web services, and so on. Trying to use the architecture for either of the other two systems will slow down the development of this system, and add complexity that will probably harm its future evolution. And although this system may be small, most enterprises have a lot of such systems, so the cumulative effect of an over-complex architecture can be significant.

## Performance

It's always difficult to talk about performance in a book such as this. The reason why it's so difficult is that any advice about performance should not be treated as fact until it is measured on your configuration. Too often I've seen designs used or rejected due to performance considerations, which turned out to be bogus once somebody actually did some measurements on the setup actually used for the application.

I've given a few guidelines in this book, including minimizing remote calls which has been good performance advice for quite a while. Even so you should verify every tip by measuring on your application. Similarly there are several occasions where code examples in this book sacrifice performance for understandability. Again it's up to you to apply the optimizations for your environment. But whenever you do a performance optimization you must measure both before and after, otherwise you may just be making your code harder to read.

There's an important corollary to this: any significant change in configuration may invalidate any facts about performance. So if you upgrade to a new version of your virtual machine, hardware, database, or almost anything else - redo your performance optimizations and make sure they are still helping. In many cases a new configuration can change things. Indeed you may find that an optimization you did in the past to improve performance actually hurts performance in the new environment.

Another problem with talking about performance is the fact that many terms are used in an inconsistent way. The most noted victim of this is "scalability", which regularly gets used to mean half a dozen

different things. To my, rather cynical eye, most comments about things that affect scalability are vague threats about ill-defined future fears about performance - and of course since they are future threats it's rather hard to verify them with measurements. As a result here's the terms I use.

**Response time** is the amount of time it takes for the system to process a request from the outside. This may be a UI action, such as pressing a button, or a server API call.

**Responsiveness** is about how quickly the system acknowledges the request, as opposed to processing it. This is important in many systems because people may get frustrated if a system has low responsiveness, even if its response time is good. If your system waits during the whole request, then your responsiveness and response time are the same. However if you indicate you've received the request before you complete, then your responsiveness is better. Providing a progress bar during a file copy improves the responsiveness of your user interface, even though it doesn't improve the response time.

**Latency** is the minimum time required to get any form of response, even if the work to be done is non-existent. Latency is usually the big issue in remote systems. If I ask a program to do nothing, but tell me when it's done doing nothing, then if the program runs on my laptop I should get an almost instantaneous response. However if the program runs on a remote computer I may get a few seconds just because of the time taken for the request and response to make their way across the wire. As an application developer, there's usually nothing I can do to improve latency. Latency is also the reason why you should minimize remote calls.

**Throughput** is how much stuff you can do in a given amount of time. If you're measuring copying a file, throughput might be measured in bytes per second. For enterprise applications a typical measure is transactions per second, but the problem with this is that it depends on the complexity of your transaction. For your particular system you should pick a common set of transactions.

The **load** of a system is a statement of how much stress a system is under. This might be measured in how many users are currently connected to the system. The load is usually a context for some other measurement, such as a response time. So you may say that the response time for some request is 0.5 seconds with 10 users and 2 seconds with 20 users. The **load sensitivity** is an expression of how the response time varies with the load. A system that has a response time of 0.5 seconds for 10 through 20 users has a lower load sensitivity than a system with a response time of 0.2 seconds for 10 users that rises to 2 seconds for 20 users. The **capacity** of a system is an indication of a maximum effective throughput or load. This might be an absolute maximum, or a point at which the response time dips below an acceptable threshold.

**Scalability** is a measure of how adding hardware effects the performance. A scalable system is one that allows you to add hardware and get a commensurate performance improvement, such as doubling your server capacity doubling your throughput..

The problem is that design decisions don't affect all these performance factors equally. Say we have two software systems running on a server: swordfish has a capacity of 20 tps, while camel's capacity is 40 tps. Which has better performance, which is more scalable? We can't answer the scalability question from this data, and we can only say that camel has a higher capacity on a single server. Now if we add another server we notice that swordfish now handles 35 tps and camel handles 50 tps. Camel's capacity is still better, but swordfish looks like it may scale better. Let's say we continue adding servers and discover that Swordfish gets 15 tps per extra server and Camel gets 10 tps per extra server. Given this data we can say swordfish is more scalable, even though Camel has better capacity for less than five servers.

When building enterprise systems, it often makes sense to build for hardware scalability rather than capacity or even load sensitivity. Scalability gives you the option of better performance if you need it. Scalability is also often easier to do. Often designers do complicated things that improve the capacity on a particular hardware platform, when it might actually be cheaper to buy more hardware. If Camel has a greater cost than Swordfish, and that greater cost is equivalent to a couple of servers, then Swordfish ends up being cheaper even if you only need 40 tps. It's fashionable to complain about relying on better hardware to make our software run properly, and I join this choir whenever I have to upgrade my laptop just to handle the latest version of Word. But newer hardware is often cheaper than the price of making software run on less powerful systems. Similarly adding more servers is often a cheaper price than adding more programmers - providing a system is scalable.

## Patterns

Patterns have been around for a long time, so part of me does not want to regurgitate the history of them yet another time. But this is an opportunity for me to provide my view of patterns and what it is that makes patterns a worthwhile way to approach describing design.

There's no generally accepted definition of a pattern but perhaps the best place to start is Christopher Alexander, and inspiration for many pattern enthusiasts: "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice" [\[Alexander et al, pattern language\]](#). Alexander is an architect so he was talking about buildings, but the definition works pretty nicely for software as well. The focus of the pattern is a particular solution, one that is both common and effective at dealing with one or more recurring problems. Another way of looking at it is that it is a chunk of advice, and the art of creating patterns is to divide up many pieces of advice into relatively independent chunks, so that you can refer to them and discuss them more or less separately.

A key part of patterns is that they are rooted in practice. You find patterns by looking at what people do, observing things that work, and then looking for the "core of the solution". It isn't an easy process, but once you've found some good patterns they become a valuable thing. For me their value lies in being able to create a book that is a reference book. You don't need to read all of this book, or of any patterns book, to find it useful. You need to read enough of the book to have a sense of what the patterns are, the problems they solve, and some sense of how they do it. But you don't need to know all the details. You just need enough so that if you run into one of the problems, you can find the pattern in the book. Only then do you need to really understand the pattern and its solution.

Once you need the pattern, you then have to figure out how to apply to your circumstances. A key thing about patterns is that you can never just apply the solution blindly, which is why patterns tools have been such miserable failures. A phrase I like to use is that patterns are "half baked" - meaning that you always have to finish them off in the oven of your own project. Every time I use a pattern I tweak it a little here and a little there. So you see the same solution many times over, but it's never exactly the same.

Each pattern is relatively independent, but they are not isolated from each other. So often one pattern leads to another or one pattern usually only occurs if another is around. So you'll usually only see [Class Table Inheritance](#) if there's a [Domain Model](#) in your design. The boundaries between the patterns are naturally very fuzzy, but I've tried to make each pattern as self-standing as I can. So if someone says

"use a [Unit of Work](#) you can look it up and see how to apply without having to read the entire book.

If you're an experienced designer of enterprise applications, you'll probably find most of these patterns are familiar to you. I hope you won't be too disappointed (I did try to warn you in the preface). Patterns aren't original ideas, they are very much observations of what happens in the field. As a result patterns authors don't say they "invented" a pattern, rather we say we "discovered" a pattern. Our role is simply to note the common solution, look for its core, and then write down the resulting pattern. For an experienced designer, the value of the pattern is not that it gives you a new idea, the value lies in helping you communicate your idea. If you and your colleagues all know what a [Remote Facade](#) is, you can communicate a lot by saying "this class is a [Remote Facade](#)". It also allows you to say to someone newer, "use a [Data Transfer Object](#) for this" and they can come to this book to look it up. As a result patterns create a vocabulary to talk about design, which is why naming is such an important issue.

## The Structure of the Patterns

Every author has to choose his pattern form. Some base their forms on the classic patterns book such as [\[Alexander et al, pattern language\]](#), [\[Gang of Four\]](#), or [\[POSA\]](#). Others make up their own. I long wrestled with what makes the best form. On the one hand I don't want something as bitty as the GOF form, on the other I do need to have sections that support a reference book. So this is what I've used for this book.

The first item is the name of the pattern. Pattern names are crucially important, because part of the purpose of patterns is to create a vocabulary that allows designers to communicate more effectively. So if I tell you my web server is built around a [Front Controller](#) and a [Transform View](#) and you know these patterns, you have a very clear idea of the architecture of my web server.

Next are two items that go together: the intent and the sketch. The intent sums up the pattern in a sentence or two. The sketch is a visual representation of the pattern, often but not always a UML diagram. The idea behind these is to be a brief reminder of what the pattern is about, so you can quickly recall it. If you already "have the pattern", meaning you know the solution even if you don't know the name, then the intent and sketch should be all you need to know what the pattern is.

The next section describes a motivating problem for the pattern. This may not be the only problem that the pattern solves, but it's one that I think best motivates the pattern.

*How it Works* describes the solution. In here I put a discussion of how it works, variations that I've come across, and various implementation issues that come up. The discussion is as independent as possible of any particular platform, where there are platform specific sections I've indented them off so you can see them and easily skip over them. Where useful, I've put in UML diagrams to help explain them.

*When to Use It* describes when the pattern should be used. Here I talk about the trade offs that make you select this solution compared to others. Many of the patterns in this book are alternatives: such [Page Controller](#) and [Front Controller](#). Few patterns are always the right choice, so whenever I find a pattern I always ask myself "when would I not do this?" That question often leads me to alternative patterns.

I like to add one or more *examples*. Each example is a *simple* example of the pattern in use, illustrated with some code in Java or C#. I chose those languages because these seem to be languages that largest

amount of professional programmers can read. It's absolutely essential to understand that the example is not the pattern. When you use the pattern it won't look exactly like this example - so don't treat it as some kind of glorified macro. I've deliberately kept the example as simple as possible, so you can see the pattern in as clear a form as I can imagine. There are all sorts of issues that are ignored that will become important when you use it, but these will be particular to your own environment, which is why you always have to tweak the pattern.

One of the consequences of this is that I've worked hard to keep each example as simple as I can, while still illustrating the core message of the pattern. As such I've often chosen an example that's simple and explicit, rather than one that demonstrates how a pattern works with many of the many wrinkles required in a production system. It's a tricky balance between simple and simplistic, but it's also true that too many realistic yet peripheral issues can make it harder to understand the key points of a pattern.

This is also why I've gone for simple independent examples, instead of a connected running examples. Independent examples are easier to understand in isolation, but give less guidance on how you put them together. A connected example shows how things fit together, but it's hard to understand any one pattern without understanding all the others involved in the example. While in theory it's possible to produce examples that are both connected yet understandable independently, doing so is very hard. Or at least too hard for me, so I chose the independent route.

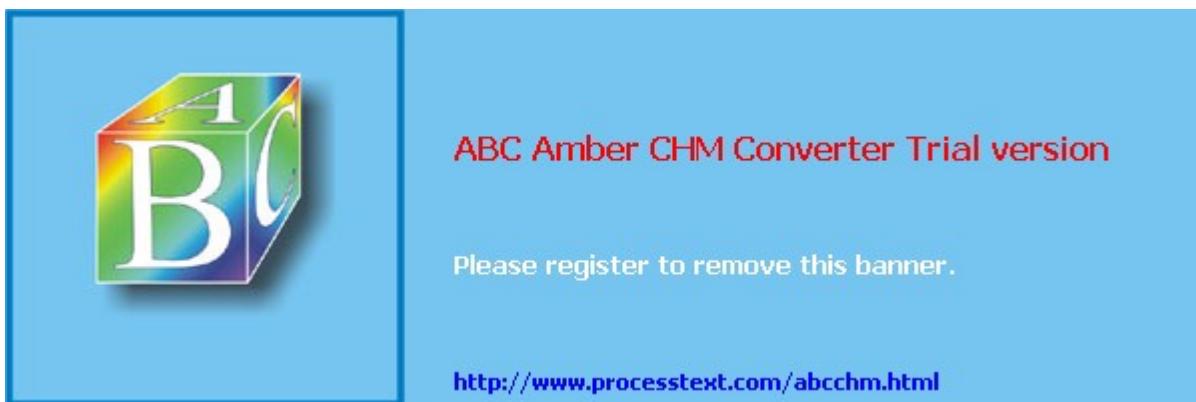
The *Further Reading* section points you to other discussions of this pattern. This is not a comprehensive bibliography on the pattern, I've limited my references to pieces that I think are important in helping you understand the pattern. So I've eliminated discussion that I don't think add much to what I've written, and of course eliminated discussions of those patterns I haven't read. I also haven't mentioned items that I think are going to be hard to find, or unstable web links that I fear may disappear by the time you read this book.

Not all the sections appear in all of the patterns. If I couldn't think of a good example or motivation text, I left it out.



---

© Copyright [Martin Fowler](#), all rights reserved



# Preface

---

In the spring of 1999 I flew to Chicago to consult on a project being done by ThoughtWorks, a small but rapidly growing application development company. The project was one of those ambitious enterprise application projects: a back-end leasing system. Essentially what this system does is to deal with everything that happens to a lease after you've signed on the dotted line. It has to deal with sending out bills, handling someone upgrading one of the assets on the lease, chasing people who don't pay their bills on time, and figuring out what happens when someone returns the assets early. That doesn't sound too bad until you realize that leasing agreements are infinitely varied and horrendously complicated. The business "logic" rarely fits any logical pattern, because after all its written by business people to capture business, where odd small variations can make all the difference in winning a deal. Each of those little victories is yet more complexity to the system.

That's the kind of thing that gets me excited. How to take all that complexity and come up with system of objects that can make more tractable. Developing a good [Domain Model](#) for this is difficult, but wonderfully satisfying.

Yet that's not the end of the problem. Such a domain model has to persist to a database, and like many projects we were using a relational database. We also had to connect this model to a user interface, provide support to allow remote applications to use our software, and integrate our software with third party packages. All of this on a new technology called J2EE which nobody in the world had any real experience in using.

Even though this technology was new, we did have the benefit of experience. I'd been doing this kind of thing for ages now with C++, Smalltalk, and CORBA. Many of the ThoughtWorkers had a lot of experience with Forte. We already had the key architectural ideas in our heads, we just had to figure out how to apply them to J2EE. Looking back on it three years later the design is not perfect, but it's stood the test of time pretty damn well.

That's the kind of situation that is where this book comes in. Over the years I've seen many enterprise application projects. These projects often contain similar design ideas which have proven to be effective ways to deal with the inevitable complexity that enterprise applications possess. This book is a starting point to capture these design ideas as patterns.

The book is organized in two parts. The first part is a set of narrative chapters on a number of important topics in the design of enterprise applications. They introduce various problems in the architecture of enterprise applications and their solutions. However the narrative chapters don't go into much detail on these solutions. The details of the solutions are in the second part, organized as patterns. These patterns are a reference and I don't expect you to read them cover to cover. My intention is that you can read the narrative chapters in part one from start to finish to get a broad picture of what the book covers, then you can dip into the patterns chapters of part two as your interest and needs drive you. So this book is a

short narrative book and a longer reference book combined into one.

This is a book on enterprise application design. Enterprise applications are about the display, manipulation and storage of large amounts of often complex data. Examples include reservation systems, financial systems, supply chain systems, and many of the systems that run modern business. Enterprise applications have their own particular challenges and solutions. They are a different animal to embedded systems, control systems, telecoms, or desktop productivity software. So if you work in of these other fields, there's nothing really in this book for you (unless you want to get a feel for what enterprise applications are like.) For a general book on software architecture I'd recommend [\[POSA\]](#).

There are many architectural issues in building enterprise applications. I'm afraid this book isn't a comprehensive guide to them. In building software I'm a great believer in iterative development. At the heart of iterative development is the notion that you should deliver software as soon as you have something useful to the user, even if it's not complete. Although there are many differences between writing a book and writing software, this notion is one that I think the two share. So this book is an incomplete but (I trust) useful compendium of advice on enterprise application architecture. The primary topics I talk about are:

- layering of enterprise applications
- how to structure domain (business) logic
- the structure of a web user interface
- how to link in-memory modules (particularly objects) to a relational database
- how to handle session state in stateless environments
- some principles of distribution

The list of things I don't talk about is rather longer. I really fancied writing about organizing validation, incorporating messaging and asynchronous communication, security, error handling, clustering, architectural refactoring, structuring rich-client user interfaces, amongst others. I can only hope to see some patterns appear for this work in the near future. However due to space, time, and lack of cogitation you won't find them in this book. Perhaps I'll do a second volume someday and get into these topics, or maybe someone else will fill these, and other, gaps.

Of these dealing with message based communication is a particularly big issue. Increasingly people who are integrating multiple applications are making use of asynchronous message based communication approaches. There's much to said for using them within an application as well.

This book is not intended to be specific for any particular software platform. I first came across these patterns while working with Smalltalk, C++, and CORBA in the late 80's and early 90's. In the late 90's I started to do extensive work in Java and found these patterns applied well both to early Java/CORBA systems and later J2EE based work. More recently I've been doing some initial work with Microsoft's .NET platform and find the patterns apply again. My ThoughtWorks colleagues have also introduced their experiences, particularly with Forte. I can't claim generality across all platforms that ever have been or will be used for enterprise applications, but so far these patterns have shown enough recurrence to be useful.

I have provided code examples for most of these patterns. My choice of language for the code examples is based on what I think most readers are likely to be able to read and understand. Java's a good choice here. Anyone who can read C or C++ can read Java, yet Java is much less complex than C++. Essentially most C++ programmers can read Java but not vice versa. I'm an object bigot, so inevitably lean to an OO language. As a result most of the code examples are in Java. As I was working on the book Microsoft started stabilizing their .NET environment, and their C# language has most of the same

properties as Java for an author. So I did some of the code examples in C# as well, although that does introduce some risk since developers don't have much experience with .NET yet and so the idioms for using it well are less mature. Both are C-based languages so if you can read one you should be able to read both, even if you aren't deeply into that language or platform. My aim was to use a language that the largest amount of software developers can read, even if it's not their primary or preferred language. (My apologies to those who like Smalltalk, Delphi, Visual Basic, Perl, Python, Ruby, COBOL or any other language. I know you think you know a better language than Java or C#, all I can say is I do too!)

## Who this book is for

I've written this book for programmers, designers, and architects who are building enterprise applications and who want to either improve their understanding of these architectural issues or improve their communication about them.

I'm assuming that most of my readers will fall into two groups: either those with modest needs who are looking to build their own software to handle these issues, or readers with more demanding needs who will be using a tool. For those of modest needs, my intention is that these patterns should get you started. In many areas you'll need more than the patterns will give you, but my intention is to provide more of a head start in this field than I got. For tool users I hope this book will be useful to give you some idea of what's happening under the hood, but also help you in making choices between which of the tool supported patterns to use. Using, say, an object-relational mapping tool still means you have to make decisions about how to map certain situations. Reading the patterns should give you some guidance in making the choices.

There is a third category, those with demanding needs who want to build their own software for these problems. The first thing I'd say here is look carefully at using tools. I've seen more than one project get sucked into a long exercise at building frameworks which weren't what project was really about. If you're still convinced, go ahead. Remember in this case that many of the code examples in this book are deliberately simplified to help understanding, and you'll find you'll need to do a lot tweaking to handle the greater demands that you'll face.

Since patterns are common solutions to recurring problems, there's a good chance that you'll have already come across some of them. If you've been working in enterprise applications for a while, you may well know most of them. I'm not claiming to have anything new in this book. Indeed I claim the opposite - this is a book of (for our industry) old ideas. If you're new to this field I hope you'll like this book to help you learn about these techniques. If you're more familiar with the techniques I hope you'll like this book because it helps you communicate and teach these ideas to others. An important part of patterns is trying to build a common vocabulary, so you can say that this class is a [\*Remote Facade\*](#) and other designers know what you mean.

## Acknowledgements

As with any book, what's written here has a great deal to do with the many people that have worked with me in various ways over the years. Lots of people have helped me in lots of ways. Often I don't recall important things that people have said that go into this book. But in this section I can acknowledge

those contributions I do remember.

I'll start with my contributors: David Rice and Matt Foemmel, colleagues of mine at ThoughtWorks. Not just have they given me many good ideas that are key to this book, they have also contributed some chapters and examples to the content. Their direct and recent project experience adds a great deal to those sections.

I could almost list the ThoughtWorks telephone directory here, for so many of my colleagues have helped this project by talking over their designs and experiences with me. Many patterns formed in my mind by having the opportunity to the many talented designers that we have, so I have little choice but to thank the whole company.

Kyle Brown, Rachel Reinitz, and Bobby Woolf have gone out of their way to have long and detailed review sessions with me in North Carolina. Their fine tooth-comb has injected all sorts of wisdom, not including this particularly heinous mixed metaphor. In particular I've enjoyed several long telephone calls with Kyle that contributed more than I can list.

As usual I owe more than I can say to my first class panel of official reviewers:

- John Brewer
- John Crupi
- Alan Knight
- Rob Mee
- Gerard Meszaros
- David Siegel
- Kai Yu

Early in 2000 I prepared a talk for Java One with Alan Knight and Kai Yu which was the earliest genesis of this material. As well as thanking them for their help in that, I should also thank Josh Mackenzie, Rebecca Parsons, and Dave Rice for their help refining these talks, and the ideas, later on. Jim Newkirk did a great deal in helping me get used to the new world of .NET.

As I was writing this book, I put drafts on the web. During this time many people sent me emails pointing out problems, asking questions, or talking about alternatives. These people include (in rough order of hearing from them) Ivan Mitrovic, Bjorn Beskow, Daniel Drasin, Eric Kaun, Thomas Neumann, Peter Gassmann, Russell Freeman, Brad Wiemerslage, John Coakley, Mark Bernstein, Chester Chen, Christian Heller, Jesper Ladegaard, Kyle Hermenean, Don Dwiggins, Knut Wannheden, Akira Hirasawa, Volker Termath, Christopher Thames, Bryan Boreham, Michael Yoon, Tobin Harris, Kirk Knoernschild, Matthew Roberts, Joel Rieder, Jeremy Miller, Russel Healey, Pascal Costanza, Paul Campbell, Bob Corrick, Graham Berrisford, Trevor Pinkney, Peris Brodsky, Volker Turau, Dan Green, Ken Rosha, Michael Banks, Paolo Marino, Jason Gorman.

There are many others who've given input whose names I either never knew or can't remember, but my thanks is no less heartfelt.

An important part of my sanity for the code examples in this book was being able to easily run automated tests. Although I'm sure there are errors which got past them, they certainly caught plenty. So I'd like to thank Kent Beck and Erich Gamma for writing JUnit ([junit.org](http://junit.org)), and Philip Craig for his work on NUnit ([nunit.sourceforge.net](http://nunit.sourceforge.net)).

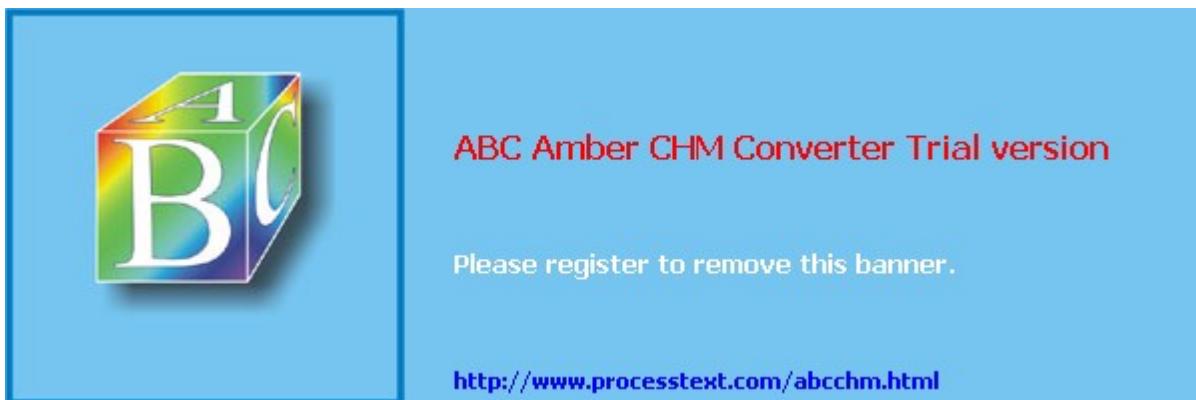
# Colophon

This was the first book that I've written using XML and related technologies. The master text was written as XML documents using trusty TextPad. I used a home grown DTD. While I was working I used XSLT to generate the web pages for the web site. For the diagrams I relied on my old friend Visio using Pavel Hruba's wonderful UML templates (much better than those that come with the tool, I have a link on my web site if you want them.) I wrote a small program that automatically imported the code examples into the output, this saved me from the usual nightmare of code cut and paste. For my first draft I tried XSL-FO with Apache FOP. It wasn't quite up to the job (yet) so for later work I wrote scripts in XSLT and Ruby to import the text into FrameMaker.



---

© Copyright [Martin Fowler](#), all rights reserved

A blue rectangular banner with a white border. On the left side is a 3D cube graphic with the letters 'A', 'B', and 'C' on its faces, colored with a rainbow gradient. To the right of the cube, the text "ABC Amber CHM Converter Trial version" is displayed in red. Below this, in a smaller black font, is the text "Please register to remove this banner." At the bottom right of the banner is the URL "http://www.processtext.com/abcchm.html" in blue.

# Layering

---

Layering is one of the most common techniques that software designers use to break apart a complicated software system. You see it in machine architectures where layers descend from a programming language with operating system calls, into device drivers and CPU instruction sets, into logic gates inside chips. Networking has FTP layered on top of TCP, on top of IP, on top of ethernet.

When thinking of a system in terms of layers, you imagine the principal subsystems in the software arranged in some form of layer cake, where each layer rests upon a lower layer. In this scheme the higher layer uses various services defined by the lower layer, but the lower layer is unaware of the higher layer. Furthermore each layer usually hides its lower layers from the layers above, so layer 4 uses the services of layer 3 which uses the services of layer 2, but layer 4 is unaware of layer 2. (Not all layering architectures are opaque like this, but most are - or rather most are mostly opaque).

Breaking down a system into layers has a number of important benefits

- You can understand a single layer as a coherent whole without knowing much about the other layers. You can understand how to build an FTP service on top of TCP without knowing the details of how ethernet works
- You can substitute layers with alternative implementations of the same basic services. An FTP service can run over ethernet, PPP, or whatever a cable company uses without change.
- You minimize dependencies between the layers. If the cable company changes its physical transmission system, providing they make IP work we don't have to alter our FTP service.
- Layers make good places for standardization. TCP and IP are standards because they define how their layers should operate.
- Once you have a layer built you can use it for many higher level services. So TCP/IP is used by FTP, telnet, SSH, http. Otherwise all of these higher level protocols would have to write their own lower level protocols

Layering is an important technique, but there are downsides.

- Layers encapsulate some things well, but not all. As a result you sometimes get cascading changes. The classic example of this in a layered enterprise application is adding a field that needs to display on the UI, be in the database, and thus be added to every layer in between.
- Extra layers can harm performance. At every layer things typically need to be transformed from one representation to another. However the encapsulation of underlying function often gives you efficiency gains that more than compensate. A layer that controls transactions can be optimized and will then make everything faster.

But the hardest part of a layered architecture is deciding what layers to have and what the responsibility of each layer should be.

# Evolution of layers in enterprise applications

Although I'm too young to have done any work in the early days of batch systems, I don't sense that people thought much of layers in those days. You wrote a program that manipulated some form of files (ISAM, VSAM, etc) and that was your application. No layers present.

The notion of layers became more apparent in the 90's with the rise of **client-server** systems. These were two layer systems: the client held the user-interface and other application code and the server was usually a relational database. Common client tools were VB, Powerbuilder and Delphi. These made it particularly easy to build data-intensive applications as they had UI widgets that were aware of SQL. This allowed you build a screen by dragging controls onto a design area and then using property sheets to connect the controls to the database.

If the application was all about the display and simple update of relational data, then these client-server systems worked very well. The problem came with domain logic: business rules, validations, calculations and the like. Usually people would write these on the client. But this was awkward and was usually done by embedding the logic directly into the UI screens. As the domain logic got more complex, this code became very difficult to work with. Furthermore embedding logic in screens made it easy to duplicate code, which meant that simple changes resulted in hunting down similar code in many screens.

An alternative was to put the domain logic in the database as stored procedures. However stored procedures give limited structuring mechanisms, which again led to awkward code. Also many people liked relational databases because SQL was a standard which would allow them to change their database vendor. Despite the fact that relatively few people actually do this, many more like the freedom to do it without too high a porting cost. Stored procedures are all proprietary, so would remove that option.

At the same time that client-server was gaining popularity, the object-oriented world was rising. The object community had an answer to the problem of domain logic, move to a three layer system. In this approach you have a presentation layer for your UI, a domain layer for your domain logic, and a data source. This way you can move all of that intricate domain logic out of the UI and put into a layer where you can objects to properly work with it.

Despite this the object bandwagon made little headway. The truth was that many systems were simple, or at least started that way. And although the three layer approach had many benefits the tooling for client-server was compelling if your problem was simple. The client-server tools also were difficult, or even impossible, to use in a three layer configuration.

I think the seismic shock here was the rise of the web. Suddenly people wanted to deploy client-server applications with a web browser. However if all your business logic was buried in a rich client then all your business logic needed to be redone to have a web interface. A well designed three layer system could just add a new presentation and be done with it. Furthermore, with Java, we saw an object-oriented language hit the mainstream. The tools that appeared to build web pages were much less tied to SQL, and thus more amenable to a third layer.

When people discuss layering, there's often some confusion over the terms layer and tier. Often the two are used as synonyms. However most people see tier as implying a physical separation. Client-server systems are often described as two-tier systems, and the separation is physical: the client is a desktop

and the server is server. I use layer to stress that you don't have to run the layers on different machines. A domain layer would often run on either a desktop or the database server. In this situation you have two nodes but with three distinct layers. With a local database I could run all three layers on a laptop, but it would still be three distinct layers.

## The Three Principal Layers

So for this book I'm centering my discussion around a layered architecture of three primary layers: presentation, domain, and data source (I'm following the names of [\[Brown et al\]](#)).

**Presentation** logic is about how to handle the interaction between the user and the software. This can be as simple as a command line or a text based menu system. These days it's more likely to be a rich client graphics UI or an HTML based browser UI. (In this book I use **rich client** to mean a windows / swing / fat client style UI, as opposed to an HTML browser.) The primary responsibilities of the presentation is to display information to the user and to interpret commands from the user into actions upon the domain and data source.

**Data source** logic is about communicating with other systems that carry out tasks on behalf of the application. These can be transaction monitors, other applications, messaging systems and so forth. For most enterprise applications the biggest piece of data source logic is a database which is primarily responsible for storing persistent data.

The remaining piece is the **domain logic**, also referred to as business logic. This is the work that this application needs to do for the domain you're working with. It involves calculations based on inputs and stored data, validation of any data that comes in from the presentation, and figuring out exactly what data source logic to dispatch depending on commands received from the presentation.

A single application can often have multiple packages of each of these three subject areas. An application designed to be manipulated by end users through a rich client interface, but also for manipulation though a command line would have two presentations: one for the rich client interface and one for the command line. Multiple data source components may be present for different databases, but particularly for communication with existing packages. Even the domain may be broken into distinct areas which are relatively separate from each other. Certain data source packages may only be used by certain domain packages.

So far I've talked about a user, this naturally raises the question of what happens when there isn't a human being driving the software. This could be something new and fashionable like a web service, or something mundane and useful like a batch process. In this case the user is the client program. At this point it becomes apparent that there is a lot of similarity between the presentation and data source layers in that they both are about connection to the outside world. This is the logic behind Alistair Cockburns Hexagonal Architecture pattern which visualizes any system as a core surrounded by interfaces to external systems. In the hexagonal architecture everything external is fundamentally an outside interface and thus it's a symmetrical view rather than my asymmetric layering scheme.

I find the asymmetry useful, however, because I think there is a useful distinction to be made between an interface that you provide as a service to others, and a your use of someone else's service. Driving down to the core, this is the real distinction I make between presentation and data source. Presentation is an external interface for a service your system offers to someone else, whether it be a complex human or a

simple remote program. Data source is the interface to things that are providing a service to you. I find it useful to think about these differently because the difference in client is alters the way you think about the service.

| Layer        | Responsibilities   |
|--------------|--|
| Presentation | Provision of services, display of information (eg in windows or HTML, handle user request (mouse clicks, keyboard hits, http requests, command line invocations) |
| Domain       | The logic that is the real point of the system   |
| Data source  | Communication with databases, messaging systems, transaction managers, other packages  |

One of the hardest parts of working with domain logic seems to be that people often find it difficult to recognize what is domain logic and what is other forms of logic. A good example of this was when I was told about a system where there was a list of products where all the products that sold over 10% more than they did last month was colored in red. To do this they placed logic in the presentation layer that compared this months sales to last month's sales and if the difference was more than 10% they set the color to red.

The trouble is that's putting domain logic into the presentation. To properly separate the layers you need have a method in the domain layer to indicate if a product has improving sales. This method does the comparison between the two months and returns a boolean value. The presentation layer then simply calls this boolean method, and if true, highlights the product in red. That way the process is broken into its two parts: deciding whether there is something highlightable and choosing how to highlight.

I'm uneasy with being overly dogmatic about this. When reviewing this book, Alan Knight commented that he was "torn between whether just putting that into the UI is the first step on a slippery slope to hell or a perfectly reasonable thing to do that only a dogmatic purist would object to". The reason we are uneasy is because it's both!

## When to separate the layers

Although we can identify the three common responsibility layers of presentation, domain and data source for each of the three examples, each example has its own questions as to how they are separated. If each responsibility layer is quite complex then it makes sense to break these into their own separate modules. An application with complex logic would have distinct package structures for the presentation, domain, and data source. Indeed it would probably have further mediating layers. But simpler systems might not. If all you are doing is viewing and simple data entry then it may well be reasonable to put all the logic in a series of server pages, particularly if you have a tool that makes it easy to write these server pages.

The cut over point for separating these responsibilities isn't one that can be easily defined. I can't say that if your domain logic has complexity greater than 7.4 then you should separate it out from the presentation. (Or I guess I can if I leave the calculation of the complexity as an exercise for the reader.) My practice is to almost always separate the presentation from the domain/data source. The only case I'd break that rule is if the complexity of the domain/data source is close to nothing and I have tools that

make it easy to tie things together. A classic example of this is the many client-server systems done in such environments as Visual Basic or Powerbuilder. They make it very easy to build rich client interfaces on top of SQL databases. Providing the presentation matches very closely to the database structure, and there's hardly any validation or calculation involved, then I'd go ahead and use it. But as soon as validation or calculation starts getting complicated I would look to separating it out into separate objects.

I'm inclined to be more tolerant of separating the domain from the data source. For many simpler applications the two can look quite similar, so then I'm more inclined to keep them together. But as soon as the way I organize my business logic starts looking different from the way the data source is defined I'll want to put them into separate modules.

When describing the layering structure of an application I like to use a UML package diagram to show the packages and their dependencies. I haven't used one here because the actual dependencies depend a great deal on the actual patterns that you are using. One pretty absolute rule is that nothing should depend on the presentation. No module in the domain or data source layers should ever invoke anything on the presentation. This sometimes causes problems when the presentation needs to update on a change in the domain, when this occurs you'll need to use an [Observer](#) to avoid a dependency into the presentation.

Other than that we need to go into the patterns in more detail before we can sketch out any dependencies.

## Choosing where to run your layers

For most of this book I'm talking about logical layers: dividing a system into separate pieces to reduce the coupling between different parts of a system. Separation between layers is useful even if they are all running on one physical machine. But there are places where the physical structure of a system makes a difference.

For most IS applications the decision is whether to run processing on a client, desktop machine, or whether to run processing on a server.

Often the simplest case is to run everything on servers. Using a HTML front end which uses a web browser is a good way to do this. The great advantage of running everything on the server is that everything is easy to upgrade and fix because it's in a limited amount of places. You don't have to worry about deployment to many desktops and keeping them all in sync with the server. You don't have to worry about compatibilities with other desktop software.

The general argument in favor of running on a client turns on responsiveness or disconnected operation. Any logic that runs on the server needs a server round trip to respond to anything the user does. If the user wants to quickly fiddle with things and see immediate feedback that round trip gets in the way. It also needs a network connection to run. The network may like to be everywhere, but as I type this it isn't at 31,000 ft. It may be everywhere soon, but there's people who want to do work now and not wait for wireless coverage to reach Dead End Creek. Disconnected operation brings particular challenges, and I'm afraid I decided to put those out of the scope of this book.

With those general forces in place we can then look at the options layer by layer.

The data source pretty much always will only run on servers. The exception is where you might duplicate server functionality onto a suitably powerful client. Usually this case appears when you want disconnected operation. In this case changes to the data source on the disconnected client need to be synchronized with the server. As I mentioned earlier, I decided to leave those issues to another day - or another author.

The decision of where to run the presentation depends mostly on what kind of user interface you want. Running a rich client pretty much means running the presentation on the client. Running a web interface pretty much means running on the sever. There are exceptions, remote operation of client software (such as X servers in the Unix world) running a web server on the desktop, but these exceptions are rare.

If you're building a B2C system- you have no choice. Any Tom, Dick or Harriet could be connecting to your servers and you don't want to turn anyone away because they insist on doing their online shopping with a TRS-80. In this case you do all processing on the server and offer up HTML for the browser to deal with. Your limitation with the HTML option is that every bit of decision making needs a round trip from the client to the server and that can hurt responsiveness. You reduce some of that with browser scripting and downloadable applets, but they reduce your browser compatibility and tend to add other headaches. The more pure HTML you can go the easier life is.

That ease of life is appealing even if every one of your desktops is lovingly hand-built by your IS department. Keeping clients up to date and avoiding compatibility errors with other software dogs even simple rich client systems.

The primary reason that people want a rich client presentation is because some tasks are complicated for users to do, and to have a usable application they'll need more than what a web GUI can give. Increasingly, however, people are getting used to ways to make web front ends more usable and that reduces the need for a rich client presentation. As I write this I'm very much in favor of the web presentation if you can, and the rich client if you must.

This leaves us with the domain logic. You can run business logic either all on the server, all on the client, or split it. Again all on the server is the easiest way for ease of maintenance. The demand to move it to the client is either for responsiveness or for disconnected use.

If you have to run some logic on the client you can consider running all of it there - at least that way it's all in one place. Usually this goes hand in hand with a rich client - running a web server on a client machine isn't going to help responsiveness much, although it can be a way to deal with disconnected operation. In this case you can still keep your domain logic in separate modules from the presentation, and indeed you can with either [Transaction Scripts](#) or a [Domain Model](#). The problem with putting all the domain logic on the client is that this way you have more to upgrade and maintain.

Splitting across the both desktop and server sounds like the worst of both worlds, since that way you don't know really where any piece of logic may be. The main reason to do it is when you only have a small amount of domain logic that needs to run on the client. The trick then is to isolate this piece of logic into a self contained module that isn't dependent on any other part of the system. That way you can run that module on either client or server. Doing this will require a good bit of annoying jiggery-pokery - but it's a good way of doing the job.

Once you've chosen your processing nodes, you should then try to keep all the code on one node on a single process. Don't try to separate the layers into separate processes unless you absolutely have to, since doing that will both cause performance degradation and add complexity as you have to add things like [Remote Facades](#) and [Data Transfer Objects](#).

# More Layering schemes

I've picked these three layers because these three subject matters are always present, and I think that the three represent the simplest layering scheme that makes sense for enterprise applications. You can cut the layers differently, and often these other cuts are useful, but I've found these three are always useful to have in your mind while the others are often useful but not always useful.

I'll illustrate this by looking at some layering schemes that have been promoted in some useful books on IS architecture. First up is what I'll call the Brown model which is discussed in [\[Brown et al\]](#). The Brown model has five layers: Presentation, Controller/Mediator, Domain, Data Mapping, and Data Source. Essentially this model places additional mediating layers between the basic three layers. The controller/mediator mediates between the presentation and domain layers, while the data mapping layer mediates between the domain and data source layers.

I find that having the mediating layers is something that's useful some of the time, but not all of the time. As a result I've described mediators in terms of patterns. The [\*Application Controller\*](#) is the mediator between the presentation and domain, and the [\*Data Mapper\*](#) is the mediator between the data source and the domain. For organizing this book I've described [\*Application Controller\*](#) in the presentation section and [\*Data Mapper\*](#) in the data source section.

| Brown               | Fowler  |
|---------------------|---|
| Presentation        | Presentation                                  |
| Controller/Mediator | <a href="#"><i>Application Controller</i></a> |
| Domain              | Domain  |
| Data Mapping        | <a href="#"><i>Data Mapper</i></a>            |
| Data Source         | Data source                                   |

So for me the addition of mediating layers, frequently but not always useful, represents an optional extra in the design. My approach is to always think of the three base layers, look to see if any of them are getting too complex, and if so add the mediating layer to separate the functionality.

Another good layering scheme for J2EE appears in [\*CoreJ2EE patterns\*](#). Here the layers are client, presentation, business, integration, and resource. Simple correspondences exist for the business and integration layers. They refer to resource layer as those external services that the integration layer is connecting to. The main difference is that they split the presentation layer between the part that runs on the client (client) and the part that runs on a server (presentation). This is often a useful split to do, but again it's not one that's needed all the time.

| Core J2EE    | Fowler  |
|--------------|---|
| Client       | Presentation that runs on client (eg rich client systems)             |
| Presentation | Presentation part that runs on server (eg http handers, server pages) |

|             |   |
|-------------|---|
| Business    | Domain  |
| Integration | Data source   |
| Resource    | The external resource that the data source is in communication with |

The [Microsoft DNA](#) architecture defines three layers: presentation, business, and data access. They correspond pretty directly to the three layers I use here. The biggest shift occurs in the way that data is passed up from the data access layers. In Microsoft DNA all the layers operate on record sets that are the results of SQL queries issued by the Data Access layer. This introduces an apparent coupling in that both the business and presentation layers know about the database.

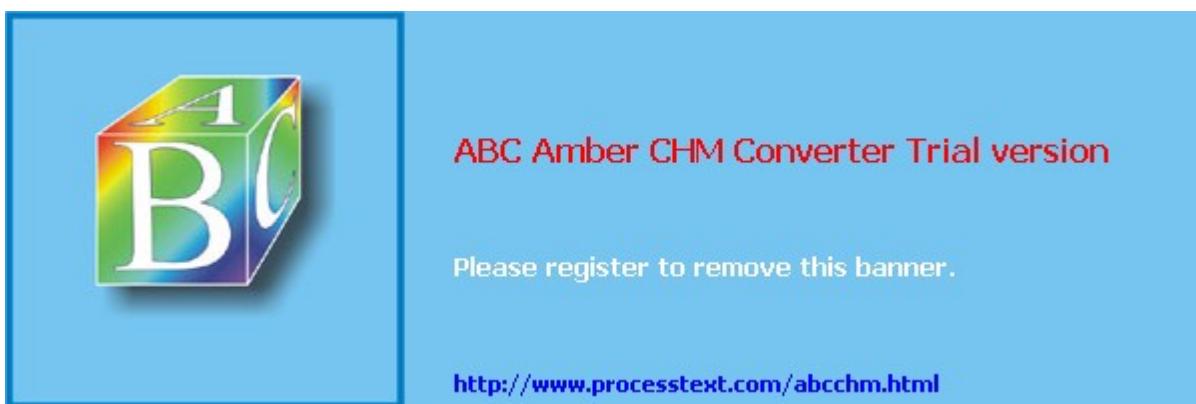
The way I look at this is that in DNA the record set acts as [Data Transfer Object](#) between layers. The business layer can modify the record set on its way up to the presentation, or even create one itself (although that is rarer). Although this form of communication is in many ways unwieldy it has the big advantage of allowing the presentation to use data-aware GUI controls, even on data that's been modified by the business layer.

| Microsoft DNA | Fowler       |
|---------------|--------------|
| Presentation  | Presentation |
| Business      | Domain       |
| Data Access   | Data source  |

In this case the domain layer is structured in the form of [Table Modules](#) and the data source layer uses [Table Data Gateways](#).



© Copyright [Martin Fowler](#), all rights reserved



# Organizing Domain Logic

---

The simplest approach to storing domain logic is the [\*Transaction Script\*](#). A [\*Transaction Script\*](#) is essentially a procedure that takes the input from the presentation, processes it with validations and calculations, stores data in the database, invokes any operations from other systems and replies with more data to the presentation perhaps doing more calculation to help organize and format the reply data. The fundamental organization is of a single procedure for each action that a user might want to do. Hence we can think of it being a script for an action, or business transaction. It doesn't have to be a single inlined procedure of code, pieces get separated out into subroutines and these subroutines can be shared between different [\*Transaction Scripts\*](#), but the driving force is still that of a procedure for each action. So a retailing system might have [\*Transaction Scripts\*](#) for checkout, add something to the shopping cart, display delivery status, and so on.

The advantages of a [\*Transaction Script\*](#) include:

- It's a simple procedural model that most developers understand
- It works well with a simple data source layer using [\*Row Data Gateway\*](#) or [\*Table Data Gateway\*](#)
- It's obvious how to set the transaction boundaries: start with opening a transaction and end with closing it. It's easy for tools to do this behind the scenes.

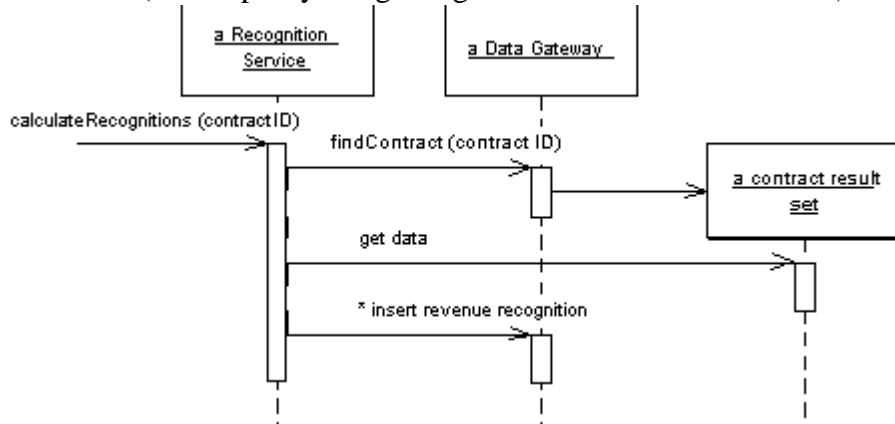
Sadly there are also plenty of disadvantages, these disadvantages tend to appear as the complexity of the domain logic increases. Often there will be duplicated code as several transactions need to do similar things. Some of this can be dealt with by factoring out common subroutines, but even so much of the duplication is tricky to remove and harder to spot. The resulting application can end up being quite a tangled web of routines without a clear structure.

Of course complex logic is where objects come in, and the object-oriented way to handle this problem is with a [\*Domain Model\*](#). For a [\*Domain Model\*](#) the primary organization is of building up a model of our domain organized primarily about the nouns in the domain. So a leasing system would have classes for lease, asset, etc. The logic for handling validations and calculations would be placed into this domain model. A shipment object might contain the logic to calculate the shipping charge for a delivery. There might still be routines for calculating a bill, but such a procedure would quickly delegate to a method in the [\*Domain Model\*](#).

Using a [\*Domain Model\*](#) as opposed to a [\*Transaction Script\*](#) is the essence of the paradigm shift that object-oriented people talk about so much. Rather than one routine having all the logic for a user action, each object takes a part of the logic that's relevant to the object. If you're not used to a [\*Domain Model\*](#) then learning to work with it can be very frustrating as you chase from object to object trying to find where the behavior is.

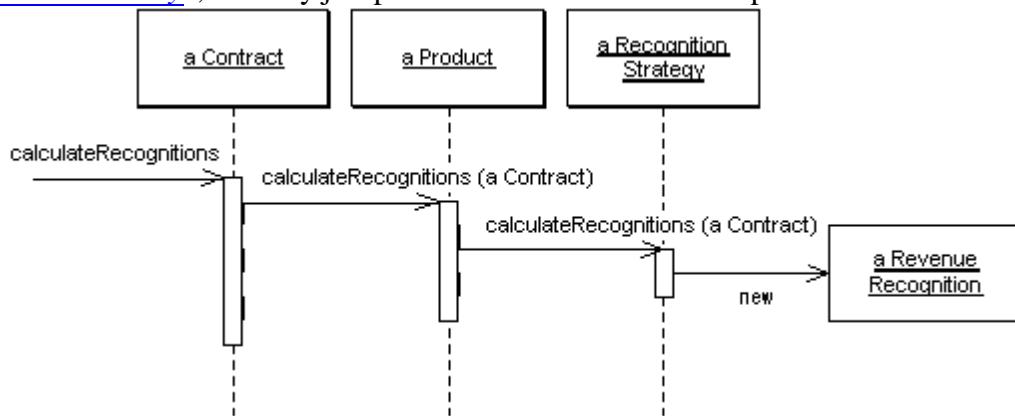
It's hard to capture the essence of the difference with a simple example, but in the discussions of the

patterns I've tried to do that by building a simple piece of domain logic both ways. The easiest way to see the difference is by looking at the sequence diagrams for the two approaches. The essential problem is that different kinds of product have different algorithms for figuring out how to recognize revenue on some given contract. The calculation method has to determine what kind of product a given contract is for, apply to correct algorithm, and then create revenue recognition objects to capture the results of the calculation. (For simplicity I'm ignoring the database interaction issues.)



*Figure 1: A transaction script's way of calculating revenue recognitions*

In Figure 1 the [Transaction Script](#)'s method does all the work. The underlying objects are just [Table Data Gateways](#), and they just pass data to the transaction script.



*Figure 2: A domain model's way of calculating revenue recognitions*

Contrastingly, in the Figure 2 we see multiple objects each forwarding part of the behavior to another, until a strategy object creates the results.

The value of a [Domain Model](#) lies in the fact that once you've got used to things, there are many techniques that allow you to handle more and more complex logic in a well organized way. As we get more and more algorithms for calculating revenue recognition, we can add these by adding new recognition strategy objects. With the [Transaction Script](#) we are adding more conditions to the conditional logic of the script. Once your mind is as warped to objects as mine is, you'll find you prefer a [Domain Model](#) even in fairly simple cases.

The costs of a [Domain Model](#) come from the complexity of using it and complexity of your data source layer. It takes time for people new to rich object models to get used to using a rich [Domain Model](#). Often developers may need to spend several months working on a project that uses a [Domain Model](#) before their paradigms are shifted. However once you've got used to your first [Domain Model](#) you're usually infected for life and it becomes easy to work with them in the future - that's how object bigots

like me are made. However a significant minority of developers seem to be unable to make the shift.

Even once you've made the shift, you still have to deal with the database mapping. The richer your [Domain Model](#), the more complex your mapping to a relational database (usually using [Data Mapper](#)). A sophisticated data source layer is much like a fixed cost, it takes a fair amount of money (if you buy one) or time (if you build one) to get a good data source layer together, but once you have you can do a lot with it.

There is a third choice for structuring domain logic - [Table Module](#). At very first blush the [Table Module](#) looks like a [Domain Model](#) since both will have classes for contracts, products and revenue recognitions. The vital difference is that whilst a [Domain Model](#) has one instance of contract for each contract in the database, a [Table Module](#) has only one instance. A [Table Module](#) is designed to work with [Record Set](#). So the client of a contract [Table Module](#) will first issue queries to the database to form a [Record Set](#), then it will create a contract object and pass it the [Record Set](#) as an argument. The client can then invoke operations on the contract to do various things (Figure 3). If it wants to do something to an individual contract it must pass in an id.

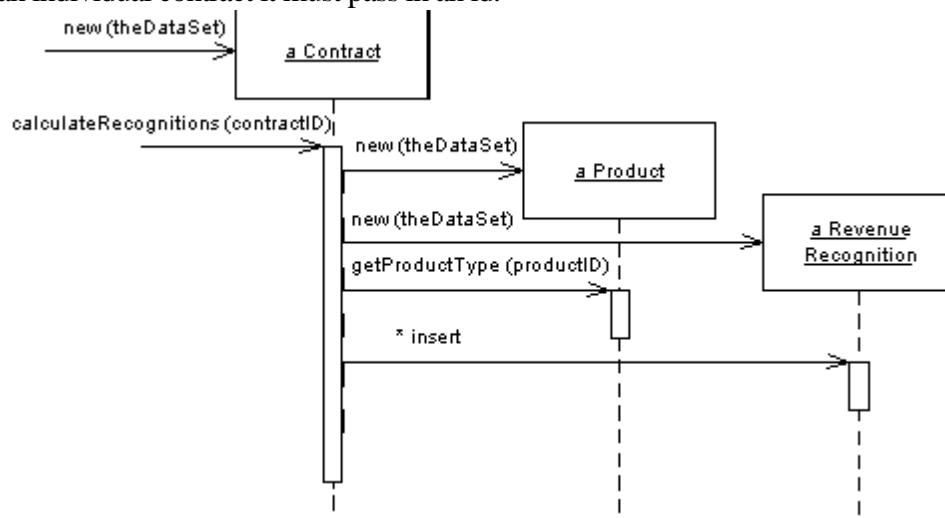


Figure 3: Calculating revenue recognitions with a [Table Module](#)

A [Table Module](#) is in many ways a middle ground between a [Transaction Script](#) and a [Domain Model](#). Organizing the domain logic around tables provides more structure than straight procedures and makes it easier to find and remove duplication. However you can't use a number of the techniques that a [Domain Model](#) can use for finer grained structure of the logic: such as inheritance, strategies and other OO patterns.

The biggest advantage of a [Table Module](#) is how it fits into the rest of the architecture. Many GUI environments are built to work on the results of a SQL query organized in a [Record Set](#). Since a [Table Module](#) also works on a [Record Set](#) you can easily run a query, manipulate the results in the [Table Module](#) and pass the manipulated data to the GUI for display. You can also use the [Table Module](#) on the way back for further validations and calculations. A number of platforms, particularly Microsoft's COM and .NET use this style of development.

So far I've talked about the three styles of domain logic as exclusive alternatives. While it helps to explain them by showing them in a pure form, applications commonly mix [Transaction Scripts](#) with [Domain Model](#). (Other mixes are possible, but are rarer.) In a mixed approach the question is how much behavior to put in the script and how much in the domain objects. It's a continuum, but three useful points to discuss are a dominant [Transaction Script](#) a dominant [Domain Model](#) and the

controller-entity mix.

With a dominant *Transaction Script* you have most of your domain logic in *Transaction Scripts*, and you have some common behavior in relatively simple domain objects. Since the domain objects are simple, they usually map one-to-one with the database and thus you can use a simpler data source layer such as *Active Record*.

A dominant *Domain Model* will have most logic in domain objects with just some coordination code in *Transaction Scripts*. The middle way here is the **controller-entity** style the names of which come from a common practice influenced heavily by [Jacobson et al]. The point here is to have any logic that's particular to a single transaction or use-case placed in *Transaction Scripts*, which are commonly referred to as controllers. These are different controllers to the input controller in *Model View Controller* or the *Application Controller* so I use the term **use-case controller**. Behavior that's used in more than one use case goes on the domain objects which are called entities.

Although the controller entity approach is a common one, it's not one that I've ever liked much. The use-case controllers, like any *Transaction Script* tend to encourage duplicate code. My view is that if you decide to use a *Domain Model* at all you really should go the whole hog and make it dominant. The one exception to this is if you've started with a design that uses *Transaction Script* with *Row Data Gateway*, then it makes sense to move duplicated behavior on to the *Row Data Gateways*. This will turn them into a simple *Domain Model* using *Active Record*. But I would only do that to improve a design that's showing cracks, I wouldn't start that way.

So how do you choose between the three? It's not an easy choice, but it very much depends on how complex your domain logic is. Figure 4 is one of these non-scientific graphs that really irritate me in powerpoint presentations as it has utterly unquantified axes. However it helps to visualize my sense of how the three compare. With simple domain logic the *Domain Model* is less attractive because the cost of understanding it and the complexity of the data source add a lot of effort to developing it that doesn't get paid back. However as the complexity of the domain logic increases, then the other approaches tend to hit a wall where adding more features gets exponentially more difficult.

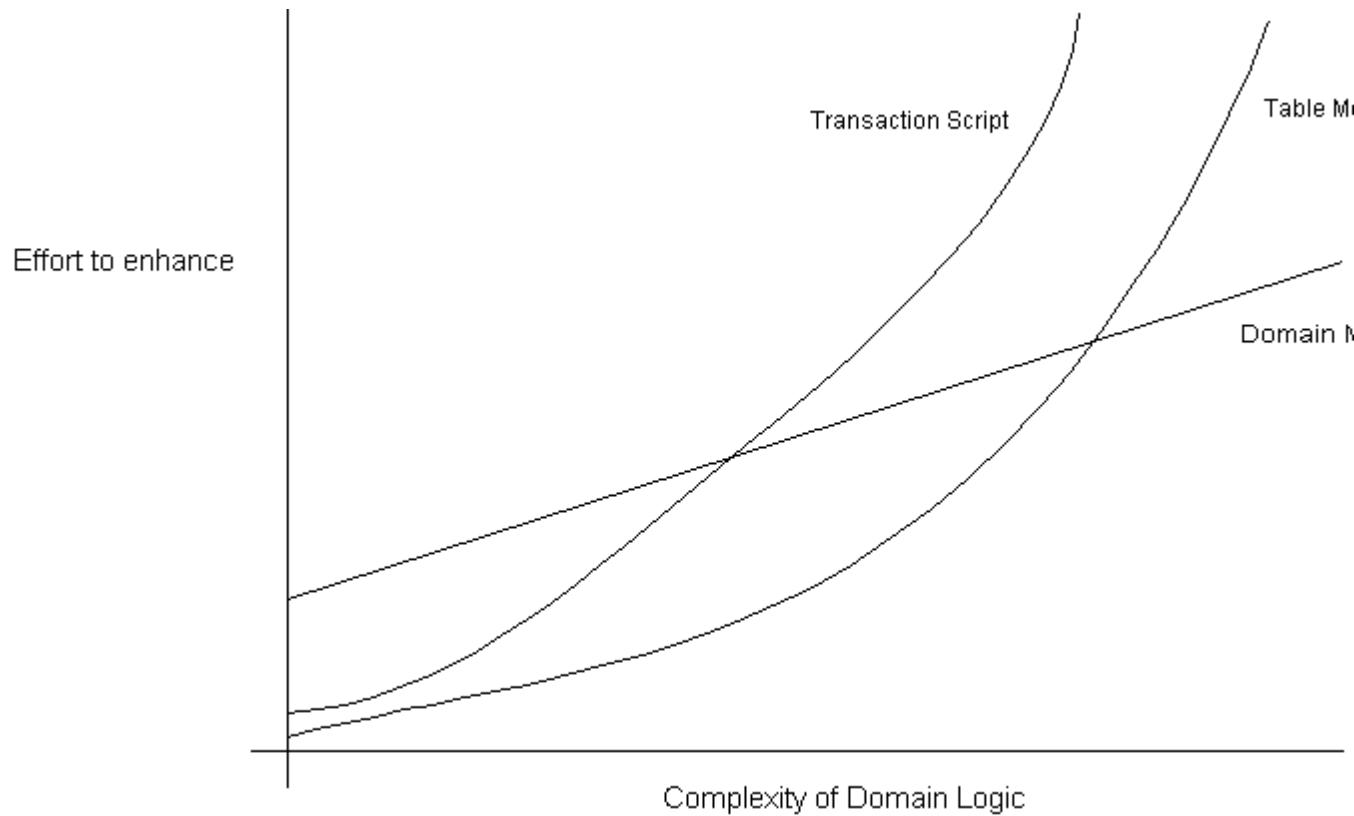


Figure 4: A sense of the relationships between complexity and effort for different domain logic styles

Your problem, of course, is to figure out where on that x axis your application lies. Sadly there's no good way of measuring that, so your only alternative is to find some experienced people who can make an initial analysis of the requirements and make a judgement call.

There are some factors that alter the curves a bit. TA team that's familiar with [Domain Model](#) will lower the initial cost of using a [Domain Model](#). It won't lower it to same starting point as the others, because of the data source complexity, but the better the team is, the more I'm inclined to use a [Domain Model](#).

The attractiveness of a [Table Module](#) depends very much on the support for a common [Record Set](#) structure in your environment. If you have an environment like .NET and Visual Studio where lots of tools work around a [Record Set](#) then that makes a [Table Module](#) much more attractive. Indeed I don't see a reason to use [Transaction Scripts](#) in a .NET environment. However if there's no special tooling for [Record Sets](#), then I wouldn't bother with [Table Module](#).

Once you've made your decision it isn't completely cast in stone, but it is more tricky to change. So it's worth up front thought to decide which way to go. If you find you did go the wrong way, then if you started with [Transaction Script](#) don't hesitate to refactor towards [Domain Model](#). If you started with [Domain Model](#) however, then going to [Transaction Script](#) is usually less worthwhile unless you can simplify your data source layer.

## The interface to the domain logic

When you have server domain logic that has to communicate with the client, an important question is what kind of interface should the server have? The main choice here is whether to have an http based interface or a object-oriented interface.

The most common case of an http interface is that to a web browser, and of course if you have an HTML presentation that's your main option. Http based interfaces can be used for more than just web presentations. The obvious recent candidate is the web service, which is communication between systems over http, typically using XML. XML based http communication is handy for several reasons. It easily allows a lot of data to be sent, in a structured form, in a single round trip. Since remote calls need to be minimized, that's a good thing. The fact that XML is a common format with parsers available in many platforms allows systems built on very different systems to communicate, as does the fact that http is pretty universal these days. The fact that XML is textual makes it easy to see what's going across the wire. Http is also easy to get through firewalls while security and political reasons often make it difficult to open up other ports.

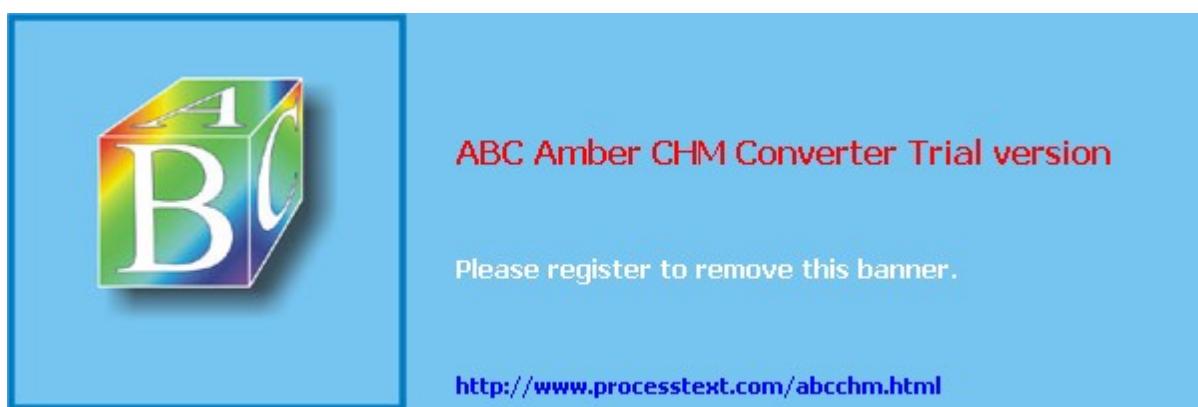
Despite these reasons, however, an object-oriented interface of classes and methods has value too. An http interface isn't very explicit. It's difficult to see exactly what requests can be processed with what data. An object-oriented interface has methods with names that take parameters with names. This makes it much easier to see what can be done with the interface. Communications that don't go through a firewall can be made much more explicit by using such an interface. The methods also provide good spots to provide controls such as security and transactions.

Of course you can have the best of both worlds by layering an http interface over a object-oriented interface. All calls to the web server are translated by the web server into calls on an underlying object oriented interface. To an extent this gives you the best of both worlds, but it does add complexity since you'll need both the web server and the machinery for a remote OO interface. You should only do this if you need both an http and a remote OO API, or if the facilities of the remote OO API for security and transaction handling make it easier to deal with these issues than using non-remote objects.

In any case two patterns you need to be familiar with for this work are [Remote Facade](#) and [Data Transfer Object](#), I talk more about those in a later chapter.



© Copyright [Martin Fowler](#), all rights reserved



# Web Presentation

---

One of the biggest changes to enterprise applications in the last few years is the rise of web-browser based user interfaces. They bring with them a lot of advantages: no client software to install, a common UI approach, and easy universal access. There's also a lot of environments which make it easy to build a web app.

Preparing a web app begins with the sever software itself. Usually this has some form of configuration file which indicates which URLs are to be handled by which programs. Often a single web server can handle many kinds of programs. These programs are often dynamic and can be added to a server by placing them in an appropriate directory. The web server's job is to interpret the URL of a request and hand over control to a web server program. There are two main forms of structuring a program in a web server: as a script or as a server page.

The script form is a program, usually with functions or methods to handle the http call - examples include CGI scripts and Java servlets. The program text can then do pretty much anything a program can do and the script can be broken down into subroutines, and create and use other services. It gets data from the web page by examining the http request object which is a HTML string. In some environments it does this by regular expression searching of the text string - perl's ease of doing this makes perl a popular choice for CGI scripts. Other platforms, such as Java servlets, do this parsing for the programmer, which allows the script programmer to access the information from the request through a keyword interface: which at least means less regular expressions to mess with. The output of the web server is another HTML string - the response - which the script can write to using the usual write stream operations in the language.

Writing an HTML response through stream commands is uncomfortable for programmers, and nearly impossible for non-programmers who would otherwise be comfortable preparing HTML pages. This led to the idea of server pages, where the program is structured around the returning text page. You write the return page in HTML and insert into the HTML scriptlets of code to execute at certain points. Examples of this approach include PHP, ASP, and JSP.

The server page approach works well when there is little processing of the response, such as "show me the details of album # 1234". Things get a lot more messy when you have to make decisions based on the input: such as a different format for displaying classical or jazz albums.

Because the script style works best for interpreting the request and sever page style works best for formatting a response, there's the obvious option to use a script for request interpretation and a sever page for response formatting. This separation is in fact an old idea which first surfaced in user interfaces with [Model View Controller](#). Combine this with the essential notion that non-presentation logic should be factored out and we have a very good fit for the concepts of [Model View Controller](#).

Model View Controller is a widely referenced pattern, but on that's often misunderstood. Indeed before web apps appeared on the scene most presentation I sat through of Model View Controller would get it wrong. A main reason for the confusion was the use of the word "controller". Controller is used in a number of different contexts, and I've usually found it used in a different way to that described in Model View Controller. As a result I prefer to use the term **input controller** for the controller in Model View Controller.

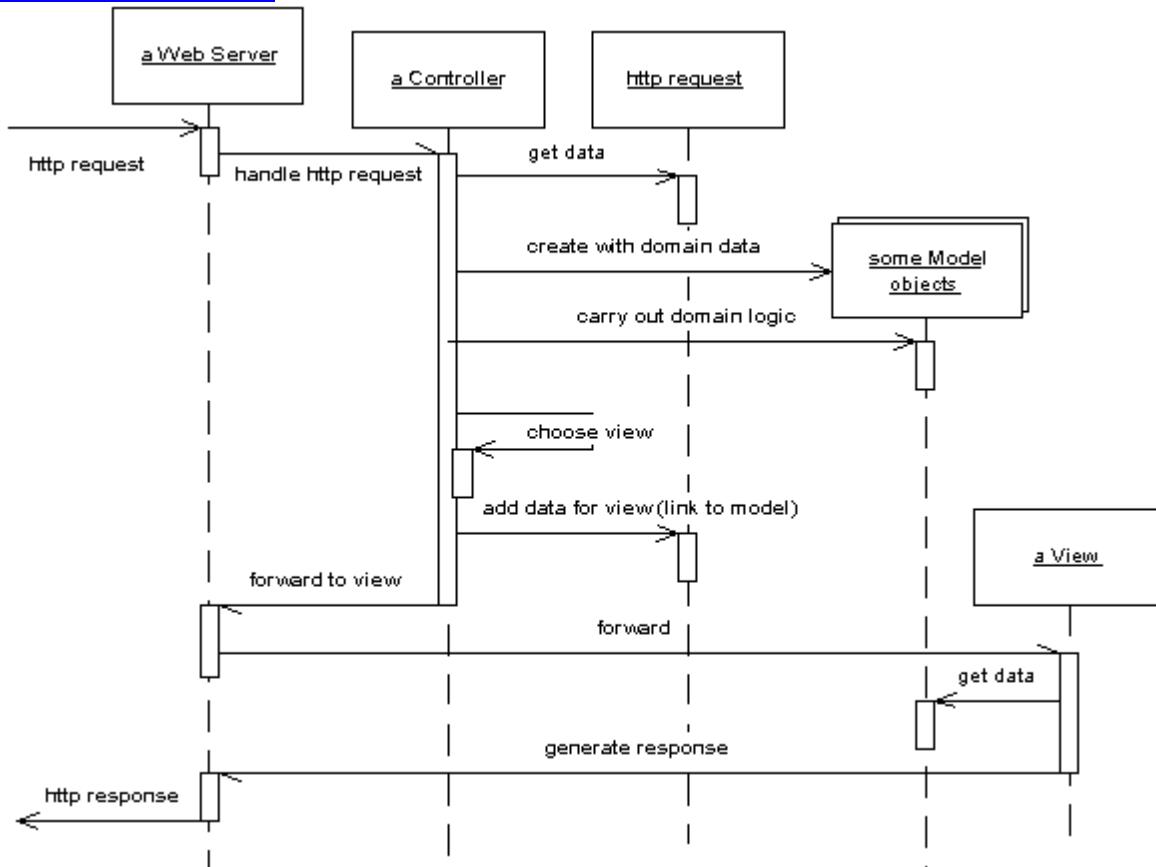


Figure 1: A broad brush picture of how the model, view, and input controller roles work together in a web server. The controller handles the request, gets the model to do the domain logic, then gets the view to create a response based on the model

The request comes in to an input controller, which pulls information off the request. It then forwards the business logic to an appropriate model object. The model object talks to the data source and does everything indicated by the request as well as gathering information for the response. When it's done it returns control to the input controller. The input controller looks at the results and decides which view is needed to display the response. It then passes control, together with the response data, to the view. The input controller's hand off to the view often isn't a straight call, often involves forwarding with the data placed in an agreed place on some form of http session object that's shared between the input controller and the view.

The first, and most important reason for applying Model View Controller is to ensure that models are completely separated from the web services. A good thought experiment for this is to ask yourself how much change you would need to make to existing code to add a command line interface to the program. Separating out the processing into separate Transaction Script or Domain Model objects will make it easier to test them as well. This is particularly important if you are using a server page as your view.

At this point we come to a second use of the word "controller". A lot of user-interface designs look to

separate the presentation objects from the domain objects with an intermediate layer of [Application Controller](#) objects. The purpose of an [Application Controller](#) is to handle the flow of an application, deciding which screens should appear in which order. An [Application Controller](#) may appear as part of the presentation layer, or you can think of it as a separate layer which mediates between the presentation and domain layers. [Application Controllers](#) may be written to be independent of any particular presentation, in which case they can be reused between presentations. This works well if you have different presentations with the same basic flow and navigation, although often it's best to give different presentations a different flow.

Not all systems need an [Application Controller](#). They are useful only if your system has a lot of logic about the order of screens and the navigation between them. If someone can pretty much see any screen in any order, then you'll probably have little need for an [Application Controller](#).

## View Patterns

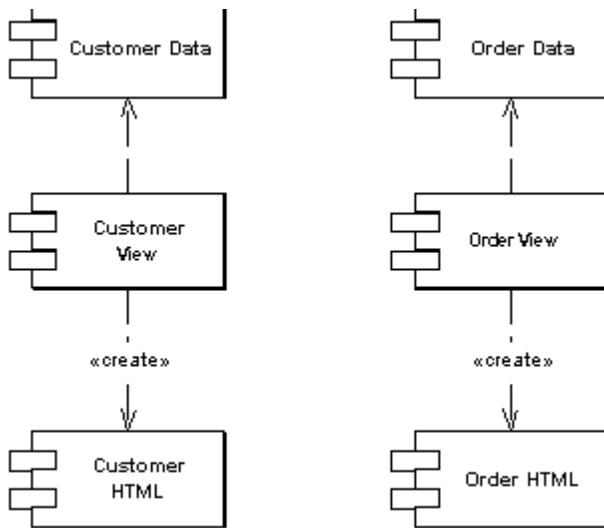
On the view side there are three patterns to think about: [Transform View](#), [Template View](#) and [Two Step View](#). These represent essentially two choices, whether to use [Transform View](#) or [Template View](#), and whether two use one stage or a [Two Step View](#). The basic patterns for [Transform View](#) and [Template View](#) are single stage. [Two Step View](#) is a variation you can apply to either [Transform View](#) or [Template View](#).

I'll start with the choice between [Template View](#) and [Transform View](#). [Template View](#) allows you write the presentation in the structure of the page and embed markers into the page to indicate where dynamic content needs to go. There are quite a few popular platforms that are based on this pattern. Many of the more popular of are the sever pages technologies (ASP, JSP, PHP) that allow you to put a full programming language into the page. This clearly provides a lot of power and flexibility, sadly it also leads to very messy code that's difficult to maintain. As a result if you use server page technology you must be very disciplined to keep programming logic out of the page structure, often by using a companion object.

The [Transform View](#) uses a transform style of program, the usual example is XSLT. This can be very effective if you are working with domain data that is in XML format, or can easily be converted to XML. An input controller picks the appropriate XSLT stylesheet and applies it to XML that's gleaned from the model.

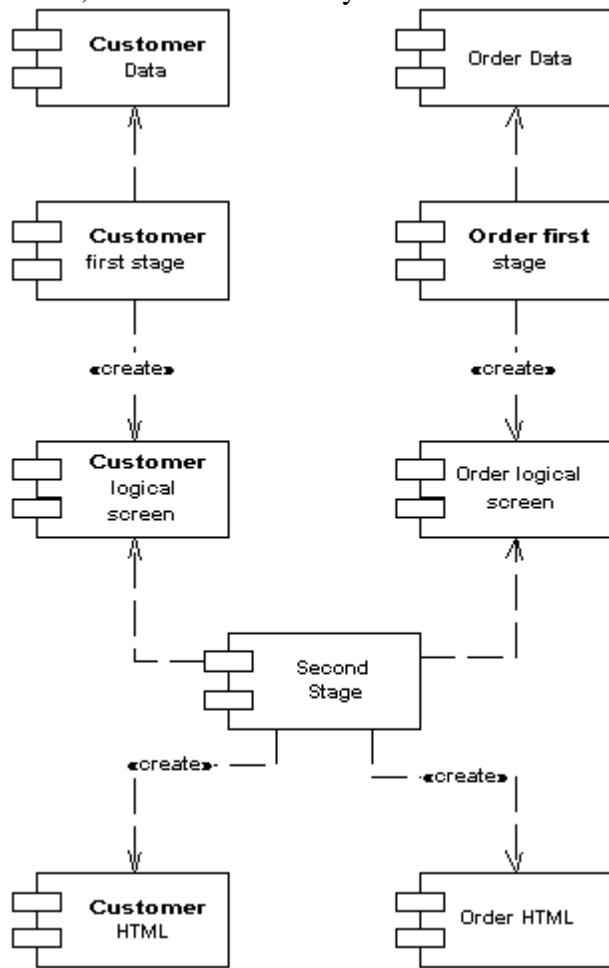
If you use procedural scripts as your view, you can write the code in either [Transform View](#) or [Template View](#) style - or indeed in some interesting mix of the two. I've noticed that most scripts predominantly follow one of these two patterns as their main form.

The second decision is whether to be single stage or use [Two Step View](#)



*Figure 2: A single stage view*

A single stage view mostly has one view component for each screen in the application. The view takes domain oriented data and renders it into HTML. I say mostly because similar logical screens may share views, but most of the time you can think of it as one view per screen.



*Figure 3: A two stage view*

A two stage view breaks this process into two stages, producing a logical screen from the domain data,

then rendering that logical screen into HTML. There is one first stage view for each screen, but only one second stage view for the whole application.

The advantage of the [\*Two Step View\*](#)'s is that it puts the decision of what HTML to use in a single place, this makes global changes to the HTML easy since there's only one object to alter and every screen on the site alters. Of course you only get that advantage if your logical presentation stays the same, so it works best with sites where different screens use the same basic layout. Highly design intensive sites won't be able to come up with a good logical screen structure.

[\*Two Step View\*](#) work even better if you have a web application where its services are being used by multiple front end customers, such as multiple airlines fronting the same basic reservation system. Within the limits of the logical screen, each front end can have a different appearance by using a different second stage.

## Input Controller Patterns

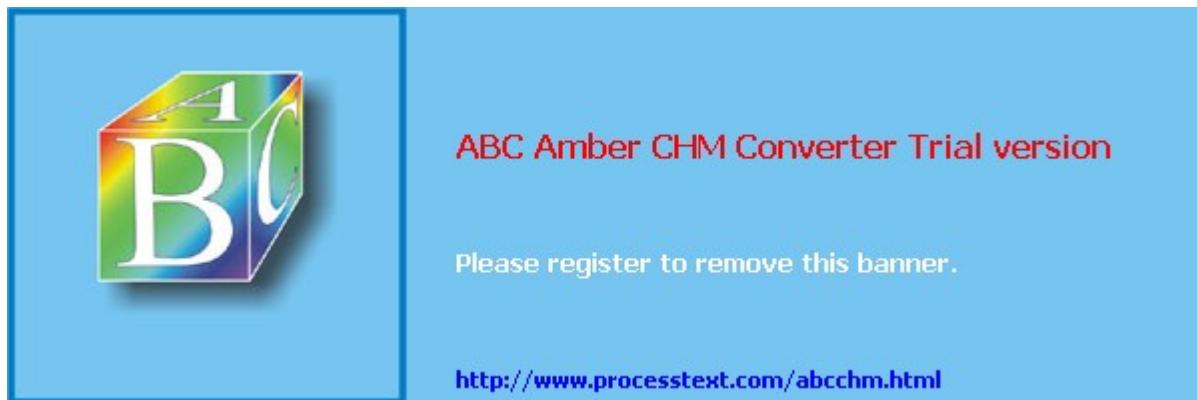
There are two patterns for the input controller. In the most common case you have an input controller object for every page on your web site. In the simplest case this [\*Page Controller\*](#) can be a server page itself: combining the roles of view and input controller. In many implementations it makes things easier to split out the input controller into a separate object. The input controller then can create appropriate models to do the processing and instantiate a view to return the result. Often you'll find there isn't quite a one-to-one relationship between [\*Page Controllers\*](#) and views. A more precise thought is that you have a [\*Page Controller\*](#) for each action: where an action is a button or link. Most of the time the actions correspond to pages, but occasionally they don't - such as a link that may go to a couple of different pages depending some condition.

With any input controller there are two responsibilities: handling the http request and deciding what to do with it. Often it makes sense to separate these two roles. You can use a server page to handle the request, it then delegates to a separate helper object to decide what to do with it. [\*Front Controller\*](#) goes further in this separation by having only one object that handles all requests. This single handler interprets the URL to figure out what the kind of request it's dealing with and creates a separate object to process the request. This allows you to centralize all http handling within a single object avoiding the need to reconfigure the web server whenever you change the action structure of the site.

## Further Reading

Most books on web server technologies provide a chapter or two on how to produce good designs with them, although these are often buried in the technological descriptions. An excellent chapter to read on Java web design is Chapter 9 of [\[Brown et al\]](#). The best source for further patterns is [\[Alur, Crupi, and Malks\]](#), most of these patterns can be used in non-Java situations. I stole the terminology on separating input and application controllers from [\[Knight and Dai\]](#)

© Copyright [Martin Fowler](#), all rights reserved



# Mapping to a Relational Database

---

From the earliest days of using objects with business software, I've wrestled with the issues of how to make objects interact with a relational database. The problem stems from the fact they lead to quite different data structures. Relational databases are table oriented: visualizing data as a collection of one-dimensional tables which can be manipulated through relational calculus. Objects include collection data structures such as lists and maps, use inheritance and direct links between objects. On top of this we have the fact that objects are created and modified in memory and we have to coordinate what happens in memory with what is happening on disk. Since the database is usually a separate process, we have to do this with as few remote calls as possible.

The first question to ask in relational mapping is whether you can avoid it at all. During the early days of objects many people realized that there was a fundamental "impedance mismatch" between objects and relations. So there followed a spate of effort on object-oriented databases, which essentially brought the OO paradigm to disk storage. With an OO database you don't have to worry about mapping. You work with a large structure of interconnected objects, and the database figures out when to move objects on or off disks. You can use transactions to group together updates and permit sharing of the data store. To programmers it seems like an infinite amount of transactional memory that's transparently backed by disk storage.

The chief advantage of using OO databases is that it improves productivity. Although I'm not aware of any controlled tests, anecdotal sayings put the effort of mapping to a relational database as around a third of programming effort - a cost that you have to continue to pay during maintenance.

However most projects don't use OO databases. The primary reason against OO databases is risk. Relational databases are a well understood and proven technology backed by big vendors who have been around a long time. SQL provides a relatively standard interface for all sorts of tools. (If you're concerned about performance all I can say is that I haven't seen any conclusive data comparing performance of OO and relational systems.)

So if you're still here I'll assume you're needing to work with a relational database. Indeed even if you are able to use an OO database as your primary persistence mechanism, you'll probably still need to talk to existing relational databases.

Any project that works with a relational database needs to decide whether to purchase a tool for the purpose. These tools tend to be expensive, but have a lot of valuable features. Books like this can't keep up with technology fast enough to provide any guidance on specific tools, but reading this chapter and its associated patterns will give you an appreciation of the many issues in object-relational mapping. If you

decide to roll your own solution, they will give you a head start. If you decide to purchase a tool, they will help you decide how to use the various features the tool gives you, as well as some appreciation for what goes on under the hood.

The tools I'm familiar with are designed to work with a [Domain Model](#) which is the most complex case. I've seen several projects build their own layer to do this, and most people have badly underestimated the amount and difficulty of the work required to do it. So if you are using a [Domain Model](#) I definitely advise you to look seriously at a tool. Although they may look expensive, you have to factor that against the considerable labor cost of doing it yourself, and maintaining it later on.

## Architectural Patterns

In this book I divide the basic choice into four:[Active Record](#), [Row Data Gateway](#), [Table Data Gateway](#), and [Data Mapper](#).

The starting point for the four are [Row Data Gateway](#) and [Table Data Gateway](#) which are both based on [Gateway](#). In both cases you have an in-memory class which maps exactly to a table in the database. For each table you have a class with one field per column in the database. The gateways and the tables are thus *isomorphic*, they have the same form. The other characteristic about gateways is that they contains all the database mapping code for an application, that is all the SQL, which keeps the SQL out of any other code. The final, and equally important point is that it contains no domain logic. The gateway is purely there for database access, nothing else.

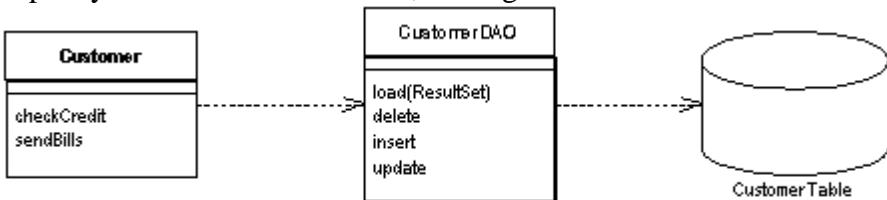


Figure 1: A data access object acts as a [Gateway](#) for the database tables

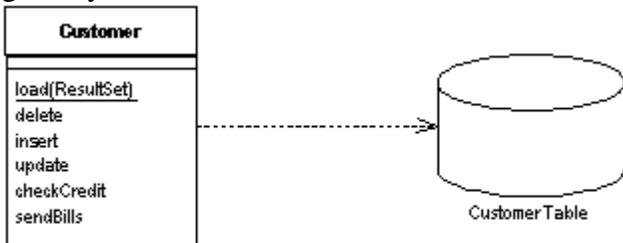
The choice between [Row Data Gateway](#) and [Table Data Gateway](#) depends upon your domain logic and also on how strongly record sets fit into the architecture of your application. Microsoft architectures, for instance, use record sets widely. Much of the UI machinery is based on manipulating record sets. In this environment, where record sets are widely used and easy to program with, a [Table Data Gateway](#) makes a lot of sense. It is the natural choice to work with a [Table Module](#).

A [Row Data Gateway](#) is a good choice when the result sets are more difficult to work with, because you get a real object for each record in the data source. I often use [Row Data Gateway](#) with [Transaction Script](#) designs where I find the record set constructs to be less easy to manipulate than I would like.

Even for simple applications I tend to one of the gateway patterns - as a glance at my ruby and python scripts would confirm for you. I find the clear separation of SQL and procedural logic to be very helpful.

If your application has a [Domain Model](#) and that domain model looks very much like the database model, then you can consider [Active Record](#). Essentially [Active Record](#) combines the gateway and the domain object into a single class that includes both database access and business logic. That's why it's a

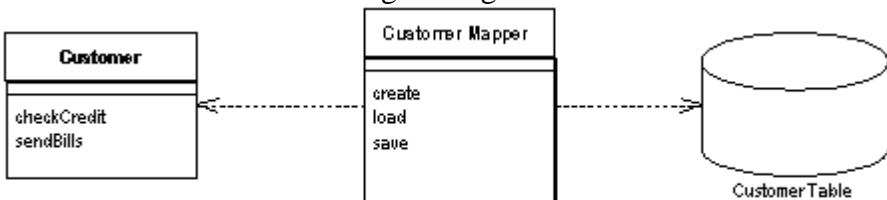
common pattern, and also why I don't like using it. Instead I prefer to keep those responsibilities separate. If your business logic is pretty simple and a [Domain Model](#) that's isomorphic with the database schema makes sense, then you might consider it. But I almost always prefer a separate gateway class.



*Figure 2: In the [Active Record](#) a customer domain object knows how to interact with database tables.*

The biggest exception to this is when you're refactoring a [Transaction Script](#) and [Row Data Gateway](#) design to move duplicate domain logic out of the [Transaction Script](#). It makes sense to put that logic into the [Row Data Gateway](#) thus turning them into [Active Records](#). If the domain logic isn't too complicated then this can be a reasonable move, better than having duplicate logic in the [Transaction Scripts](#), but as the domain logic gets more complicated it's best to look at altering the domain logic design to a proper [Domain Model](#) or a [Table Module](#).

Our third choice is the [Data Mapper](#). This is the most complex, but also most flexible choice. The big difference between it and two data gateways is the inversion of dependency and control. With a data gateway the domain logic has to know about the structure of the data in the database, even if it doesn't have to deal with SQL. In [Data Mapper](#) the domain objects can be completely ignorant of the database layout and when and how data gets moved from the database to the domain model. This makes it easier to write and test the domain objects. It also makes it easier to change either the [Domain Model](#) or the database schema without forcing a change on the other.



*Figure 3: A [Data Mapper](#) insulates the domain objects and the database from each other.*

The driving choice is how you organize your domain logic. If you use [Table Module](#) then [Table Data Gateway](#) is the axe of choice. If you have a domain model that is reasonably complex then I'd pick [Data Mapper](#). [Active Record](#) is a contender if the domain logic is very simple and you have control over the schema, but even then I'd tend to use [Table Data Gateway](#). With a [Transaction Script](#) I would use [Table Data Gateway](#) if the platform makes it convenient to use record sets and [Row Data Gateway](#) otherwise.

These patterns aren't entirely mutually exclusive. In much of this discussion we are thinking of the primary persistence mechanism. By this we mean that if you have some kind of in memory model, your primary persistence mechanism is how you save the data in that model to the database. For that you'll pick one of these patterns, you don't want to mix them as that ends up getting very messy. However even if you're using [Data Mapper](#) as your primary persistence mechanism you may use a data [Gateway](#) to wrap

tables or services that are being treated as external interfaces.

A further separation of SQL that I've seen a few times is to embed the SQL code into separate files as strings, and getting the database interaction classes to pull in the strings when they are needed. This allows people who only know SQL to write SQL without knowing the host language, it also allows them to format the SQL across multiple lines, which is difficult in many languages. The SQL can easily be put into a simple XML based file which can define the various SQL statements with named tags. If you're considering doing this you should remember that the host classes are still highly coupled to the separated SQL - a change in one is a change in the other. For this reason I don't tend to use this approach unless there's a very strong desire in the team to separate the physical languages, as I prefer to keep the highly coupled items in the same file. But it's not something I find a big problem either, so go ahead and do this if it works for you.

If your database mapping logic isn't too complex, it's relatively straightforward to roll your own. However as the mapping gets more complicated you should start to consider using commercial OR mapping tools. Such tools will do a lot of the hard work for you, at the cost of infiltrating their proprietary nature into your application. Don't under-estimate the complexity of building a powerful OR mapping layer, I've come across plenty of projects sunk by the size and complexity of the task.

I've also built a few of these layers myself. In these cases the most important thing is to not to try to write a general object-relational mapping mechanism. Write an OR mapping that works only for your application. This will lead you to many simplifications that make the task much easier, and with OR mapping you need all the simplification you can get.

## The Behavioral Problem

When people talk about OR mapping, usually they focus on the structural aspects - how exactly you relate tables to objects. However I've found the hardest part of the exercise is architectural and behavioral aspects. I've already talked about the main architectural approaches - the next thing to think about is the behavioral problem.

The behavioral problem is how to get the various objects to load and save themselves to the database. At first sight this doesn't seem much of a problem. A customer object can have load and save methods that do this task. Indeed with [Active Record](#) this is an obvious route to take.

However as you start to deal with more complicated situations various problems rear their ugly head. If you're using [Data Mapper](#) the domain objects cannot make calls to the mapping layer since they have no visibility to it. This isn't a tricky problem to solve.

If you load in a bunch of objects to memory and modify them, you have to keep track of which one's you've modified and make sure to write them all back out to the database. If you only load a couple of records this is easy. As you load more and more objects it gets to be more of an exercise. This is particularly the case when you create some rows and modify others, since you'll need the keys from the created rows before you can modify the rows that refer to them. This is a slightly tricky problem to solve.

As you read objects and modify them, you have to ensure that the database state that you are working with stays consistent. If you read some objects, it's important to ensure the reading is isolated so that no

other process changes any of the objects you've read while you are working on them. Otherwise you could have inconsistent and invalid data in your objects. This is the whole issue of concurrency, which is the topic of the [next chapter](#).

A pattern that's an essential part to solving both of these problems is [\*Unit of Work\*](#). A [\*Unit of Work\*](#) keeps track of all objects read from the database, together with all of the objects are modified in any way. It also handles how the updates occur to the database. Instead of the application programmer invoking explicit save methods, the programmer tells the unit of work to commit. The unit of work then sequences all the appropriate behavior to the database. This puts all of the complex commit processing in one place. The [\*Unit of Work\*](#) is an essential pattern whenever the behavioral interactions with the database become awkward.

A good way of thinking about [\*Unit of Work\*](#) is to think of it as an object that acts as the controller of the database mapping. Without a [\*Unit of Work\*](#), typically the domain layer acts as the controller: deciding when to read and write to the database. The [\*Unit of Work\*](#) results from factoring the database mapping controller behavior out into its own object.

## Reading in Data

Reading data from a database involves issuing SQL queries. SQL is not a difficult language to use for many of the simpler actions that people often do. However putting SQL queries around an application can lead to problems. SQL is decidedly non-standard in practice, so scattering SQL around an application makes it difficult to find So it's useful to wrap SQL queries into the appropriate OR-mapping classes. Not just does this keep the SQL together, it also provides a better interface to queries and makes it easier to isolate SQL for testing.

When reading in data, I like to think of the methods as find methods. These find methods wrap SQL select statements with a method structured interface. So you might have methods such as `find(id)` or `findForCustomer(customer)`. Clearly these finder methods can get pretty unwieldy if you have twenty three different clauses in your select statements, but these are thankfully rare.

Where you put the finder methods depends on the interfacing pattern that's used. If your database interaction classes are table-based, that is you have one instance of the class per table in the database, then you can combine the finder methods in with the inserts and updates. If your interaction classes are row based, that is you have one interaction class per row in the database then this doesn't work. In this case you can make the find operations static, but this will stop you from making the database operations substitutable - which means you can't swap out the database for testing. In this case it makes sense to have separate finder objects.

Whichever approach you use, you need to ensure that you don't load the same object twice. If you do that you'll have two in memory objects that correspond to a single database row. Update them both, and everything gets very confusing. So you use an [\*Identity Map\*](#) to keep track of what's been loaded to avoid loading things twice. If you're using [\*Unit of Work\*](#), it's an obvious place to hold the [\*Identity Map\*](#). Without the [\*Unit of Work\*](#) you can keep the [\*Identity Map\*](#) in the finder implementation.

When reading in data, performance issues can often loom large. This leads to a few rules of thumb.

Always try to pull back multiple rows at once. In particular never do repeated queries on the same table

to get multiple rows. It's almost always better to pull back too much data than too little. (Although you have to be wary of locking too many rows if you're using locking.) So consider a situation where you need to get fifty people that you can identify by primary key in your domain model, but you can only construct a query such that you get two hundred people which from which you'll do some further logic to isolate the fifty you need. It's usually better to use one query that brings back unnecessary rows than to issue fifty individual queries.

Another way to avoid going to the database more than once is to use joins so that you can pull multiple tables back with a single query. The resulting record set looks odd, but can really speed things up.

However if you're using joins, bear in mind that databases are optimized to handle up to three or four joins per query. Beyond that performance suffers, although you can restore a good bit of this with cached views.

But in all cases you should profile your application with your specific database and data. General rules can guide thinking, but your particular circumstances will always have their own variations. Database systems and application servers often have sophisticated caching schemes, and there's no way I can predict what will happen for your application. So set aside time to do performance profiling and tuning.

## Structural Mapping Patterns

Structural mapping patterns are the ones most written about, and most understood. They describe how the data in a relational database maps to data in objects.

The central issue here is the different way in which objects and relations handle links. This leads to two problems. Firstly there is a difference in representation. Objects handle links by storing references, either references held by the runtime of memory managed environments or memory addresses. Relations handle links by forming a key into another table. The second problem is that objects can easily use collections to handle multiple references from a single field, while normalization forces all relation links to be single valued. This leads to reversals of the data structure between objects and tables. An order object would naturally have a collection of line item objects which do not need any reference back to the order. However the table structure is the other way around, the line item must include a foreign key reference to the order, since the order cannot have a multi-valued field.

The way to handle the representation problem is to keep the relational identity of each object as an [\*Identity Field\*](#) in the object, and looking up these values to map back and forth between the object references and the relational keys. It's a tedious process, but not that difficult once you understand the basic technique. When you read objects from the disk you use an [\*Identity Map\*](#) as a look-up table from relational keys to objects. Each time you come across a foreign key in the table, you use the look up table to wire up the appropriate inter-object reference. If you don't have the key in the [\*Identity Map\*](#) you need to either go to the database to get it, or use a [\*Lazy Load\*](#). When saving an object, each time you save the object you save it into the row with the right key. Any inter-object reference is replaced with the target object's id field.

On this foundation the collection handling requires a little more jiggery-pokery. If an object has a collection you need to issue another query to find all the rows that link to the id of the source object (or again you can avoid the query now with [\*Lazy Load\*](#)). Each object that comes back gets created and added to the collection. Saving the collection involves saving each object in the collection and making

sure it has a foreign key to the source object. Messy, but simple when you get the hang of it - which is why some form of metadata based approach becomes an obvious move for larger systems (I'll elaborate on that later).

When you're working with collections a common gotcha is to rely on the ordering within the collection. In OO languages it's common to use ordered collections such as lists and arrays - indeed it often makes it easier to test. But it's very difficult to maintain an arbitrarily ordered collection when saved to a relational database. For this reason it's worth considering using unordered sets for storing collections. Another option is to ensure that whenever you do a collection query decide on a sort order

In some cases referential integrity can make updates more complex. Modern systems allow you to defer referential integrity checking to the end of the transaction. If you have this capability there's no reason not to use it. Otherwise the database will check on every write. In this case you have to be careful to do your updates in the right order. How to do this is out of the scope of this book but one technique is to do a topological sort of your updates. Another is to hard code which tables get written in which order. This technique can sometimes reduce deadlock problems.

*Identity Field* is used for inter-object references that turn into foreign keys, but not all object relationships need to be persisted that way. Small *Value Objects*, such as date ranges and money objects clearly should not be represented as their own table in the database. So instead you take all the fields of the *Value Object* and embed them into the linked object as a *Embedded Value*. Since *Value Objects* have value semantics you can happily create them each time you get a read and you don't need to bother with an *Identity Map*. Writing them out is also easy, just deference the object and spit out its fields into the owning table

You can do this kind of thing on a larger scale by taking a whole cluster of objects and saving them as a single column in a table as a *Serialized LOB*. LOB stands for large object which can either be a binary object, (BLOB) or it can be textual. Serializing a clump of objects as an XML document is an obvious route to take for a hierachic object structure. This way you can grab a whole bunch of small linked objects in a single read. Often databases perform poorly with small highly interconnected objects, where you spend a lot of time making many small database calls. Hierachic structures such as org charts and bills of materials are common cases where a *Serialized LOB* can save a lot of database round trips.

The downside is that SQL isn't aware of what's happening, so you can't make portable queries against the data structure. Again XML may come to the rescue here, allowing you to embed XPath query expressions into SQL calls, although the embedding is largely non-standard at the moment. As a result currently *Serialized LOB* is best used when you don't want to query for the parts of the stored structure.

## Inheritance

In the above hierarchies I'm talking about compositional hierarchies, such as a parts tree, which relational system traditionally do a poor job at. There's another kind of hierarchy that causes relational headaches, that of class hierarchies linked by inheritance. Since there's no standard way to do inheritance in SQL, again we've got a mapping to perform. For any inheritance structure there's basically three options. You can have a one table for all the classes in the hierarchy (*Single Table Inheritance*), one table for each concrete class (*Concrete Table Inheritance*) or one table per class in the hierarchy (*Class Table Inheritance*).

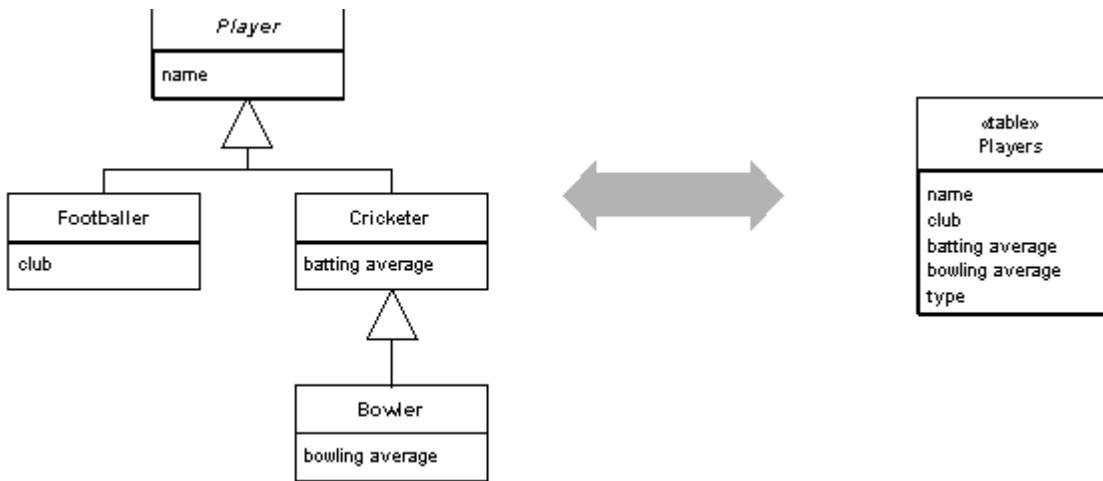


Figure 4: Single Table Inheritance: using one table to store all the classes in a hierarchy

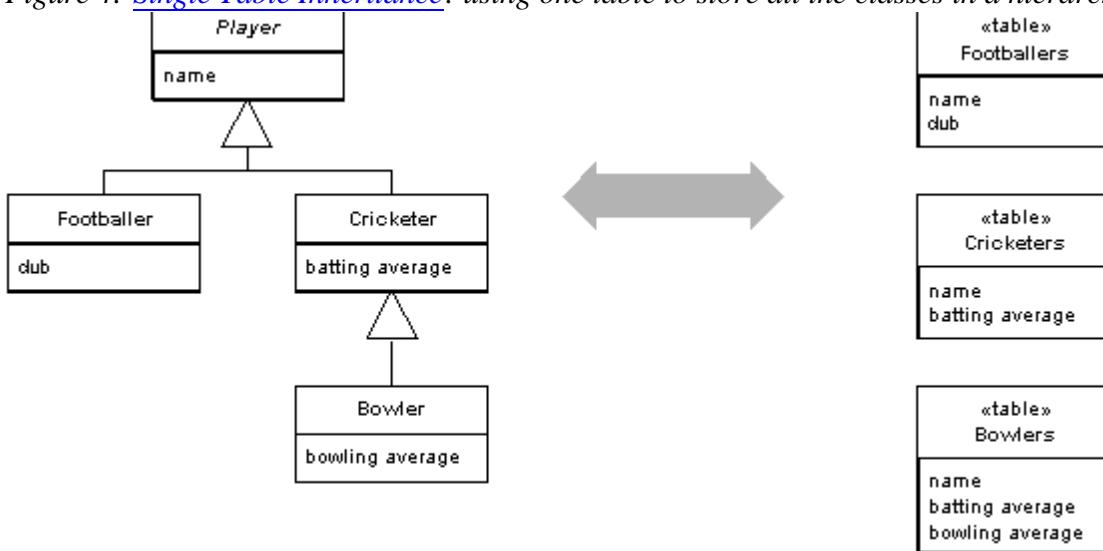


Figure 5: Concrete Table Inheritance: using one table to store each concrete class in a hierarchy

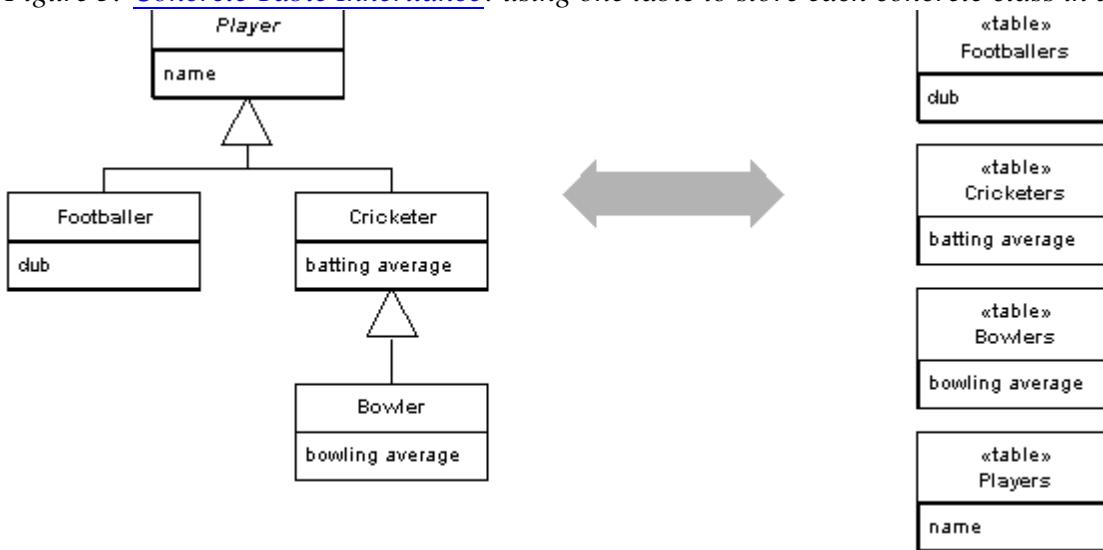


Figure 6: Class Table Inheritance: using a table for each class in a hierarchy

The trade-offs are all between duplication of data structure and speed of access. [Class Table Inheritance](#) is the simplest relationship between the classes and the tables, but needs multiple joins to load a single object, which usually reduces performance. [Concrete Table Inheritance](#) avoids the joins, allowing you pull a single object from one table. But [Concrete Table Inheritance](#) is brittle to changes. Any change to a superclass and you have to remember to alter all the tables (and the mapping code). Altering the hierarchy itself can cause even bigger changes. Also the lack of a superclass table can make key management awkward and seriously get in the way of referential integrity. In some databases [Single Table Inheritance](#)'s biggest downside is wasted space, since each row has to have columns for all possible subtypes, which leads to empty columns. But many databases do a very good job of compressing out wasted space in tables. Other problems with [Single Table Inheritance](#) include it's size making it a bottleneck for accesses, but it has the great advantage that it does put all the stuff in one place which makes modification easier and avoids joins.

The three options aren't mutually exclusive. In one hierarchy you can mix patterns: for instance you could have several classes pulled together with [Single Table Inheritance](#) and use [Class Table Inheritance](#) for a few unusual cases. Of course mixing patterns adds complexity.

There's no clear cut winner here, you need to take into account your own circumstances and preferences, much like all the rest of these patterns. My first choice tends to be [Single Table Inheritance](#) as its easy to do and is resilient to many refactorings. I then tend to use the other two as needed to help solve the inevitable issues with irrelevant and wasted columns. Often the best thing to do is to talk to the DBAs, they'll often have good advice as to the sort of access that most makes sense for the database.

All the examples above, and in the patterns, use single inheritance. Although multiple inheritance is becoming less fashionable these days and most languages are increasingly avoiding it, the issue still appears in OR mapping when you use interfaces, as in Java and .NET. The patterns here don't go into this topic specifically but essentially you cope with multiple inheritance using variations of the above trio of patterns. [Single Table Inheritance](#) puts all superclasses and interfaces into the one big table, [Class Table Inheritance](#) makes a separate table for each interface and superclass, and [Concrete Table Inheritance](#) includes all interfaces and superclasses in each concrete table.

## Partial Loading

The last structural pattern to mention is [Lazy Load](#). If you have all your objects connected together then any read of any object could pull an enormous object graph out of the database. To avoid such inefficiencies you need to reduce what you bring back, yet still keep the door open to pull back more data if you need it later on. [Lazy Load](#) relies on having a place holder for a reference to an object. There are several variations on the theme, but all of them have the object reference modified so that instead of pointing to the real object it marks a placeholder. Only if you try to follow the link does the real object get pulled in from the database. By using [Lazy Load](#) at suitable points you can bring back just enough from the database with each call.

## Double Mapping

Occasionally I run into situations where the same kind of data needs to be pulled from multiple sources.

There may be more than one database that holds the same data, but has small differences in the schema due to some copy and paste reuse. (In this situation the amount of annoyance is inversely proportional to the amount of the difference.) Another possibility is using different mechanisms, sometimes storing the data in a database and sometimes through messages. You may want to pull similar data from both XML messages, CICS transactions and relational tables.

The simplest option is to have multiple mapping layers, one for each data source. However if data is very similar this can lead to a lot of duplication. In this situation you might consider a two layer mapping scheme. The first step converts data from the in-memory form to logical data store form. The logical data store form is designed to maximize the similarities in the data source formats. The second step then maps from the logical data store schema to the actual physical data store schema. The second step contains the differences.

The extra step only pays itself when you have many commonalities, so you should use this when you have similar but annoyingly different physical data stores. When doing this treat the mapping from the logical data store to the physical data store as a [Gateway](#) and use any of the mapping techniques to map from the application logic to the logical data store.

## Going about building the mapping

When you map to a relational database there are essentially three situations that you encounter

- You get to choose the schema yourself
- You have to map to an existing schema, which cannot be changed
- You have to map to an existing schema, but changes to it are negotiable

The simplest case is where you are doing the schema yourself and you have little to moderate complexity in your domain logic, resulting in a [Transaction Script](#) or [Table Module](#) design. In this case you can design the tables around the data using classic database design techniques. Use a [Row Data Gateway](#) or [Table Data Gateway](#) to pull the SQL away from the domain logic.

If you are using a [Domain Model](#) then you should beware of a design that looks like a database design. In this case build your [Domain Model](#) without regard to the database so that you can best simplify the domain logic. Treat the database design as a way of persisting the objects' data. [Data Mapper](#) gives you the most flexibility here, but it is more complex. If a database design that is isomorphic to the [Domain Model](#) makes sense, you might consider a [Active Record](#) instead.

Although the build the model first is reasonable way of thinking about it, the advice only applies within short iterative cycles. Spending six months building a database free [Domain Model](#) and then deciding to persist it once you're done is highly risky. The danger is that the resulting design will have crippled performance problems that take too much refactoring to fix. Instead build the database up with each iteration, of no more than six weeks in length - preferably less. That way you'll get rapid and continuous feedback about how your database interactions work in practice. Within any particular task you should think about the [Domain Model](#) first, but integrate each piece of [Domain Model](#) in with the database as you go.

When the schema's already there, then your choices are similar, but the process is slightly different. With simple domain logic you build [Row Data Gateway](#) or [Table Data Gateway](#) classes that mimic the database, and layer domain logic on top of that. With more complex domain logic you'll need a [Domain](#)

Model which is highly unlikely to match the database design. So gradually build up the Domain Model and include Data Mappers to persist the data to the existing database.

In this book most of my examples use hand written code. With mapping that's often simple and repetitive this can lead to code that's simple and repetitive - and repetitive code is a sign of something wrong with the design. There is much you can do by factoring out common behaviors with inheritance and delegation - good honest OO practices. But there's also call for a more sophisticated approach using Metadata Mapping.

Metadata Mapping is based on boiling down the details of the mapping into a metadata file. This metadata file contains the details of the mapping: how columns in the database map to fields in objects. The point of this is that once you have the metadata you can then avoid the repetitive code by using either code generation or reflective programming.

Using metadata buys you a lot of expressiveness from a little metadata. One line of metadata can say something like

```
<field name = "customer" targetClass = "Customer", dbColumn = "custID",
targetTable = "customers" lowerBound = "1" upperBound = "1" setter =
"loadCustomer" />
```

From that you can define the read code, the write code, automatic generation of ad hoc joins, all of the SQL, enforcement of the multiplicity of the relationship, and even fancy things like computing write orders under the presence of referential integrity. This is why commercial OR mapping tools tend to use metadata.

Despite the many advantages of metadata, in this book I've focused on hand-written examples. This is because I think that's easier to understand first. Once you get the hang of the patterns and can hand write them for your application, you'll be able to figure out how to use metadata to make matters easier.

## Database Connections

Most database interfaces rely on some kind of database connection object to act as the link between application code and the database. Typically a connection must be opened before you can execute commands against the database, indeed usually you need an explicit connection to create and execute a command. This same connection must be open the whole time you execute the command. Queries return a Record Set. Some interfaces provide for disconnected Record Sets, these can be manipulated after the connection is closed. Other interfaces only provide connected Record Sets, which implies that the connection must remain open while the Record Set is manipulated. If you are running inside a transaction, then usually transactions are bound to a particular connection and the connection must remain open during the whole transaction.

In many environments it's expensive to create a connection. In these cases it's worth creating a pool of connections. In this situation developers request a connection from the pool and release the connection when they are done, instead of the creating and closing. Most platforms these days give you pooling, so you'll rarely have to do it yourself. If you do have to do it yourself, check first to see if pooling actually does help performance. Increasingly environments make it quicker to create a new connection so there's no need to pool.

Environments that do give you pooling often put it behind an interface that looks like creating a new connection. That way you don't know whether you are getting a brand new connection, or one allocated from a pool. That's a good thing, as the choice about whether to pool or not is properly encapsulated. Similarly closing the connection may not actually close, but just return it to the pool for someone else to use. In this discussion I'll use open and close, but you can substitute these for getting from a pool and releasing back to the pool.

Expensive or not to create, connections need management. Since they are expensive resources to manage, connections must be closed as soon as you are done using them. Furthermore if you are using a transaction, usually you need to ensure that every command inside a particular transaction goes with the same connection.

The most common advice is to get a connection explicitly, using a call to a pool or connection manager, and then supply the connection to each database command you wish to make. Once you are done with the connection, you should close it. This advice leads to a couple of issues: making sure you have the connection everywhere you need it, and ensuring you don't forget to close it at the end.

To ensure you have it where you need it there are two choices. One is to pass the connection around as an explicit parameter. The problem with this is that the connection gets added to all sorts of method calls where its purpose is only to be passed to some other method five layers down the call stack. This, of course, is the situation to bring out [Registry](#). Since you don't want multiple threads using the same connection, you'll want a thread-scoped [Registry](#).

If you're half as forgetful as I am, explicit closing isn't such a good idea. It's just too easy to forget to do it when you should. You also can't close the connection with every command, because you may be running inside a transaction and just closing the connection will usually cause the transaction to rollback.

Memory is also a resource that needs to be freed up when you're not using it. Modern environments these days provide automatic memory management and garbage collection. So one way to ensure connections get closed is to use the garbage collector. In this approach either the connection itself, or some object that refers to the connection, closes the connection during garbage collection. The good thing about this is that it uses the same management scheme that's used for memory, and so it's both convenient and familiar. The problem with this is that the close of the connection only happens when the garbage collector actually reclaims the memory, and this can be quite a bit later than when the connection lost its last reference. As a result unreferenced connections may sit around a while before they get closed. Whether this is a problem or not depends very much on your specific environment.

On the whole I don't like relying on garbage collection, other schemes - even explicit closing - are better. But garbage collection does make a good backup in case the regular scheme fails. After all it's better to have the connections close eventually than to have them hanging around forever.

Since connections are so tied to transactions a good way to manage the connections is to tie them to a transaction. Open a connection when you begin a transaction, and close it when you commit or rollback. Have the transaction know what connection its using, and that way you can ignore the connection completely and just deal with the transaction. Since the transaction's completion has a visible effect, it's easier to remember to commit it and to spot if you forget. If you use a [Unit of Work](#) it makes a natural fit to manage both transaction and connection.

If you do things outside of a transaction, such as reading immutable data, then you use a fresh connection for each command. Pooling can deal with any issues in creating short-lived connections.

If you're using a disconnected [Record Set](#) you can open a connection to put the data in the record set and close it while you manipulate the data in the [Record Set](#). Then when you're done with the data you can open a new connection, and transaction, to write the data out. If you do this you'll need to worry about the data being changed while the [Record Set](#) was being manipulated, a topic I'll talk about with concurrency control

The specifics of connection management are very much a feature of your database interaction software, so the strategy you use is often dictated to you by your environment.

## Some miscellaneous points

You'll notice that some of the code examples use select statement in the form select \* from while others use named columns. Using select \* can have some serious problems in some database drivers which would break if a new column was added or a column was reordered. Although more modern environments don't suffer from this, it's not wise to use select \* if you are using positional indices to get information from columns, as a column reorder will break code. It's okay to use column name indices with a select \* and indeed column name indices are clearer to read, however column name indices may be slower - although that probably won't make much difference given the time for the SQL call, as usual measure to be sure. If you do use column number indices, you need to make sure the accesses to the result set are very close to the definition of the SQL statement so they don't get out of sync if the columns get reordered, consequently if using [Table Data Gateway](#) you should use column name indices as the result set is used by every piece of code that runs a find operation on the gateway.

For connections, I just conjure them up with a call to a "DB" object which is a [Registry](#). How you get a connection will depend on your environment so you'll substitute this with whatever you need to do. I haven't involved transactions in any of the patterns other than those on concurrency - again you'll need to mix in whatever your environment needs.

## Further Reading

Object-relational mapping is a fact of life for most people, so it's no surprise that there's been a lot written on the subject. The surprise is that there isn't even a single coherent, complete, and up to date source in a book, which is why I've devoted so much of this one to this tricky yet interesting subject.

The nice thing about database mapping is that there's a lot of ideas out there to steal from. The most victimized intellectual banks are: [\[Brown and Whitenack\]](#), [\[Ambler\]](#), [\[Yoder\]](#), and [\[Coldeway and Keller\]](#). I'd certainly urge you to have a good surf through this material to supplement the patterns in this book.





ABC Amber CHM Converter Trial version

Please register to remove this banner.

<http://www.processtext.com/abcchm.html>

# Concurrency

---

by Martin Fowler and David Rice

Concurrency is one of the most tricky aspects of software development. Whenever you have multiple processes or threads manipulating the same data, you run into concurrency problems. Concurrency is hard to think about, since it's difficult to enumerate the possible scenarios that can get you into trouble. Whatever you do, there always seems to be something you miss. Furthermore concurrency is hard to test for. We are great fans of a large bevy of automated tests to act as a foundation for software development, but it's hard to get tests to give us the security we need for concurrency problems.

One of the great ironies of enterprise application development is that few branches of software development use concurrency more, yet worry about it less. The reason enterprise developers can get away with a naive view of concurrency is transaction managers. Transactions provide a framework that helps avoid many of the most tricky aspects of concurrency in an enterprise application setting. As long as you do all your manipulation of data within a transaction, no really bad things will happen to you.

Sadly this doesn't mean we can ignore concurrency problems completely. The primary reason for this is that there are many interactions with a system that cannot be placed within a single database transaction. This forces us to manage concurrency in situations where we have data that spans transactions. The term we use is **offline concurrency**: concurrency control for data that's manipulated during multiple database transactions.

The second area where concurrency rears its ugly head for enterprise developers is application server concurrency: supporting multiple threads in an application server system. We've spend much less time on this because dealing with it is much simpler, or you can use server platforms that deal with much of this for you.

Sadly, to understand these concurrency issues, you need to understand at least some of the general issues of concurrency. So we begin this chapter by going over these issues. We don't pretend that this chapter is a general treatment of concurrency in software development, to that we'd need at least a complete book. What this chapter does is introduce concurrency issues for enterprise application development. Once we've done that we'll introduce the patterns for handling offline concurrency and say our brief words on application server concurrency.

In much of this chapter we'll illustrate the ideas with examples from a topic that we hope you are very familiar with, the source code control systems used by teams to coordinate changes to a code base. We do this because it is relatively easy to understand as well as well as familiar. After all, if you aren't familiar with source code control systems, you really shouldn't be developing enterprise applications.

# Concurrency Problems

To begin getting into concurrency, we'll start by going through the essential problems of concurrency. We call these the essential problems, because these are the fundamental problems that concurrency control systems try to prevent. They aren't all the problems of concurrency, because often the control mechanisms provide a new set of problems in their solution! But they do focus on the essential point of concurrency control.

**Lost updates** are the simplest idea to understand. Say Martin edits a file to make some changes to the checkConcurrency method - a task that takes a few minutes. While he's doing this David alters the updateImportantParameter method in the same file. David starts and finishes his alteration very quickly, so quickly that while he starts after Martin he finishes before Martin. This is unfortunate because when Martin read the file it didn't include David's update, so when Martin writes the file, it writes over the version that David updated and David's update is lost for ever.

An **inconsistent read** occurs when you read two things that were correct pieces of information, but they weren't correct at the same time. Say Martin wishes to know how many classes are in the concurrency package. The concurrency package contains two sub packages for locking and multiphase. Martin looks in the locking package and sees seven classes. At this point he gets a phone call from Roy on some abstruse question. While Martin's answering it David finishes dealing with that pesky bug in the four phase lock code and adds two classes to the locking package and adds three classes to the five that were in the multiphase package. Phone call over Martin now looks in the multiphase package to see how many classes there are and sees eight, producing a grand total of fifteen concurrency classes.

Sadly fifteen classes was never the right answer. The correct answer was twelve before David's update and seventeen afterwards. Either answer would have been correct, even if not current. But fifteen was never correct. This problem is called an inconsistent read because the data that Martin read was inconsistent.

Both of these problems cause a failure of **correctness** (or safety). They result in incorrect behavior that would not have occurred without two people trying to work with the same data at the same time. However if correctness was the only issue, these problems wouldn't be that serious. After all we could arrange things so that only one of us can work the data at once. While this helps with correctness, it reduces the ability to do things concurrently. The essential problem of any concurrent programming is that it's not enough to worry about correctness, you also have to worry about **liveness**: how much concurrent activity can go on. Often people need to sacrifice some correctness to gain more liveness, depending on the seriousness and likelihood of the failures and the need for people to work on their data concurrently.

These aren't all the problems you get with concurrency, but we think of these as the basic ones. To avoid these problems we use various mechanisms to control concurrency, but sadly there's no free lunch. The solutions to these problems introduce problems of their own. But at least the problems introduced by the control mechanisms are less serious than the basic ones. However this does bring up an important point - if you can tolerate the problems above, you can avoid any form of concurrency control. This is rare, but occasionally you find circumstances that permit it.

# Execution Contexts

Whenever processing occurs in a system, it occurs in some form of context, and usually in more than one. There's no standard terminology for execution contexts, so here we'll define the ones that we are assuming in this book.

From the perspective of interacting with the outside world, two important contexts are the request and the session. A **request** corresponds to a single call from the outside world which the software works on and optionally sends back a response. During a request the processing is largely in the servers court and the client is assumed to wait for a response. Some protocols allow the client to interrupt a request before it gets a response, but this is fairly rare. Rather more often a client may issue another request that may interfere with one it has just sent. So a client may request to place an order and then issue a separate request to cancel that same order. From the client's view they may be obviously connected, but depending on your protocol that may not be so obvious to the server.

A **session** is a long running interaction between a client and server. A session may consist of a single request, but more commonly it consists of a series of requests, a series that user regards as a consistent logical sequence. Commonly it will begin with a user logging in, doing various bits of work which may involve issuing queries, and may involve one or more business transactions (see later), at the end the user logs out. Or the user just goes away and assumes the system interprets that as logging out.

Server software in an enterprise application sees both requests and sessions from two angles, as the server from the client and as the client to other systems. As a result you'll often see multiple sessions: http sessions from the client and multiple database sessions with various databases.

Two important terms from operating systems are processes and threads. A **process** is a, usually heavyweight, execution context that provides a lot of isolation for the internal data it works on. A **thread** is a lighter-weight active agent that's set up so that multiple threads can operate in a single process. People like threads because that way you can support multiple requests inside a single process - which is good utilization of resources. However threads usually share memory. Such sharing leads to concurrency problems. Some environments allow you to control what data a thread may access, allowing you to have **isolated threads** that don't share memory.

The difficulty with execution contexts comes when they don't line up as well as we might like. In theory it would be nice if each session had an exclusive relationship with a process for its whole lifetime. Since processes are properly isolated from each other, this helps reduce concurrency conflicts. Currently we don't know of any server tools allow you to work this way. A close alternative is to start a new process for each request, which was the common mode for early Perl web systems. People tend to avoid that now because starting processes tie up a lot of resources, but it's quite common for systems to have a process only handle one request at a time - and that can save many concurrency headaches.

When you're dealing with databases there's another important context - a transaction. Transactions pull together several requests which the client would like to see treated as if they were done in a single request. Transactions can either occur from the application to database: a system transaction, or from the user to an application: a business transaction: we'll dig into these terms more later on.

## Isolation and Immutability

The problems of concurrency have been around for a while, and software people have come up with various solutions. For enterprise applications there are two that are particularly important: isolation and immutability.

The various concurrency problems occur when more than one active agent, such as a process or thread, has access to the same piece of data. So one way to deal with concurrency concerns is **isolation**: partition the data so that any piece of data can only be accessed by one active agent. This is indeed how processes work in operating system memory. The operating system allocates memory exclusively to a single process. Only that process can read or write the data linked to it. Similarly you find file locks in many popular productivity applications. If Martin opens a file, nobody else can open that same file. They may be allowed to open a read-only copy of the file as it was when Martin started, but they don't change it and they don't get to see the file between my changes.

Isolation is a vital technique because it reduces the chance of errors. Too often we've seen people get themselves into trouble with concurrency because they try to use a technique that forces everyone to worry about concurrency all the time. With isolation you arrange things so that the programs enters an isolated zone, then within that zone it doesn't have to worry about concurrency issues. So a good concurrency design is to find ways of creating such zones, and ensure as much programming as possible is done in one of these zones.

You only get concurrency problems if the data you're sharing may be modified. So one way to avoid concurrency conflicts is to recognize **immutable** data. Obviously we can't make all data immutable, as the whole point of many systems is to modify data. But by identifying some data that is immutable, or at least immutable as far as almost all users are concerned, we can then relax all the concurrency issues for that data and share it widely. Another option is to separate applications that are only reading data, and have them use copied data sources from which we can relax all concurrency controls.

## Optimistic and Pessimistic Concurrency Control

So what happens when we have mutable data that cannot isolate? In broad terms there's two forms of concurrency control that we can use: optimistic and pessimistic.

Let's suppose that Martin and David both want to edit the Customer file at the same time. With **optimistic locking** both of them can make a copy of the file and edit it freely. If David is the first to finish he can check in his work without trouble. The concurrency control kicks in when Martin tries to commit his changes. At this point the source code control system detects that there is a conflict between Martin's changes and David's changes. Martin's commit is rejected and it's up to Martin to figure out how to deal with the situation. With **pessimistic locking** whoever checks out the file first prevents anyone else from editing the file. So if Martin is first to check out, David cannot work with the file until Martin is finished with it and commits his changes.

A good way of thinking about this is that an optimistic lock is about conflict detection, while a pessimistic lock is about conflict prevention. As it turns out real source code control systems can use either style, although these days most developers prefer to work with optimistic locks for source code.

Both approaches have their pros and cons. The problem with the pessimistic locks is that it reduces concurrency. While Martin is working on a file he locks it, so everybody else has to wait. If you've worked with pessimistic source code control mechanisms, you know how frustrating this can be. With enterprise data it's often worse, because if someone is editing data, then nobody else is allowed to read the data, let alone edit it.

Optimistic locks thus allow people to make progress much better, because the lock is only held during the commit. However the problem the optimistic locks is what happens when you get a conflict.

Essentially everybody after David's commit has to check out the version of the file that David checked in, figure out how to merge their changes with David's changes, and then check in a newer version. With source code, this happens to be not too difficult to do. Indeed in many cases the source code control system can automatically do the merge for you. Even when it can't auto merge, tools can make it much easier to see the differences. But business data is usually too difficult to automatically merge, so often all you can do is throw away everything and start again.

So the essence of the choice between optimistic and pessimistic is the frequency and severity of conflicts. If conflicts are sufficiently rare, or the consequences are no big deal, then you should usually pick optimistic locks because they give you better concurrency, as well as usually being easier to implement. However if the results of a conflict are painful for the users, then you'll need to avoid them with a pessimistic technique.

Neither of these approaches are exactly free of problems. Indeed by using them you can easily introduce problems which cause almost as much trouble as the basic concurrency problems that you are trying to solve anyway. We'll leave a detailed discussion of all these ramifications to a proper book on concurrency, but here's a few highlights to bear in mind.

## Preventing Inconsistent Reads

Consider this situation. Martin edits the Customer class which makes calls on the Order class. Meanwhile David edits the order class and changes the interface. David compiles and checks in, Martin then compiles and checks in. Now the shared code is broken because Martin didn't realize that the order class altered underneath him. Some source code control systems will spot this inconsistent read, but others require some kind of manual discipline to enforce it, such as updating your files from the trunk before you check in.

In essence this is the inconsistent read problem, and it's often easy to miss because most people tend to focus on lost updates as the essential problem in concurrency. Pessimistic locks have a well-worn way of dealing with this problem through read and write locks. To read data you need a read (or shared) lock, to write data you need a write (or exclusive) lock. Many people can have read locks on the data at once, but if anyone has a read lock then nobody can get a write lock. Conversely once somebody has a write lock, then nobody else can have any lock. By doing this you can avoid inconsistent reads with pessimistic locks.

Optimistic locks usually base their conflict detection based on some kind of version marker on the data. this can be a timestamp, or a sequential counter. To detect lost updates, the system checks the version marker of your update with the version marker of the shared data. If they are the same the system allows the update and updates the version marker.

Detecting an inconsistent read is essentially similar: in this case every bit of data that was read also needs

to have its version marker compared with the shared data. Any differences indicate a conflict.

However controlling access to every bit of data that's read often causes unnecessary problems due to conflicts or waits on data that doesn't actually matter that much. You can reduce this burden by separating out data you've *used* from data you've merely read. A pick list of products is something where it doesn't matter if a new product appeared in the list after you started your changes. But a list of charges that you are summarizing for a bill may be more important. The difficulty is that this requires some careful analysis of what exactly is used for what. A zip code in a customer's address may seem innocuous, but if a tax calculation is based on where somebody lives, that address has to be controlled for concurrency. So figuring out what you need to control and what you don't is an involved exercise whichever form of concurrency control you use.

Another way to deal with inconsistent read problems is to use **Temporal Reads**. These prefix each read of data with some kind of time stamp or immutable label. The database then returns the data as it was according to that time or data. Very few databases have anything like this, but developers often come across this in source code control systems. The problem is that the data source needs to provide a full temporal history of changes which takes time and space to process. This is reasonable for source code, but both more difficult and more expensive for databases. You may need to provide this capability for specific areas of your domain logic: see [\[snodgrass\]](#) and [\[Fowler, temporal patterns\]](#) for ideas on how to do that.

## Deadlocks

A particular problem with pessimistic techniques is **deadlock**. Say Martin starts editing the customer file and David is editing the order file. David realizes that to complete his task he needs to edit the customer file too, but since Martin has a lock on it he can't and has to wait. Then Martin realizes he has to edit the order file, which David has locked. They are now deadlocked, neither can make progress until the other completes. Said like this, they sound easy to prevent, but deadlocks can occur with many people involved in a complex chain, and that makes it more tricky.

There are various techniques you can use to deal with deadlocks. One is to have software that can detect a deadlock when it occurs. In this case you pick a **victim** who has to throw away his work and his locks so the others can make progress. Deadlock detection is very difficult to do and causes pain for the victims. A similar approach is to give every lock a time limit. Once you hit the time limit you lose your locks and your work - essentially becoming a victim. Time outs are easier to implement than a deadlock detection mechanism, but if anyone holds locks for a while you'll get some people being victimized when there wasn't actually a deadlock present.

Time outs and detection look to deal with a deadlock when it occurs, other approaches try to stop deadlocks occurring at all. Deadlocks essentially occur when people who already have locks try to get more (or to upgrade from read to write locks.) So one way of preventing deadlocks is to force people to acquire all their locks at once at the beginning of their work, and once they have locks to prevent them gaining more.

You can force an order on how everybody gets locks, an example might be to always get locks on files in alphabetical order. This way once David had a lock on the Order file, he can't try to get a lock on the Customer file because it's earlier in the sequence. At that point he becomes a victim.

You can also make it so that if Martin tries to acquire a lock and David already has a lock, then Martin

automatically becomes a victim. It's a drastic technique, but is simple to implement. And in many cases such a scheme actually works just fine.

If you're very conservative you can use multiple schemes. So you force everyone to get all their locks at the beginning, but add a timeout as well in case something goes wrong. That may seem like using a belt and braces, but such conservatism is often wise with deadlocks because deadlocks are pesky things that are easy to get wrong.

It's very easy to think you have a deadlock-proof scheme, and then to find some chain of events you didn't consider. As a result for enterprise application development we prefer very simple and conservative schemes. They may cause unnecessary victims, but that's usually much better than the consequences of missing a deadlock scenario.

## Transactions

The primary tool for handling concurrency in enterprise applications is the transaction. The word transaction often brings to mind an exchange of money or goods. Walking up to an ATM machine, entering your PIN, and withdrawing cash is a transaction. Paying the three dollar toll at the Golden Gate Bridge is a transaction. Buying a beer at the local pub is a transaction.

Looking at typical financial dealings such as these provides a good definition for the word transaction. First, a transaction is a bounded sequence of work. Both start and end points are well defined. An ATM transaction begins when the card is inserted and ends when cash is delivered or an inadequate balance is discovered. Second, all participating resources are in a consistent state both when the transaction begins and when the transaction ends. A man purchasing a beer has a few bucks less in his wallet but has a nice pale ale in front of him. The sum value his assets has not changed. Same for the pub - pouring free beer would be no way to run a business.

In addition, each transaction must complete on an all-or-nothing basis. The bank cannot subtract from an account holder's balance unless the ATM machine actually delivers the cash. While the human element might make this last property optional during the above transactions, there is no reason software cannot make a guarantee on this front.

Software transactions are often described in terms of the ACID properties:

- **Atomicity:** Each step in the sequence of actions performed within the boundaries of a transaction must complete successfully or all work must rollback. Partial completion is not a transactional concept. So if Martin is transferring some money from his savings to his checking account and the server crashes after he's withdrawn the money from his savings, the system behaved as if he never did the withdrawal. Committing says both things occurred, a rollback says neither occurred, it has to be both or neither.
- **Consistency:** A system's resources must be in a consistent, non-corrupt state at both the start and completion of a transaction.
- **Isolation:** The result of an individual transaction must not be visible to any other open transactions until that transaction commits successfully.
- **Durability:** Any result of a committed transaction must be made permanent. This translates to 'must survive a crash of any sort.'

Most enterprise applications run into transactions in terms of databases. But there are plenty of other

things than can be controlled using transactions, such as message queues, printers, ATMs, and the like. As a result technical discussions of transactions use the term 'transactional resource' to mean anything that is transactional: that is uses transactions to control concurrency. Transactional resource is a bit of a mouthful, so we just tend to use database, since that's the most common case. But when we say database, the same applies for any other transactional resource.

To handle the greatest throughput, modern transaction systems are designed around the transactions being as short as possible. As a result the general advice is to never make a transaction span multiple requests, making a transaction span multiple requests is generally known as a **long transaction**.

As a result a common approach is to start a transaction at the beginning of a request and complete it at the end. This **request transaction** is a nice simple model and a number of environments make it easy to do declaratively, by just tagging methods as transactional.

A variation on this is to open a transaction as late as possible. With a **late transaction** you may do all the reads outside of a transaction and only open up a transaction when you do updates. This has the advantage of minimizing the time spent in a transaction. If there's a lengthy time lag between the opening of the transaction and the first write, this may improve liveness. However this does mean that you don't have any concurrency control until you begin the transaction, which leaves you open to inconsistent reads. As a result it's usually not worth doing this unless you have very heavy contention, or if you are doing it anyway due to business transactions that span multiple requests (which is the next topic).

When you use transactions, you need be somewhat aware of what exactly is being locked. For many database actions the transaction system locks the rows that are involved, which allows multiple transactions to access the same table. However if a transaction locks a lot of rows in a table, then the database finds it's got more locks than it can handle and escalates the lock to the entire table - locking out other transactions. This **lock escalation** can have a serious effect on concurrency. In particular it's a reason why you shouldn't have some "object" table for data at the domain's [Layer Supertype](#) level. Such a table is a prime candidate for lock escalation, and locking that table shuts everybody else out of the database.

## Reducing Transaction Isolation for Liveness

It's common to restrict the full protection of transactions so that you can get better liveness. This is particularly the case when it comes to handling isolation. If you have full isolation you get serializable transactions. Transactions are **serializable** if they can be executed concurrently and get a result that is the same as you would get from some sequence of executing the transactions serially. So if we take our earlier example of Martin counting his files, serializable would guarantee that he would get a result that corresponds to either completing his transaction entirely before David's transaction starts (twelve) or after David's finishes (seventeen) Serializability cannot guarantee which result, as in this case, but it at least guarantees a correct one.

Most transactional systems use the SQL standard which defines four levels of isolation. Serializable is the strongest level, each level below allows a particular kind of inconsistent read to enter the picture. We'll explore these with the example of Martin counting files while David modifies them. There are two packages: locking and multiphase. Before David's update there are 7 files in the locking package and 5 in the multiphase package, after his update there are 9 in the locking package and 8 in the multiphase package. Martin looks at the locking package, David then updates both, then Martin looks at the multiphase package.

If the isolation level is serializable then the system guarantees that Martin's answer is either twelve or seventeen, both of which are correct. Serializability cannot guarantee that every run through this scenario would give the same result, but it would always get either the number before David's update, or the number afterwards.

The first level below serializability is the isolation level of **repeatable read** that allows **phantoms**. Phantoms occur when you are adding some elements to a collection and the reader sees some of them, but not all of them. The case here is that Martin looks at the files in the locking package and sees 7. David then commits his transaction. Martin then looks at the multiphase package and sees 8. Hence Martin gets an incorrect result. Phantoms occur because they are valid for some of Martin's transaction but not all of it, and they are always things that are inserted.

Next down the list is the isolation level of **read committed** that allows **unrepeatable reads**. Imagine that Martin looks at a total rather than the actual files. An unrepeatable read would allow him to read a total of 7 for locking, then David commits, then he reads a total of 8 for multiphase. It's called an unrepeatable read because if Martin were to re-read the total for the locking package after David committed he would get the new number of 9. His original read of 7 can't be repeated after David's update. It's easier for databases to spot unrepeatable reads than it is to spot phantoms, so the repeatable read isolation level gives you more correctness than read committed but with less liveness.

The lowest level of isolation is **read uncommitted** which allows **dirty reads**. Read uncommitted allow you to read data that another transaction hasn't actually committed yet. This causes two kinds of errors. Martin might look at the locking package when David adds the first of his files but before he adds the second. As a result he sees 8 files in the locking package. The second kind of error would come if David added his files, but then rolled back his transaction - in which case Martin would see files that were never really there.

| Isolation Level  | Dirty Read | unrepeatable read | phantom |
|------------------|------------|-------------------|---------|
| Read Uncommitted | Yes        | Yes               | Yes     |
| Read Committed   | No         | Yes               | Yes     |
| Repeatable Read  | No         | No                | Yes     |
| Serializable     | No         | No                | No      |

To be sure of correctness you should always use the isolation level of serializable. But the problem is that choosing a serializable level of transaction isolation really messes up the liveness of a system. So much so that often you have to reduce the serializability in order to increase throughput. You have to decide what risks you want take and make your own trade-off of errors versus performance.

You don't have to use the same isolation level for all transactions, so you should look at each transaction you have decide how to balance liveness versus correctness for that transaction.

## Business and System Transactions

What we've talked about so far, and most of what most people talk about, are what we call **system transactions**, or transactions supported by RDBMS systems and transaction monitors. A database transaction is a group of SQL commands delimited by instructions to begin and end the transaction. If

the fourth statement in the transaction results in an integrity constraint violation the database must rollback the effects of the first three statements in the transaction and notify the caller that the transaction has failed. If all four statements had completed successfully all would be made visible to others at the same time, rather than one at a time. Support for system transactions, in the form of RDBMS systems and application server transaction managers, are so commonplace that they can pretty much be taken for granted. They work well and are well understood by application developers.

However, a system transaction has no meaning to the user of a business system. To the user of an online banking system a transaction consists of logging in, selecting an account, setting up some bill payments, and finally clicking the 'OK' button to pay the bills. This is what we call a **business transaction**. That a business transaction displays the same ACID properties as a system transaction seems a reasonable expectation. If the user cancels before paying the bills any changes made on previous screens should be cancelled. Setting up payments should not result in a system-visible balance change until the 'OK' button is pressed.

The obvious answer to supporting the ACID properties of a business transaction is to execute the entire business transaction within a single system transaction. Unfortunately business transactions often take multiple requests to complete. So using a single system transaction to implement a business transaction results in a long system transaction. Most transaction systems don't work very efficiently with long transactions.

This doesn't mean that you should never use long transactions. If your database has only modest concurrency needs, then you may well be able to get away with it. And if you can get away with it, we'd suggest you do it. Using a long transaction means you avoid a lot of awkward problems. However the application won't be scalable, because those long transactions will turn the database into a major bottleneck. In addition the refactoring from long to short transactions is both complex and not well understood.

As a result many enterprise applications cannot risk long transactions. In this case you have to break the business transaction down into a series of short transactions. This means that you are left to your own devices to support the ACID properties of business transactions between system transactions, the problem that we call **offline concurrency**. System transactions are still very much part of the picture. Whenever the business transaction interacts with a transactional resource, such as a database, that interaction will execute within a system transaction in order to maintain the integrity of that resource. However, as you'll read below it is not enough to string together a series of system transactions to properly support a business transaction. The business application must provide a bit of glue between the system transactions.

Atomicity and durability are the ACID properties most easily supported for business transactions. Both are supported by running the commit phase of the business transaction, when the user hits 'save,' runs within a system transaction. Before the session attempts to commit all its changes to the record set it will first open a system transaction. The system transaction guarantees that the changes will commit as a unit and that they will be made permanent. The only potentially tricky part here is maintaining an accurate change set during the life of the business transaction. If the application uses a [Domain Model](#) a [Unit of Work](#) can track changes accurately. Placing business logic in a [Transaction Script](#) requires a more manual tracking of changes. But that is probably not much of a problem as the use of transaction scripts implies rather simple business transactions.

The tricky ACID property to enforce with business transactions is isolation. Failures of isolation lead to failures of consistency. Consistency dictates that a business transaction not leave the record set in an invalid state. Within a single transaction the application's responsibility in supporting consistency is to

enforce all available business rules. Across multiple transactions the application's responsibility is to ensure that one session doesn't step all over another session's changes, leaving the record set in the invalid state of having lost a user's work.

As well as the obvious problems of clashing updates, there is the more subtle problems of inconsistent reads. When data is read over several system transactions, there's no guarantee that the data will be consistent. The different reads could even introduce data in memory that's sufficiently inconsistent to cause application failures.

Business Transactions are closely tied to sessions. In the user's view of the world, each session is a sequence of business transactions (unless they are only reading data). So as a result we usually make the assumption that all business transactions execute in a single client session. While it's certainly possible to design a system that has multiple sessions in for one business transaction, that's a very good way of getting yourself badly confused - so we'll assume you won't do that.

## Patterns for Offline Concurrency Control

As much as possible, you should let the transaction system you use deal with concurrency problems. Handling concurrency control that spans system transactions plonks you firmly into the murky waters of doing your own concurrency. Waters full of virtual sharks, jellyfish, piranhas, and other less friendly creatures. Unfortunately the mismatch between business and system transactions means you sometimes just have to wade into these waters. The patterns that we've provided here are some techniques that we've found helpful in dealing with concurrency control that spans system transactions.

Remember that these are techniques that you should only use if you have to. If you can make all your business transaction fit into a system transaction by ensuring that all your business transactions fit within a single request, then do that. If you can get away with long transactions by forsaking scalability, then do that. By leaving concurrency control in the hands of your transaction software you'll avoid a great deal of trouble. These techniques are what you have to do when you can't do that. Because of the tricky nature of concurrency, we have to stress again that the patterns are a starting point not a destination. We've found these useful, but we don't claim to have found a cure for the ills of concurrency.

Our first choice for handling offline concurrency problems is [\*Optimistic Offline Lock\*](#), which essentially uses optimistic concurrency control across the business transactions. We like this as a first choice because it's an easier approach to program and yields the best liveness. The limitation of [\*Optimistic Offline Lock\*](#) is that you only find out that a business transaction is going to fail when you try to commit that transaction. In some circumstances the pain of that late discovery is too much: users may have put an hour's work into entering details about a lease, or you get lots of failures and users lose faith in the system. Your alternative then is [\*Pessimistic Offline Lock\*](#), this way you find out early if you are in trouble, but lose out because it's harder to program and it reduces your liveness.

With either of these approaches you can save considerable complexity by not trying to manage locks on every object. A [\*Coarse-Grained Lock\*](#) allows you to manage the concurrency of a group of objects together. Another step you can do to make life easier for application developers is to use [\*Implicit Lock\*](#) which saves them from having to manage locks directly. Not just does this save work, it also avoids bugs when people forget - and these bugs are hard to find.

A common statement about concurrency is that it's something that is a purely technical decision that can

be decided on after requirements are complete. We disagree. The choice of optimistic or pessimistic controls affect the whole user experience of the system. Intelligent design of [\*Pessimistic Offline Lock\*](#) needs a lot of input about the domain from the users of the system. Similarly domain knowledge is needed to choose good [\*Coarse-Grained Lock\*](#).

Whatever you do, futzing with concurrency is one of the most difficult programming tasks you can do. It's very difficult to test concurrent code with confidence. Concurrency bugs are hard to reproduce and very difficult to track down. These patterns have worked for us so far, but this is particularly difficult territory. If you need to go down this path it's particularly worth getting some experienced help. At the very least consult the books we've mentioned at the end of this chapter..

## Application Server Concurrency

So far we've talked about concurrency mainly in terms of the concurrency of multiple sessions running against a shared data source. Another form of concurrency is the process concurrency of the application server itself: how does that server handle multiple requests concurrently and how this affects the design of the application on the server. The big difference with the other concurrency issues we've talked about so far is that application server concurrency doesn't involve transactions, so working with this means stepping away from the relatively controlled transactional world.

Explicit multi-threaded programming, with locks and synchronization blocks, is complicated to do well. It's easy to introduce defects that are very hard to find, since concurrency bugs are almost impossible to reproduce - resulting in a system that works correctly 99% of the time, but throws random fits. Such software is incredibly frustrating both to use and to debug. As a result our policy is to avoid the need for explicit handling of synchronization and locks as much as possible. Application developers should almost never have to deal with these explicit concurrency mechanisms.

The simplest way to handle this is to use **process-per-session** where each session runs in its own process. The great advantage of this is that the state of each process is completely isolated from the other processes, so the application programmers don't have to worry at all about multi-threading. As far as memory isolation goes, it's almost equally effective to have each request start a new process, or to have one process tied to the session that's idle between requests. Many early web systems would start a new Perl process for each request.

The problem with process-per-session is that it uses up a lot resources, since processes are expensive beasties. To be more efficient you can pool the processes, such that each process only handles a single request at one time, but can handle multiple requests from different sessions in a sequence. This approach of pooled **process-per-request** will use much less processes to support a given amount of sessions. Your isolation is almost as good: you don't have many of the nasty multi-threading issues. The main problem over process-per-session is that you have to ensure any resources used to handle a request are released at the end of the request. The current Apache mod-perl uses this scheme, as do a lot of serious large scale transaction processing systems.

Even process-per-request will need many processes running to handle a reasonable load, you can further improve throughput with by having a single process run multiple threads. With this **thread-per-request** approach, each request is handled by a single thread within a process. Since threads use much less server resources than a process, you can handle more requests with less hardware this way, so your server is more efficient. The problem with using thread-per-request is that there's no

isolation between the threads, any thread can touch any piece of data that it can get access to.

In our view there's a lot to be said for using process-per-request. Although it leads to less efficiency than thread-per-request, using process-per-request is equally scalable. You also get better robustness, if one thread goes haywire it can bring down an entire process, so using a process-per-request limits the damage. Particularly with a less experienced team, the reduction of threading headaches (and the time and cost of fixing bugs) is worth the extra hardware costs. We find that few people actually do any performance testing to assess the relative costs of thread-per-request and process-per-request for their application.

Some environments provide a middle ground of allowing isolated areas of data to be assigned to a single thread, COM does this with the single-threaded apartment and J2EE does this with Enterprise Java Beans (and in the future with isolates). If your platform has something like this available, this can allow you to have your cake and eat it - whatever that means..

If you use thread-per-request then the most important thing is to create and enter an isolated zone where application developers can mostly ignore multi-threaded issues. The usual way to do this is have the thread create new objects as it starts handling the request, and ensure these objects aren't put anywhere (such as a static variable) where other threads can see them. That way the objects are isolated because other threads have no way of referencing them.

Many developers are concerned about creating new objects because they've been told that object creation is an expensive process. As a result people often pool objects. The problem with pooling is that you have to synchronize access to the pooled objects in some way. But the cost of object creation is very dependent upon the virtual machine and memory management strategies. With modern environments object creation is actually pretty fast [missing reference] (off the top of your head: how many Java date objects do you think we can create in one second on Martin's 600Mhz P3 with Java 1.3? We'll tell you later.) Creating fresh objects for each session avoids a lot of concurrency bugs and can actually improve scalability.

While this tactic works for many cases, there are still some areas that developers need to avoid. One is any form of static, class based variables or global variables. Any use of these has to be synchronized, so it's usually best to avoid them. This is also true of singletons. If you need some kind of global memory use a [Registry](#), which you can implement in such a way that it looks like a static variable, but actually uses thread-specific storage.

Even if you're able to create objects for the session, and thus make a comparatively safe zone, there are some objects that are expensive to create and thus need to be handled differently - the most common example of this is a database connection. To deal with this you can place these objects in an explicit pool where you acquire a connection while you need it and return it to the pool when done. These operations will need to be synchronized.

## Further Reading

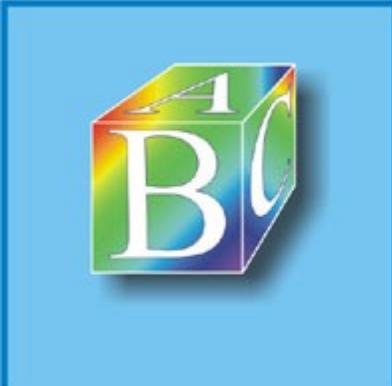
In many ways, this chapter only skims the surface of a much more complex topic. To investigate further we'd suggest starting with [\[Bernstein and Newcomer\]](#), [\[Lea\]](#), and [\[Schmidt\]](#).

---



---

© Copyright [Martin Fowler](#), all rights reserved

A blue rectangular banner with a white border. On the left side is a 3D-style cube with faces colored red, green, blue, and yellow, with the letters 'A', 'B', and 'C' visible on its faces. To the right of the cube, the text "ABC Amber CHM Converter Trial version" is displayed in red. Below this, in smaller black text, is "Please register to remove this banner." At the bottom right of the banner is the URL "http://www.processtext.com/abcchm.html".

---

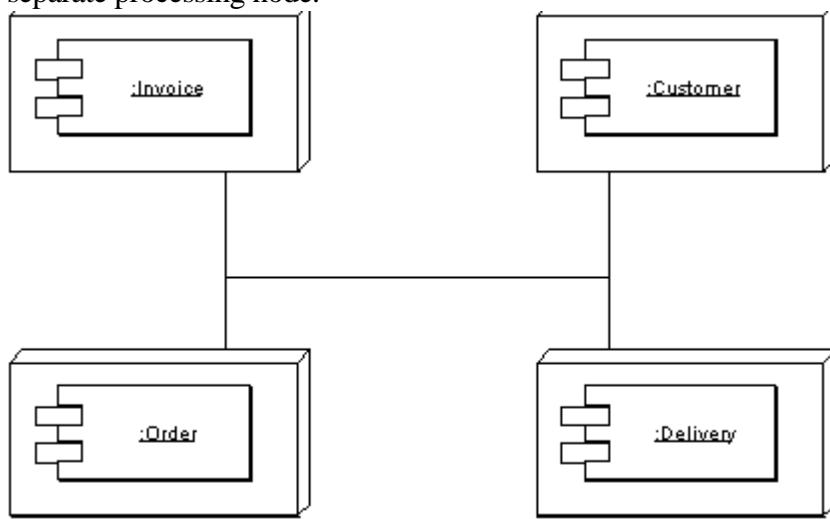
# Distribution Strategies

---

Objects have around for a while, and sometimes it seems that ever since they were created folks wanted to distribute them. However distribution of objects, or indeed of anything else, has a lot more pitfalls than many people realize - especially when under the influence of vendors' cozy brochures. This chapter is about some of these hard lessons - lessons I've seen many of my clients learn the hard way.

## The Allure of Distributed Objects

This is a recurring presentation that I used to see two or three times a year when doing design reviews. Proudly the system architect of a new OO system lays out his plan for a new distributed object system. Let's pretend it's a some kind of ordering system with customers, orders, products, and deliveries. They show me a design that looks rather like Figure 1>. The design has separate remote objects for customers, orders, products, and deliveries. Each one is a separate component that can be placed on a separate processing node.



*Figure 1: Distributing an application by putting different components on different nodes.*

So I ask: "why do you do this?"

"Performance, of course" the architect replies, looking at me a little oddly. "We can run each component on a separate box. If one component gets too busy we add extra boxes for that component - enabling us to load balance our application." The look is now curious as if he wonders if I really know anything about real distributed object stuff at all.

Meanwhile I'm faced with an interesting dilemma. Do I just say out and out that this design sucks like an inverted hurricane and get showed the door immediately? Or do I try to slowly shine the light onto my client. The latter is more remunerative but is much tougher, since the client is usually very pleased with his architecture and it takes a lot to give up on a fond dream.

So assuming you haven't shown this book the door I guess you'll want to know why this distributed architecture sucks. After all many vendors of tools will tell you that the whole point of distributed objects is that you can take a bunch of objects and position them how you like on processing nodes: and their powerful middleware provides transparency. Transparency that allows objects to call each other within a process or between a process without them knowing if the callee is in the same process, another process, or another machine.

Such transparency is valuable, but while many things can be made transparent in distributed objects - performance isn't usually one of them. Although our prototypical architect was distributing things the way he was for performance reasons - in fact his design will usually cripple performance, or make the system much harder to build and deploy, or usually both.

## Remote and Local Interfaces

The primary reason why the distribution by class model doesn't work, is because of a fundamental fact of computers. A procedure call within a process is very, very fast. A procedure call between two separate processes is orders of magnitude slower. Make that process a process running on another machine and you can add another order of magnitude or two - depending on the network topography involved.

As a result of this you need a different kind of interface for an object that's intended to be used remotely from one that intended to be used locally within the same process.

A local interface is best as a fine grained interface. So if I have an address class - a good interface would have separate methods for getting the city, getting the state, setting the city, setting the state etc. A fine grained interface is good because it follows the general OO principle of lots of little pieces that can be combined and overridden in various ways to extend the design into the future.

A fine grained interface does not work well, however, when you have a remote interface. When method calls are slow, you want to obtain or update the city, state, and zip in one call rather than three calls. The resulting interface is coarse-grained, designed not for flexibility and extendibility but for minimizing the calls. Here you'll see an interface along the lines of get address details and update address details. This coarse grained interface is much more awkward to program to, but for performance you need to have it.

Of course what middleware vendors will tell you is that there is no overhead to using the their middleware for remote and local calls. If it's a local call it's done with the speed of a local call. If it's a remote call it's done more slowly. So you only pay the price of a remote call when you need a remote call. This much is, to some extent, true. But it doesn't avoid the essential point that any object that may be used remotely should have a coarse-grained interface, while every object that isn't used remotely should have a fine-grained interface. Whenever two objects communicate you have to choose which to use. If the object could ever be in separate processes you have to use the coarse-grained interface and pay the cost of the harder programming model. Obviously it only makes sense to pay that cost when you

need to - so you need to minimize the amount of inter-process collaborations.

As a result you can't just take a group of classes that you design in the world of a single process, throw CORBA or some such at them, and come up with a distributed model. Distribution design is more than that. If you base your distribution strategy on a classes - you'll end up with a system that does a lot of remote calls and thus needs awkward coarse-grained interfaces. In the end, even with coarse-grained interfaces on every remotable class you'll still end up with too many remote calls, and system that's awkward to modify as a bonus.

Hence we get to Fowler's **First Rule of Distributed Object Design**: don't distribute your objects!

So how do you effectively use multiple processors? In most cases the way to go is to use clustering. Put all the classes into a single process and then run multiple copies of that process on the various nodes. That way each process uses local calls to get the job done and thus does things faster. You can also use fine-grained interfaces for all the classes within the process and thus get better maintainability with a simpler programming model.

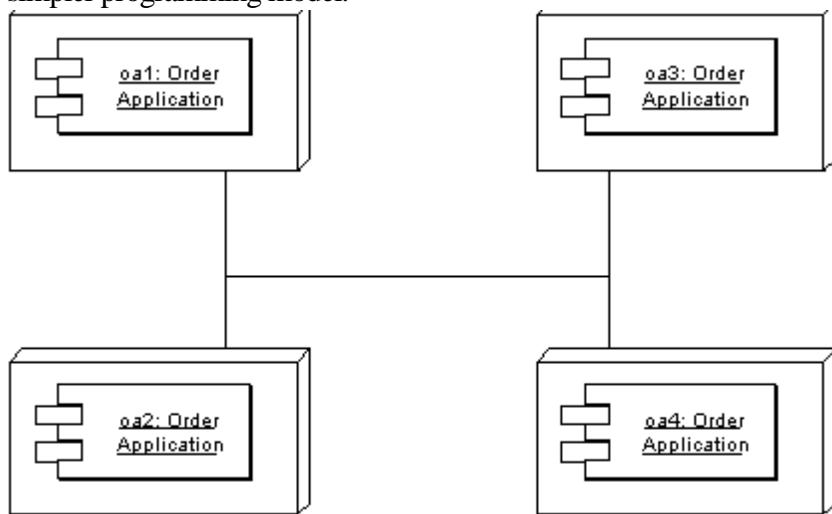


Figure 2: Clustering involves putting several copies of the same application on the different nodes.

## Where you have to distribute

So you want to minimize distribution boundaries and utilize your nodes through clustering as much as possible. The rub is that there are limits to that approach. There are places where you need to separate the processes. If you're sensible you'll fight like a cornered rat to eliminate as many of them as you can - but you won't eliminate them all.

- One obvious separation is that between the traditional clients and servers of business software. PCs on users' desktops are different nodes to shared repositories of data. Since they are different machines you need separate processes that communicate. The client/server divide is a typical inter-process divide.
- A second divide often occurs between server based application software (the application server) and the database. Of course you don't have to do this. You may run all your application software in the database process itself - using such things as stored procedures. But often that is not so practical, so you have to have separate processes. They may run on the same machine -

but once you have separate processes you immediately have to have to pay most of the costs in remote calls. Fortunately SQL is designed as a remote interface, so you can usually arrange things to minimize that cost.

- Another separation in process may occur in a web system between the web server and the application server. All things being equal it's best to run the web and applications server in a single process. But all things are not always equal.
- You may have to separate due to vendor differences. If you are using a software package, it will often run in its own process. So again you are distributing, but at least a good package will have a coarse-grained interface.
- And finally there may be some genuine reason that you have to split your own application server software. You should sell any grand-parent you can get your hands on to avoid this, but cases do come up. Then you just have to hold your nose and divide your software into remote, coarse-grained components.

The overriding theme is, in Colleen Roe's memorable phrase, to be parsimonious with object distribution. Sell your favorite grandma first if you possibly can.

## Working with the Distribution Boundary

So as you design your system, you need to limit your distribution boundaries as much as possible, but where you have them you need to take them into account. Every remote call travels over the cyber equivalent of a horse and carriage. All sorts of places in the system will change shape to minimize those remote calls. That's pretty much the expected price.

However you can still design within a single process using fine-grained objects. The key is to use fine-grained objects internally and place coarse grained objects at the distribution boundaries whose sole role is to provide a remote interface to these fine-grained objects. These coarse grained objects don't really do anything, so they act as a facade for the fine-grained objects, a facade that's only there for distribution purposes, the usual term for them is [Remote Facade](#).

Using a [Remote Facade](#) helps minimize the difficulties that the coarse-grained interface introduces. This way only the objects that really need a remote service need the coarse grained method, and it is obvious to the developers that they are paying that cost. Transparency has its virtues, but you don't want to be transparent about a potential remote call.

However by keeping the coarse grained interfaces as mere facades, you allow people to use the fine-grained objects whenever they know they are running in the same process. This makes the whole distribution policy much more explicit.

Hand in hand with [Remote Facade](#) is [Data Transfer Object](#). Not just do you need coarse-grained methods, you also need to transfer coarse grained objects. When you ask for an address, you need to send that information in one block. You can't usually send the domain object itself, because the domain object is tied in a web of fine-grained local inter-object references. So you take all the data that the client needs and bundle it up into a particular object for the transfer, hence the term [Data Transfer Object](#) (Many people in the enterprise Java community use the term *value object* for this, but this causes a name clash with other meanings of the term [Value Object](#)). The [Data Transfer Object](#) appears on both sides of the wire, so it's important that it doesn't reference anything that isn't shared over the wire. This boils down to the fact that a [Data Transfer Object](#) usually only references other [Data](#)

[Transfer Objects](#)s and fundamental objects such as strings.

Another route to distribution is to have a broker that migrates objects between processes. The idea here is to use a [Lazy Load](#) scheme where instead of lazy reading from a database, you move objects across the wire. The hard part of this is ensuring you don't end up with lots of remote calls. I've not seen anyone try to do this in an application, but some OR mapping tools (eg TOPLink) have this facility, and I've heard some good reports about it.



---

© Copyright [Martin Fowler](#), all rights reserved



# Putting it all Together

---

So far these narratives have looked at one aspect of a system and explored the various options that you can use in handling these issues. Now it's time to sweep all of these issues together and start to answer the tricky question of what patterns you should use together when designing an enterprise application.

The advice in this chapter is in many ways a repeat of the advice given in the earlier narrative chapters. I must admit I've had prevarications on whether this chapter was needed, but I felt it was good to put all the discussion in context, now that I hope you have at least an outline of the full scope of the patterns in this book.

As I write this, I'm very conscious of the limitations of my advice. Frodo said of the elves "go not to the Elves for counsel, for they will say both no and yes". While I'm not claiming any immortal knowledge I certainly understand their answer that advice is often a dangerous gift. If you're reading this to make architectural decisions for your project, you know far more about the issues of your project than I do. One of the biggest frustrations in being a pundit is that people will often come up to me at a conference, or in an email message, and ask for advice on their architectural or process decisions. There's no way you can give particular advice on the basis of a five minute description. I write this chapter with even less knowledge of your predicament.

So read this chapter in the spirit that it's given. I don't know all the answers, and I certainly don't know your questions. Use this advice to prod your thinking, but don't use it as a replacement for your thinking. In the end you have to make, and live with, the decisions yourself.

But one good thing is that your decisions are not cast forever in stone. Architectural refactoring is hard, and we are still ignorant of its full costs. But it isn't impossible. Here the best advice I can give is that even if you dislike the full story of [extreme programming](#), you should still consider seriously three technical practices: continuous integration, test driven development, and refactoring. These will not be a panacea, but they will make it much easier to change your mind when you discover you need to. And you will need to, unless you are either more fortunate, or more skillful, than anyone I've met to date.

## Starting With the Domain Layer

The start of the process is deciding which domain logic approach to go with. The three main contenders are: [Transaction Script](#), [Table Module](#), and [Domain Model](#).

As I've indicated in [that chapter](#), the strongest force that drives you through this trio is the complexity of the domain logic, and this is something that is currently impossible to quantify, or even qualify with any

degree of reasonable precision. But other factors also play in the decision, in particular the difficulty of the connection with a database.

The simplest of the three is [\*Transaction Script\*](#). It fits with the procedural model that most people are still most comfortable with. It nicely encapsulates the logic of each system transaction into a comprehensible script. It's easy to build on top of a relational database. Its great failing is the fact that it doesn't deal well with complex business logic, being particularly susceptible to duplicate code. If you have a simple catalog application with little more than a shopping cart running off a simple pricing structure, then [\*Transaction Script\*](#) will fill the bill perfectly, but as your logic gets more complicated then your difficulties multiply exponentially.

At the other end of the scale is the [\*Domain Model\*](#). Hard core object bigots like myself will have an application no other way. After all if an application is simple enough to write with [\*Transaction Scripts\*](#), why should our immense intellects bother with such an unworthy problem? And my experiences lead me to have no doubt that with really complex domain logic, there's nothing can handle this hell better than a rich [\*Domain Model\*](#). Once you get used to working with a [\*Domain Model\*](#) even simple problems can be tackled with ease.

Yet the [\*Domain Model\*](#) carries its faults. High on the list is the difficulty of learning them. Object bigots often look down their noses at people who just don't get objects. But the consequence is that a [\*Domain Model\*](#) requires skill if it's to be done well. Done poorly it's quite a disaster. The second big difficulty of a [\*Domain Model\*](#) is the connection to a relational database. Of course a real object zealot finesse this problem with the subtle flick of an object database. But for many, mostly non-technical, reasons an object database isn't a possible choice for enterprise applications. The result is the messy connection to a relational database. Let's face it, object models and relational models don't fit well together. The complexity of many of the OR mapping patterns I describe is the result.

[\*Table Module\*](#) represents an attractive middle ground between these poles. It can handle domain logic better than [\*Transaction Scripts\*](#), and while it can't touch a real [\*Domain Model\*](#) on handling complex domain logic it fits in really well with a relational database - and many other things too. If you have an environment, such as .NET, where many tools orbit around the all-seeing [\*Record Set\*](#) then [\*Table Module\*](#) works very nicely. It plays to the strengths of the relational database and yet represents a reasonable factoring of the domain logic.

In this argument we see that the tools you have also affect your architecture. Sometimes you're able to choose the tools based on the architecture - and in theory that's the way you should go. But in practice you often have to fit your architecture to match your tools. Of the three [\*Table Module\*](#) is the one whose star rises most when you have tools that match it. It's a particularly strong choice for .NET environments, since so much of the platform is geared around [\*Record Set\*](#).

If you've read the discussion in the domain logic chapter, much of this will seem familiar. Yet I think it was worth repeating here because I really do think this is the central decision. From here we go downwards to the database layer, but now the decisions are shaped by the context of your domain layer choice.

## Down to the Data Source

Once you've chose your domain layer, you now have to figure out how to connect it to your data

sources. Your decisions are based on your domain layer choice, so I'll tackle this in separate sections, driven by the your choice of domain layer.

## Transaction Script

The simplest [\*Transaction Scripts\*](#) contain their own database logic, but I prefer to avoid doing that even in the simplest cases. Separating the database out separates two parts that make sense separate, so even in the simplest applications I make the separation. The database patterns to choose from here are [\*Row Data Gateway\*](#) and [\*Table Data Gateway\*](#).

The choice between them depends much on the facilities of your implementation platform, and on where you expect the application to go in the future. Using a [\*Row Data Gateway\*](#) gives you a nicely explicit interface for dealing with the data, as each record is read into an object with a clear and explicit interface. With [\*Table Data Gateway\*](#) you may have less code to write, since you don't need all the accessor code to get at the data, but you end up with a much more implicit interface that relies on accessing a record set structure that's little more than a glorified map.

The key decision, however, lies in the rest of your platform. If you have a platform that provides a lot of tools that work well with [\*Record Set\*](#), particularly UI tools or transactional disconnected record sets, then that would tilt me decisively in the direction of a [\*Table Data Gateway\*](#).

You usually don't need any of the other OR mapping patterns in this context. All the structural mapping issues are pretty much absent since the in-memory structure maps to the database structure so well. You might consider a [\*Unit of Work\*](#), but usually it's easy to keep track of what's changed in the script. You don't need to worry about most concurrency issues because the script often corresponds almost exactly to a system transaction, so you can just wrap the whole script in a single transaction. The common exception is where one request pulls data back for editing and the next request tries to save the changes. In this case [\*Optimistic Offline Lock\*](#) is almost always the best choice. Not just is it easier to implement, it also usually fits users expectations and avoids the problem of a hanging session leaving all sorts of things locked.

## Table Module

The main reason to choose [\*Table Module\*](#) is the presence of a good [\*Record Set\*](#) framework. In this case you'll want a database mapping pattern that works well with [\*Record Sets\*](#), so that leads you inexorably towards [\*Table Data Gateway\*](#). These two patterns fit naturally together as if they were made in heaven for each other.

There's not really anything else you need to add on the data source side with this pattern. In the best cases the [\*Record Set\*](#) has some kind of concurrency control mechanism built in, which effectively turns it into a [\*Unit of Work\*](#), which further reduces hair loss.

## Domain Model

Now things get interesting. In many ways the big weakness of [\*Domain Model\*](#) is the fact that connection

to the database is complicated. The degree of complication depends upon the complexity of the [Domain Model](#).

If your [Domain Model](#) is fairly simple, say a couple of dozen classes which are pretty close to the database, then an [Active Record](#) makes sense. If you want to decouple things a bit you can use either [Table Data Gateway](#) or [Row Data Gateway](#) to do that. Whether you separate or not is not a huge deal either way.

As things get more complicated, then you'll need to consider [Data Mapper](#). [Data Mapper](#) is really the approach that delivers on the promise of keeping your [Domain Model](#) as independent as possible of all the other layers. But [Data Mapper](#) is also the most complicated one to implement. Unless you either have a strong team, or you can find some simplifications that make the mapping easier to do, I'd strongly suggest getting a tool to handle the mapping.

Once you choose [Data Mapper](#) then most of the patterns in the OR mapping section come into play. In particular I heartily recommend using [Unit of Work](#), which does a great deal to act as a focal point for concurrency control.

## Up to the Presentation

In many ways the presentation is relatively independent of the choice of the lower layers. Your first question is whether to provide a rich client interface or a HTML browser interface. A rich client will give you a nicer UI, but you then have to have a certain amount of control and deployment of your clients. My preference is to pick an HTML browser if you can get away with it, and a rich client if that's not possible. A rich client will usually be more effort to program, but that's usually because rich clients tend to be more sophisticated, not so much the inherent complexities of the technology.

I haven't explored any rich client patterns in this book, so if you choose rich client I don't really have anything further to say.

If you go the HTML route then you have to decide how to structure your application. I certainly recommend the [Model View Controller](#) as the underpinning for your design. That done, you are left with two decisions, one for the controller and one for the view.

Your tooling may well make your choice for you. If you use Visual Studio, then the easiest way to go is [Page Controller](#) and [Template View](#). If you use Java, then you have a choice of web frameworks to consider. A popular choice at the moment is Struts, that will lead you to a [Front Controller](#) and [Template View](#).

Given a freer choice, I'd recommend [Page Controller](#) if your site is more document-oriented, particularly if you have a mix of static and dynamic pages. More complex navigation and UI leads you towards a [Front Controller](#)

On the view front, the choice between [Template View](#) and [Transform View](#) really depends on whether your team prefers to use server pages or XSLT in your programming. [Template Views](#) have the edge at the moment, although I rather like the added testability of [Transform View](#). If you have the need to display a common site with multiple look and feels you should consider [Two Step View](#)

How you communicate with the lower layers depends on what kind of layers they are, and whether they

are always going to be in the same process. My preference is to have everything all run in one process if you can, that way you don't have to worry about slow inter-process calls. If you can't do that then you should wrap your domain layer with [Remote Facade](#) and use [Data Transfer Object](#) to communicate to the web server. There's also an argument for using [Remote Facades](#) within a single process if you have a [Domain Model](#) as the [Remote Facade](#) can help hide the complexities of the [Domain Model](#) from view. That choice really depends on how comfortable you are working with the [Domain Model](#) directly, adding a [Remote Facade](#) can add a lot of extra coding.

## Some Technology Specific Advice

In most of this book I'm trying to bring out the common experience of doing projects in many different platforms. Experience with Forte, CORBA, and Smalltalk translates very effectively into developing with Java and .NET. The only reason I've concentrated on Java and .NET environments in this book is because they look like they will be the most common platforms for enterprise application development in the future. (Although I'd like to see the dynamically typed scripting languages, in particular Python and Ruby, give them a run for their money.)

So in this section I want to apply the above advice to these two platforms. As soon as I do this I'm in danger of dating my advice. Technologies change much more rapidly than these patterns. So as you read this, remember I'm writing this in early 2002 when everyone is saying that economic recovery is just around the corner.

### Java and J2EE

As I write this the big debate in the Java world is exactly how valuable is Enterprise Java Beans? The EJB spec has finally appeared and tools are coming out. But you don't *need* EJB to build a good J2EE application, despite what EJB vendors tell you. You can do a great deal with POJOs (Plain Old Java Objects) and JDBC

The design alternatives for J2EE vary on what kind of patterns you are using, and again they break out driven by the domain logic.

If you use [Transaction Script](#) on top of some form of [Gateway](#), then with EJB the common approach at the moment is to use session beans as [Transaction Script](#) and entity beans as [Row Data Gateway](#). This is a pretty reasonable architecture if your domain logic is sufficiently modest to only need [Transaction Scripts](#). The problem with this strongly beany approach is that it's hard to get rid of the EJB server if you find you don't need it and you don't want to cough up the license fees. The non EJB approach is a POJO for the [Transaction Script](#) on top of either a [Row Data Gateway](#) or a [Table Data Gateway](#). If JDBC 2.0 row sets get more accepted, then that's a reason to use them as [Record Sets](#) and that leads to a [Table Data Gateway](#). If you're not sure about EJB you can use the non EJB approach and wrap them with session beans acting as [Remote Facades](#).

If you are using a [Domain Model](#) then the current orthodoxy is to use entity beans as your [Domain Model](#). If your [Domain Model](#) is pretty simple and matches your database well, then that makes reasonable sense and your entity beans will then be [Active Records](#). It's still good practice to wrap your entity beans with session beans acting as [Remote Facades](#). However if you [Domain Model](#) is more

complex, then I would want the [Domain Model](#) to be entirely independent of the EJB structure, so that you can write, run, and test your domain logic without having to deal with the vagaries of the EJB container. In that model I would use POJOs for the [Domain Model](#) and wrap them with session beans acting as [Remote Facades](#). If you choose not to use EJB I would run the whole app in the web server and avoid any remote calls between presentation and domain. If you're using POJO [Domain Model](#) I'd also use POJOs for the [Data Mappers](#) - either using an OR mapping tool or rolling something myself if I felt really up to it.

At the moment, the [Table Module](#) isn't terribly common in the Java world. It'll be interesting to see if more tooling surrounds the JDBC row set - if it does it could become a viable approach. In this case the POJO approach fits best, although you can also wrap the [Table Module](#) with session beans acting as [Remote Facades](#) and returning [Record Sets](#).

## .NET

Looking at .NET, Visual Studio, and the history of application development in the Microsoft world, the dominant pattern in [Table Module](#). Although object bigots tend to dismiss this as just meaning that Microsofties don't get objects, the [Table Module](#) does present a valuable compromise between [Transaction Script](#) and [Domain Model](#), with an impressive set of tools that take advantage of the ubiquitous data set acting as a [Record Set](#)

As a result the [Table Module](#) has to be the default choice for this platform. Indeed I see no point at all in using [Transaction Scripts](#) except in the very simplest of cases, and even then they should act on and return data sets.

This doesn't mean that you can't use [Domain Model](#), indeed you can build a [Domain Model](#) just as easily in .NET as you can in any other OO environment. But the tools don't give you the extra help that they do for [Table Modules](#), so I'd tolerate more complexity before I felt the need to shift to a [Domain Model](#).

The current hype in .NET is all about web services, but I would not use web services inside an application, I'd use it, as in Java, as a presentation to allow applications to integrate. There's no real reason to separate the web server and domain logic into separate processes in a .NET application, so [Remote Facade](#) is less useful within the application.

## Web Services

As I write this, the general consensus amongst pundits is that web services will make reuse a reality and drive system integrators out of business. I'm not holding my breath for either of those. Within these patterns web services don't play a huge role because they are about application integration rather than how you build a simple application. You shouldn't try to break a single application up into web services talking to each other unless you really need to. Rather build your application and expose various parts of it as web services treating the web services as [Remote Facades](#). Above all don't let all the buzz about how easy it is to build web services let you forget about the [First Rule of Distributed Object Design](#).

© Copyright [Martin Fowler](#), all rights reserved



# Transaction Script

---

*Organize Business Logic by procedures that carry out what needs to be done in a transaction*

```
recognizedRevenue (contractNumber: long, asOf: Date) : Money
calculateRevenueRecognitions (contractNumber long) : void
```

Most business applications can be thought of as a series of transactions. A transaction may view some information organized in a particular way. Another will make changes to the database. Each interaction between a client system and a server system contains a certain amount of logic. In some cases this can be as simple as just displaying some information that's in the database. In others this may involve many steps of validations, and calculations.

A *Transaction Script* organizes all this logic primarily as a single procedure, making calls directly to the database or through a thin database wrapper. Each transaction will have its own *Transaction Script*, although common subtasks can be broken out into sub-procedures. Each *Transaction Script* is a public method on a class.

## How it Works

With *Transaction Script* this logic is primarily organized by the transactions that you do with the system. If your need is to book a hotel room, the logic to check the room is available, calculate the rates, and update the database is found inside the book hotel room procedure.

For simple cases, there isn't really much to say about how you organize this. Of course, like any other program, you should structure the code into modules in a way that makes sense. Unless the transaction is particularly complicated that won't be too much of a challenge. One of the benefits of this approach is that you don't need to worry about what other transactions are doing. Your task is to get the input, interrogate the database, munge, and save results to the database.

Where you put the *Transaction Script* will depend on how you organize your layers. It may be in a server page, CGI script, distributed session object. Whatever your architecture there is some transaction controller that initiates everything that goes on. For a simple *Transaction Script* the business logic can be placed in that transaction controller module, either as inline code within the handling of the request, or (better) broken into some separate routines.

The challenges for implementing *Transaction Script* come in two forms which often come together: complicated transaction logic and duplicated logic.

You know you have complicated transaction logic when the customer says something like "for this use case step, here's the ten pages that explains what goes on". At this point the most important thing to do is get any complicated code isolated from where it fits into your architecture. Create a module that is has no dependencies on the upper layers of the system. The *Transaction Script* will know about the database, or some [Gateways](#). Your transaction controller will then call this module at appropriate points. The module might be a single routine, or a bunch of routines.

Separating the business logic from the layers will make it easier to modify and test the business logic. It's easier to modify because when you're modifying the logic, you don't have to worry about the issues that come from the module's place in your layers. Also you can address layering issues without worrying about the details of the business logic. It's more testable because the module can be tested outside the confines of your layered architecture. This may improve the cycle time of development process quite significantly.

You can organize your *Transaction Scripts* into classes in two ways. One is to have one *Transaction Script* per class. In this case your classes are [commands](#) and should have a common [Layer Supertype](#) that defines the execute method. The advantage of this is that it allows you to manipulate instances of scripts as objects at runtime, although I've rarely seen people need to do this with the kinds of systems that use *Transaction Scripts* to organize domain logic. The alternative is to have several *Transaction Scripts* in a single class, where each class defines a subject area of related *Transaction Scripts*. This is the simpler, and more common technique. Of course you can ignore classes completely in many languages and just use global functions. However you'll often find that instantiating a new object helps contain threading issues as it makes it easier to isolate data.

## When to Use it

The glory of *Transaction Script* is its simplicity. Organizing logic this way is natural for applications with only a small amount of logic and implies very little overhead: both in performance and in understanding.

But as the business logic gets more complicated it gets progressively harder to keep the business logic in a well designed state. One particular problem to watch carefully for is duplication between transactions. Since the whole point is handle one transaction, if there's common code then it tends to be duplicated.

Careful factoring can alleviate many of these problems, but more complex business domains really need to build a [Domain Model](#). A [Domain Model](#) will give you much more options in structuring the code, increasing readability and decreasing duplication.

It's hard to quantify the cut over level, especially when you're more familiar with one than the other. You can refactor a *Transaction Script* oriented design to a [Domain Model](#) style of design. But it's a harder change than it would otherwise need to be, so an early shot is often the best way to move forwards.

But however much of an object bigot you become, don't rule out *Transaction Script*. There are a lot of simple problems out there, and a simple solution will get you up and running much faster.

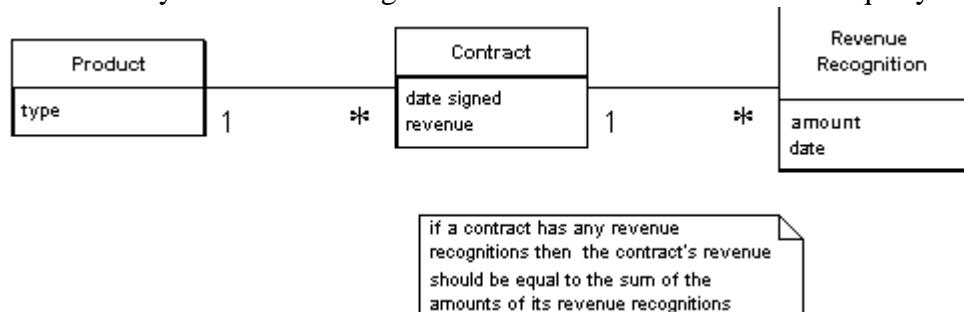
## The Revenue Recognition Problem

For this pattern, and the others that talk about domain logic, I'm going to use the same problem. To avoid typing the problem statement several times, I'm just putting it in here.

Revenue recognition is a common problem in business systems, it's all about when you can actually count the money you receive on your books. If I sell you a cup of coffee, it's a simple matter. I give you the coffee, I take the money, and I can count the money to the books that nanosecond. But for many things it gets complicated. Say you pay me a retainer to be available that year. Even if you pay me some ridiculous fee today, I may not be able to put it on my books right now because the service is to be performed over the course of a year. One approach might be that I should only count one twelfth of that fee for each month over the course of the year, since you might pull out of the contract after a month and you realize that writing has atrophied my programming skills.

The rules for revenue recognition are many, various, and volatile. Some are set by regulation, some by professional standards, some by company policy. Tracking revenue ends up being quite complex problem.

Not that I fancy delving into the complexity right now. Instead we'll imagine a company that sells three kinds of products: word processors, databases, and spreadsheets. The rules are that when you sign a contract for a word processor, you can book all the revenue right away. If it's a spreadsheet you can book one third today, one third in sixty days and one third in ninety days. If it's a database you can book one third today, one third in thirty days, and one third in sixty days. There is no basis for these rules other than my own fevered imagination. I'm told that the real rules are equally rational.



*Figure 1: A conceptual model for simplified revenue recognition. Each contract has multiple revenue recognitions that indicate when the various parts of the revenue should be recognized.*

## Example: Revenue Recognition (Java)

This example uses two transaction scripts: one to calculate the revenue recognitions for a contract, and the other to tell us how much revenue on a contract has been recognized by a certain date. We'll make the unrealistic simplifying assumption that all contracts are only about one product.

The database structure has three tables: one for the products, one for the contracts, and one for the revenue recognitions.

```

CREATE TABLE products (ID int primary key, name varchar, type varchar)
CREATE TABLE contracts (ID int primary key, product int, revenue decimal,
  
```

```
dateSigned date)
CREATE TABLE revenueRecognitions (contract int, amount decimal, recognizedOn
date, PRIMARY KEY (contract, recognizedOn))
```

The first script calculates the amount of recognition due by a particular day. I can do this in two stages, the first part to select the appropriate rows in the revenue recognitions table, the second part to sum up the amounts.

Many *Transaction Script* designs have *Transaction Scripts* that operate directly on the database, putting SQL code into the procedure. Here I'm using a simple [Table Data Gateway](#) to wrap the SQL queries. Since this example is so simple, I'm using a single gateway rather than one for each table. I can define an appropriate find method on the gateway.

```
class Gateway...
public ResultSet findRecognitionsFor(long contractID, MfDate asof) throws
SQLException{
PreparedStatement stmt = db.prepareStatement(findRecognitionsStatement);
stmt = db.prepareStatement(findRecognitionsStatement);
stmt.setLong(1, contractID);
stmt.setDate(2, asof.toSqlDate());
ResultSet result = stmt.executeQuery();
return result;
}
private static final String findRecognitionsStatement =
"SELECT amount " +
"FROM revenueRecognitions " +
"WHERE contract = ? AND recognizedOn <= ?";
```

```
private Connection db;
```

I then use the script to sum up based on the result set passed back from the gateway.

```
class RecognitionService...
public Money recognizedRevenue(long contractNumber, MfDate asOf) {
Money result = Money.dollars(0);
try {
ResultSet rs = db.findRecognitionsFor(contractNumber, asOf);
while (rs.next()) {
result = result.add(Money.dollars(rs.getBigDecimal("amount")));
}
return result;
} catch (SQLException e) {throw new ApplicationException (e);
}
```

For calculating the revenue recognitions on an existing contract, I use a similar split. The script on the service carries out the business logic.

```
class RecognitionService...
public void calculateRevenueRecognitions(long contractNumber) {
try {
ResultSet contracts = db.findContract(contractNumber);
contracts.next();
Money totalRevenue = Money.dollars(contracts.getBigDecimal("revenue"));
MfDate recognitionDate = new MfDate(contracts.getDate("dateSigned"));
String type = contracts.getString("type");
if (type.equals("S")){

```

```
Money[] allocation = totalRevenue.allocate(3);
db.insertRecognition
(contractNumber, allocation[0], recognitionDate);
db.insertRecognition
(contractNumber, allocation[1], recognitionDate.addDays(60));
db.insertRecognition
(contractNumber, allocation[2], recognitionDate.addDays(90));
} else if (type.equals("W")){
db.insertRecognition(contractNumber, totalRevenue, recognitionDate);
} else if (type.equals("D")) {
Money[] allocation = totalRevenue.allocate(3);
db.insertRecognition
(contractNumber, allocation[0], recognitionDate);
db.insertRecognition
(contractNumber, allocation[1], recognitionDate.addDays(30));
db.insertRecognition
(contractNumber, allocation[2], recognitionDate.addDays(60));
}
} catch (SQLException e) {throw new ApplicationException (e);
}
}
```

Notice I'm using [Money](#) to carry out the allocation. When splitting an amount three ways it's very easy to lose a penny.

The [Table Data Gateway](#) provides support on the SQL. Firstly there's a finder for a contract.

```
class Gateway...
public ResultSet findContract (long contractID) throws SQLException{
PreparedStatement stmt = db.prepareStatement(findContractStatement);
stmt.setLong(1, contractID);
ResultSet result = stmt.executeQuery();
return result;
}
private static final String findContractStatement =
"SELECT * " +
"FROM contracts c, products p " +
"WHERE ID = ? AND c.product = p.ID";
```

And secondly there's a wrapper for the insert.

```
class Gateway...
public void insertRecognition (long contractID, Money amount, MfDate asof)
throws SQLException {
PreparedStatement stmt = db.prepareStatement(insertRecognitionStatement);
stmt.setLong(1, contractID);
stmt.setBigDecimal(2, amount.amount());
stmt.setDate(3, asof.toSqlDate());
stmt.executeUpdate();
}
private static final String insertRecognitionStatement =
"INSERT INTO revenueRecognitions VALUES (?, ?, ?)";
```

In a Java system, the recognition service might be a regular class, or it could be a session bean.

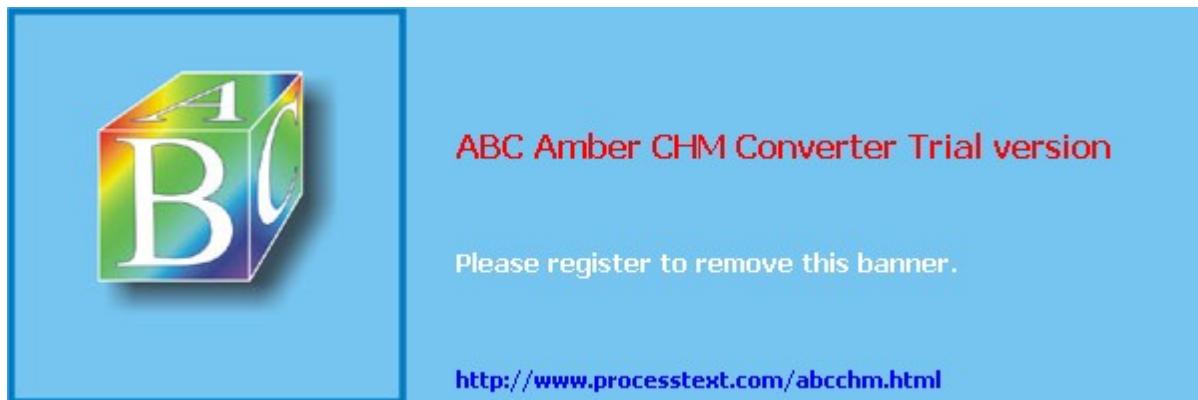
As you compare this to the example in [Domain Model](#) you probably think, unless your mind is as

twisted as mine, that this is much simpler. The harder thing to imagine is what happens as the rules get more complicated. Typical revenue recognition rules get very involved, varying not just by product but also by date (if the contract was signed before April 15 then this rule applies....) It's difficult to keep a coherent design with *Transaction Script* once things get that complicated, which is why object bigots like me prefer using a [Domain Model](#) in those circumstances.



---

© Copyright [Martin Fowler](#), all rights reserved

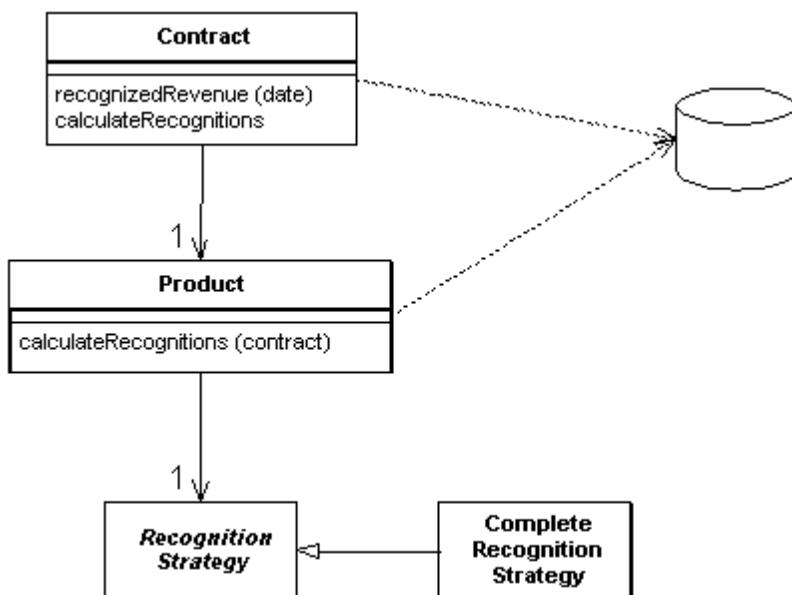


---

# Domain Model

---

*Build an object model of the domain that incorporate both behavior and data.*



At its worst, business logic can be a very complex matter. Rules and logic describe many different cases and slants of behavior. It is this complexity that objects were designed to work with. A *Domain Model* creates a web of interconnected objects, where each object represents some meaningful individual, which may be as large as a corporation, or as small as a single line on an order.

## How it Works

There's whole books written on this, so I hardly know where to start.

Putting a *Domain Model* into an application involves inserting a whole layer of objects. These objects model the business area that you're working in. You'll find objects that mimic the data in the business, and objects that capture the rules that the business uses. Mostly the data and process is combined together to cluster the processes close to the data the processes work with.

An OO domain model will often look similar to a database model, yet it will still have a lot of differences. A *Domain Model* mingles data and process, has multi-valued attributes, and there's inheritance.

Since the behavior of the business is subject to a lot of change, it's important to be able to modify, build, and test this layer easily. As a result you'll want the minimum of coupling from the *Domain Model* to other layers in the system. You'll notice that a guiding force of many layering patterns is to keep the as few dependencies as possible from the domain model to other parts of the system.

When you use a *Domain Model* there are a number of different scopes you might use. The simplest case is a single user application where the whole object graph is read from a file and put into memory. A desktop application may work this way, but it's less common for a multi-tiered IS application simply because there are too many objects. Putting every object into memory takes too long and consumes too much memory. The beauty of object-oriented databases is that they give the impression of doing this, while moving objects between memory and disk.

Without an OODB you have to do this yourself. Usually each session will involve pulling in an object graph of all the objects involved in that session. This will certainly not be all the objects, and usually not all the classes. So if you are looking at a set of contracts you might only pull in the products referenced by contracts within your working set. If you are just performing calculations on contracts and revenue recognition objects you may not pull in any product objects at all. Exactly what you pull into memory is governed by your database mapping objects.

If you need the same object graph between calls to the server, you'll need to save the server state somewhere, which is the subject the section on [saving server state](#).

A common concern with domain logic is bloated domain objects. As you're building a screen to manipulate orders you notice that some of the behavior of the order is only needed for this one screen. If you put these responsibilities on the order the risk is that the order class will become too big, because it's full of responsibilities that are only used in a single use case. This concern leads people to consider whether some responsibility is general, in which case it should sit in the order class; or specific, in which case it should sit in some usage specific class - which might be a [Transaction Script](#) or perhaps the presentation itself.

The problem with separating usage specific behavior is that it tends to lead to duplication. Behavior that separated from the order is harder to find, so people tend to not see it and duplicate it instead. The duplication can quickly lead to more complexity and inconsistency. On the other hand I've found that a bloated domain object is much less of a problem than I used to think. Bloating seems to occur much less frequently than predicted. If bloating does occur, it's relatively easy to see and not difficult to fix. So my advice is not to try to separate usage specific behavior. Put all behavior in the object that is the natural fit. Fix the bloating when, and if, it becomes a problem.

## Java

There's always a lot of heat generated when people talk about developing a *Domain Model* in J2EE. Many of the teaching materials and introductory J2EE books suggest that you use entity beans to develop a domain model. But there are some serious problems with this approach. Entity Beans are remotable, and thus suggest a coarse-grained interface, but a *Domain Model* is at its best when you use fine grained objects. One solution to this is to use session beans to wrap the entity beans so that entity beans are never accessed remotely. While this is good practice, there is still some overhead in calling entity bean methods.

Another strike against entity beans is that, at least in J2EE version 1, the container managed persistence

mapping is very limited. This really means that you are forced to make the entity beans map one to one with the database tables, which only works for an [\*Active Record\*](#) style mapping. You can get more flexibility by using bean managed persistence, but if you are using BMP and not remoting your entity beans their value drops considerably.

True, entity beans handle in-memory caching, which reduces the need for a [\*Unit of Work\*](#), but they are also very difficult to debug, require a lot of classes to use, awkward in handling association and inheritance relationships, increase the build times, and often cause as many performance problems as they can solve.

The alternative is to use normal Java objects. This always causes surprise because it's amazing how many people think you can't run regular Java objects in an EJB container. I've come to the conclusion that people forget about regular Java objects because they haven't got a fancy name - so while preparing for a talk Rebecca Parsons, Josh Mackenzie and I gave them one: POJO (Plain Old Java Object). A POJO domain model is easier to put together, quick to build, can run and test outside of an EJB container, and isn't dependent on EJB (maybe that's why EJB vendors don't encourage you to use them.)

Having said that I've seen projects do well with entity beans if they have modest domain logic and you have a simple relationship with the database. Most of the time, I'd prefer to take the POJO route.

Of course all this is moot with the appearance of EJB 2.0. It's hard for me to comment on this at the moment, however, since the EJB 2.0 spec has had more final drafts than The Who had farewell concerts. What I really want is a *Domain Model* that is as independent as possible from the vagaries of development platforms.

## When to Use it

If the how for a *Domain Model* is difficult because it's such a big subject, the when is hard due to both the vagueness and simplicity of the advice. In a nutshell it all comes down to the complexity of the behavior in your system. If you have complicated and ever changing business rules involving validation, calculations and derivations... chances are you'll want an object model to handle them. On the other hand if you just have a simple not null checks and a couple of sums to calculate, then a [\*Transaction Script\*](#) is a better bet.

A factor that comes into this is how used the development team is with using domain objects. Learning how to design and use a *Domain Model* is a significant exercise - one that led to many articles on the "paradigm shift" of using objects. It certainly takes practice and coaching to get used to using a *Domain Model*, but once you're used to it I've found that few people want to go back to a [\*Transaction Script\*](#) for any but the simplest problems.

If you're using *Domain Model* then my first choice for database interaction is [\*Data Mapper\*](#). This will help keep your *Domain Model* independent from the database and is the best approach to handle cases where the *Domain Model* and database schema diverge.

## Example: Revenue Recognition (Java)

One of the biggest frustrations of describing a *Domain Model* is the fact that any example I show is necessarily simple, so you can understand it; yet that simplicity hides the strength of a *Domain Model*. You only really appreciate the strengths of a *Domain Model* when you have a really complicated domain.

But even if the example can't really do justice to why you'd want a *Domain Model*, at least the example will give you a sense of what it can look like. So I'm using the [same example](#) that I used for [Transaction Script](#), a little matter of revenue recognition. You can compare it with the example there.

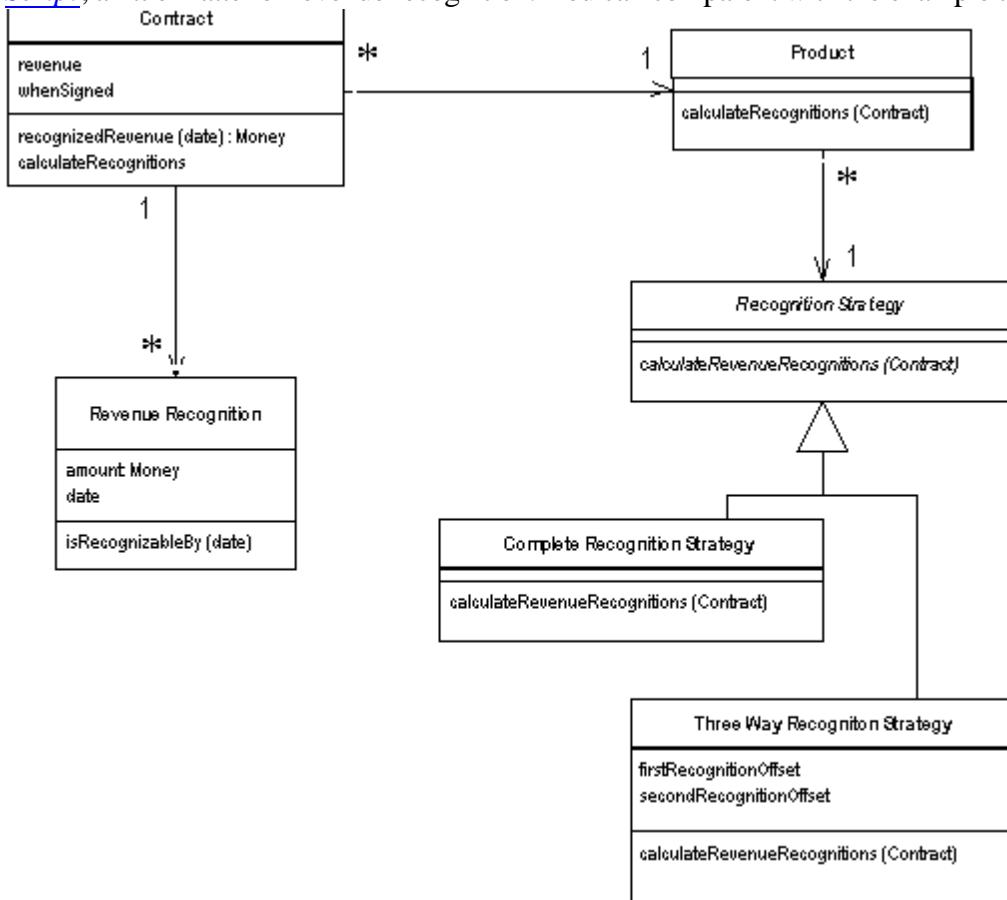


Figure 1: Class diagram of the example classes for a domain model

An immediate thing to notice is that every class, even in this small example, contains both behavior and data. Even the humble Revenue Recognition class contains a simple method to find out if that object's value is recognizable on a certain date.

```

class RevenueRecognition...
private Money amount;
private MfDate date;

public RevenueRecognition(Money amount, MfDate date) {
    this.amount = amount;
    this.date = date;
}

public Money getAmount() {
    return amount;
}
  
```

```
}
```

```
boolean isRecognizableBy(MfDate asOf) {
    return asOf.after(date) || asOf.equals(date);
}
```

Calculating how much revenue is recognized on a particular date involves both the contract and revenue recognition classes.

```
class Contract...
private List revenueRecognitions = new ArrayList();

public Money recognizedRevenue(MfDate asOf) {
    Money result = Money.dollars(0);
    Iterator it = revenueRecognitions.iterator();
    while (it.hasNext()) {
        RevenueRecognition r = (RevenueRecognition) it.next();
        if (r.isRecognizableBy(asOf))
            result = result.add(r.getAmount());
    }
    return result;
}
```

A common thing you find in domain models is how multiple classes interact in order to do even the simplest tasks. This is what often leads to the complaint that with OO programs you spend a lot of time hunting around from class to class trying to find the program. There's a lot of sense to this complaint. The value comes as the decision whether something is recognizable by a certain date gets more complex and as other objects need to know. Containing the behavior on the object that needs to know both avoids duplication and reduces coupling between the different objects.

Looking at calculating and creating these revenue recognition objects further demonstrates this notion of lots of little objects. In this case the calculation and creation begins with the customer and is handed off via the product to a strategy hierarchy. The strategy pattern is a well known OO pattern that allows you combine a group of operations in a small class hierarchy. Each instance of product is connected to a single instance of Recognition Strategy which determines which algorithm is used to calculate revenue recognition. In this case we have two subclasses of Recognition Strategy for the two different cases. The structure of the code looks like this.

```
class Contract...
private Product product;
private Money revenue;
private MfDate whenSigned;
private Long id;

public Contract(Product product, Money revenue, MfDate whenSigned) {
    this.product = product;
    this.revenue = revenue;
    this.whenSigned = whenSigned;
}
class Product...
private String name;
private RecognitionStrategy recognitionStrategy;

public Product(String name, RecognitionStrategy recognitionStrategy) {
    this.name = name;
    this.recognitionStrategy = recognitionStrategy;
```

```
}

public static Product newWordProcessor(String name) {
return new Product(name, new CompleteRecognitionStrategy());
}

public static Product newSpreadsheet(String name) {
return new Product(name, new ThreeWayRecognitionStrategy(60, 90));
}

public static Product newDatabase(String name) {
return new Product(name, new ThreeWayRecognitionStrategy(30, 60));
}

class RecognitionStrategy...
abstract void calculateRevenueRecognitions(Contract contract);
class CompleteRecognitionStrategy...
void calculateRevenueRecognitions(Contract contract) {
contract.addRevenueRecognition(new RevenueRecognition(contract.getRevenue(),
contract.getWhenSigned()));
}
class ThreeWayRecognitionStrategy...
private int firstRecognitionOffset;
private int secondRecognitionOffset;

public ThreeWayRecognitionStrategy(int firstRecognitionOffset,
int secondRecognitionOffset)
{
this.firstRecognitionOffset = firstRecognitionOffset;
this.secondRecognitionOffset = secondRecognitionOffset;
}

void calculateRevenueRecognitions(Contract contract) {
Money[] allocation = contract.getRevenue().allocate(3);
contract.addRevenueRecognition(new RevenueRecognition
(allocation[0], contract.getWhenSigned()));
contract.addRevenueRecognition(new RevenueRecognition
(allocation[1], contract.getWhenSigned().addDays(firstRecognitionOffset)));
contract.addRevenueRecognition(new RevenueRecognition
(allocation[2], contract.getWhenSigned().addDays(secondRecognitionOffset)));
}
```

The great value of the strategies is that they provide well-contained plug points to extend the application. Adding a new revenue recognition algorithm involves creating a new subclass and overriding the calculateRevenueRecognitions method. This makes it easy to extend the algorithmic behavior of the application.

When you create products, you hook them up with the appropriate strategy objects. In this case I'm doing this in my test code.

```
class Tester...
private Product word = Product.newWordProcessor("Thinking Word");
private Product calc = Product.newSpreadsheet("Thinking Calc");
private Product db = Product.newDatabase("Thinking DB");
```

Once everything is setup, then calculating the recognitions doesn't involve any knowledge of the strategy subclasses.

```
class Contract...
public void calculateRecognitions() {
product.calculateRevenueRecognitions(this);
}
class Product...
void calculateRevenueRecognitions(Contract contract) {
recognitionStrategy.calculateRevenueRecognitions(contract);
}
```

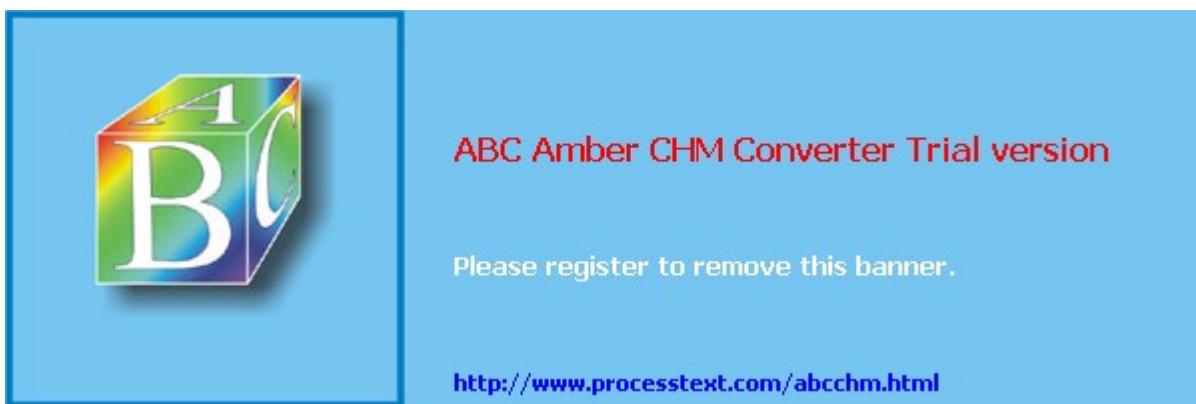
The OO habit of successive forwarding from object to object both moves the behavior to the object most qualified to handle it, but also resolves much of the conditional behavior. You'll notice there's no conditionals in this calculation. The decision path was set up when the products were created with the appropriate strategy, once setup like this the algorithms just follow the path. Domain models work very well when you have examples of similar conditionals because the similar conditionals can be factored out into the object structure itself. This moves complexity out of the algorithms and into the relationships between objects. The more similar the logic, the more you find the same network of relationships used by different parts of the system. Any algorithm that's dependent on the type of recognition calculation can follow this particular network of objects.

You'll notice that in this example, I've not shown anything about how the objects get retrieved from, and written to, the database. This is for a couple of reasons. Firstly mapping a *Domain Model* to a database is always somewhat hard, so I'm chickening out and not providing an example. Secondly in many ways the whole point of a *Domain Model* is to hide the database, both from upper layers but also from people working the *Domain Model* itself. So hiding it here reflects what it's like to actually program in this environment.



---

© Copyright [Martin Fowler](#), all rights reserved

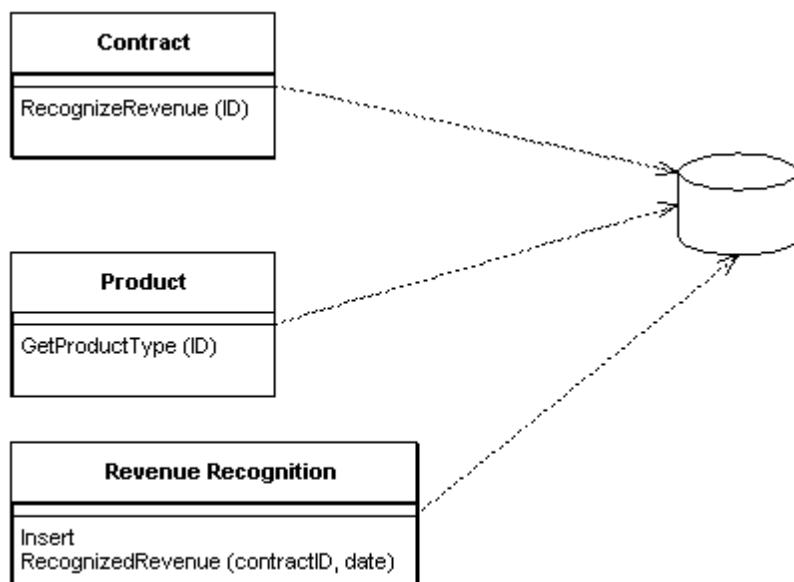


---

# Table Module

---

*Provide a single object for all the behavior on a table*



One of the key messages of object-orientation is to bundle together data with the behavior that uses that data. The traditional object-oriented approach is based on objects with identity, along the lines of [Domain Model](#). So if we have an employee class, any instance of that class corresponds to a particular employee. This scheme works well because once we have a reference to an employee, we can execute operations, follow relationships, and gather data on that individual.

One of the problems with [Domain Model](#) is the interface with relational databases. In many ways the [Domain Model](#) approach treats the relational database like some old granny that's shut up in an attic and nobody wants to talk about. As a result you often need considerable programmatic gymnastics to pull data in and out of the database, transforming between two different representations of the data.

A *Table Module* organizes domain logic with one class per table in the database, and a single instance of a class contains the various procedures that will act on the data. The primary distinction with [Domain Model](#) is that if you have many orders, a [Domain Model](#) will have one order object per order, while a *Table Module* will have one object to handle all the orders.

## How it Works

The strength of *Table Module* is that it allows you to package the data and behavior together, while at the same time playing to the strengths of a relational database. On the surface *Table Module* looks much the same as regular objects. The key difference is that *Table Module* has no notion of an identity for the objects that it's working with. So if you wanted to obtain the address of an employee you would have a method like `anEmployeeModule.getAddress(long employeeID)`. Every time you want to do something to a particular employee you have to pass in some kind of identity reference. Often this will be the primary key that's used in the database.

Usually you use *Table Module* with a backing data structure that's table-oriented. The tabular data is usually the result of a SQL call, and held in some record set object that mimics a SQL table. The *Table Module* gives you an explicit method based interface that acts on the tabular data.

The *Table Module* may be an instance or it may be a collection of static methods. The advantage of an instance is that it allows you to initialize the *Table Module* with an existing record set, perhaps the result of a query. We can then use the instance to manipulate the rows in the record set. Using instances also makes it possible to use inheritance, so we can write a ManagerModule that contains additional behavior.

The *Table Module* may include queries as factory methods. The alternative route is to have a [\*Table Data Gateway\*](#). The disadvantage of using an extra [\*Table Data Gateway\*](#) class and mechanism in the design. The advantage of the [\*Table Data Gateway\*](#) is that it allows you to use a single *Table Module* on data from different data sources, since you use a different [\*Table Data Gateway\*](#) for each data source.

## When to Use it

*Table Module* is very much based on table-oriented data, so obviously it makes sense when you are accessing tabular data. It also puts that data structure very much into the center of the code, so you also want the way you access the data structure to be fairly straightforward.

The most well-known situation where I've come across this pattern is in Microsoft COM designs. In COM the record set is the primary repository of data in an application. Record sets can be passed to the UI where data aware widgets display information. Microsoft's ADO libraries give you a good mechanism to access the relational data as record sets. In this situation *Table Module* allows you to fit business logic into the application in a well organized way, without losing the way the various elements work on the tabular data.

So you can access data from the database and pass it to a *Table Module*. You can then use the *Table Module* to perform calculations of derived data which adds to the underlying data set. Then you can pass the data set to the UI for display and modification using the table aware widgets. The table aware widgets can't tell if the record sets came directly from the relational database, or if a *Table Module* manipulated the data on the way out. After modification in the GUI, the data set goes back to the *Table Module* for validation before it's saved to the database.

However *Table Module* does not give you the full power of objects in organizing complex logic. You can't have direct instance to instance relationships, and polymorphism does not work well. So for handling complicated domain logic, a [\*Domain Model\*](#) is a better choice. Essentially you have to trade off

Domain Model's ability to handle complex logic versus Table Module's easier integration with the underlying table oriented data structures.

If the objects in a Domain Model and the database tables are relatively similar, then it may be better to use a Domain Model that uses Active Record. Table Module works better than using a combination of Domain Model and Active Record when other parts of the application are based around a common table-oriented data structure. That's why you don't usually see Table Module very much in the Java environment, although that may change as row sets become more widely used.

## Example: Revenue Recognition with a Table Module (C#)

Time to revisit the [revenue recognition example](#) I used in the other domain modelling patterns, but this time with a Table Module. To recap, our mission is to recognize revenue on orders when the rules vary depending on the type of product. In this fantasy example we have different rules for word processors, spreadsheets, and databases.

Table Module is based on a data schema of some kind, usually a relational data model (although in the future we may well see XML as model used in a similar way). In this case I'll use the relational schema from Figure 1.

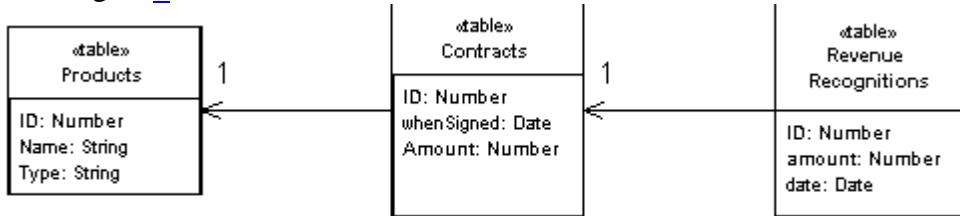


Figure 1: Database schema for revenue recognition

The classes that manipulate this data are of pretty much the same form, there is one Table Module class for each table. In the .NET architecture there is a data set object which provides an in-memory representation of a database structure. So it makes sense to create classes that operate on this data set. Each Table Module class has a data member of a Data Table, which is the .NET system class corresponding to a table within the data set. This ability to read a table is common to all the Table Modules and so can appear in the superclass

```

class TableModule...
protected DataTable table;
protected TableModule(DataSet ds, String tableName) {
    table = ds.Tables[tableName];
}
  
```

The subclass constructor calls the superclass constructor with the correct table name.

```

class Contract...
public Contract (DataSet ds) : base (ds, "Contracts") { }
  
```

This allows me to create a new *Table Module* by just passing in a data set to the *Table Module's* constructor

```
contract = new Contract(dataset);
```

This keeps the code that creates the data set away from the *Table Modules*, following the guidelines of using ADO.NET.

A useful capability is to use the C# indexer to get to a particular row in the data table given the primary key.

```
class Contract...
public DataRow this [long key] {
get {
String filter = String.Format("ID = {0}", key);
return table.Select(filter)[0];
}
}
```

The first piece of functionality is to calculate the revenue recognition for a contract, updating the revenue recognition tables accordingly. The amount recognized depends on the kind of product we have. Since this behavior mainly uses data from the contract table, I decided to add the method to the contract class.

```
class Contract...
public void CalculateRecognitions (long contractID) {
DataRow contractRow = this[contractID];
Decimal amount = (Decimal)contractRow["amount"];
RevenueRecognition rr = new RevenueRecognition (table.DataSet);
Product prod = new Product(table.DataSet);
long prodID = GetProductId(contractID);
if (prod.GetProductType(prodID) == ProductType.WP) {
rr.Insert(contractID, amount, (DateTime) GetWhenSigned(contractID));
} else if (prod.GetProductType(prodID) == ProductType.SS) {
Decimal[] allocation = allocate(amount,3);
rr.Insert(contractID, allocation[0], (DateTime) GetWhenSigned(contractID));
rr.Insert(contractID, allocation[1], (DateTime)
GetWhenSigned(contractID).AddDays(60));
rr.Insert(contractID, allocation[2], (DateTime)
GetWhenSigned(contractID).AddDays(90));
} else if (prod.GetProductType(prodID) == ProductType.DB) {
Decimal[] allocation = allocate(amount,3);
rr.Insert(contractID, allocation[0], (DateTime) GetWhenSigned(contractID));
rr.Insert(contractID, allocation[1], (DateTime)
GetWhenSigned(contractID).AddDays(30));
rr.Insert(contractID, allocation[2], (DateTime)
GetWhenSigned(contractID).AddDays(60));
} else throw new Exception("invalid product id");
}
private Decimal[] allocate(Decimal amount, int by) {
Decimal lowResult = amount / by;
lowResult = Decimal.Round(lowResult,2);
Decimal highResult = lowResult + 0.01m;
Decimal[] results = new Decimal[by];
int remainder = (int) amount % by;
for (int i = 0; i < remainder; i++) results[i] = highResult;
for (int i = remainder; i < by; i++) results[i] = lowResult;
return results;
}
```

Usually I would use Money here, but for variety's sake, I'll show this using a decimal. I use a similar allocation method to the one I would use for a Money.

To carry this out, we need some behavior that's defined on the other classes. The product needs to be able to tell us which product type it is. We can do this with an enum for the product type and a lookup method.

```
public enum ProductType {WP, SS, DB};  
class Product...  
public ProductType GetProductType (long id) {  
String typeCode = (String) this[id]["type"];  
return (ProductType) Enum.Parse(typeof(ProductType), typeCode);  
}
```

GetProductType encapsulates the data in the data table. There is an argument for doing this for all the columns of data, as opposed to accessing them directly as I did with the amount on the contract. While encapsulation is generally a Good Thing, I don't do it here because it doesn't fit in with the assumption of the environment that different parts of the system access the data set directly. There's no encapsulation when the data set moves over to the UI. So column access functions really only make sense when there's some additional functionality to be done, such as converting a string to a product type.

The other additional behavior is inserting a new revenue recognition record.

```
class RevenueRecognition...  
public long Insert (long contractID, Decimal amount, DateTime date) {  
DataRow newRow = table.NewRow();  
long id = GetNextID();  
newRow[ "ID" ] = id;  
newRow[ "contractID" ] = contractID;  
newRow[ "amount" ] = amount;  
newRow[ "date" ]= String.Format("{0:s}", date);  
table.Rows.Add(newRow);  
return id;  
}
```

Again the point of this method is less to encapsulate the data row, and more to have a method instead of several lines of code that would get repeated.

The second behavior is to sum up all the revenue recognized on a contract by a given date. Since this uses the revenue recognition table it makes sense to define the method there.

```
class RevenueRecognition...  
public Decimal RecognizedRevenue (long contractID, DateTime asOf) {  
String filter = String.Format("ContractID = {0} AND date <= #{1:d}#",  
contractID,asOf);  
DataRow[] rows = table.Select(filter);  
Decimal result = 0m;  
foreach (DataRow row in rows) {  
result += (Decimal)row[ "amount" ];  
}  
return result;  
}
```

This fragment takes advantage of the really nice feature of ADO.NET that allows you to define a where clause and then select a subset of the data table to manipulate. Indeed you can go further and use an aggregate function.

```
class RevenueRecognition...
public Decimal RecognizedRevenue2 (long contractID, DateTime asOf) {
String filter = String.Format("ContractID = {0} AND date <= #{1:d}#", contractID,asOf);
String computeExpression = "sum(amount)";
Object sum = table.Compute(computeExpression, filter);
return (sum is System.DBNull) ? 0 : (Decimal) sum;
}
```



---

© Copyright [Martin Fowler](#), all rights reserved

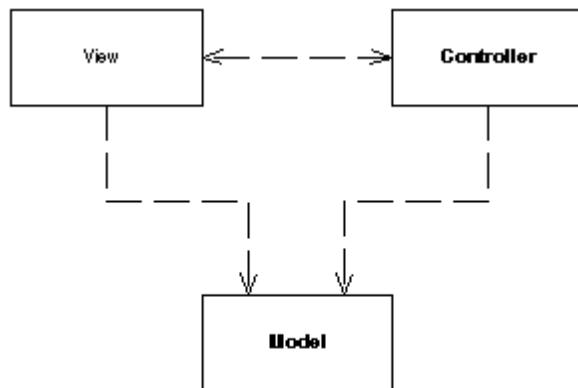


---

# Model View Controller

---

*Split user interface interaction into three distinct roles*



Model View Controller (MVC) has been one of the most quoted (and most misquoted) patterns around. It started life as a framework developed by Trygve Reenskaug for the Smalltalk platform in the late 70's. Since then it's played an influential role for most UI frameworks and thinking about UI design.

## How it Works

MVC considers three roles. The model is an object that represents some information about the domain. It is a non-visual object and contains all the data and behavior other than that used for the UI. In its most pure OO notion the model is an object within a [Domain Model](#). You might also think of a [Transaction Script](#) as the model providing the [Transaction Script](#) contained no UI machinery. Such a definition stretches the notion of model, but fits the role breakdown of MVC.

The view represents the display of the model in the UI. So if our model is a customer object our view might be a frame full of UI widgets or an HTML page rendered with information from the model. The view is only about display of information, any changes to the information are handled by the third member of the MVC trinity: the controller. The controller takes user input and manipulates the model and causes the view to update appropriately. The UI is therefore a combination of the view and the controller.

As I think about MVC I see two principal separations: separating the presentation from the model and separating the controller from the view. Of these the separation of presentation from model is one of the most fundamental heuristics of good software design. This separation is important for several reasons

- Fundamentally they are about different concerns. When you are developing a view you are thinking about the mechanisms of UI and how to lay out a good user interface. When you are working with a model you are thinking about business policies, perhaps database interactions. Certainly you will use different very different libraries when working with one or the other. Often you find people prefer one area to another and you'll see people specialize in one side of the line.
- Depending on context users like to see the same basic model information in different ways. Separating the two allows you develop multiple presentations, indeed entire different kinds of interfaces and yet use the same model code. In its most noticeable form this could be providing a rich client, web browser, remote API, and command line interface to the same model. Even within a sole web interface you might have different customer pages at different points in an application.
- Non visual objects are usually easier to test than visual ones. Separating presentation and model allows you to test all the domain logic easily without resorting to awkward GUI scripting tools and the like.

A key point in this separation is the direction of the dependencies: the presentation depends on the model but not the other way around. People programming in the model should be entirely unaware of what presentation is being used. This both simplifies their task and makes it easier to add new presentations later on. It also means that presentation changes can be made freely without altering the model.

This principle introduces a common issue. With a rich client interface of multiple windows it's quite likely that there will be several presentations of a model on a screen at once. If a user makes a change to the model from one presentation, then the others need to change. To do this without creating a dependency you'll usually find the [observer pattern](#), such as event propagation or a listener. The presentation acts as the observer of the model: whenever the model changes it sends out an event and the presentations refresh the information.

The second separation, that of view and controller, is less important. Indeed the irony is that almost every version of Smalltalk didn't actually make a view/controller separation. The classic example of why you'd want to separate them is to support editable and non-editable behavior. You can do this with one view and two controllers for the two cases, where the controllers are [strategies](#) for the view. But in practice most systems have only one controller per view, so this separation is usually not done. However this separation has come back into vogue with web interfaces where it does become useful to separate the controller and view again.

The fact that most GUI frameworks combine view and controller has led to the many misquotations of MVC that I've run into. The model and the view is obvious - but where is the controller. The common idea is that it sits between the model and the view as in the [Application Controller](#), a fact that isn't helped that the word "controller" is used in both contexts. Whatever the merits of a [Application Controller](#) it's a very different beast to an MVC controller

For the purposes of this set of patterns these principles are really all you need to know. If you want to dig deeper into MVC the best available reference is [\[POSA\]](#).

## When to Use it

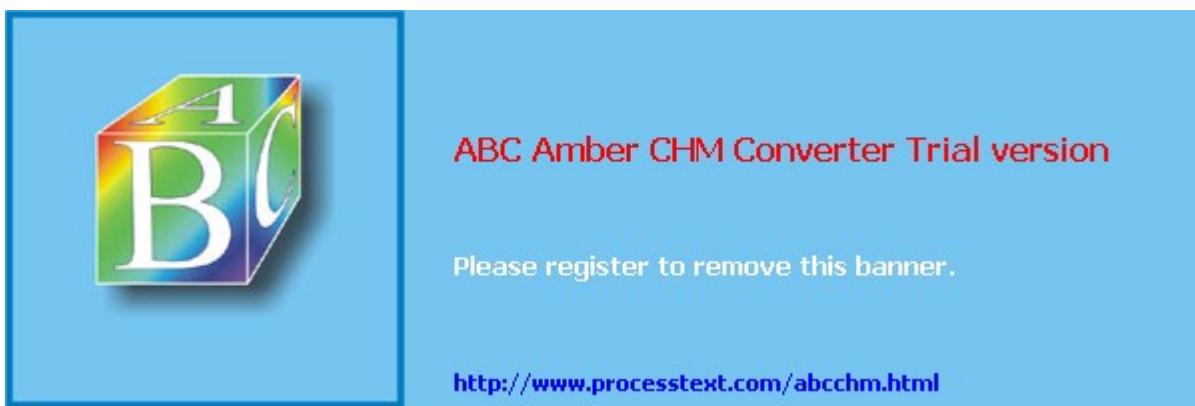
The value of MVC really lies in its two principles. Of these the separation of presentation and model is

one of the most important design principles in software and the only time you shouldn't follow it is in very simple systems where the model wouldn't have any real behavior in it anyway. As soon as you get some non visual logic you should look to apply the separation. Unfortunately a lot of UI frameworks make it difficult, and those that don't often are taught without a separation.

The separation of view and controller is less important, so I'd only recommend doing it when it is really helpful. For rich client systems that ends up being hardly ever. It is however common in web front ends where the controller is separated out - most of the patterns on web design here are based on that principle.



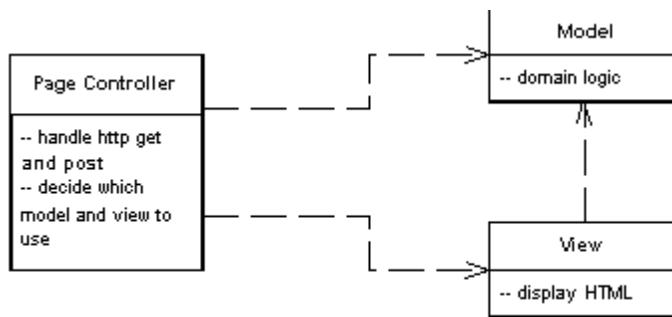
© Copyright [Martin Fowler](#), all rights reserved



# Page Controller

---

*An object that handles a request for a specific page or action on a web site*



Most people's basic web experience is with static HTML pages. When you request static HTML you pass to the web server the name and path for a HTML document stored on the web server. The key notion is that each page on the web site is a separate document on the server. With dynamic pages things can get much more interesting since there is a much more complex relationship between path names and the file that responds. However the approach of one path leading to one file that handles the request is a simple model to understand.

As a result *Page Controller* has one input controller for each logical page of the web site. That controller may be the page itself, as it often is in server page environments, or it may be a separate object that corresponds to that page.

## How it Works

The basic idea behind a *Page Controller* is to have one module on the web server act as the controller for each page on the web site. In practice, it doesn't work out to exactly one per page, since sometimes you may hit a link and get a different page depending on some dynamic information. More strictly the controllers tie to each *action* on the web site, where an action may be clicking a link or a button.

The *Page Controller* can be structured either as a script (CGI script, servlet, ...), or a server page (ASP, PHP, JSP, ...). Using a server page usually combines the *Page Controller* and a [\*Template View\*](#) into the same file. While this works well for the [\*Template View\*](#) it works less well for the *Page Controller* since it is more awkward to properly structure the module. If the page is a simple display then this is not a problem. If there is logic involved in either pulling data out of the request, or deciding which actual view to display, then you can end up with awkward scriptlet code in the server page.

One way of dealing with this scriptlet code is to use a helper object. In this case the first thing the server page does is to call the helper object to handle all the logic. The helper may return control to the original server page, or it may forward to a different server page to act as the view. In this case the server page is the handler of the request, but most of the controller logic lies in the helper object

Another approach is to let a script be the handler and controller. The web server passes control to the script, the script carries out the controller's responsibilities, and finally forward to an appropriate view to display any results.

The basic responsibilities of a *Page Controller* are:

- Decode the URL and extract any form data to figure out all the data for the action
- Create and invoke any model objects to process the data. All relevant data from the HTML request should be passed to the model so that the model objects do not need any connection to the HTML request
- Determine which view should display the result page and forward the model information to that view.

The *Page Controller* need not be a single class, it can invoke helper objects. This is particularly useful if several handlers need to do similar tasks. A helper class can then be a good spot to put any code that would otherwise be duplicated.

There's no reason why you can't have some URLs handled by server pages and some URLs handled by scripts. Any URLs that have little or no controller logic are best handled with a server page, since that provides a simple mechanism that's easy to understand and modify. Any URLs with more complicated logic go to a script. I've often come across teams who want to handle everything the same way, either everything is a server page or everything is a script. Any advantages of consistency in such an application are usually offset by the problems of either scriptlet laden server pages or lots of simple pass-through scripts.

## When to Use it

The main decision point is whether to use *Page Controller* or *Front Controller*. Of the two *Page Controller* is the most familiar one to work with and leads to a natural structuring mechanism where particular actions are handled by particular server pages or script classes. You thus have to trade off the greater complexity of the *Front Controller* against the various advantages that it offers, most of which make a difference in web sites that have more navigational complexity.

*Page Controller* works particularly well in a site where most of the controller logic is pretty simple. In this case most URLs can be handled with a server page with the more complicated cases in helpers. When your controller logic is simple, *Front Controller* adds a lot of overhead.

It's not uncommon to have a mix in a site where some requests are dealt with by *Page Controllers* and others are dealt with by *Front Controllers*, particularly when a team is refactoring from one to another. The two patterns actually mix together without too much trouble.

## Example: Simple Display with a servlet

# controller and a JSP view (Java)

A simple example of an action controller is to display some information about something. Here we'll use an example of displaying some information about a recording artist. The URL would run something along the lines of <http://www.thingy.com/recordingApp/artist?name=danielaMercury>

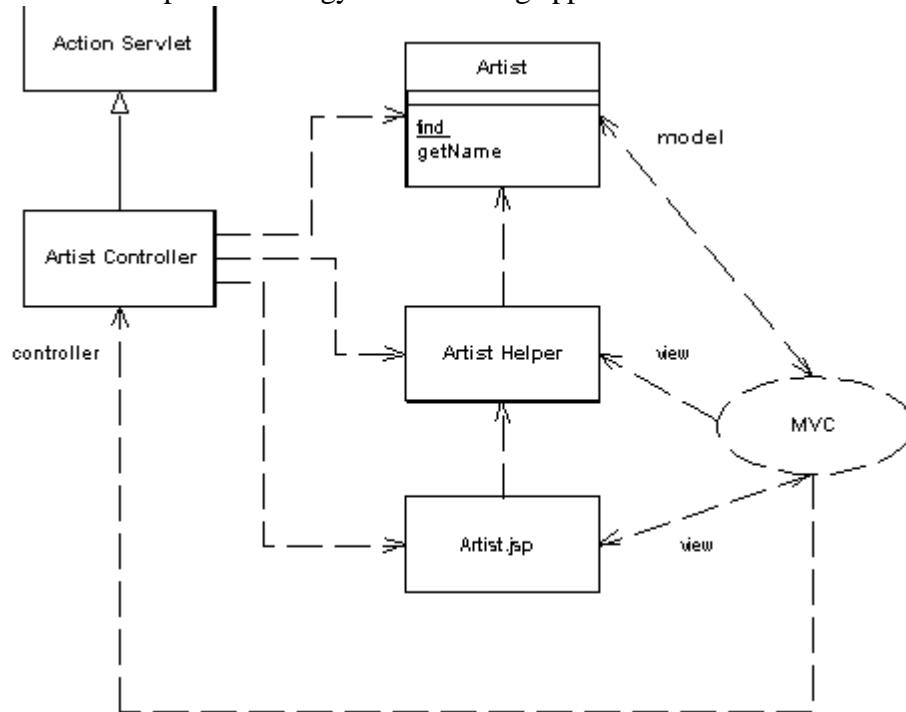


Figure 1: Classes involved in a simple display with an action controller servlet and JSP view

The web server needs to be configured to recognize "/artist" as a call to ArtistController. In Tomcat you do this with the following code in the web.xml file.

```

<servlet>
    <servlet-name>artist</servlet-name>
    <servlet-class>actionController.ArtistController</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>artist</servlet-name>
    <url-pattern>/artist</url-pattern>
</servlet-mapping>
  
```

The artist controller needs to implement a method to handle the request.

```

class ArtistController...
public void doGet(HttpServletRequest request, HttpServletResponse response)
throws IOException, ServletException {
Artist artist = Artist.findNamed(request.getParameter("name"));
if (artist == null)
forward("/MissingArtistError.jsp", request, response);
else {
request.setAttribute("helper", new ArtistHelper(artist));
forward("/artist.jsp", request, response);
  
```

```
}
```

Although this is a very simple case, it covers the salient points. First the controller needs to get create the necessary model objects to do their thing, in this case it just need to find the correct model object to display. The second part is to put the right information in the http request so that the JSP can display it properly. In this case it creates a helper and puts it into the request. Finally it forwards to the [Template View](#) to handle the display of the data. Forwarding is a common behavior, so it sits naturally on a superclass for all action controllers

```
class ActionServlet...
protected void forward(String target,
    HttpServletRequest request,
    HttpServletResponse response)
throws IOException, ServletException
{
RequestDispatcher dispatcher =
getServletContext().getRequestDispatcher(target);
dispatcher.forward(request, response);
}
```

The main point of coupling between the [Template View](#) and the *Page Controller* is the parameter names in the request to pass on any objects that the JSP needs to use.

In this case the controller logic is really very simple, but as we get more complex controller logic we can continue to use the servlet as a controller. We can have a similar behavior for albums, with the twist that classical albums both have a different model object and are rendered with a different JSP. To do this behavior we can again use a controller class.

```
class AlbumController...
public void doGet(HttpServletRequest request, HttpServletResponse response)
throws IOException, ServletException
{
Album album = Album.find(request.getParameter("id"));
if (album == null) {
forward("/missingAlbumError.jsp", request, response);
return;
}
request.setAttribute("helper", album);
if (album instanceof ClassicalAlbum)
forward("/classicalAlbum.jsp", request, response);
else
forward("/album.jsp", request, response);
}
```

Notice that in this case I'm using the model objects as the helper, rather than creating a separate helper class. This is worth doing if the helper class would otherwise be just a dumb forwarder to the model class. When doing this make sure that the model class doesn't have any servlet dependent code in it - if there is any servlet dependent code it should sit in a separate helper class.

## Example: Using a JSP as a handler (Java)

Using a servlet as a controller is one route to take, but the most common route is to have the server page itself be the controller. The problem with this approach is that it results in scriptlet code at the beginning of the server page, and as you may have gathered I think that scriptlet code has the same relationship to well-designed software that professional wrestling does to sport.

Despite this you can have a server page as the handler of the request, while delegating control to the helper to actually carry out the controller function. This preserves the simple property of having your URLs be denoted by server pages. In this case I'll do this for the album display, where you can display an album with a URL of the form `http://localhost:8080/isa/album.jsp?id=zero`. Most albums are displayed directly with the album JSP, but classical albums require a different display and are displayed with a classicalAlbum JSP.

This controller behavior appears in a helper class to the JSP. The helper is set up in the album JSP itself. `album.jsp...`

```
<jsp:useBean id="helper" class="actionController.AlbumConHelper"/>
<%helper.init(request, response);%>
```

The call to init sets the helper up to carry out the controller behavior.

```
class AlbumConHelper extends HelperController...
public void init(HttpServletRequest request, HttpServletResponse response) {
super.init(request, response);
if (getAlbum() == null) forward("missingAlbumError.jsp", request, response);
if (getAlbum() instanceof ClassicalAlbum) {
request.setAttribute("helper", getAlbum());
forward("/classicalAlbum.jsp", request, response);
}
}
```

Common helper behavior naturally sits on a helper superclass.

```
class HelperController...
public void init(HttpServletRequest request, HttpServletResponse response) {
this.request = request;
this.response = response;
}

protected void forward(String target,
HttpServletRequest request,
HttpServletResponse response)
{
try {
RequestDispatcher dispatcher = request.getRequestDispatcher(target);
if (dispatcher == null) response.sendError(response.SC_NO_CONTENT);
else dispatcher.forward(request, response);
} catch (IOException e) {
throw new ApplicationException(e);
} catch (ServletException e) {
throw new ApplicationException(e);
}
}
```

The key difference between the controller behavior here and that when using a servlet, is that the handler JSP is also the default view. Unless the controller forwards to a different JSP, control reverts to the original handler. This is an advantage when you have pages where most of the time the JSP directly acts as the view. In these cases there's no forwarding to be done. The initialization of the helper acts to kick off any model behavior and set things up for the view later on. It's a simple model to follow, since people generally associate a web page with the server page that is the view for the page. This also often fits naturally with web server configuration.

The call to initialize the handler is rather clumsy, in a JSP environment this awkwardness can be handled much better with a custom tag. Such a tag can automatically create an appropriate object, put it in the request, and initialize it. With that all you need is to put a simple tag into the JSP page.

```
<helper:init name = "actionController.AlbumConHelper" />
```

The custom tag's implementation then does the work.

```
class HelperInitTag extends HelperTag...
private String helperClassName;

public void setName(String helperClassName) {
this.helperClassName = helperClassName;
}

public int doStartTag() throws JspException {
HelperController helper = null;
try {
helper = (HelperController) Class.forName(helperClassName).newInstance();
} catch (Exception e) {
throw new ApplicationException("Unable to instantiate " + helperClassName, e);
}
initHelper(helper);
pageContext.setAttribute(HELPER, helper);
return SKIP_BODY;
}

private void initHelper(HelperController helper) {
HttpServletRequest request = (HttpServletRequest) pageContext.getRequest();
HttpServletResponse response = (HttpServletResponse)
pageContext.getResponse();
helper.init(request, response);
}

class HelperTag...
public static final String HELPER = "helper";
```

If I'm going to use a custom tag like this, I might as well make custom tags for property access too.

```
class HelperGetTag extends HelperTag...
private String propertyName;

public void setProperty(String propertyName) {
this.propertyName = propertyName;
}

public int doStartTag() throws JspException {
try {
```

```
pageContext.getOut().print(getProperty(propertyName));
} catch (IOException e) {
throw new JspException("unable to print to writer");
}
return SKIP_BODY;
}

class HelperTag...
protected Object getProperty(String property) throws JspException {
Object helper = getHelper();
try {
final Method getter = helper.getClass().getMethod(gettingMethod(property),
null);
return getter.invoke(helper, null);
} catch (Exception e) {
throw new JspException
("Unable to invoke " + gettingMethod(property) + " - " + e.getMessage());
}
}

private Object getHelper() throws JspException {
Object helper = pageContext.getAttribute(HELPER);
if (helper == null) throw new JspException("Helper not found.");
return helper;
}

private String gettingMethod(String property) {
String methodName = "get" + property.substring(0, 1).toUpperCase() +
property.substring(1);
return methodName;
}
```

(You may be thinking that it's better to use the Java beans mechanism than to just invoke a getter using reflection. If so, you're probably right... and also probably intelligent enough to figure out how to change the method to do that)

With the getting tag defined, I can use it to pull information out of the helper. The tag is both shorter and removes the chances of me mizpelling 'helper'.

```
<B><helper:get property = "title"/></B>
```

## Example: Page Handler with a Code Behind (.NET)

The web system in .NET is designed to work with the *Page Controller* and [Template View](#) patterns, although you can certainly decide to handle web events with a different approach. With this example, I'll take the preferred style of .NET, with the presentation layer built on top of a domain using [Table Module](#), using data sets as the main carrier of information between the layers.

For our example this time we'll have a page that displays runs scored and the run rate for one innings of a cricket match. As I know I'll have many readers who are afflicted with no material experience of this art form, I can summarize by saying that the runs scored is the score of batsman and the run rate is how

many runs they score divided by the number of balls they face. The runs scored and balls faced are in the database, the run rate needs to be calculated by the application - a tiny but pedagogically useful piece of domain logic.

The handler in this design is an ASP.NET web page, captured in a .aspx file. As with other server page constructs, the aspx file allows you to embed programming logic directly into the page as scriptlets. Since you know I'd rather drink bad beer than write scriptlets, you know there's little chance that I'd want to do that. My savior in this case is the code behind mechanism that allows you to associate a regular file and class with the aspx page. You signal this in the header of the aspx page.

```
<%@ Page language="c#" Codebehind="bat.aspx.cs" AutoEventWireup="false"
trace="False" Inherits="batsmen.BattingPage" %>
```

The page is setup as a subclass of the code behind class, and as such can use all the protected properties and methods of the code behind. The page object is the handler of the request, and the code behind can define how to handle the request by defining a `Page_Load` method. If most pages follow a common flow, I can define a [Layer Supertype](#) that has a [template method](#) for this.

```
class CricketPage...
protected void Page_Load(object sender, System.EventArgs e) {
db = new OleDbConnection(DB.ConnectionString);
if (hasMissingParameters())
errorTransfer (missingParameterMessage);
DataSet ds = getData();
if (hasNoData (ds))
errorTransfer ("No data matches your request");
applyDomainLogic (ds);
.DataBind();
prepareUI(ds);
}
```

The template method breaks the handling of the request down into a number of common steps. This way we can define a single common flow for handling web requests, while allowing each *Page Controller* to supply implementations for the specific steps. If you do this, you will find that once you've written a few *Page Controllers*, you will find what common flow to use for the template method. If any page needs to do something completely different, it can always override the page load method.

The first task is to do validation on the parameters coming into the page. In a more realistic example this might do initial sanity checking of various form values, but in this case we are just decoding a URL of the form `http://localhost/batsmen/bat.aspx?team=England&innings=2&match=905`. The only validation in this example is that the various parameters required for the database query are present. As usual I've been overly simplistic in the error handling until somebody writes a good set of patterns on validation. So here the particular page defines a set of mandatory parameters and the [Layer Supertype](#) has the logic for checking them.

```
class CricketPage...
abstract protected String[] mandatoryParameters();
private Boolean hasMissingParameters() {
foreach (String param in mandatoryParameters())
if (Request.Params[param] == null) return true;
return false;
}
private String missingParameterMessage {
get {
String result = "<P>This page is missing mandatory parameters:</P>" ;
}}
```

```
result += "<UL>";
foreach (String param in mandatoryParameters())
if (Request.Params[param] == null)
result += String.Format("<LI>{0}</LI>", param);
result += "</UL>";
return result;
}
}

protected void errorTransfer (String message) {
Context.Items.Add("errorMessage", message);
Context.Server.Transfer("Error.aspx");
}

class BattingPage...
override protected String[] mandatoryParameters() {
String[] result = {"team", "innings", "match"};
return result;
}
```

The next stage is to pull the data out of the database into a ADO.NET's disconnected data set object. In this case this is a single query to the batting table.

```
class CricketPage...
abstract protected DataSet getData();
protected Boolean hasNoData(DataSet ds) {
foreach (DataTable table in ds.Tables)
if (table.Rows.Count != 0) return false;
return true;
}

class BattingPage...
override protected DataSet getData() {
OleDbCommand command = new OleDbCommand(SQL, db);
command.Parameters.Add(new OleDbParameter("team", team));
command.Parameters.Add(new OleDbParameter("innings", innings));
command.Parameters.Add(new OleDbParameter("match", match));
OleDbDataAdapter da = new OleDbDataAdapter(command);
DataSet result = new DataSet();
da.Fill(result, Batting.TABLE_NAME);
return result;
}
private const String SQL =
@"SELECT * from batting
WHERE team = ? AND innings = ? AND matchID = ?
ORDER BY battingOrder";
```

Now the domain logic gets its turn to play. The domain logic is organized as a [Table Module](#). The controller passes the retrieved data set to the [Table Module](#) to process.

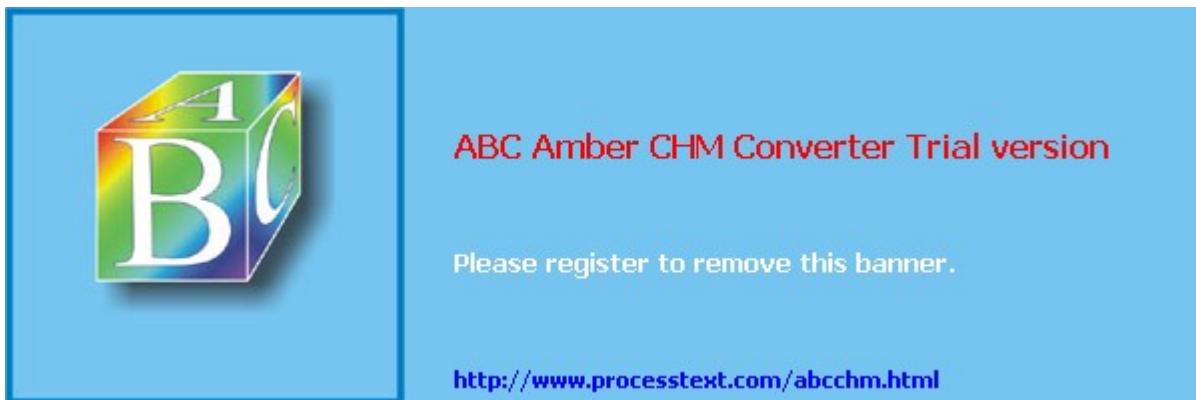
```
class CricketPage...
protected virtual void applyDomainLogic (DataSet ds) {}
class BattingPage...
override protected void applyDomainLogic (DataSet dataSet) {
batting = new Batting(dataSet);
batting.CalculateRates();
}
```

At this point the controller part of the page hander is done. By this I mean that in classic [Model View Controller](#) terms the controller should now hand over to the view to do display. In this design the BattingPage acts as both the controller and the view and the last call to prepareUI is part of the view behavior, so I can say farewell to this example in this pattern. However I suspect you'll find this to lack a certain dramatic closure, so you can find the example continued in [Template View](#).



---

© Copyright [Martin Fowler](#), all rights reserved

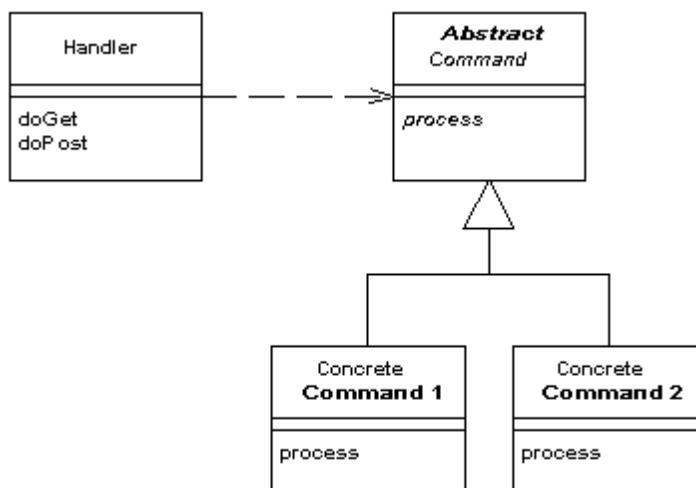


---

# Front Controller

---

*A controller that handles all requests for a web site*



In a complex web site, there are many similar things you need to do when handling a web request. These things include handling security, internationalization, providing particular views for certain kinds of users, etc. If the input controller behavior is scattered across multiple objects, then much of this behavior can end up duplicated, and it's difficult to change behavior at run time.

The *Front Controller* consolidates all of this by channeling all the requests through a single object. This handler can carry out common behavior which can be modified at runtime with decorators. The handler then dispatches to command objects for behavior particular to a request.

## How it Works

A *Front Controller* handles all calls for a web site. It's usually structured in two parts: a web handler and a hierarchy of commands. The web server software directs the http request to the handler. The handler pulls just enough information from the URL and request to decide what kind of action to initiate. The handler then delegates the request to a command, which usually isn't a web handler, to carry out the action (Figure 1).

The web handler is almost always implemented as a class, rather than a server page, as it doesn't produce any response. The commands are also classes rather than server pages, and indeed don't need

to be web handlers at all, although they are usually passed the http information. The web handler itself is usually a fairly simple program which does nothing other than decide which command should be run.

The web handler can make the decision of which command to run either statically or dynamically. The static version involves parsing the URL and using conditional logic to decide which command to run. The dynamic version usually involves taking a standard piece of the URL and using dynamic instantiation to create a command class.

The static case has the advantage of explicit logic, compile time error checking on the dispatch, and lots of flexibility in what your URLs look like. The dynamic case allows you to add new commands without changing the web handler.

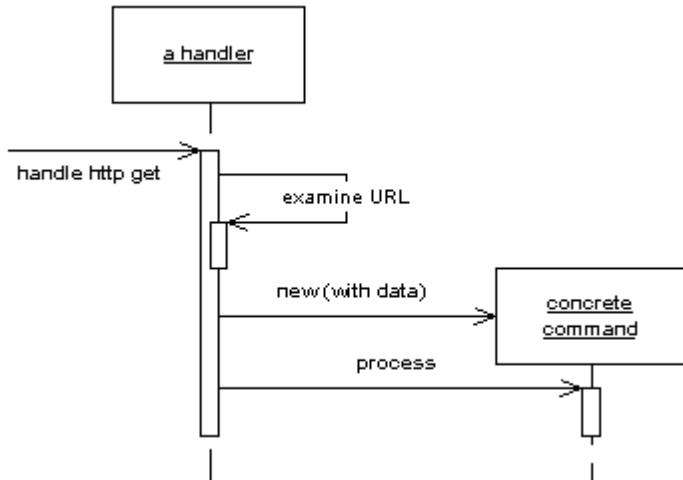


Figure 1: How the front controller works

A particularly useful pattern to use in conjunction with *Front Controller* is *Intercepting Filter*, described in [\[Alur, Crupi, and Malks\]](#). An intercepting filter is essentially a decorator that wraps the handler of the front controller. This allows you build a *filter chain* (or pipeline) of filters to handle different issues such as authentication, logging, locale identification. Using filters allows you to dynamically set up the filters to use at configuration time.

Rob Mee showed me an interesting variation of *Front Controller* using a two stage web handler. In this case the web handler is further separated into a degenerate web handler and a dispatcher. The degenerate web handler pulls the basic data out of the http parameters and hands it to the dispatcher in such a way that the dispatcher is completely independent of the web server framework. This makes testing easier because test code can drive the dispatcher directly without having to run in web server.

Remember that both the handler and the commands are part of the controller. As a result the commands can (and should) choose which view to use for the response. The only responsibility of the handler is in choosing which command to execute. Once it's done that, it plays no further part in that request.

## When to Use it

The *Front Controller* is a more complicated design than the it's obvious counterpart, the [\*Page Controller\*](#). It therefore needs a few advantages to be worth the effort.

Only one *Front Controller* needs to be configured into the web server, the web handler then does the rest of the dispatching. This simplifies the configuration of the web server, which is an advantage if the web server is awkward to configure. With dynamic commands you can add new commands without changing anything. It also eases porting, since you only have to register the handler in a web server specific way.

Since you create new command object with each request, you don't have to worry about making the command classes thread safe. This can avoid the headaches of multi-threaded programming. You do have to make sure that you don't share any other objects, such as the model objects.

A commonly stated advantage of a *Front Controller* is that it allows you to factor out code that's otherwise duplicated in the [\*Page Controller\*](#). To be fair, however, you can also do much of this by using a superclass [\*Page Controller\*](#).

Since there is just one controller, you can easily enhance its behavior at run time by using [decorators](#). You can have decorators for authentication, character encoding, internationalization and add them either using a configuration file or even while the server is running. ([\[Alur, Crupi, and Malks\]](#) describe this approach in detail under the name *Intercepting Filter*.)

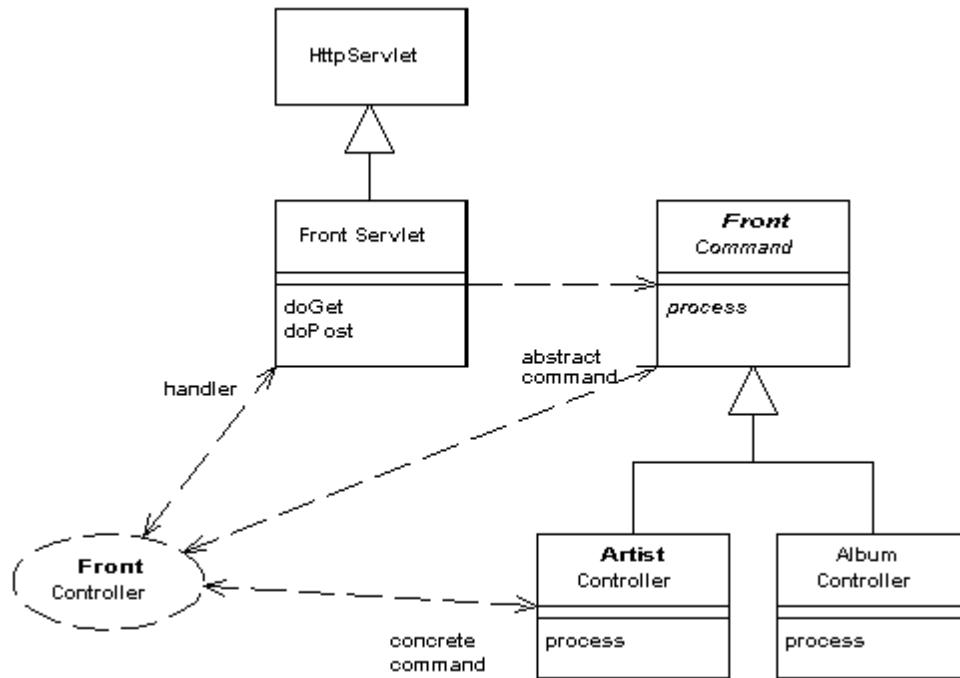
## Further Reading

[\[Alur, Crupi, and Malks\]](#) gives a detailed description of how to implement *Front Controller* in Java. They also describe *Intercepting Filter* which goes very well with *Front Controller*.

A number of Java web frameworks use this pattern. An excellent example of this is [Struts](#)

## Example: Simple Display (Java)

Here's a simple case of using *Front Controller* for the original and innovative task of displaying some information about a recording artist. We'll use dynamic commands with a URL of the form `http://localhost:8080/isa/music?name=astor&command=Artist`. The command parameter tells the web handler which command to use.



*Figure 2: The classes that implement Front Controller*

We'll begin with the handler, which I've implemented as a servlet.

```

class FrontServlet...
public void doGet(HttpServletRequest request, HttpServletResponse response)
throws IOException, ServletException {
FrontCommand command = getCommand(request);
command.init(getServletContext(), request, response);
command.process();
}

private FrontCommand getCommand(HttpServletRequest request) {
try {
return (FrontCommand) getCommandClass(request).newInstance();
} catch (Exception e) {
throw new ApplicationException(e);
}
}

private Class getCommandClass(HttpServletRequest request) {
Class result;
final String commandClassName =
"frontController." + (String) request.getParameter("command") + "Command";
try {
result = Class.forName(commandClassName);
} catch (ClassNotFoundException e) {
result = UnknownCommand.class;
}
return result;
}
  
```

The logic is quite straightforward. The handler takes the command name and tries to instantiate a class named by concatenating the command name and "Command". Once it has created the new command it initializes it with the necessary information from the http server. In this case I've passed in what I need for

this simple example, you may well need more - such as the http session.

If you can't find a command, I've used the [Special Case](#) pattern and returned an unknown command. As is often the case, using [Special Case](#) allows you to avoid a lot of extra error checking.

Commands share a fair bit of data and behavior. All of them need to be initialized with information from the web server.

```
class FrontCommand...
protected ServletContext context;
protected HttpServletRequest request;
protected HttpServletResponse response;

public void init(ServletContext context,
    HttpServletRequest request,
    HttpServletResponse response)
{
    this.context = context;
    this.request = request;
    this.response = response;
}
```

They can also provide common behavior, such as a forward method, and define an abstract process command for the actual commands to override.

```
class FrontCommand...
abstract public void process() throws ServletException, IOException ;

protected void forward(String target) throws ServletException, IOException
{
    RequestDispatcher dispatcher = context.getRequestDispatcher(target);
    dispatcher.forward(request, response);
}
```

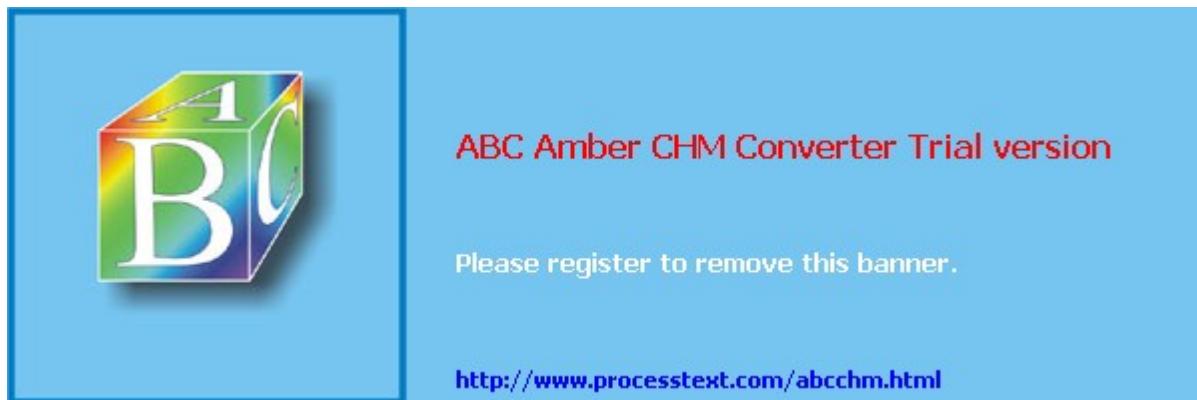
The command object is then very simple, at least in this case. It just implements the process method. This involves invoking the appropriate behavior on the model objects, putting the information needed for the view into the request, and then forwarding to a [Template View](#).

```
class ArtistCommand...
public void process() throws ServletException, IOException {
    Artist artist = Artist.findNamed(request.getParameter("name"));
    request.setAttribute("helper", new ArtistHelper(artist));
    forward("/artist.jsp");
}
```

The unknown command just brings up a boring error page.

```
class UnknownCommand...
public void process() throws ServletException, IOException {
    forward("/unknown.jsp");
}
```

© Copyright [Martin Fowler](#), all rights reserved

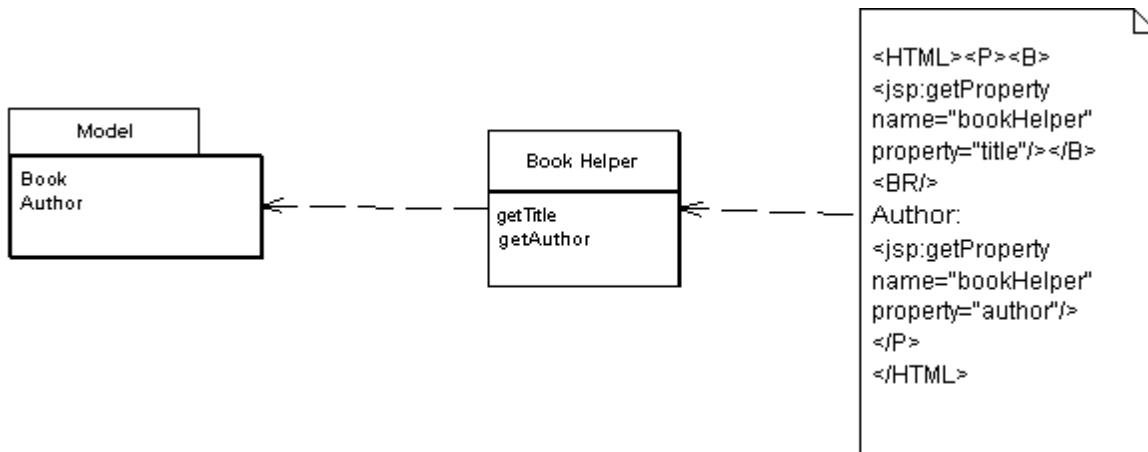


---

# Template View

---

*Render information into HTML by embedding markers into an HTML page*



Writing a program that spits out HTML is often more difficult than you would like. Although programming languages are better at creating text than they used to be (some of us remember character handling in Fortran and standard Pascal), creating and concatenating string constructs is still painful. If there isn't much to do it isn't too bad - but a whole HTML page is a lot of text manipulation

With static HTML pages, those that don't change from request to request, you can use nice editors that produce the HTML pages in a wisewig manner. Even those of us that like raw text editors find it easier to just type in the text and tags rather than fiddle with string concatenation in a programming language.

Of course the issue is with dynamic web pages: those that take the results of something like database queries and embed those into the HTML page. The page looks different with each result, and as a result regular HTML editors aren't enough for the job.

The best way to work is to compose the dynamic web page the same way that you write a static page, but put in markers that can be resolved into calls to gather dynamic information. Since the static part of the page acts as a template for the particular response, I call this a *Template View*.

## How it Works

The basic idea of *Template View* is to embed markers into a static HTML page when it's written. When the page is used to service a request, the makers are replaced by the results of some computation, such

as a database query. This way the page can be laid out in the usual manner, often using wisiwig editors, often by people who aren't programmers. The makers then communicate with real programs to put in the results.

There's a lot of tools that use *Template View*. As a result this pattern isn't about how to build one yourself, more about how to use one effectively and what the alternative is.

## Embedding the Markers

There are a number of ways these markers can be placed into the HTML. One form is to use HTML-like tags. This works well with wisiwig editors because they realize that anything between  $\<\math>$  is special and either ignore it, or treat it differently. If the tags follow the rules for well-formed XML you can also use XML tools on the resulting document (providing your HTML is XHMTL, of course). Another way to do it is to use special text markers in the body text. Wisiwig editors then treat that as regular text, still ignoring it but probably doing annoying things like spell checking it. The advantage is that the syntax can be easier than the clunky HTML/XML syntax.

One of the most popular forms of *Template View* is a **server page**: something like ASP, JSP, or PHP. These actually go a step further than the basic form of a *Template View* in that they allow you to embed arbitrary programming logic, referred to as **scriptlets**, into the page. In my view, however, this feature is actually a big problem and you're better off limiting yourself to basic *Template View* behavior when you use server page technology.

The most obvious disadvantage of putting a lot of scriptlets into a page is that it eliminates the possibility of having non-programmers edit the page. This is particularly important when you're using graphic designers to do the page design. However the biggest problems of embedding scriptlets into the page come from the fact that a page is poor module for a program. Even when using a object-oriented language the page construct loses you most of the structural features that make it possible to do a modular design: either in OO or procedural style.

Even worse, putting a lot of scriptlets into the page makes it too easy to mingle together the different layers of an enterprise application. When domain logic starts turning up on server pages it becomes far too difficult to structure it well, and far too easy to duplicate logic across different server pages. Indeed the worst code I've seen in the last few years has been server page code.

## Helper Object

The key to avoiding is to provide a regular object as a **helper** to each page. This helper has all the real programming logic inside it. The page only has calls into the helper. This simplifies the page and makes it more of a pure *Template View*. The resulting simplicity makes it easier for non-programmers to edit the page and programmers to concentrate on the helper. Depending on the actual tool you are using, you can often reduce all the templates in a page to HTML/XML tags, which keeps the page more consistent and more amenable to tool support.

This sounds like a simple and commendable principle, but as ever there are a quite a few dirty issues involved to make things more complicated. The simplest markers are those that get some information from the rest of the system and put into the correct place on the page. These makers are easily translated

into calls to the helper. The calls result in text, or something that's trivially turned into text, and the engine places the text on the page.

## Conditional Display

A more knotty issue is conditional page behavior. The simplest case is the situation where something is only displayed if some condition is true. The easiest thing to imagine where would be some kind of conditional tag along the lines of <IF condition = "\$pricedrop > 0.1"> ...show some stuff </IF>. The trouble with this is that when you start having conditional tags like this, you start going down the path of turning the templates into a programming language of themselves. This leads you into all the same problems that you get by embedding scriptlets into the page. If you need a full programming language you might as well use scriptlets, but you know what I think of that idea!

As a result I see purely conditional tags as a bad smell, something you should try to avoid. Sometimes you can't avoid it, but you should look to see if you can come up with something more focused than a general purpose <IF> tag.

If you are displaying some text conditionally, one option is to move the condition into the helper. The page then always inserts the result of the call to helper, it's just that if the condition isn't true the helper sends back an empty string. This way the logic is all in the helper. This approach works best if there is no markup for the returned text, or it is enough to return empty markup which gets ignored by the browser.

An example of where this doesn't work is where you might want to highlight good selling items in a list by putting their names in bold. In this situation we always need the names to be displayed, but sometimes we want the special markup. One way to deal with this situation is to have the helper generate the markup. This keeps all the logic out of the page, at the cost of moving the choice of highlighting mechanism away from the page designer and into the programming code.

In order to keep the choice of HTML in the hands of the page design, you need some form of conditional tag. However it's important to look beyond simple <IF> tags. A good route to go is to consider a focused tag. So rather than have a tag that looks like.

```
<IF expression = "isHighSelling()"><B></IF><property name = "price"/><IF expression = "isHighSelling()"></B></IF>
```

you could have a more focused tag such as

```
<highlight condition = "isHighSelling" style = "bold"><property name = "price"/></highlight>
```

In either case it's important that the condition be done based on a single boolean property of the helper. Putting any more complex expression into the page is a case of putting logic into the page itself.

Another example may be something like putting information on a page that depends on the locale that the system is running in. Consider some text that should only be shown in the united states or Canada. Rather than have

```
<IF expression = "locale = 'US' || 'CA'"> ...special text </IF>
```

Look for something like

```
<locale includes = "US, CA"> ...special text </locale>
```

## Iteration

Iterating over a collection presents similar issues. If you want a table where each line corresponds to a line item on an order, you'll need a construct that allows you easily to display information for each line. Here it's hard to avoid a general iterate over a collection tag, and indeed they usually work simply enough to fit in quite well.

Of course the kinds of tag you have to work with are often limited by the environment that you find yourself in. Some environments give you a fixed set of templates to work with. In which case you may be more constrained than you would like in following these kinds of guidelines. In other environments, however, you may have more choice in what kinds of tag to use, many of them allow you to define your own libraries of tags.

## When to Process

The name *Template View* brings out the fact that the primary purpose of *Template View* is to play the view role in [\*Model View Controller\*](#). For many systems the *Template View* should only be the view. In simpler systems it may be reasonable for *Template View* to play the controller role, and possibly even the model role, although I would strive to separate model processing as much as possible. In cases where the *Template View* is taking on responsibilities beyond the view, it's important to ensure that these responsibilities are handled by the helper, not by the page. Controller and model responsibilities involve program logic, and like all program logic this should sit in the helper.

Any template system needs extra processing by the web server. This can either be done by compiling the page after it's created, compiling the page on its first request, or by interpreting the page on each request. Obviously the latter isn't a good idea if the interpretation takes a while to do.

## Using Scripts

Although server pages are one of the most common forms of *Template View* around these days, you can write scripts in a *Template View* style. I've seen a fair bit of perl done this way. The trick, most noticeably demonstrated by perl's CGI.pm is to avoid concatenating strings by having function calls that output the appropriate tags to the response. This way you can write the script in your programming language and avoid the mess of interspersing print strings with programming logic.

## When to Use it

For implementing the view in [\*Model View Controller\*](#) the main choice is between *Template View* and [\*Transform View\*](#). The strength of *Template View* is that it allows you to compose the content of the page by looking at the structure of the page itself. This seems to be easier for most people, both to do and to learn. In particular it nicely supports the idea of a graphic designer laying out a page with a programmer working on the helper.

*Template View* has two significant weaknesses. Firstly the common implementations make it too easy to put complicated logic onto the page, thus making it hard to maintain - particularly by non-programmers. You need good discipline to keep the page simple and display oriented, putting logic in the helper. The second weakness of *Template View* is that it is harder to test than [\*Transform View\*](#). In most implementations *Template View* are designed to work within a web server and are very difficult or impossible to test outside of a web server. [\*Transform View\*](#) implementations are much easier to hook into a testing harness and test without a running web server.

In thinking about a view you also need to consider [\*Two Step View\*](#). Depending on the template scheme you have you may be able to implement [\*Two Step View\*](#) using specialized tags. However you may find it easier to implement [\*Two Step View\*](#) based on a [\*Transform View\*](#). If you are going to need [\*Two Step View\*](#) you need to take that into account when making your choice.

## Example: Using a JSP as a view with a separate controller (Java)

When you're using a JSP as a view only, it will always be invoked from a controller rather than directly from the servlet container. In this case it's important to pass to the JSP any information it will need to figure out what to display. A good way to do this is to have the controller create a helper object and pass that to the JSP using the http request. We'll use the simple display example from [\*Page Controller\*](#). The web handling method for the servlet looks like this

```
class ArtistController...
public void doGet(HttpServletRequest request, HttpServletResponse response)
throws IOException, ServletException {
Artist artist = Artist.findNamed(request.getParameter("name"));
if (artist == null)
forward("/MissingArtistError.jsp", request, response);
else {
request.setAttribute("helper", new ArtistHelper(artist));
forward("/artist.jsp", request, response);
}
}
```

As far as the *Template View* is concerned the important behavior is creating the helper and placing in the request. The sever page can now reach the helper with the useBean tag.

```
<jsp:useBean id="helper" type="actionController.ArtistHelper"
scope="request"/>
```

With the helper in place we can now use it to access the information that we need to display. The model information that the helper needs was passed to it when it was created.

```
class ArtistHelper...
private Artist artist;

public ArtistHelper(Artist artist) {
this.artist = artist;
}
```

We can use the helper to get appropriate information from the model. In the simplest case we can provide a method to get some simple data, such as the artist's name.

```
class ArtistHelper...
public String getName() {
return artist.getName();
}
```

We can then access this information by a Java expression

```
<B> <%=helper.getName()%></B>
```

or a property

```
<B><jsp: getProperty name="helper" property="name" /></B>
```

The choice between properties or expressions depends on who is editing the JSP. For a programmer the expression is easy to read and more compact, but HTML editors may not be able to handle them. Non-programmers will probably prefer the tags, since that fits in the general form of HTML and leaves fewer room for confusing errors.

Using a helper is one way to remove awkward scriptlet code. If you want to show a list of albums for an artist you need to run a loop. You can do this by using a scriptlet in the server page.

```
<UL>
<%
for (Iterator it = helper.getAlbums().iterator(); it.hasNext();) {
Album album = (Album) it.next();%>
<LI><%=album.getTitle()%></LI>

<%}      %>
</UL>
```

But frankly I find this mix of Java and HTML really hard to read. An alternative is to move the for loop to the helper

```
class ArtistHelper...
public String getAlbumList() {
StringBuffer result = new StringBuffer();
result.append("<UL>");
for (Iterator it = getAlbums().iterator(); it.hasNext();) {
Album album = (Album) it.next();
result.append("<LI>");
result.append(album.getTitle());
result.append("</LI>");
}
result.append("</UL>");
return result.toString();
}

public List getAlbums() {
return artist.getAlbums();
}
```

which I find easier to follow since the amount of HTML is quite small. It also allows you to use a property to get this list.

A third alternative available to JSP, of course, is to use a custom tag for iteration.

```
<UL><tag:forEach host = "helper" collection = "albums" id = "each">
<LI><jsp:getProperty name="each" property="title"/></LI>
</tag:forEach></UL>
```

This is a much nicer alternative as it keeps scriptlets out of the JSP and HTML out of the helper.

## Example: ASP.NET Server Page (.NET)

In this example, I'm continuing the example I started in [Page Controller](#). To remind you, this example shows the scores made by batsmen in a single innings of a cricket match. For those who think that cricket is a small noisy animal, I'll pass over the long rhapsodies about the world's most immortal sport and boil it all down to the fact that the page displays three essential pieces of information

- An id number to reference which match it is
- Which team's scores are shown, and which innings the scores are for
- A table showing each batsman's name, his score, and shows his run rate: the number of balls he faced divided by the runs he scored

If you don't understand what these statistics mean, don't worry about it. Cricket is full of statistics, perhaps its greatest contribution to humanity is providing odd statistics for eccentric papers.

The discussion in [Page Controller](#) covers how a web request is handled. To sum up, the object that acts as both the controller and the view is the aspx ASP.NET page. To avoid holding the controller code in a scriptlet, instead you define a separate code behind class.

```
<%@ Page language="c#" Codebehind="bat.aspx.cs" AutoEventWireup="false"
trace="False" Inherits="batsmen.BattingPage" %>
```

The page can access the methods and properties of the code behind class directly. Furthermore the code behind can define a Page\_Load method to handle the request. In this case I've defined the page load method as a template method [\[Gang of Four\]](#) on a [Layer Supertype](#)

```
class CricketPage...
protected void Page_Load(object sender, System.EventArgs e) {
db = new OleDbConnection(DB.ConnectionString);
if (hasMissingParameters())
errorTransfer (missingParameterMessage);
DataSet ds = getData();
if (hasNoData (ds))
errorTransfer ("No data matches your request");
applyDomainLogic (ds);
.DataBind();
prepareUI(ds);
}
```

For the purposes of *Template View* I can ignore all but the last couple of lines of the page load. The call to DataBine allows various page variables to be properly bound to their underlying data sources. That will do for the simpler cases, for more complicated cases the last line calls a method in the particular page's code behind to prepare any objects for use by the page.

The match id number, team, and innings are single values for the page, all of which came into the page as parameters in the http request. I can provide these values by using properties on the code behind class.

```
class BattingPage...
protected String team {
get {return Request.Params["team"]; }
}
protected String match {
get {return Request.Params["match"]; }
}

protected String innings {
get {return Request.Params["innings"]; }
}
protected String ordinalInnings{
get {return (innings == "1") ? "1st" : "2nd"; }
}
```

With the properties defined, I can use them in the text of the page.

```
<P>
Match id:
<asp:label id="matchLabel" Text="<%# match %>" runat="server"
font-bold="True">
</asp:label>&nbsp;
</P>
<P>
<asp:label id=teamLabel Text="<%# team %>" runat="server" font-bold="True">
</asp:label>&nbsp;
<asp:Label id=inningsLabel Text="<%# ordinalInnings %>" runat="server">
</asp:Label>&nbsp;innings</P>
<P>
```

The table is a little more complicated, but actually works out easy in practice because of the graphical design facilities in Visual Studio. Visual Studio provides a data grid control which can be bound to a single table from a data set. I can do this binding in the prepareUI method that's called by the page load method.

```
class BattingPage...
override protected void prepareUI(DataSet ds) {
DataGrid1.DataSource = ds;
DataGrid1.DataBind();
}
```

The batting class is a [Table Module](#) that provides domain logic for the batting table in the database. It's data property is the data from that table, enriched by domain logic from the [Table Module](#). In this case the enrichment is the run rate, which is calculated rather stored in the database.

With the ASP.NET data grid you can select which columns from the table you wish to display in the web page, together with information about the appearance of the table. In this case we can select the name, runs, and rate columns.

```
<asp:DataGrid id="DataGrid1" runat="server" Width="480px" Height="171px"
BorderColor="#336666" BorderStyle="Double" BorderWidth="3px"
BackColor="White" CellPadding="4" GridLines="Horizontal"
AutoGenerateColumns="False">
<SelectedItemStyle Font-Bold="True" ForeColor="White"
BackColor="#339966"></SelectedItemStyle>
<ItemStyle ForeColor="#333333" BackColor="White"></ItemStyle>
<HeaderStyle Font-Bold="True" ForeColor="White"
BackColor="#336666"></HeaderStyle>
<FooterStyle ForeColor="#333333" BackColor="White"></FooterStyle>
<Columns>
<asp:BoundColumn DataField="name" HeaderText="Batsman">
<HeaderStyle Width="70px"></HeaderStyle>
</asp:BoundColumn>
<asp:BoundColumn DataField="runs" HeaderText="Runs">
<HeaderStyle Width="30px"></HeaderStyle>
</asp:BoundColumn>
<asp:BoundColumn DataField="rateString" HeaderText="Rate">
<HeaderStyle Width="30px"></HeaderStyle>
</asp:BoundColumn>
</Columns>
<PagerStyle HorizontalAlign="Center" ForeColor="White" BackColor="#336666"
Mode="NumericPages"></PagerStyle>
</asp:DataGrid></P>
```

The HTML for this data grid looks intimidating, but in Visual Studio you don't manipulate the HTML directly, you manipulate it through property sheets in the development environment - as you do for much of the rest of the page.

This ability to have web form controls on the web page that understands the ADO.NET abstractions of data sets and data tables is the strength, and limitation, of this scheme. It's strength is that when you can transfer information through data sets, all of this can be nicely supported through the kind of tools that Visual Studio has. Its limitation is that it only works seamlessly when you use patterns such as [Table Module](#). If you have very complex domain logic, you'll find that a [Domain Model](#) becomes helpful, and then to take advantage of the tools the [Domain Model](#) needs to create its own data set.





ABC Amber CHM Converter Trial version

Please register to remove this banner.

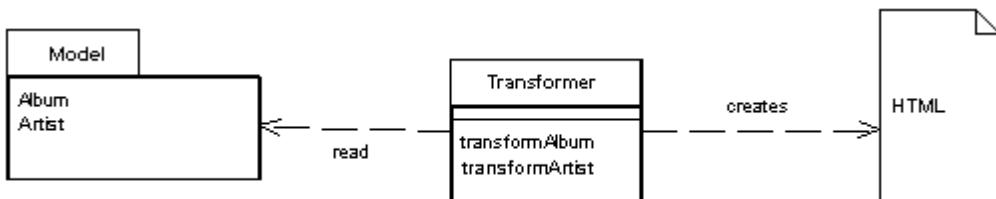
<http://www.processtext.com/abcchm.html>

---

# Transform View

---

*A view that process domain data element by element and transforms it into HTML*



When you issue requests for data to the domain and data source layers, you'll get back all the data you need to satisfy a request, but without the formatting you need to make a proper web page. The role of the view in [Model View Controller](#) is to render this data into a web page. Using *Transform View* means thinking of this as a transformation where you have the model's data as input and the HTML as output.

## How it Works

The basic notion of *Transform View* is to write a program that will look at domain oriented data and convert it to HTML. The program walks the structure of the domain data and as it recognizes each form of domain data it writes out the particular piece of HTML for that data. If you think about this in an imperative way you might have a method called `renderCustomer` that takes a customer object and renders it into HTML. If the customer contains a bunch of orders, the `renderCustomer` method would loop over the orders calling `renderOrder`.

The key difference between *Transform View* and [Template View](#) is the way in which the view is organized. A [Template View](#) is organized around the output. A transform view is organized around separate transforms for each kind of input element. The transform is controlled by something like a simple loop that looks at each input element, finds the appropriate transform for that element, and then calls that transform on the input element. A typical *Transform View*'s elements can be arranged in any order without affecting the resulting output.

You can write a *Transform View* in any language, however at the moment the dominant choice for writing *Transform Views* is XSLT. The interesting thing about this is that XSLT is a functional programming language, similar to Lisp, Scheme, Haskell and other languages that never quite made it into the IS mainstream. As such it has a different kind of structure to it. Rather than explicitly calling routines, the language recognizes elements in the domain data and then invokes the appropriate rendering

transformations.

To carry out an XSLT transform we need to begin with some XML data. The simplest way this can happen is if the natural return type of the domain logic is either XML or something that is automatically transformable to XML. A good example of an automatic transform is a .NET object that can transform itself to XML. Failing that we need to produce the XML ourselves. A good way to do this is to populate a [\*Data Transfer Object\*](#) that can serialize itself into XML. That way the data can be assembled using a convenient api. In simpler cases a [\*Transaction Script\*](#) can return XML directly.

The XML that's fed into the transform does not need to be a string, unless a string form is needed to cross a communication line. It's usually quicker and easier to produce a DOM and hand that to the transform.

Once we have the XML we pass it to XSLT engine, increasingly these are available commercially. The logic for the transform is captured in an XSLT stylesheet which we also pass to the transformer. The transformer then applies the stylesheet to the input XML to yield the output HTML, which we can write directly to the HTTP response.

## When to Use it

The choice between a *Transform View* and a [\*Template View\*](#) mostly comes down to which environment is preferable for the team working on the view software to use. The presence of tools is a key factor here. Increasingly there are HTML editors which you can use to write [\*Template Views\*](#). Tools for XSLT are, at least so far, much less sophisticated. XSLT can also be an awkward language to master, due its functional programming style coupled with its awkward XML syntax.

One of the strengths of XSLT is its portability. It can be used with almost any web platform. You can use the same XSLT to transform XML created from J2EE or .NET. This can help putting a common HTML view onto data from different sources.

XSLT is also often easier if you are building a view on an XML document. Other environments usually require you to transform the XML document into an object, or indulge in walking the XML DOM, which is often complicated. XSLT fits naturally in an XML world.

*Transform View* avoids two of the biggest problems with [\*Template View\*](#). It's easier to keep the transform focused only on rendering HTML, this avoids having too much other logic in the view. It's also easy to run the *Transform View* and capture the output for testing. This makes it easier to test the view and you don't need a web server to run the tests.

*Transform View* transforms directly from domain-oriented XML into HTML. If you need to change the overall appearance of a web site, this can lead to you having to change multiple transform programs. Using common transforms, such as you can do with XSLT includes, helps reduce this problem. Indeed it's much easier to call common transformations using *Transform View* than it is using [\*Template View\*](#). If you need to make global changes easily, or support multiple appearances for the same data, you might consider [\*Two Step View\*](#) which uses a two stage process.

## Example: Simple Transform (Java)

Setting up a simple transform involves preparing Java code to invoke the right stylesheet to form the response, and preparing the stylesheet to format the response. In these cases most of the response to a page is pretty generic, so it makes sense to use [\*Front Controller\*](#). I'll only describe the command here, you should look at [\*Front Controller\*](#) to see how the command object fits in with the rest of the request response handling.

All the command object does is invoke the methods on the model to obtain an XML input document, then pass that XML document through the XML processor.

```
class AlbumCommand...
public void process() {
try {
Album album = Album.findNamed(request.getParameter("name"));
Assert.notNull(album);
PrintWriter out = response.getWriter();
XsltProcessor processor = new SingleStepXsltProcessor("album.xsl");
out.print(processor.getTransformation(album.toXmlDocument()));
} catch (Exception e) {
throw new ApplicationException(e);
}
}
```

The XML document may look something like this

```
<album>
<title>Zero Hour</title>
<artist>Astor Piazzola</artist>
<trackList>
<track><title>Tanguedia III</title><time>4:39</time></track>
<track><title>Milonga del Angel</title><time>6:30</time></track>
<track><title>Concierto Para Quinteto</title><time>9:00</time></track>
<track><title>Milonga Loca</title><time>3:05</time></track>
<track><title>Michelangelo '70</title><time>2:50</time></track>
<track><title>Contrabajisimo</title><time>10:18</time></track>
<track><title>Mumuki</title><time>9:32</time></track>
</trackList>
</album>
```

The translation of the XML document is done by an XSLT program. Each template match matches a particular part of the XML and produces the appropriate HTML output for the page. In this case I've kept the formatting to a excessively simple level to just show the essentials. The following template clauses match the basic elements of the XML file.

```
<xsl:template match="album">
<HTML><BODY bgcolor="white">
<xsl:apply-templates/>
</BODY></HTML>
</xsl:template>
<xsl:template match="album/title">
<h1><xsl:apply-templates/></h1>
</xsl:template>
```

```
<xsl:template match="artist">
<P><B>Artist: </B><xsl:apply-templates/></P>
</xsl:template>
```

These template matches handle the table. The table here has alternating rows highlighted in different colors. This is a good example of something that isn't possible with Cascading Style Sheets but is reasonable to do with XML.

```
<xsl:template match="trackList">

```



---

© Copyright [Martin Fowler](#), all rights reserved

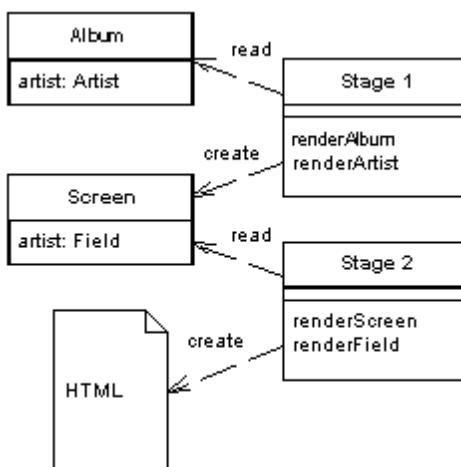
A blue rectangular banner with rounded corners. On the left side, there is a 3D-style cube with the letters 'A', 'B', and 'C' on its faces, colored in a rainbow gradient. To the right of the cube, the text "ABC Amber CHM Converter Trial version" is displayed in red. Below this, in smaller white text, is the instruction "Please register to remove this banner.". At the bottom right of the banner, the URL "http://www.processtext.com/abcchm.html" is shown in blue.

---

# Two Step View

---

*Turn domain data into HTML in two steps: first by forming some kind of logical page, then rendering the logical page into HTML.*



If you have a web application with many pages, you often want to have a consistent look and organization to the site. If every page looks different, you end up with a different look and feel that users find confusing. You may also want to make global changes to the appearance site easily.

Common web site approaches using [Template View](#) or [Transform View](#) make this difficult because presentation decisions are often duplicated across multiple pages or transform modules. A global change can force you to change several files.

*Two Step View* deals with this problem by splitting the transformation into two stages. The first transforms the model data into a logical presentation without any specific formatting, and the second stage converts that logical presentation with the actual formatting needed. This way you can make a global change by altering the second stage, or support multiple output look and feels with one second stage each.

## How it Works

The key to this pattern is to make the transformation to the HTML a two stage process. The first stage assembles the information that should be displayed in a logical screen structure that is suggestive of the display elements, yet does not contain any HTML. The second stage takes that presentation-oriented structure and renders it into HTML.

This intermediate form is a kind of logical screen. Its elements might include things like fields, headers, footers, tables, choices, and the like. As such it's certainly presentation-oriented and certainly constrains the screens to follow a definite style. You can think of the presentation-oriented model as one that defines the various kinds of widgets that you can have and the data the widgets contain, but does not specify what their appearance is in HTML.

This presentation-oriented structure is assembled by specific code written for each screen. The first stage's responsibility is to access a domain-oriented model, either a database, a domain model, or a domain-oriented [Data Transfer Object](#), extract the relevant information for that screen, and then put that information into the presentation-oriented structure.

The second stage is then turns this presentation-oriented structure into HTML. It knows about each element in the presentation-oriented structure and how to show that element as HTML. As such a system with many screens can be rendered to HTML by a single second stage - so all the HTML formatting decisions are made in one place. Of course the constraint is that the resulting screen must be derivable from the presentation-oriented structure.

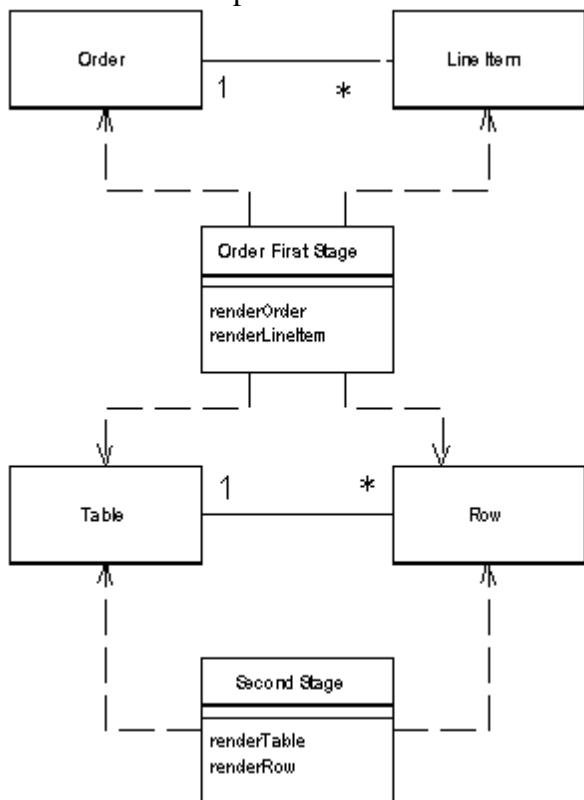


Figure 1: Sample classes for two step rendering.

There are several ways in which you can build a *Two Step View*. Perhaps the easiest to think about is using two step XSLT. A single step XSLT follows the approach in [Transform View](#). Each page has a single XSLT stylesheet that transforms the domain oriented XML into HTML. The two step approach uses two XSLT stylesheets. The first stage transforms the domain-oriented XML into a presentation-oriented XML. The second stage stylesheet then renders that presentation-oriented XML into HTML.

Another way is to use classes. Here you define the presentation-oriented structure as a set of classes, with a table class, a row class, etc. The first stage takes domain information and instantiates these

presentation-oriented classes into a structure that models a logical screen. Then you render these classes into HTML, either by getting each presentation-oriented class to generate HTML for itself, or by having a separate HTML renderer class to do the job.

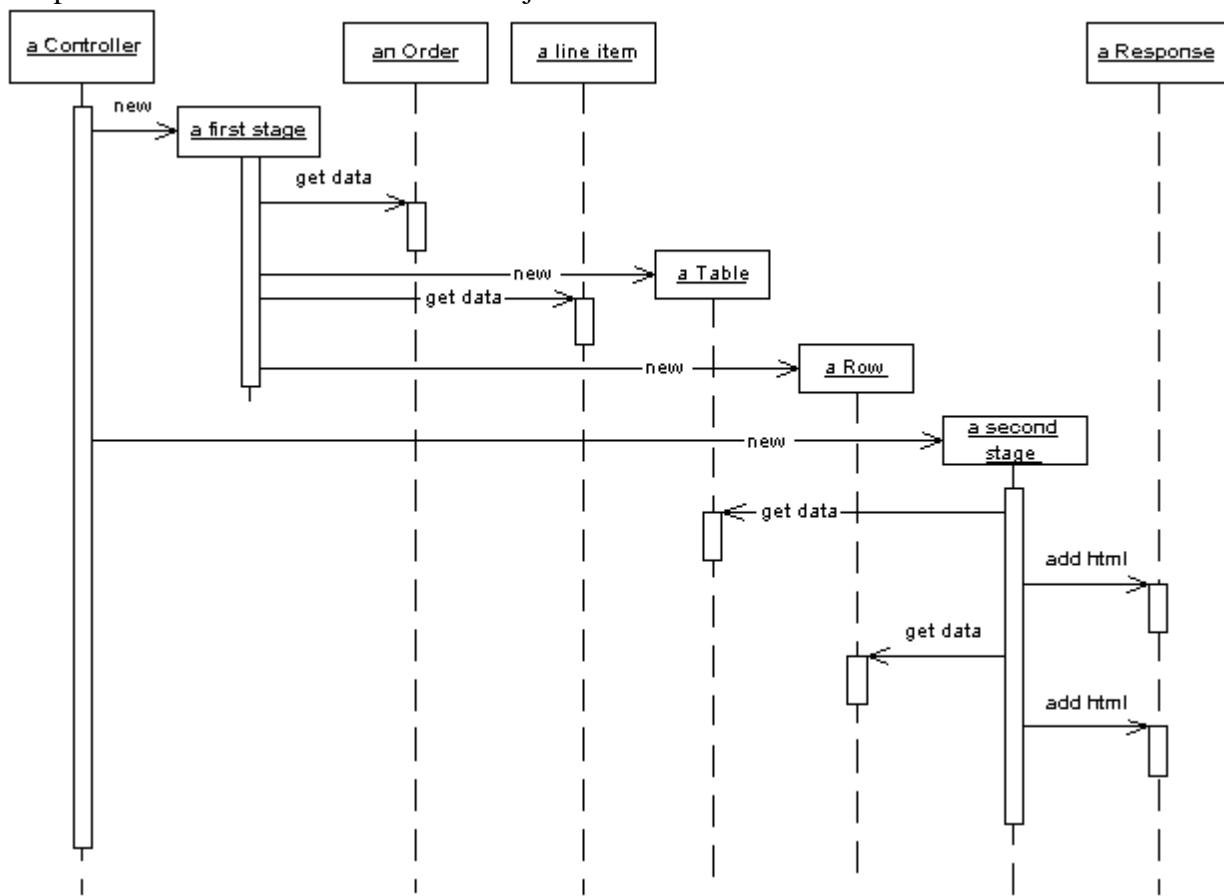


Figure 2: Sequence diagram for two step rendering

Both of the above approaches are based on [Transform View](#). You can also do a [Template View](#) based approach. To do this you need to pick templates that are based around the idea of a logical screen. For example you might have a template that looks like

```
<field label = "Name" value = "getName" />
```

The template system then converts these logical tags to HTML. In such a scheme the page definition wouldn't include any HTML, it would include these logical screen tags - as a result it would probably be an XML document. Of course this has the disadvantage that you lose the ability to use wisiwig HTML editors.

## When to Use it

The key value in *Two Step View* comes from the separation of first and second stages, allowing you to make global changes more easily. It helps to think of two situations: multi-appearance web application and single-appearance web application. Multi-appearance web apps are the rarer breed, but one that's growing. A multi-appearance web application is one where the same basic functionality is provided through multiple organizations, and each organization wants its own distinct look to the application. A

current example of this is airline travel sites where as you look at them you can tell from the page layout and design that they are all variations on one base travel site. I suspect many airlines would want that same functionality but with a distinctly different and individual appearance.

The single-appearance case is the more common one, where there's only one organization fronting the web app and they would like a consistent look throughout the site. That's the easiest case to consider first.

With a single stage view (either a [Template View](#) or a [Transform View](#)) you build one view module per page of the web application.

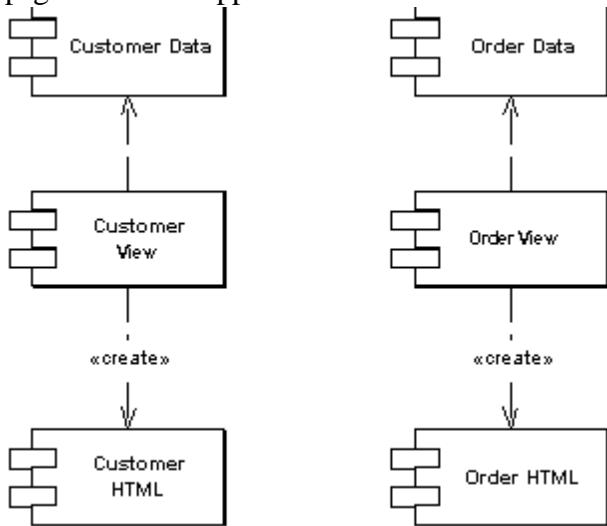


Figure 3: Single stage view with one appearance

With a *Two Step View* you have two stages to the view. You have one first stage module per page in the web application and only second stage module for the entire application. Your pay-off in using *Two Step View* is that any change you can make to the appearance of the site in the second stage is much easier to make, since one change in the second stage affects the entire site.

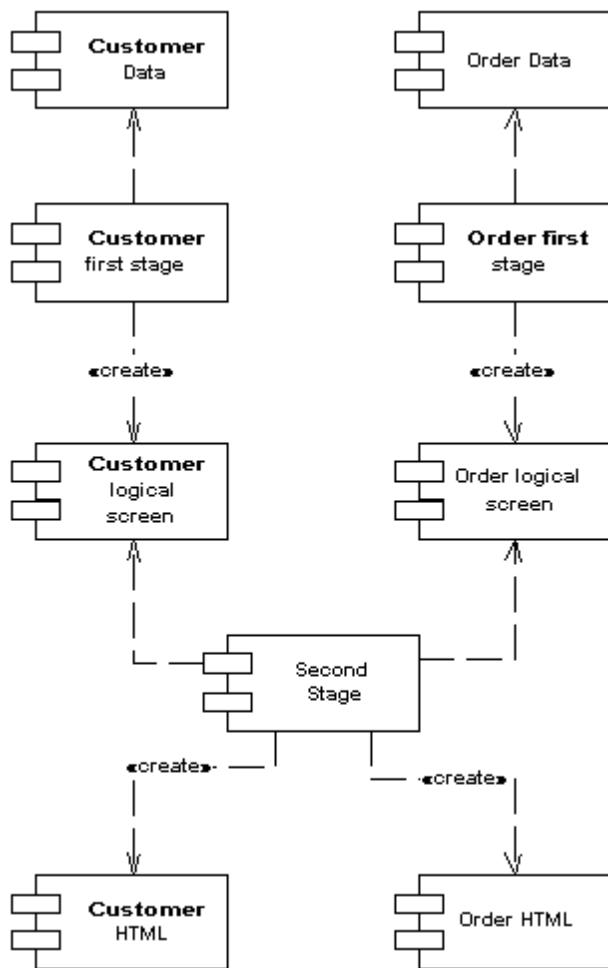


Figure 4: Two stage view with one appearance

With a multi-appearance app this advantage is compounded. You a single-stage view for each combination of screen and appearance. So ten screens and three appearances require 30 single stage view modules. Using *Two Step View*, however you can get away with ten first stages and three second-stages. The more screens and appearances you have, the bigger the saving.

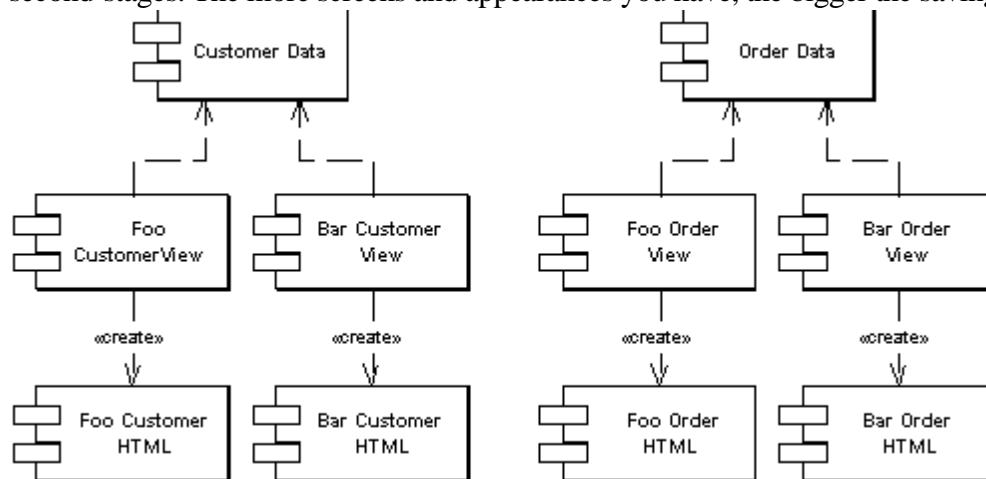


Figure 5: Single stage view with two appearances

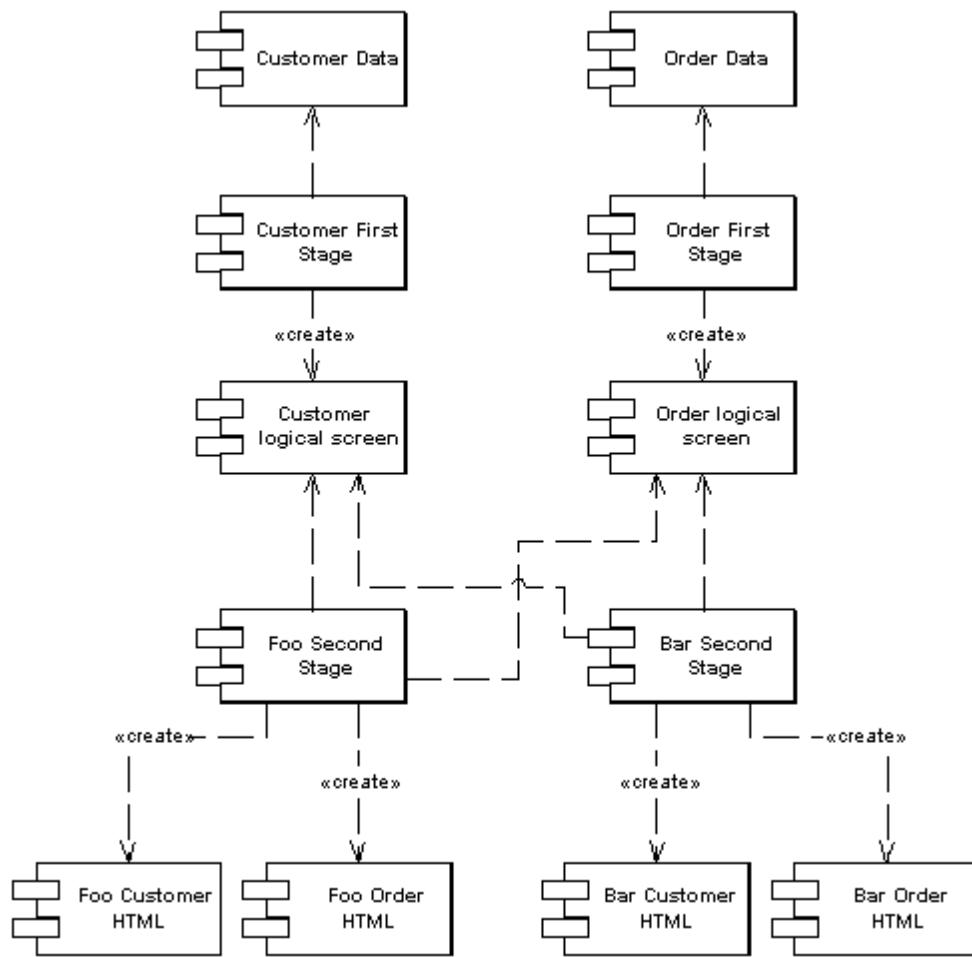


Figure 6: Two stage view with two appearances

However your ability to pull this off is entirely dependent on how well you can make the presentation-oriented structure to really serve the needs of the appearance. A design heavy site, where each page is supposed to look different, won't work well with *Two Step View* because it's to find enough commonality between the screens to get a simple enough presentation-oriented structure. Essentially the design of the site is constrained by the presentation-oriented structure - for many sites that is too much of a limitation.

Another limitation of *Two Step View* is the tools to use it. Tools are widely available for designers with no programming skills to lay out HTML pages using [Template View](#). *Two Step View* forces programmers to write the renderer and controller objects. Thus programmers have to be involved in any design change.

It's also true that *Two Step View*, with its multiple layers, presents a harder programming model to learn, although once you are used to it it's no more difficult - and may help to reduce repetitive boilerplate code.

## Example: Two Stage XSLT (XSLT)

This approach to a *Two Step View* uses a two stage XSLT transformation. The first stage transforms

the domain specific XML into a logical screen XML, the second stage transforms the logical screen XML into HTML.

The initial domain oriented XML looks like this:

```
<album>
<title>Zero Hour</title>
<artist>Astor Piazzola</artist>
<trackList>
<track><title>Tanguedia III</title><time>4:39</time></track>
<track><title>Milonga del Angel</title><time>6:30</time></track>
<track><title>Concierto Para Quinteto</title><time>9:00</time></track>
<track><title>Milonga Loca</title><time>3:05</time></track>
<track><title>Michelangelo '70</title><time>2:50</time></track>
<track><title>Contrabajisimo</title><time>10:18</time></track>
<track><title>Mumuki</title><time>9:32</time></track>
</trackList>
</album>
```

The first stage XSLT processor transforms this into the following screen oriented XML.

```
<screen>
<title>Zero Hour</title>
<field label="Artist">Astor Piazzola</field>
<table>
<row><cell>Tanguedia III</cell><cell>4:39</cell></row>
<row><cell>Milonga del Angel</cell><cell>6:30</cell></row>
<row><cell>Concierto Para Quinteto</cell><cell>9:00</cell></row>
<row><cell>Milonga Loca</cell><cell>3:05</cell></row>
<row><cell>Michelangelo '70</cell><cell>2:50</cell></row>
<row><cell>Contrabajisimo</cell><cell>10:18</cell></row>
<row><cell>Mumuki</cell><cell>9:32</cell></row>
</table>
</screen>
```

To do this we need the following XSLT program.

```
<xsl:template match="album">
<screen><xsl:apply-templates/></screen>
</xsl:template>
<xsl:template match="album/title">
<title><xsl:apply-templates/></title>
</xsl:template>
<xsl:template match="artist">
<field label="Artist"><xsl:apply-templates/></field>
</xsl:template>
<xsl:template match="trackList">
<table><xsl:apply-templates/></table>
</xsl:template>
<xsl:template match="track">
<row><xsl:apply-templates/></row>
</xsl:template>
<xsl:template match="track/title">
<cell><xsl:apply-templates/></cell>
</xsl:template>
<xsl:template match="track/time">
```

```
<cell><xsl:apply-templates/></cell>
</xsl:template>
```

The screen oriented XML is very plain. To turn it into HTML we use a second stage XSLT program.

```
<xsl:template match="screen">
<HTML><BODY bgcolor="white">
<xsl:apply-templates/>
</BODY></HTML>
</xsl:template>
<xsl:template match="title">
<h1><xsl:apply-templates/></h1>
</xsl:template><xsl:template match="field">
<P><B><xsl:value-of select = "@label"/>: </B><xsl:apply-templates/></P>
</xsl:template>
<xsl:template match="table">
<table><xsl:apply-templates/></table>
</xsl:template>
<xsl:template match="table/row">
<xsl:variable name="bgcolor">
<xsl:choose>
<xsl:when test="(position() mod 2) = 1">linen</xsl:when>
<xsl:otherwise>white</xsl:otherwise>
</xsl:choose>
</xsl:variable>
<tr bgcolor="{{$bgcolor}}><xsl:apply-templates/></tr>
</xsl:template>
<xsl:template match="table/row/cell">
<td><xsl:apply-templates/></td>
</xsl:template>
```

To assemble the two stages, I've used [Front Controller](#) to help separate the code that does the work.

```
class AlbumCommand...
public void process() {
try {
Album album = Album.findNamed(request.getParameter("name"));
album = Album.findNamed("1234");
Assert.notNull(album);
PrintWriter out = response.getWriter();
XsltProcessor processor = new TwoStepXsltProcessor("album2.xsl",
"second.xsl");
out.print(processor.getTransformation(album.toXmlDocument()));
} catch (Exception e) {
throw new ApplicationException(e);
}
}
```

It's useful to compare this to the single stage approach in [Transform View](#). Consider changing the colors of the alternating rows. Using [Transform View](#) it would require editing every XSLT program. With *Two Step View* only the single second stage XSLT program needs to be changed. While it might be possible to use callable templates to do something similar, this will need a fair bit of XSLT gymnastics to pull off. The down side of *Two Step View* is that the final HTML is very much constrained by the screen oriented XML.

# Example: JSP and Custom Tags (Java)

Although the XSLT route is conceptually the easiest way to think about implementing *Two Step View*, there are plenty of other ways to do it. For this example I'll use JSPs and custom tags, although they are both more awkward and less powerful than XSLT they do show how the pattern can manifest itself in different ways.

The key rule of *Two Step View* is that the choosing of what to display and the HTML that displays it are utterly separated. For this example my first stage is handled by a JSP page and its helper, while a set of custom tags deals with the second stage.

The interesting part of the first stage is the JSP page

```
<%@ taglib uri="2step.tld" prefix = "2step" %>
<%@ page session="false"%>

<jsp:useBean id="helper" class="actionController.AlbumConHelper"/>
<%helper.init(request, response);%>

<2step:screen>
<2step:title><jsp:getProperty name = "helper" property =
"title"/></2step:title>
<2step:field label = "Artist"><jsp:getProperty name = "helper" property =
"artist"/></2step:field>
<2step:table host = "helper" collection = "trackList" columns = "title,
time"/>
</2step:screen>
```

I'm using [Page Controller](#) for the JSP page with a helper object, that much you can flick over to [Page Controller](#) to read. For this discussion the important thing to do is look at the tags that are part of the "2step" name space. These are the tags that I'm using to invoke the second stage. Notice that there is no HTML on the JSP page, the only tags that are present are either the second stage tags, or bean manipulation tags to get values out of the helper.

Each of the second stage tags has an implementation to pump out the necessary HTML for that logical screen element. The simplest of these is the title.

```
class TitleTag...
public int doStartTag() throws JspException {
try {
pageContext.getOut().print("<H1>");
} catch (IOException e) {
throw new JspException("unable to print start");
}
return EVAL_BODY_INCLUDE;
}

public int doEndTag() throws JspException {
try {
pageContext.getOut().print("</H1>");
} catch (IOException e) {
throw new JspException("unable to print end");
}
```

```
return EVAL_PAGE;
}
```

For those that haven't indulged, a custom tag works by implementing hook methods that are called at the beginning and the end of the tagged text. So this tag simply wraps its body content with an <H1> tag.

A more complex tag, such as the field, can take an attribute. The attribute is tied into the tag class using a setting method

```
class FieldTag...
private String label;

public void setLabel(String label) {
this.label = label;
}
```

With the value set, you can then use it in the output.

```
class FieldTag...
public int doStartTag() throws JspException {
try {
pageContext.getOut().print("<P>" + label + ": <B>");
} catch (IOException e) {
throw new JspException("unable to print start");
}
return EVAL_BODY_INCLUDE;
}

public int doEndTag() throws JspException {
try {
pageContext.getOut().print("</B></P>");
} catch (IOException e) {
throw new JspException("how are checked exceptions helping me here?");
}
return EVAL_PAGE;
}
```

The table tag is the most sophisticated of the tags. As well as allowing the JSP writer to choose which columns to put in the table, it also does highlighting of alternate rows. Similarly to my previous example, the highlighting is done by the second stage so that a system wide change can be done globally.

The table tag takes attributes for the name of the collection property, the object on which the collection property sits, and a comma separated list of column names.

```
class TableTag...
private String collectionName;
private String hostName;
private String columns;

public void setCollection(String collectionName) {
this.collectionName = collectionName;
}

public void setHost(String hostName) {
this.hostName = hostName;
}
```

```
public void setColumns(String columns) {  
this.columns = columns;  
}
```

I made a helper method to get a property out of an object. There's a good argument for using the various classes that support Java beans, rather than just invoking a getSomething method, but this will do for the example.

```
class TableTag...  
private Object getProperty(Object obj, String property) throws JspException {  
try {  
String methodName = "get" + property.substring(0, 1).toUpperCase() +  
property.substring(1);  
Object result = obj.getClass().getMethod(methodName, null).invoke(obj, null);  
return result;  
} catch (Exception e) {  
throw new JspException("Unable to get property " + property + " from " + obj);  
}  
}
```

This tag doesn't have a body. When it's called it pulls the named collection out of the request property and iterates through this collection to generate the rows of the table.

```
class TableTag...  
public int doStartTag() throws JspException {  
try {  
JspWriter out = pageContext.getOut();  
out.print("<table>");  
Collection coll = (Collection) getPropertyFromAttribute(hostName,  
collectionName);  
Iterator rows = coll.iterator();  
int rowNumber = 0;  
while (rows.hasNext()) {  
out.print("<tr>");  
if ((rowNumber++ % 2) == 0) out.print(" bgcolor = " + HIGHLIGHT_COLOR);  
out.print(">");  
printCells(rows.next());  
out.print("</tr>");  
}  
out.print("</table>");  
} catch (IOException e) {  
throw new JspException("unable to print out");  
}  
return SKIP_BODY;  
}  
  
private Object getPropertyFromAttribute(String attribute, String property)  
throws JspException  
{  
Object hostObject = pageContext.findAttribute(attribute);  
if (hostObject == null)  
throw new JspException("Attribute " + attribute + " not found.");  
return getProperty(hostObject, property);  
}  
  
public static final String HIGHLIGHT_COLOR = "'linen'" ;
```

During the iteration it sets every other row to the linen background to highlight them.

To print the cells for each row, I use the column names as property values on the objects in the collection.

```
class TableTag...
private void printCells(Object obj) throws IOException, JspException {
JspWriter out = pageContext.getOut();
for (int i = 0; i < getColumnList().length; i++) {
out.print("<td>");
out.print(getProperty(obj, getColumnList()[i]));
out.print("</td>");
}
}

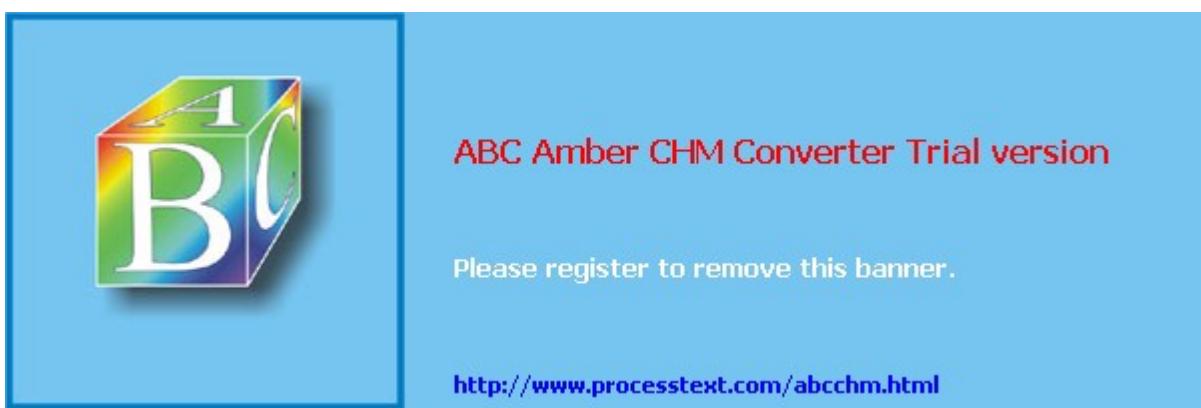
private String[] getColumnList() {
StringTokenizer tk = new StringTokenizer(columns, " ", " ");
String[] result = new String[tk.countTokens()];
for (int i = 0; tk.hasMoreTokens(); i++)
result[i] = tk.nextToken();
return result;
}
```

Compared to the XSLT implementation, one of the biggest differences is that this solution is rather less constraining on the uniformity of the site's layout. If an author of one page should want to slip some individual HTML into the page it's easier to do that. Of course while this allows tweaking of design intensive pages, it also is open to inappropriate use by people who are unfamiliar of the how the thing is to work. Sometimes constraints help prevent mistakes, that's a trade-off a team has to decide for themselves.



---

© Copyright [Martin Fowler](#), all rights reserved

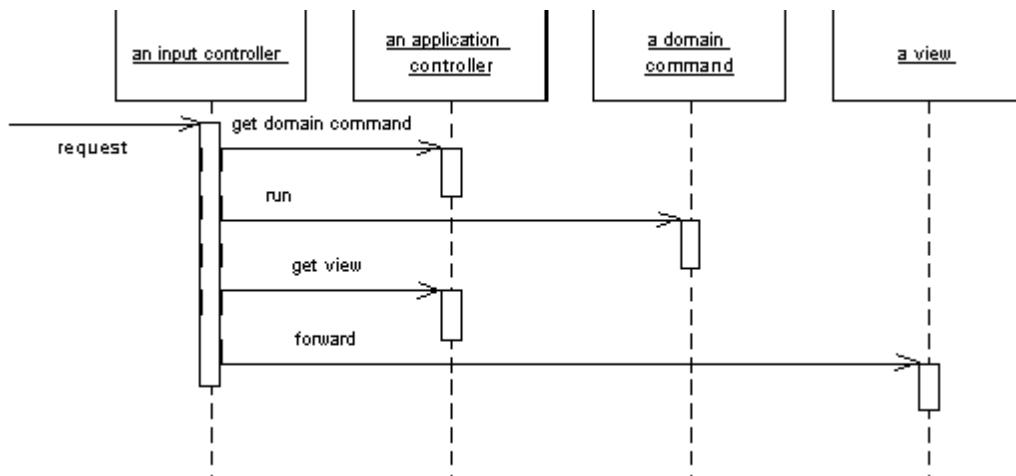


---

# Application Controller

---

*A centralized point for handling screen navigation and flow of an application*



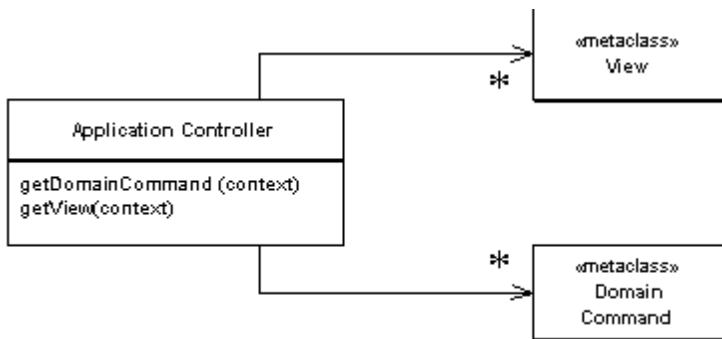
Some applications contain a significant amount of logic about what kind of screens to use at different points. This may involve invoking certain screens at certain points in an application, such as the wizard style of interaction where the user is led through a series of screens that need to be done in a certain order. Other cases we may see screens that are only brought in under certain conditions, or choices between different screens that depend on earlier input.

To some degree the various [\*Model View Controller\*](#) input controllers can make some of these decisions, but as the application gets more complex this can lead to duplicated code, as several controller for different screens need to know what to do in a certain situation.

You can remove this duplication by placing all the flow logic into an *Application Controller*. Input controllers then ask the *Application Controller* for the appropriate commands for execution against a model and the correct view to use depending on the context within the application.

## How it Works

An *Application Controller* has two main responsibilities: deciding which domain logic to run, and deciding which view to the display the response with. To this the *Application Controller* typically holds two structured collections of class references, one for domain commands to execute against in the domain layer and one of views



*Figure 1: An application controller has two collections of references to classes, one for domain logic and one for view.*

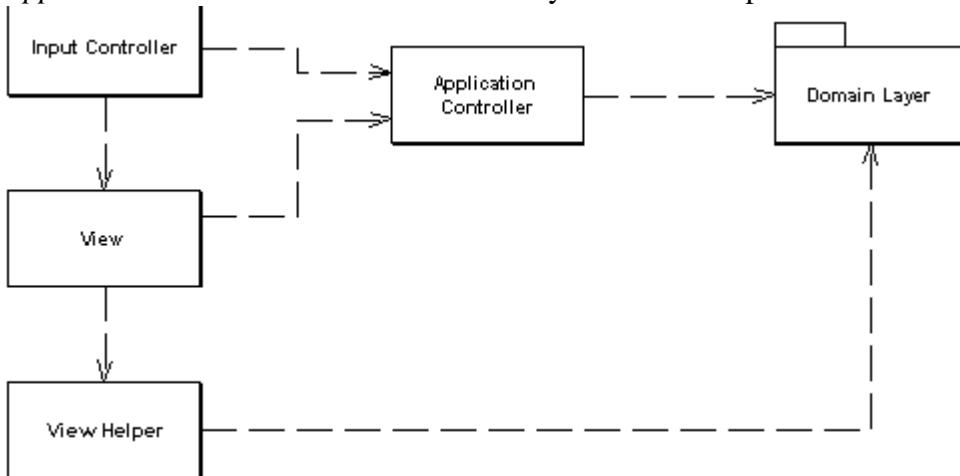
For both the domain commands and the view, the application controller needs some way of storing something that it can invoke. A [command](#) is a good choice, since it allows us to get hold of and run a block of code easily. Languages that can manipulate functions can hold a reference to a function. Another option is to hold a string that can be used to invoke a method by reflection.

The domain commands can be command objects that are part of the *Application Controller* layer, or they can be references to a [Transaction Script](#) or domain object method in the domain layer.

If you are using server pages as your views, then you can use the name of a server page. If you are using a class then a command or a string for a reflective call makes sense. You might also use an XSLT transform, in which case the *Application Controller* can hold a string to reference the transform.

One decision you'll need to make is how much to separate the *Application Controller* from the rest of the presentation. At the first level this manifests itself in whether the *Application Controller* has dependencies into the UI machinery. This might take the form of the *Application Controller* directly accessing the http session data, forwarding to a server page, or invoking methods on a rich client class.

Although I've seen direct *Application Controllers*, my preference is for the *Application Controller* to have no links to the UI machinery. For a start this makes it possible to test the *Application Controller* independently of the UI, which is a major benefit. It's also important to do this if you're going to use the same *Application Controller* with multiple presentations. As a result many people like to think of the *Application Controller* as an intermediate layer between the presentation and the domain.



*Figure 2: My preferred dependencies when using an application controller*

An application can have multiple *Application Controllers* to handle different parts of an application. This allows you to split complex logic up into several classes. In this case I usually see the work divided up into broad areas of the user interface and build separate *Application Controllers* for each area. On a simpler application I might only need a single *Application Controller*.

If you have multiple presentations - such as a web front end, a rich client, and a PDA - you may be able to use the same *Application Controller* for each presentation. Don't be too eager to do this. Often different UIs really need a different screen flow to give a really usable user interface. However reusing a single *Application Controller* may reduce the development effort, and the lower effort may be worth the cost of a more awkward user interface.

A common way of thinking about a UI is a state machine, where certain events trigger different responses depending on the state of certain key objects in the application. In this case the *Application Controller* is particularly amenable to using metadata to represent the state machine's control flow. The metadata can either be set up by programming language calls (the simplest way) or you can store the metadata in a separate configuration file.

You may find domain logic that's particular to one request placed in an *Application Controller*. As you might suspect I'm come down pretty hard against that notion. However the boundary between domain and application logic does get very murky. Say I'm handling insurance applications and I need to show a separate screen of questions only if the application is a smoker. Is this application logic or domain logic? If I have only a few such cases I'd probably put that kind of logic in the *Application Controller*, but if it occurs in lots of places I'd look to design the *Domain Model* in such a way to drive this.

## When to Use it

If the flow and navigation of your application is pretty simple, anyone can visit any screen in pretty much any order, then there's little value in a *Application Controller*. The strength of an *Application Controller* comes from definite rules about the order in which pages should be visited and different views depending on the state of objects.

A good signal to use an *Application Controller* is if you find yourself having to make similar changes in lots of different places when your application's flow changes.

## Further Reading

Most of the ideas that underlie the writing of this pattern came from reading [\[Knight and Dai\]](#). Although these ideas aren't exactly new, I found the explanation in [\[Knight and Dai\]](#) remarkably clear and compelling.

## Example: State Model *Application Controller* (Java)

State model's are a common way of thinking about user interfaces. They are particularly appropriate when you need to react differently to events depending on the state of some object. In this example I have a simple state model for a couple of commands on an asset. Our leasing experts would faint at the virulent oversimplification of this model, but it'll do to show an example of state based *Application Controller*.

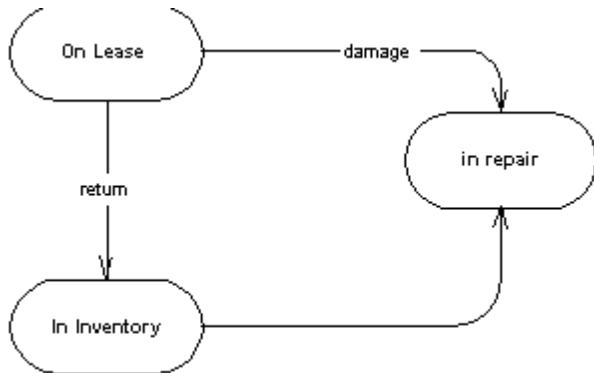


Figure 3: A simple state diagram for an asset

As far as the code is concerned our rules are this:

- When we receive a return command, and we're in the on lease state then we display a page to capture information about the return of the asset
- A return even in the in inventory state is an error, so we show an illegal action page
- When we receive a damage command we show different pages depending on whether the asset is in inventory or on lease.

The input controller is a *Front Controller*. It services the request like this.

```

class FrontServlet...
public void service(HttpServletRequest request, HttpServletResponse response)
throws IOException, ServletException
{
    ApplicationController appController = getApplicationController(request);
    String commandString = (String) request.getParameter("command");
    DomainCommand comm =
    appController.getDomainCommand(commandString, getParameterMap(request));
    comm.run(getParameterMap(request));
    String viewPage =
    "/" + appController.getView(commandString, getParameterMap(request)) + ".jsp";
    forward(viewPage, request, response);
}
  
```

The flow of the service method is pretty straightforward, we find the right application controller for a given request, we then ask the application controller for the domain command, execute the domain command, ask the application controller for a view, and finally forward to the view.

In this scheme I'm assuming a number of *Application Controllers*, all of which implement the same interface.

```

interface ApplicationController...
DomainCommand getDomainCommand (String commandString, Map params);
String getView (String commandString, Map params);
  
```

For our commands, the appropriate *Application Controller* is an asset application controller. The asset application controller uses an response class to hold the domain commands and view references. For the domain command I use a reference to a class, for the view I use a string which the front controller will turn into a URL for a JSP.

```
class Response...
private Class domainCommand;
private String viewUrl;
public Response(Class domainCommand, String viewUrl) {
this.domainCommand = domainCommand;
this.viewUrl = viewUrl;
}
public DomainCommand getDomainCommand() {
try {
return (DomainCommand) domainCommand.newInstance();
} catch (Exception e) {throw new ApplicationException (e);
}
}
public String getViewUrl() {
return viewUrl;
}
```

The application controller holds onto the responses using a map of maps indexed by the command string and an asset status.

```
class Asset ApplicationController...
private Response getResponse(String commandString, AssetStatus state) {
return (Response) getResponseMap(commandString).get(state);
}
private Map getResponseMap (String key) {
return (Map) events.get(key);
}
private Map events = new HashMap();
```

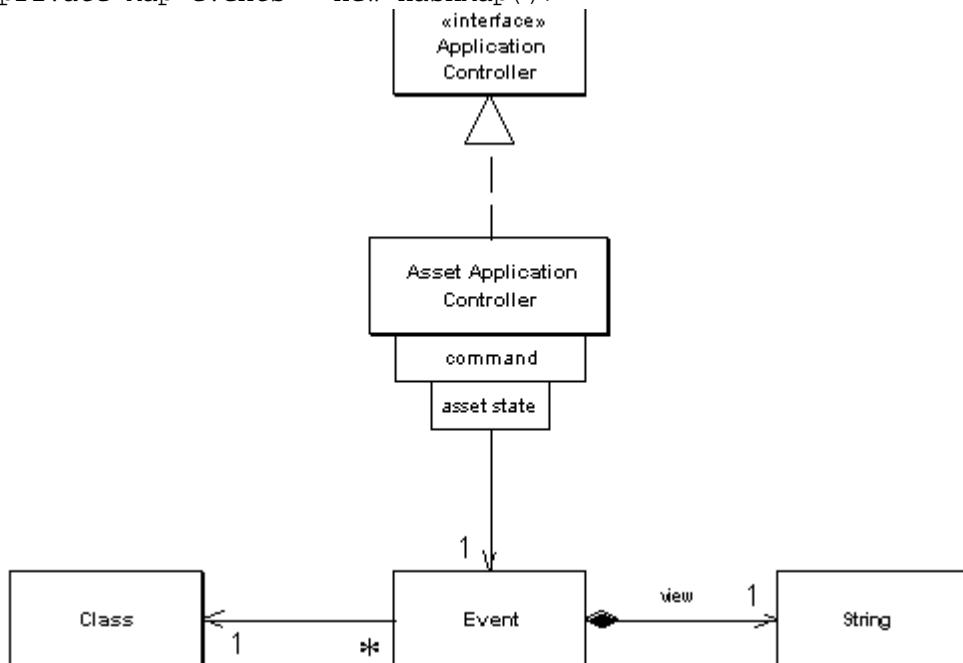


Figure 4: How the asset application controller stores its references to domain commands and

views

When asked for a domain command, the controller looks at the request to figure out the asset id, goes to the domain to find out the status of that asset, looks up the appropriate domain command class, instantiates it and returns the new object.

```
class AssetApplicationController...
public DomainCommand getDomainCommand (String commandString, Map params) {
Response reponse = getResponse(commandString, getAssetStatus(params));
return reponse.getDomainCommand();
}
private AssetStatus getAssetStatus(Map params) {
String id = getParam("assetID", params);
Asset asset = Asset.find(id);
return asset.getStatus();
}
private String getParam(String key, Map params) {
return ((String[]) params.get(key))[0];
}
```

All the domain commands follow a simple interface which allows the front controller to run them.

```
interface DomainCommand...
abstract public void run(Map params);
```

Once the domain command has done what it needs to do then the *Application Controller* comes into play again as it's asked for the view.

```
class AssetApplicationController...
public String getView (String commandString, Map params) {
return getResponse(commandString, getAssetStatus(params)).getViewUrl();
}
```

In this case the *Application Controller* doesn't return the full URL to the JSP. It returns a string that the front controller turns into an URL. I do this to avoid duplicating the URL paths in the responses. This also will make it easy to add further indirection later should I need it.

The *Application Controller* can be loaded for use with code.

```
class AssetApplicationController...
public void addResponse(String event, Object state, Class domainCommand,
String view) {
Response newResponse = new Response (domainCommand, view);
if ( ! events.containsKey(event))
events.put(event, new HashMap());
getResponseMap(event).put(state, newResponse);
}
private static void load ApplicationController(AssetApplicationController
appController) {
appController = AssetApplicationController.getDefault();
appController.addResponse("return", AssetStatus.ONLEASE,
GatherReturnDetailsCommand.class, "return");
appController.addResponse("return", AssetStatus.INVENTORY,
NullAssetCommand.class, "illegalAction");
appController.addResponse("damage", AssetStatus.ONLEASE,
InventoryDamageCommand.class, "leaseDamage");
appController.addResponse("damage", AssetStatus.INVENTORY,
LeaseDamageCommand.class, "inventoryDamage");
```

}

Doing this from a file instead isn't rocket science, but even so I'll leave it to you.



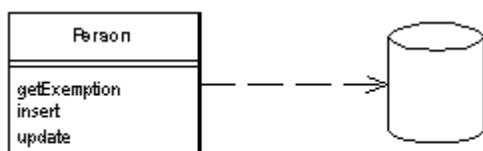
© Copyright [Martin Fowler](#), all rights reserved

A blue rectangular banner with a white border. On the left side is a 3D cube graphic with faces colored red, green, blue, and yellow, and letters A, B, and C on its visible faces. To the right of the banner, the text "ABC Amber CHM Converter Trial version" is displayed in red. Below this, in smaller black text, is "Please register to remove this banner." At the bottom of the banner, the URL "http://www.processtext.com/abcchm.html" is shown in blue.

# Active Record

---

*An object that wraps a record data structure in an external resource, such as a row in a database table, and adds some domain logic to that object.*



An object carries both data and behavior. Much of this data is persistent, and needs to be stored to a database. *Active Record* uses the most obvious approach: put data access logic into the domain object. This way all people know how to read and write their data from the database.

## How it Works

The essence of a *Active Record* is of a [\*Domain Model\*](#) where the classes in the [\*Domain Model\*](#) match very closely with the record structure of an underlying database. Each *Active Record* is responsible to saving and loading to the database, and also any domain logic that acts upon the data. This may be all the domain logic in the application, or you may find that some domain logic is held in [\*Transaction Script\*](#)s with common and data-oriented code in the *Active Record*.

The data structure of the *Active Record* should exactly match that of the database: one field in the class for each column in the table. Type the fields the way the SQL interface gives you the data - don't do any conversion at this stage. You might consider [\*Identity Field\*](#), but it may also be fine to leave the foreign keys as they are. You can use views or tables with *Active Record*, although obviously updates through views are usually limited. Views are particularly useful for reporting purposes.

The *Active Record* class typically has the following methods

- construct an instance of the *Active Record* from a SQL result set row
- construct a new instance for later insertion into the table
- static finder methods to wrap commonly used SQL queries and return *Active Record* objects
- methods to update the database and insert into the database with the data in the *Active Record*
- getting and setting methods for the fields
- methods that implement some pieces of business logic

The getting and setting methods can do some more intelligent things. They can convert from the SQL

oriented types to better in memory types. Also if you ask for a related table, the getting method can return the appropriate *Active Record*, even if you aren't using *Identity Field* on the data structure (by doing a lookup).

In this pattern the tables are a convenience, but they don't hide the fact that a relational database is present. As a result you usually see less of the other object-relational mapping patterns present when you're using *Active Record*.

It's often difficult to tell the difference between *Active Record* and *Row Data Gateway* when you are using a *Domain Model*. You are using *Active Record* if your domain objects contain the SQL themselves to access the database, and you're using *Row Data Gateway* if a separate class does this. If you have some SQL in the domain objects and some in separate database objects then you have some form of hybrid.

## When to Use it

*Active Record* is a good choice when your domain logic is not too complex, such as create, read, update and deletes.. Derivations and validations based on a single record work well in this structure.

In an initial design for a *Domain Model* the main choice is between *Active Record* and *Data Mapper*. *Active Record* has the primary advantage of simplicity. It's easy to build *Active Records* and they are easy to understand. The primary problem with them is they work well only if the *Active Record* objects correspond directly to the database tables: an isomorphic schema. If your business logic is complex then you'll soon want to use your object's mechanisms such as direct relationships, collections, and inheritance. These don't map easily onto *Active Record*. Adding them piecemeal soon gets very messy, so that's what will lead you to use *Data Mapper* instead.

Another argument against *Active Record* is the fact that it couples the object design to the database design. This makes it more difficult to refactor either design as the project goes forward.

*Active Record* is a good pattern to consider if you are using *Transaction Script* and you are beginning to feel some of the pain from code duplication, and difficulty in updating scripts and tables that *Transaction Script* often brings. In this case you can gradually start creating *Active Records* and then slowly refactor behavior into them. Often it helps to first wrap the tables as a *Gateway*, and then start moving behavior so they evolve to a *Active Record*.

## Example: A Simple Person (Java)

This is a simple, even simplistic, example to show how the bones of *Active Record* works. We begin with a simple person class that has these fields

```
class Person...
private String lastName;
private String firstName;
private int numberofDependents;
```

There's also an ID field in the superclass.

The database is set up in the same structure

```
create table people (ID int primary key, lastname varchar, firstname varchar,
number_of_dependents int)
```

To load an object the person class acts as the finder and also performs the load. It uses static methods on the person class.

```
class Person...
private final static String findStatementString =
"SELECT id, lastname, firstname, number_of_dependents" +
"FROM people" +
"WHERE id = ?";

public static Person find(Long id) {
Person result = (Person) Registry.getPerson(id);
if (result != null) return result;
PreparedStatement findStatement = null;
ResultSet rs = null;
try {
findStatement = DB.prepare(findStatementString);
findStatement.setLong(1, id.longValue());
rs = findStatement.executeQuery();
rs.next();
result = load(rs);
return result;
} catch (SQLException e) {
throw new ApplicationException(e);
} finally {
DB.cleanUp(findStatement, rs);
}
}

public static Person find(long id) {
return find(new Long(id));
}

public static Person load(ResultSet rs) throws SQLException {
Long id = new Long(rs.getLong(1));
Person result = (Person) Registry.getPerson(id);
if (result != null) return result;
String lastNameArg = rs.getString(2);
String firstNameArg = rs.getString(3);
int numDependentsArg = rs.getInt(4);
result = new Person(id, lastNameArg, firstNameArg, numDependentsArg);
Registry.addPerson(result);
return result;
}
```

Updating an object is a simple instance method

```
class Person...
private final static String updateStatementString =
"UPDATE people" +
"set lastname = ?, firstname = ?, number_of_dependents = ?" +
"where id = ?";
```

```
public void update() {
PreparedStatement updateStatement = null;
try {
updateStatement = DB.prepare(updateStatementString);
updateStatement.setString(1, lastName);
updateStatement.setString(2, firstName);
updateStatement.setInt(3, numberOfDependents);
updateStatement.setInt(4, getID().intValue());
updateStatement.execute();
} catch (Exception e) {
throw new ApplicationException(e);
} finally {
DB.cleanUp(updateStatement);
}
}
```

Insertions are also mostly pretty simple.

```
class Person...
private final static String insertStatementString =
"INSERT INTO people VALUES (?, ?, ?, ?, ?)";

public Long insert() {
PreparedStatement insertStatement = null;
try {
insertStatement = DB.prepare(insertStatementString);
setID(findNextDatabaseId());
insertStatement.setInt(1, getID().intValue());
insertStatement.setString(2, lastName);
insertStatement.setString(3, firstName);
insertStatement.setInt(4, numberOfDependents);
insertStatement.execute();
Registry.addPerson(this);
return getID();
} catch (Exception e) {
throw new ApplicationException(e);
} finally {
DB.cleanUp(insertStatement);
}
}
```

Any business logic, such as calculating the exemption, then sits directly in the Person class

```
class Person...
public Money getExemption() {
Money baseExemption = Money.dollars(1500);
Money dependentExemption = Money.dollars(750);
return
baseExemption.add(dependentExemption.multiply(this.getNumberOfDependents()));
}
```



ABC Amber CHM Converter Trial version

Please register to remove this banner.

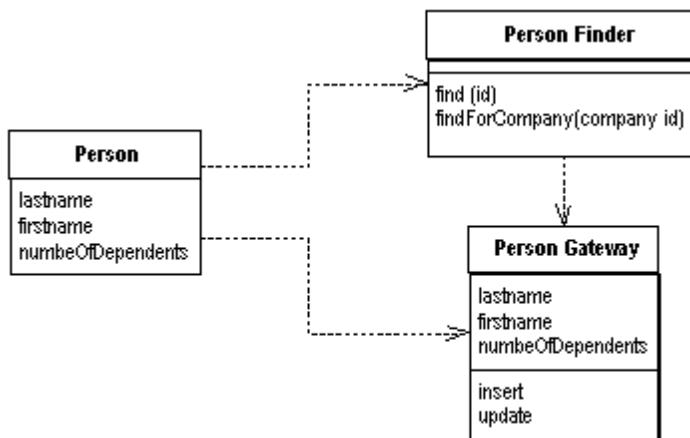
<http://www.processtext.com/abcchm.html>

---

# Row Data Gateway

---

An object that acts as a [Gateway](#) to a single record in a data source



Embedding database access code into in-memory objects can leave you with a fair few disadvantages. For a start it adds complexity. If your in-memory objects have business logic of their own, adding the database manipulation code is another aspect of complexity. Testing is awkward too, because if your in-memory objects are tied to a database tests are slower to run due to all the database access. You may have to access multiple databases with all those little annoying variations on their SQL.

A *Row Data Gateway* gives you objects that look exactly like the record in your record structure, but can be accessed with the regular programming mechanisms of your programming language. All the details of accessing the data source are hidden behind this interface.

## How it Works

A *Row Data Gateway* acts as an object that exactly mimics a single record, such as one database row. Each column in the database becomes one field in the *Row Data Gateway*. The *Row Data Gateway* will usually do any type conversion from the data source types to the in-memory types, but this type conversion is usually pretty simple. The *Row Data Gateway* holds the data about a row a client can then access the *Row Data Gateway* directly. The gateway acts as a good interface for each row of data. This approach works particularly well for [Transaction Scripts](#).

With a *Row Data Gateway* you're faced with the questions of where to put the find operations that generate the *Row Data Gateways*. You can use static find methods, but that precludes polymorphism

should you want to substitute different finder methods for different data sources. In this case it often makes sense to have separate finder objects. So each table in a relational database will have one finder class and one gateway class for the results.

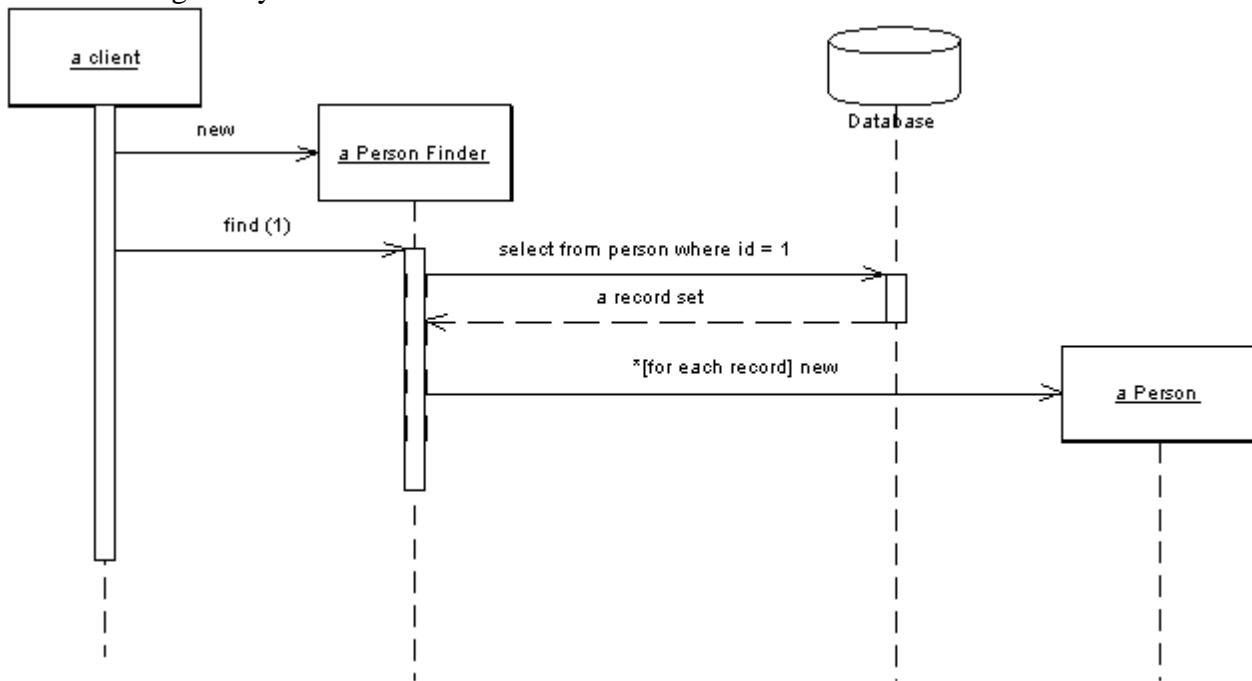


Figure 1: Interactions for a find with a row based Row Data Gateway

It's often hard to tell the difference between a *Row Data Gateway* and an [Active Record](#). The crux of the matter is whether there's any domain logic present, if so you have an [Active Record](#). A *Row Data Gateway* should contain only database access logic, and no domain logic.

## When to Use it

The choice of *Row Data Gateway* often comes in two steps, firstly whether to use a gateway at all and then whether to use *Row Data Gateway* or [Table Data Gateway](#).

I use *Row Data Gateway* most often when I'm using a [Transaction Script](#). In this case the *Row Data Gateway* nicely factors out the database access code and also allows it to be reused easily amongst different [Transaction Scripts](#).

I don't use a *Row Data Gateway* when I'm using a [Domain Model](#). If the mapping is simple then [Active Record](#) does the same job without an additional layer of code. If the mapping is complex then [Data Mapper](#) works better as it's not much more effort to write, but does a better job of decoupling the data structure from the domain objects because the domain objects don't need to know what the layout of the database is. Of course you can use the *Row Data Gateway* to shield the domain objects from the database structure, and that is a good thing to do if you are changing the database structure when using *Row Data Gateway* and don't want to change the domain logic. But doing this on a large scale leads you to three data representations: one in the business logic, one in the *Row Data Gateway*, and one in the database - and that's one too many. So I usually have *Row Data Gateways* that are the mirror of the database structure.

If you use [Transaction Script](#) with *Row Data Gateway* you may notice that you have business logic that's repeated across multiple scripts that would make sense in the *Row Data Gateway*. Moving that logic will gradually turn your *Row Data Gateway* into [Active Records](#). This is often a good thing as it reduces duplication in the business logic.

## Example: A Person Record (Java)

Here's an example for *Row Data Gateway*. The table is a simple person table.

```
create table people (ID int primary key, lastname varchar, firstname varchar,
number_of_dependents int)
```

PersonGateway is a gateway for this table. It starts with data fields and accessors

```
class PersonGateway...
private String lastName;
private String firstName;
private int numberofDependents;

public String getLastname() {
return lastName;
}

public void setLastname(String lastName) {
this.lastName = lastName;
}

public String getFirstname() {
return firstName;
}

public void setFirstname(String firstName) {
this.firstName = firstName;
}

public int getNumberofDependents() {
return numberofDependents;
}

public void setNumberofDependents(int numberofDependents) {
this.numberofDependents = numberofDependents;
}
```

The gateway class itself can handle updates and inserts

```
class PersonGateway...
private static final String updateStatementString =
"UPDATE people " +
"set lastname = ?, firstname = ?, number_of_dependents = ? " +
"where id = ?";

public void update() {
PreparedStatement updateStatement = null;
try {
```

```
updateStatement = DB.prepare(updateStatementString);
updateStatement.setString(1, lastName);
updateStatement.setString(2, firstName);
updateStatement.setInt(3, numberOfDependents);
updateStatement.setInt(4, getID().intValue());
updateStatement.execute();
} catch (Exception e) {
throw new ApplicationException(e);
} finally {DB.cleanUp(updateStatement);
}
}

private static final String insertStatementString =
"INSERT INTO people VALUES (?, ?, ?, ?)";

public Long insert() {
PreparedStatement insertStatement = null;
try {
insertStatement = DB.prepare(insertStatementString);
setID(findNextDatabaseId());
insertStatement.setInt(1, getID().intValue());
insertStatement.setString(2, lastName);
insertStatement.setString(3, firstName);
insertStatement.setInt(4, numberOfDependents);
insertStatement.execute();
Registry.addPerson(this);
return getID();
} catch (SQLException e) {
throw new ApplicationException(e);
} finally { DB.cleanUp(insertStatement);
}
}
}
```

To pull people out of the database, we have a separate PersonFinder. This works with the gateway to create new gateway objects

```
class PersonFinder...
private final static String findStatementString =
"SELECT id, lastname, firstname, number_of_dependents " +
"from people " +
"WHERE id = ?";

public PersonGateway find(Long id) {
PersonGateway result = (PersonGateway) Registry.getPerson(id);
if (result != null) return result;
PreparedStatement findStatement = null;
ResultSet rs = null;
try {
findStatement = DB.prepare(findStatementString);
findStatement.setLong(1, id.longValue());
rs = findStatement.executeQuery();
rs.next();
result = PersonGateway.load(rs);
return result;
} catch (SQLException e) {
throw new ApplicationException(e);
} finally {DB.cleanUp(findStatement, rs);
}
}
```

```
public PersonGateway find(long id) {
    return find(new Long(id));
}
class PersonGateway...
public static PersonGateway load(ResultSet rs) throws SQLException {
    Long id = new Long(rs.getLong(1));
    PersonGateway result = (PersonGateway) Registry.getPerson(id);
    if (result != null) return result;
    String lastNameArg = rs.getString(2);
    String firstNameArg = rs.getString(3);
    int numDependentsArg = rs.getInt(4);
    result = new PersonGateway(id, lastNameArg, firstNameArg, numDependentsArg);
    Registry.addPerson(result);
    return result;
}
```

The finder uses a [Registry](#) to hold [Identity Map](#)s.

We can then use the gateways from a [Transaction Script](#)

```
PersonFinder finder = new PersonFinder();
Iterator people = finder.findResponsibles().iterator();
StringBuffer result = new StringBuffer();
while (people.hasNext()) {
    PersonGateway each = (PersonGateway) people.next();
    result.append(each.getLastName());
    result.append("\t");
    result.append(each.getFirstName());
    result.append("\t");
    result.append(String.valueOf(each.getNumberOfDependents()));
    result.append("\n");
}
return result.toString();
```

If we want to use the *Row Data Gateway* from a domain object then the domain object needs to get at the data from the gateway. Instead of copying the data to the domain object we can use the row based gateway as a data holder for the domain object.

```
class Person...
private PersonGateway data;

public Person(PersonGateway data) {
    this.data = data;
}
```

Accessors on the domain logic can then delegate to the gateway for the data

```
class Person...
public int getNumberOfDependents() {
    return data.getNumberOfDependents();
}
```

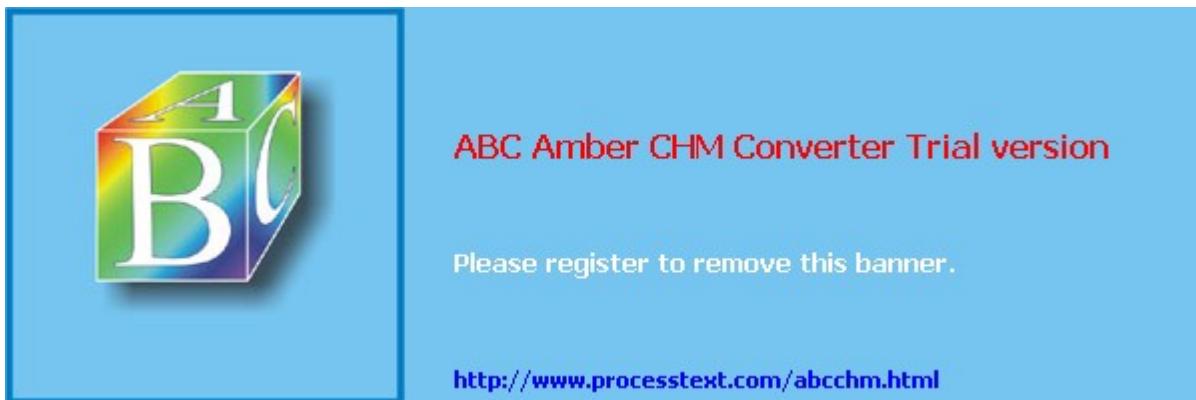
The domain logic then uses the getters to pull the data from the gateway

```
class Person...
public Money getExemption() {
Money baseExemption = Money.dollars(1500);
Money dependentExemption = Money.dollars(750);
return
baseExemption.add(dependentExemption.multiply(this.getNumberOfDependents())));
}
```



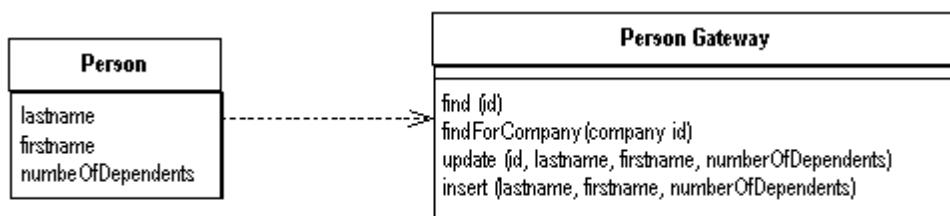
---

© Copyright [Martin Fowler](#), all rights reserved



# Table Data Gateway

An object that acts as a [Gateway](#) to a database table



It's good to keep database access code separated from the rest of an application. A simple *Table Data Gateway* holds all the SQL for accessing a single table: selects, inserts, updates and deletes. Other routine call its manipulation routines to for all changes in the database and issue queries through it's find routines, each of which passes back a suitable data structure.

## How it Works

A *Table Data Gateway* has a simple interface, usually consisting of several find methods to get data from the database, together with update, insert, and delete methods. Each method maps the input parameters into a SQL and executes the SQL against a database connection. The *Table Data Gateway* is usually stateless, as its role is to push data back and forth.

The trickiest thing about a *Table Data Gateway* is how it returns information from a query. Even a simple find-by-id query will return multiple data items. In environments where you can return multiple items you can use that for a single row. However many languages only give you a single return value, and many queries will return multiple rows.

On alternative is to return some simple data structure, such as a map. A map works, but it forces data to be copied out of the record set that comes from the database into the map. I find that using maps to pass around data is bad form, it defeats compile time checking and is not a very explicit interface - leading to bugs as people misspell what's in the map. A better alternative is to use a [Data Transfer Object](#), another object to create - but one that may well be used elsewhere.

To save all this you can return the [Record Set](#) that comes from the SQL query. This is conceptually messy, as ideally we would like the in-memory object to not have to know anything about the SQL interface. It may also make it difficult to substitute the database for a file if you cannot easily create

record sets in your own code. However in many environments, such as .NET, that use [Record Set](#) widely it's a very effective approach. A table based *Table Data Gateway* thus goes very well with [Table Module](#). If all of your updates are done through the *Table Data Gateway* the returned data can be based on views rather than the actual tables, which reduces the coupling between your code and the database.

If you're using a [Domain Model](#) you can have the *Table Data Gateway* return the appropriate domain object. The problem with this is that you then have bidirectional dependencies between the gateway and the domain objects. Since the two are closely connected that isn't necessarily a terrible thing, but it's always something I'm reluctant to do.

Most of the time you use *Table Data Gateway* you'll have one *Table Data Gateway* for each table in the database. For very simple cases, however, you can have a single *Table Data Gateway* that handles all the methods for all the tables.

## When to Use it

As with [Row Data Gateway](#) the decision to use *Table Data Gateway* is first whether to use a [Gateway](#) approach, and then which one to use.

I find *Table Data Gateway* is probably the simplest database interface pattern to use, as it maps so nicely onto a database table or record type. It also makes a natural point to encapsulate the precise access logic of the data source. I use it least with [Domain Model](#), because I find that [Data Mapper](#) gives a better isolation between the [Domain Model](#) and the database, and isn't that much more complex to use.

*Table Data Gateway* works particularly well with [Table Module](#) where the *Table Data Gateway* produces a record set data structure for the [Table Module](#) to work on. Indeed I can't really imagine any other database mapping approach for [Table Module](#).

Just like [Row Data Gateway](#), *Table Data Gateway* is very suitable for [Transaction Scripts](#). The choice between the two really boils down on how to deal with multiple rows of data. Many people like using a [Data Transfer Object](#), but that usually seems like more work than is worthwhile, unless the same [Data Transfer Object](#) is used elsewhere. I tend to prefer *Table Data Gateway* when the result set representation is convenient for the [Transaction Script](#) to work with.

Interestingly it often makes sense to use *Table Data Gateway* with [Data Mapper](#) - having the [Data Mappers](#) talk to the database via *Table Data Gateways*. Although this makes little sense when everything is hand coded, it can be very effective when if you want to use metadata for the *Table Data Gateways* but prefer hand code for the actual mapping to the domain objects.

One of the benefits of using a *Table Data Gateway* to encapsulate data base access is that the same interface can be used both for using SQL to manipulate the database, and for using stored procedures. Indeed stored procedures themselves are often organized as a *Table Data Gateway*, that way the actual table structure is encapsulated behind insert and update stored procedures. The find procedures in this case can return views, which all helps to hide the underlying table structure.

# Further Reading

[Alur, Crupi, and Malks] contains the *Data Access Object* pattern which is a *Table Data Gateway*. In the discussion they show returning a collection of *Data Transfer Objects* on the query methods. It's not clear whether they see *Data Access Object* as always being table based, the intent and discussion seems to imply either *Table Data Gateway* or *Row Data Gateway*.

I've used a different name, partly because I see this pattern as a particular usage of the more general *Gateway* concept, and felt the pattern name should reflect that. The other reason is that the term Data Access Object and it's abbreviation DAO has it's own particular meaning within the Microsoft world.

## Example: Person Gateway (C#)

*Table Data Gateway* is the usual form of database access in the windows world, so it makes sense to illustrate one with C#. In doing this, however, I have to stress that this classic form of *Table Data Gateway* isn't quite the way that fits in with the .NET environment since it doesn't take advantage of the ADO.NET data set, instead it uses the Data Reader which is a cursor like interface to database records. The data reader is the right choice to manipulating larger amounts of information where you don't want to bring everything into memory in one go.

For the example I'm using a person gateway class that connects to a person table in a database. The person gateway contains the finder code, returning ADO.NET's data reader to access the returned data.

```
class PersonGateway...
public IDataReader FindAll() {
String sql = "select * from person";
return new OleDbCommand(sql, DB.Connection).ExecuteReader();
}
public IDataReader FindWithLastName(String lastName) {
String sql = "SELECT * FROM person WHERE lastname = ?";
IDbCommand comm = new OleDbCommand(sql, DB.Connection);
comm.Parameters.Add(new OleDbParameter("lastname", lastName));
return comm.ExecuteReader();
}
public IDataReader FindWhere(String whereClause) {
String sql = String.Format("select * from person where {0}", whereClause);
return new OleDbCommand(sql, DB.Connection).ExecuteReader();
}
```

Almost always you'll want to pull back a bunch of rows with a reader. On a rare occasion you might want to get hold of an individual row of data with a method along these lines.

```
class PersonGateway...
public Object[] FindRow (long key) {
String sql = "SELECT * FROM person WHERE id = ?";
IDbCommand comm = new OleDbCommand(sql, DB.Connection);
comm.Parameters.Add(new OleDbParameter("key",key));
IDataReader reader = comm.ExecuteReader();
reader.Read();
Object [] result = new Object[reader.FieldCount];
```

```
reader.GetValues(result);
reader.Close();
return result;
}
```

The update and insert methods receive the necessary data in arguments and invokes the appropriate SQL routines.

```
class PersonGateway...
public void Update (long key, String lastname, String firstname, long
numberOfDependents){
String sql = "UPDATE person SET lastname = ?, firstname = ?,
numberOfDependents = ? WHERE id = ?";
IDbCommand comm = new OleDbCommand(sql, DB.Connection);
comm.Parameters.Add(new OleDbParameter ("last", lastname));
comm.Parameters.Add(new OleDbParameter ("first", firstname));
comm.Parameters.Add(new OleDbParameter ("numDep", numberOfDependents));
comm.Parameters.Add(new OleDbParameter ("key", key));
comm.ExecuteNonQuery();
}

class PersonGateway...
public long Insert(String lastName, String firstName, long
numberOfDependents) {
String sql = "INSERT INTO person VALUES (?,?,?,?)";
long key = GetNextID();
IDbCommand comm = new OleDbCommand(sql, DB.Connection);
comm.Parameters.Add(new OleDbParameter ("key", key));
comm.Parameters.Add(new OleDbParameter ("last", lastName));
comm.Parameters.Add(new OleDbParameter ("first", firstName));
comm.Parameters.Add(new OleDbParameter ("numDep", numberOfDependents));
comm.ExecuteNonQuery();
return key;
}
```

The deletion method just needs a key

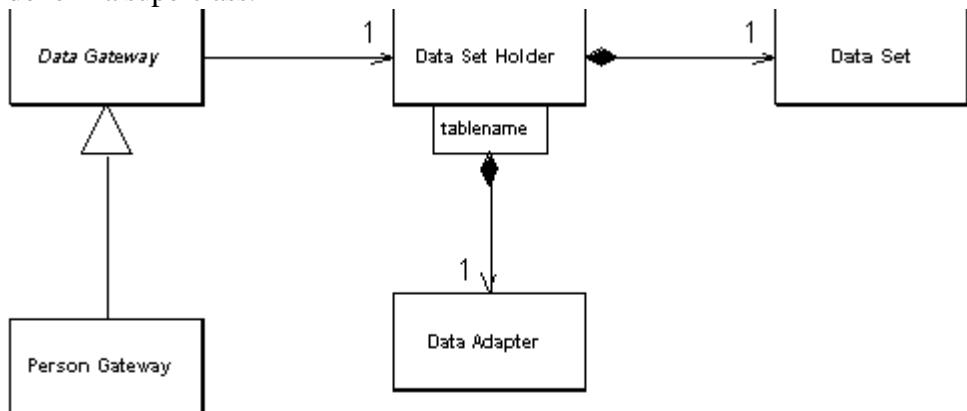
```
class PersonGateway...
public void Delete (long key) {
String sql = "DELETE FROM person WHERE id = ?";
IDbCommand comm = new OleDbCommand(sql, DB.Connection);
comm.Parameters.Add(new OleDbParameter ("key", key));
comm.ExecuteNonQuery();
}
```

## Example: Using ADO.NET Data Sets (C#)

The generic *Table Data Gateway* works with pretty much any kind of platform since it is nothing but a wrapper for SQL statements. When you use .NET you'll use data sets more often, but *Table Data Gateway* is still useful, although it comes in a different form.

A data set needs data adapters to load the data into the data set and do updates to the data. So I found it useful to define a holder for the data set and adapters that load and save the data. A gateway then uses the holder to store both the data sets and the adapters. Much of this behavior is generic, and can be

done in a superclass.



*Figure 1: Class diagram of data set oriented gateway and the supporting data holder*

The holder stores a data set and a collection of adapters indexed by the name of the table.

```

class DataSetHolder...
public DataSet Data = new DataSet();
private Hashtable DataAdapters = new Hashtable();
  
```

The gateway stores the holder and exposes the data set for its clients.

```

class DataGateway...
public DataSetHolder Holder;
public DataSet Data {
get {return Holder.Data;}
}
  
```

The gateway can act on an existing holder, or create a new one.

```

class DataGateway...
protected DataSetGateway() {
Holder = new DataSetHolder();
}
protected DataSetGateway(DataSetHolder holder) {
this.Holder = holder;
}
  
```

The find behavior can work a bit differently here. Since a data set is a container for table oriented data, and one data set can contain data from several tables. As a result it's better to load data into a data set.

```

class DataGateway...
public void LoadAll() {
String commandString = String.Format("select * from {0}", TableName);
Holder.FillData(commandString, TableName);
}
public void LoadWhere(String whereClause) {
String commandString =
String.Format("select * from {0} where {1}", TableName,whereClause);
Holder.FillData(commandString, TableName);
}
abstract public String TableName {get;}
class PersonGateway...
public override String TableName {
get {return "Person";}
}
  
```

```
}

class DataSetHolder...
public void FillData(String query, String tableName) {
if (DataAdapters.Contains(tableName)) throw new MutlipleLoadException();
OleDbDataAdapter da = new OleDbDataAdapter(query, DB.Connection);
OleDbCommandBuilder builder = new OleDbCommandBuilder(da);
da.Fill(Data, tableName);
DataAdapters.Add(tableName, da);
}
```

To update data you manipulate the data set directly in some client code.

```
person.LoadAll();
person[key]["lastname"] = "Odell";
person.Holder.Update();
```

The gateway can have an indexer to get make it easier to get to specific rows.

```
class DataGateway...
public DataRow this[long key] {
get {
String filter = String.Format("id = {0}", key);
return Table.Select(filter)[0];
}
}
public override DataTable Table {
get { return Data.Tables[TableName]; }
}
```

The update triggers update behavior on the holder.

```
class DataSetHolder...
public void Update() {
foreach (String table in DataAdapters.Keys)
((OleDbDataAdapter)DataAdapters[table]).Update(Data, table);
}
public DataTable this[String tableName] {
get {return Data.Tables[tableName]; }
}
```

Insertion can be done much the same way: get a data set, insert a new row to the data table, and fill in each column. However it can be useful to use an update method to do the insertion in one call.

```
class DataGateway...
public long Insert(String lastName, String firstname, int numberOfDependents)
{
long key = new PersonGatewayDS().GetNextID();
DataRow newRow = Table.NewRow();
newRow["id"] = key;
newRow["lastName"] = lastName;
newRow["firstName"] = firstname;
newRow["numberOfDependents"] = numberOfDependents;
Table.Rows.Add(newRow);
return key;
}
```

---



---

© Copyright [Martin Fowler](#), all rights reserved



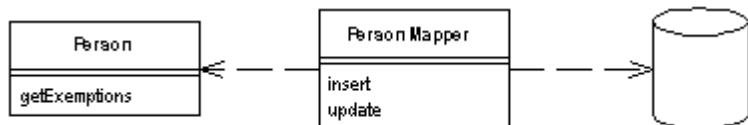
A blue rectangular banner with a white border. On the left side is a 3D-style cube with faces colored red, green, blue, and yellow, with the letters 'A', 'B', and 'C' visible on its faces. To the right of the cube, the text "ABC Amber CHM Converter Trial version" is displayed in red. Below this, in smaller black text, is "Please register to remove this banner." At the bottom right of the banner is the URL "<http://www.processtext.com/abcchm.html>".

---

# Data Mapper

---

*Transfers data from a domain object to a database*



Objects and relational databases have different mechanisms for structuring data. Many parts of objects, such as collections and inheritance are not present in relational databases. When you are building an object model with a lot of business logic it's valuable to use these mechanisms to better organize the data and the behavior that goes with it. This leads to variant schemas: where the object schema and the relational schema do not match up.

In this situation you still need to transfer data between the two schemas. This data transfer becomes a complexity in its own right. If the in-memory objects know about the relational database structure, then changes in one tend to ripple to the other.

The *Data Mapper* is a layer of software that acts as a [mediator](#) between the in-memory objects and the database. Its responsibility is to transfer data between the two, and also the two layers from each other. Using *Data Mapper* the in-memory objects need have no knowledge that there's even a database present, no SQL interface code, and certainly no knowledge of the database schema. (The database schema is always ignorant of the objects that use it.)

## How it Works

The separation between domain and data source is the main goal of a *Data Mapper*, but there are plenty of details that have to be addressed to make it happen. There's also a lot of variety in how different people have built their mapping layers. So much of the comments here are pretty broad, as I try to give a general overview of what you need to separate the cat from its skin.

We'll start with a very simple database mapper example. This is the simplest style of this layer that you might get, but is often too simple to actually be worth doing. With simple database mapping examples other patterns usually are simpler and thus better. If you are going to use *Data Mapper* at all you usually need more complicated cases. But it's easier to explain the ideas if we start simple.

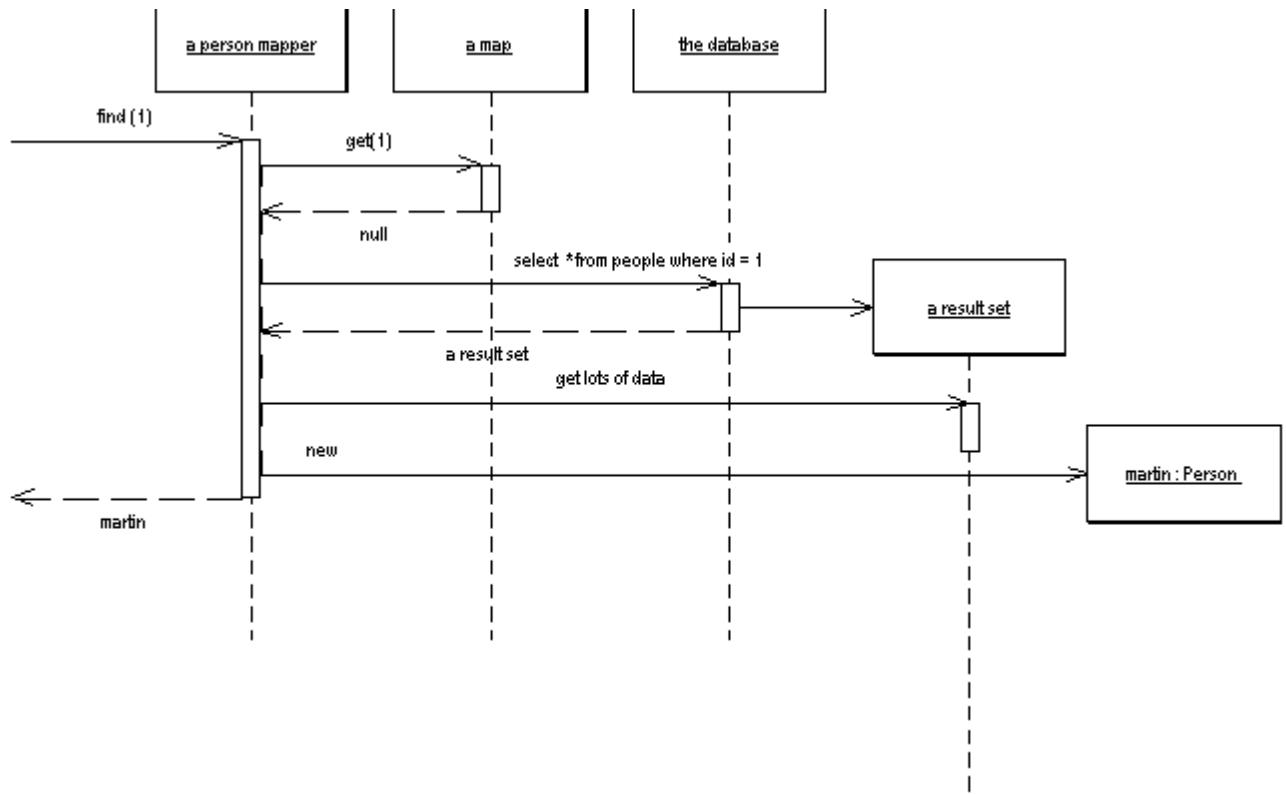


Figure 1: Retrieving Data from a Database

A simple case would have a person and person mapper class. To load a person from the database, a client would call a `find` method on the mapper Figure 1. The mapper uses an [Identity Map](#) to see if the person is already loaded, and if not it loads the person.

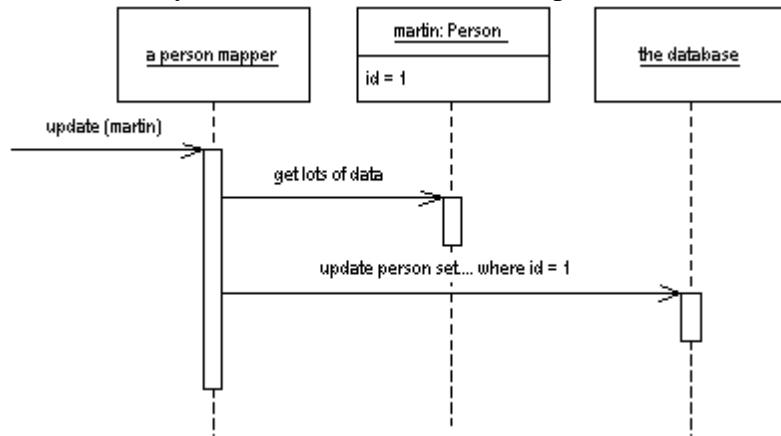


Figure 2: Updating some data

Updates are similarly simple (Figure 2). A client asks the mapper to save a domain object. The mapper pulls the data out of the domain object and shuttles it to the database.

The whole layer of *Data Mapper* can be substituted, either for testing purposes, or to allow a single domain layer to work with different databases.

In this simple case, the mapper separates the database code away from the domain objects, thus making the domain objects simpler as they focus on only one task. But soon other issues come into play which suggest other patterns.

## Handing Finders

One issue that soon raises its head is how to deal with finding objects. From time to time methods in the domain layer will need to invoke finder behavior. However if the finder behavior is defined in the mappers this will break the dependency rule that says that the domain layer should not depend on the mapper layer. To avoid this you can define interfaces for the finders in a separate package and implement them in the mapper layer ( Figure 3).

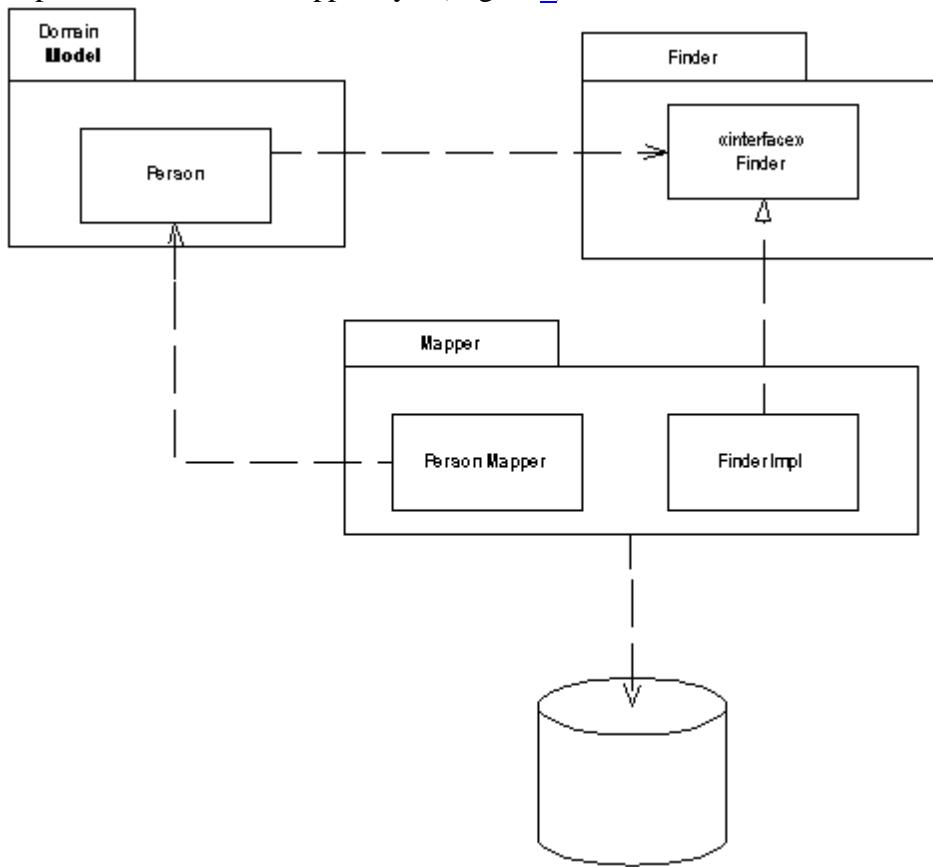


Figure 3: Moving the finder behavior into a separate package to separate the dependencies

In this case the domain object calls the finder through the interface, and the finder then calls the mapper to load the data.

This example is a gross simplification in lots of ways, but it should give you a sense of broadly what's going on. The key element to note here are the two principle roles in the layer.

- the finder: who translates method calls into SQL queries
- the mapper: who pulls data out of the row set to create the object.

When it comes to inserts and updates, there's a new level of complexity as the database mapping layer needs to understand what objects have changed, which new ones have been created, and which ones have been destroyed. It also has to fit the whole workload into a transactional framework. The [Unit of Work](#) pattern is a good way to organize this.

A simple *Data Mapper* would just map a database table to an equivalent in-memory class on a field to

field basis. Of course things aren't usually that simple. Mappers need a variety of strategies to handle classes that turn into multiple fields, classes that have multiple tables, classes with inheritance, and the joys of connecting together objects once they've been sorted out. The other patterns in this chapter deal with these. It's usually easier to deploy these patterns with a *Data Mapper*, than it is with the other organizing alternatives [Gateway](#) and [Active Record](#)

Figure 1 suggests that a single request to a finder results in a single SQL query. Often this isn't true. If we want to load a typical order with multiple order lines, then loading orders may involve loading the order lines as well. The request from the client will usually lead to a graph of objects being loaded, with the mapper designer deciding exactly how much to pull back in one go. The point of this is to minimize database queries. Thus the finders typically need to know a fair bit about the way in which the clients typically use the objects so they can make the best choices for pulling data back.

This example also leads to cases where you load multiple classes of domain objects from a single query. If you wanted to load orders and order lines, it will usually be faster to do a single query that joins the orders and order lines tables. You then use the result set would to load both the order and the order line instances.

Since objects are very interconnected, you usually have to stop pulling the data back at some point. Otherwise you are likely to pull back the entire database with a request. Again mapping layers have techniques to deal with this while minimizing the impact on the in memory objects using [Lazy Load](#). Hence the in-memory objects cannot be entirely ignorant of the mapping layer. The in-memory objects need to know about the finders, and a few other mechanisms.

An application can have one *Data Mapper* or several of them. In smaller applications having a single *Data Mapper* works well as it is easier to swap out for testing purposes. As a system grows, however, the finder will get too complicated. At this point it's worth breaking up. A common habit is to create one finder per domain class, or at least the head of a domain hierarchy. This leads to a lot of small finders, but it's easy for a developer to find the finder they need.

As with any database find behavior the finders need to use an [Identity Map](#) in order to maintain identity of the objects read from the database. You can either have a [Registry](#) of [Identity Map](#)s or have each finder hold an [Identity Map](#) (providing there is only one finder per class per session).

## Mapping data to domain fields

Mappers need to get access to the fields in the domain objects. Often this can be a problem because you need public methods to support the mappers that you don't want for domain logic. (I'm assuming you won't commit the cardinal sin of making fields public.) There's no easy to answer to this. You could use a lower level of visibility by packaging the mappers closer to the domain objects, such as the same package in Java, but this messes up the bigger dependency picture, since you don't want other parts of the system that know the domain objects to know about the mappers. You can use reflection, which often can bypass the visibility rules of the language. Reflection is slower, but the slower speed may end up as just rounding error compared to the time taken by the SQL call. You can use public methods, but guard them with a status field so that they throw an exception if they are used outside the context of a database load. If so name them in such a way that they don't get mistaken for regular getters and setters.

Tied to this is the issue of when you create the object. In essence you have two options. One is to create the object with a **rich constructor** so the object is created with at least all its mandatory data. The other

is to create an empty object and then populate it with the mandatory data. In most contexts I prefer the former since its nice to have a well formed object from the start. This also means that if you have an immutable field you can enforce this by not providing any method to change its value.

The problem with a rich constructor is that you have to be aware of cyclic references. If you have two objects that reference each other, each time you try to load one, it will try to load the other, which will try to load the first object, and so on until you run out of stack space. Avoiding this needs special case code, often using [Lazy Load](#). Since writing this special case code is messy, it's worth trying to avoid it. You can avoid it by creating an **empty object**. To do this you use a no-arg constructor to create a blank object and insert that empty object immediately into the [Identity Map](#). That way if you have a cycle the [Identity Map](#) will return an object to stop the recursive loading.

Using an empty object like this means you may need some setters for values that are really immutable when the object is loaded. A combination of a naming convention and perhaps some status checking guards can avoid this problem. You can also avoid this problem by using reflection for data loading.

## Metadata based mappings

One of the decisions you need to make is how to store the information about how the field in domain objects are mapped to columns in the database. The simplest, and often best, way to do this is with explicit code. This requires a mapper class for each domain object. The mapper does the mapping through assignments, and has fields (usually constant strings) to store the SQL for accessing the database. An alternative is to use [Metadata Mapping](#). In this approach the metadata is stored as data, either in a class or in a separate file. The great advantage of using metadata is that all the variation in the mappers can be handled through data without having to write any more source code, either by using code generation or reflective programming.

## When to Use it

The primary reason for using *Data Mapper* is when you want the database schema and the object model to evolve independently. The most common case for this is when you are using a [Domain Model](#). The primary benefit of *Data Mapper* is that when working on the domain model you can ignore the database, both in design and within the build and testing process. The domain objects have no idea what the database structure is, because all the correspondence is done by the mappers.

This helps you in the code because you can understand and work with the domain objects without having to understand how they are stored in the database. You can modify either the [Domain Model](#) and the database without having to necessarily alter the other. With complicated mappings, particularly those involving existing databases, this is very valuable.

The price, of course, is the extra layer compared to [Active Record](#). The test for using them is the complexity of the business logic. If you have fairly simple business logic then you probably won't need a [Domain Model](#) nor *Data Mapper*. More compiled logic leads you to [Domain Model](#) and therefore to *Data Mapper*.

So I wouldn't choose *Data Mapper* without [Domain Model](#), but would I use [Domain Model](#) without

*Data Mapper*? If the domain model is pretty simple, and the database is under the domain model developers' control, then it's reasonable for the domain objects to access the database directly. Effectively this puts the mapper behavior discussed here into the domain objects themselves. As things become more complicated, however, it's better to refactor the database behavior out into a separate layer.

Remember that you don't have to build a full featured database mapping layer. These are complicated beasts to build in full and there are products available that do this. For most cases I recommend buying a database mapping layer rather than building one yourself.

## Example: A simple database mapper (Java)

Here's an absurdly simple use of *Data Mapper* to give you a feel for the basic structure. Our example is a simple person with an isomorphic people table.

```
class Person...
private String lastName;
private String firstName;
private int numberDependents;
```

The database schema looks like this

```
create table people (ID int primary key, lastname varchar, firstname varchar,
number_of_dependents int)
```

We'll use the simple case here, where the person mapper class also implements the finder and [Identity Map](#). However I've added an abstract mapper [Layer Supertype](#) to indicate where I can pull out some common behavior. Loading involves checking the object isn't already in the [Identity Map](#), and then pulling the data from the database.

The find behavior starts in the person mapper which wraps calls to an abstract find method to find by id.

```
class PersonMapper...
protected String findStatement() {
return "SELECT " + COLUMNS +
"FROM people" +
"WHERE id = ?";
}

public static final String COLUMNS = " id, lastname, firstname,
number_of_dependents ";

public Person find(Long id) {
return (Person) abstractFind(id);
}

public Person find(long id) {
return find(new Long(id));
}

class AbstractMapper...
protected Map loadedMap = new HashMap();
abstract protected String findStatement();
```

```
protected DomainObject abstractFind(Long id) {
DomainObject result = (DomainObject) loadedMap.get(id);
if (result != null) return result;
PreparedStatement findStatement = null;
try {
findStatement = DB.prepare(findStatement());
findStatement.setLong(1, id.longValue());
ResultSet rs = findStatement.executeQuery();
rs.next();
result = load(rs);
return result;
} catch (SQLException e) {
throw new ApplicationException(e);
} finally {
DB.cleanUp(findStatement);
}
}
```

The find method calls the load method. This method is split between the abstract and person mappers. The abstract mapper handles checking the id, pulling the id from the data, and registering the new object in the [Identity Map](#)

```
class AbstractMapper...
protected DomainObject load(ResultSet rs) throws SQLException {
Long id = new Long(rs.getLong(1));
if (loadedMap.containsKey(id)) return (DomainObject) loadedMap.get(id);
DomainObject result = doLoad(id, rs);
loadedMap.put(id, result);
return result;
}

abstract protected DomainObject doLoad(Long id, ResultSet rs) throws
SQLException;
class PersonMapper...
protected DomainObject doLoad(Long id, ResultSet rs) throws SQLException {
String lastNameArg = rs.getString(2);
String firstNameArg = rs.getString(3);
int numDependentsArg = rs.getInt(4);
return new Person(id, lastNameArg, firstNameArg, numDependentsArg);
}
```

Notice that the [Identity Map](#) is checked twice, once by abstractFind and once by load. There is a reason for this madness.

I need to check the map in the finder, because that way if the object is already there I can save myself a trip to the database - and I always want to save myself that long hike if I can. But I also need to check in the load because I may have queries that I can't be sure of resolving in the [Identity Map](#). Say I want to find everyone whose last name matches some search pattern. I can't be sure that I have all such people already loaded, so I have to go to the database and run a query.

```
class PersonMapper...
private static String findLastNameStatement =
"SELECT " + COLUMNS +
"FROM people " +
"WHERE UPPER(lastname) like UPPER(?) " +
"ORDER BY lastname";
```

```
public List findByLastName(String name) {
PreparedStatement stmt = null;
ResultSet rs = null;
try {
stmt = DB.prepare(findLastNameStatement);
stmt.setString(1, name);
rs = stmt.executeQuery();
return loadAll(rs);
} catch (SQLException e) {
throw new ApplicationException(e);
} finally {
DB.cleanUp(stmt, rs);
}
}

class AbstractMapper...
protected List loadAll(ResultSet rs) throws SQLException {
List result = new ArrayList();
while (rs.next())
result.add(load(rs));
return result;
}
```

When I do this I may pull back some rows in the result set that correspond to people I've already loaded. I have to ensure I don't make a duplicate, so I have to check the [Identity Map](#) again.

Writing a find method this way in each subclass that needs it involves some simple, but repetitive coding. I can eliminate that by providing a general method.

```
class AbstractMapper...

public List findMany(StatementSource source) {
PreparedStatement stmt = null;
ResultSet rs = null;
try {
stmt = DB.prepare(source.sql());
for (int i = 0; i < source.parameters().length; i++)
stmt.setObject(i+1, source.parameters()[i]);
rs = stmt.executeQuery();
return loadAll(rs);
} catch (SQLException e) {
throw new ApplicationException(e);
} finally {
DB.cleanUp(stmt, rs);
}
}
```

For this to work I need an interface that wraps both the SQL string and the loading of parameters into the prepared statement

```
interface StatementSource...
String sql();
Object[] parameters();
```

I can then use this facility by providing a suitable implementation as an inner class.

```
class PersonMapper...
public List findByLastName2(String pattern) {
return findMany(new FindByLastName(pattern));
}

static class FindByLastName implements StatementSource {
private String lastName;
public FindByLastName(String lastName) {
this.lastName = lastName;
}
public String sql() {
return
"SELECT " + COLUMNS +
"FROM people " +
"WHERE UPPER(lastname) like UPPER(?) " +
"ORDER BY lastname";
}
public Object[] parameters() {
Object[] result = {lastName};
return result;
}
}
```

This kind of work can be done in other places where we have repetitive statement invocation code. On the whole I've the examples here more straight to make it easier to follow them, but if you find yourself writing a lot of repetitive straight-ahead code then you should consider doing something similar.

To perform an update I do the JDBC stuff in the [Layer Supertype](#) and the selection of fields in subtype.

```
class AbstractMapper... class PersonMapper...
private static final String updateStatementString =
"UPDATE people " +
"SET lastname = ?, firstname = ?, number_of_dependents = ? " +
"WHERE id = ?";

public void update(Person subject) {
PreparedStatement updateStatement = null;
try {
updateStatement = DB.prepare(updateStatementString);
updateStatement.setString(1, subject.getLastName());
updateStatement.setString(2, subject.getFirstName());
updateStatement.setInt(3, subject.getNumberOfDependents());
updateStatement.setInt(4, subject.getID().intValue());
updateStatement.execute();
} catch (Exception e) {
throw new ApplicationException(e);
} finally {
DB.cleanUp(updateStatement);
}
}
```

For the insert, again there's some code that can be factored into the [Layer Supertype](#)

```
class AbstractMapper...
public Long insert(DomainObject subject) {
PreparedStatement insertStatement = null;
try {
insertStatement = DB.prepare(insertStatement());
```

```
subject.setID(findNextDatabaseId());
insertStatement.setInt(1, subject.getID().intValue());
doInsert(subject, insertStatement);
insertStatement.execute();
loadedMap.put(subject.getID(), subject);
return subject.getID();
} catch (SQLException e) {
throw new ApplicationException(e);
} finally {
DB.cleanUp(insertStatement);
}
}

abstract protected String insertStatement();
abstract protected void doInsert(DomainObject subject, PreparedStatement
insertStatement)
throws SQLException;
class PersonMapper...
protected String insertStatement() {
return "INSERT INTO people VALUES (?, ?, ?, ?, ?)";
}

protected void doInsert(
DomainObject abstractSubject,
PreparedStatement stmt)
throws SQLException
{
Person subject = (Person) abstractSubject;
stmt.setString(2, subject.getLastName());
stmt.setString(3, subject.getFirstName());
stmt.setInt(4, subject.getNumberOfDependents());
}
```

## Example: Separating the finders (Java)

In order to allow domain objects invoke finder behavior I can separate the finder interfaces from the mappers. I can put these finder interfaces in a separate package that is visible to the domain layer, or in this case I can put them in the domain layer itself.

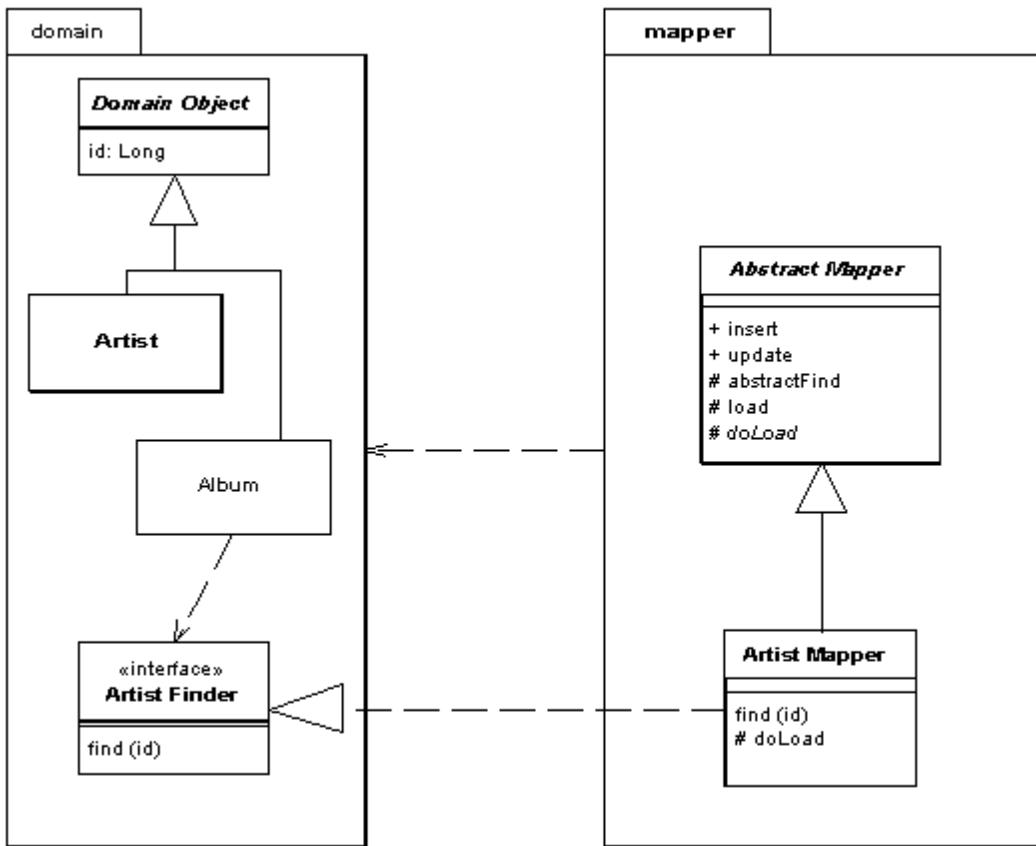


Figure 4: Defining a finder interface in the domain package

One of the most common kinds of find is one that finds an object according to a particular surrogate ID. Much of the processing in this is quite generic, so can be handled by a suitable [Layer Supertype](#)s. All it needs is a [Layer Supertype](#) for domain objects that know about ids.

The specific behavior for finding lies in the finder interface. This is usually best not made generic, because you need to know what the return type is.

```

interface ArtistFinder...
Artist find(Long id);

Artist find(long id);

```

The finder interface is best declared in the domain package with finders held in a [Registry](#). In this case I've made the mapper class implement the finder interface.

```

class ArtistMapper implements ArtistFinder...
public Artist find(Long id) {
    return (Artist) abstractFind(id);
}
public Artist find(long id) {
    return find(new Long(id));
}

```

The bulk of the find method is done by the mapper's [Layer Supertype](#). This involves checking the [Identity Map](#) to see if the object is already in memory. If not it completes a prepared statement that's loaded in by the artist mapper and executes it.

```
class AbstractMapper...
abstract protected String findStatement();
protected Map loadedMap = new HashMap();

protected DomainObject abstractFind(Long id) {
DomainObject result = (DomainObject) loadedMap.get(id);
if (result != null) return result;
PreparedStatement stmt = null;
ResultSet rs = null;
try {
stmt = DB.prepare(findStatement());
stmt.setLong(1, id.longValue());
rs = stmt.executeQuery();
rs.next();
result = load(rs);
return result;
} catch (SQLException e) {
throw new ApplicationException(e);
} finally {cleanUp(stmt, rs);
}
}

class ArtistMapper...
protected String findStatement() {
return "select " + COLUMN_LIST + " from artists art where ID = ?";
}

public static String COLUMN_LIST = "art.ID, art.name";
```

The find part of the behavior is about getting either the existing or a new object, the load part is about putting the data from the database into a new object.

```
class AbstractMapper...
protected DomainObject load(ResultSet rs) throws SQLException {
Long id = new Long(rs.getLong("id"));
if (loadedMap.containsKey(id)) return (DomainObject) loadedMap.get(id);
DomainObject result = doLoad(id, rs);
loadedMap.put(id, result);
return result;
}

abstract protected DomainObject doLoad(Long id, ResultSet rs) throws
SQLException;
class ArtistMapper...
protected DomainObject doLoad(Long id, ResultSet rs) throws SQLException {
String name = rs.getString("name");
Artist result = new Artist(id, name);
return result;
}
```

Notice that the load method also checks the [Identity Map](#). Although this is redundant in the this case, the load can also be called by other finders that haven't already done this check.

In this scheme all a subclass has to do is to develop a doLoad method to load the actual data that's needed, and return a suitable prepared statement from the findStatement method.

You can also do a find based on a query. In this case consider a database of tracks and albums.. We want a finder that will find all the tracks on a specified album. Again the interface declares the finders.

```
interface TrackFinder...
Track find(Long id);
Track find(long id);
List findForAlbum(Long albumID);
```

Since this is a specific find method for this class, this find method is implemented in a specific class, such as the track mapper class, rather than a [Layer Supertype](#). Like any finder, there are two methods to the implementation. One is setting up the prepared statement, the other is a method to wrap the call to the prepared statement, and interpret the results.

```
class TrackMapper...
public static final String findForAlbumStatement =
"SELECT ID, seq, albumID, title " +
"FROM tracks " +
"WHERE albumID = ? ORDER BY seq";

public List findForAlbum(Long albumID) {
PreparedStatement stmt = null;
ResultSet rs = null;
try {
stmt = DB.prepare(findForAlbumStatement);
stmt.setLong(1, albumID.longValue());
rs = stmt.executeQuery();
List result = new ArrayList();
while (rs.next())
result.add(load(rs));
return result;
} catch (SQLException e) {
throw new ApplicationException(e);
} finally {cleanUp(stmt, rs);
}
}
```

The finder calls a load method for each row in the result set. This method has the responsibility of creating the in memory object and loading it with the data. As the in the previous example, some of this can be handled in a [Layer Supertype](#), including checking the [Identity Map](#) to see if something is already loaded.

## Example: Creating an empty object (Java)

There are two basic approaches for loading an object. One is to create a fully valid object with a constructor, which is what I've done in the examples above. This results in the following loading code.

```
class AbstractMapper...
protected DomainObject load(ResultSet rs) throws SQLException {
Long id = new Long(rs.getLong(1));
if (loadedMap.containsKey(id)) return (DomainObject) loadedMap.get(id);
DomainObject result = doLoad(id, rs);
loadedMap.put(id, result);
return result;
}
```

```
abstract protected DomainObject doLoad(Long id, ResultSet rs) throws
SQLException;
class PersonMapper...
protected DomainObject doLoad(Long id, ResultSet rs) throws SQLException {
String lastNameArg = rs.getString(2);
String firstNameArg = rs.getString(3);
int numDependentsArg = rs.getInt(4);
return new Person(id, lastNameArg, firstNameArg, numDependentsArg);
}
```

The alternative approach is to create an empty object and load it with the setters later.

```
class AbstractMapper...
protected DomainObjectEL load(ResultSet rs) throws SQLException {
Long id = new Long(rs.getLong(1));
if (loadedMap.containsKey(id)) return (DomainObjectEL) loadedMap.get(id);
DomainObjectEL result = createDomainObject();
result.setID(id);
loadedMap.put(id, result);
doLoad(result, rs);
return result;
}
abstract protected DomainObjectEL createDomainObject();
abstract protected void doLoad(DomainObjectEL obj, ResultSet rs) throws
SQLException;
class PersonMapper...
protected DomainObjectEL createDomainObject() {
return new Person();
}

protected void doLoad(DomainObjectEL obj, ResultSet rs) throws SQLException {
Person person = (Person) obj;
person.dbLoadLastName(rs.getString(2));
person.setFirstName(rs.getString(3));
person.setNumberOfDependents(rs.getInt(4));
}
```

You'll notice I'm using a different kind of domain object [Layer Supertype](#) here. This is because I want to control the use of the setters. Let's say that I want the last name of a person to be an immutable field. If this is the case I don't want to change the value of the field once it's loaded.

I can do this by adding a status field to the domain object.

```
class DomainObjectEL...
private int state = LOADING;
private static final int LOADING = 0;
private static final int ACTIVE = 1;
public void beActive() {
state = ACTIVE;
}
```

I can then check the value of this during a load

```
class Person...
public void dbLoadLastName(String lastName) {
assertStateIsLoading();
```

```
this.lastName = lastName;
}
class DomainObjectEL...
void assertStateIsLoading() {
Assert.isTrue(state == LOADING);
}
```

The thing I don't like about this is that we now have a method in the interface that most clients of the person class cannot use. This is an argument for the mapper using reflection to set the field, which will completely bypass the Java's protection mechanisms.

Is the status based guard worth the trouble? I'm not entirely sure. On the one hand it will catch bugs due to people calling update methods at the wrong time. But the question is whether the seriousness of the bugs is worth the cost of the mechanism. At the moment I don't have a strong opinion either way.



---

© Copyright [Martin Fowler](#), all rights reserved



---

# Metadata Mapping

---

*Hold details of object-relational mapping in metadata.*



Much of the code that deals with object-relational mapping describes how fields in the database correspond to the field of in-memory objects. The resulting code tends to be tedious and repetitive to write.

A *Metadata Mapping* allows the developers to define the mappings in a simple tabular form which can then be processed by generic code to carry out the details of reading, inserting, and updating the data.

## How it Works

There are two main parts to implementing a *Metadata Mapping*: deciding how to represent the metadata, and deciding how to use the metadata to carry out the data mapping operations.

### Represent the metadata

The simplest way to represent the metadata is to use objects in your programming language. So to represent the mappings for the person class, you would have a person mapper class. Unlike an explicit [Data Mapper](#) however, this mapper class only has a method to return the map. A generic mapper, which can be a [Layer Supertype](#), then interrogates the map to carry the actual mapping operations.

Using the programming language may lead to a non-ideal syntax, and forces the mapping to be maintainable by programmers; but it avoids introducing some new language or data file into the process. Changes to the mapping have to be dealt with by deploying new classes and compiling these classes. This compilation is only for the mapping classes and shouldn't affect the rest of the system, but it may be a challenge in some environments that make deploying new or changes classes difficult.

The immediate alternative to using classes is to use a data file format. Any format can be used, but at the moment the obvious one to use is XML, since the parsing and editing can use commonly available tools.

There is an overhead, compared to classes, in doing the parsing; but the maps can be shared across and entire server process and each map only has to be parsed once per process - so the overhead turns out to be not that considerable. Even if it's difficult to update code when the maps change, it's easy to update XML mapping files.

Another alternative is to keep the mapping data in the database itself. This keeps it together with the data, so if the database schema changes the mapping information is right there. It will require an database query to get the mapping information, but again since the mapping changes rarely you can usually cache the data for the whole process and thus only need to read the mapping tables when starting or resetting the server process.

## Performing the mapping

As well as representing the metadata, you have to choose how to make use of it. There are two main options you can follow: code generation and reflective programming.

With **code generation** you write a program whose output is the source code of classes that do the mapping based on the information in the metadata. These classes look like hand-written classes, but they are entirely generated automatically during the build process, where they are usually generated just prior to compilation. The resulting mapper classes are deployed with the sever code.

If you use code generation, you should make sure that the code generation step is fully integrated into your build process with whatever build scripts you are using. The generated classes should never be edited by hand, and thus should need to be held in source code control.

A **reflective program** may ask an object for a method named "setFoo", and then run an invoke method upon the setFoo method passing in some argument. By treating methods (and fields) as data the reflective program can read in field and method names from a file and use them to carry out the mapping. I usually counsel against using reflection, partly because it's slow but mainly because it often causes code that's hard to debug. But reflection is actually quite appropriate for database mapping. Since you are reading the names of fields and methods in from a file, you are taking full advantage of the flexibility of reflection. And although reflection is slow, it's much less of an issue because the SQL calls themselves usually dominate the performance issue.

Code generation is a less-dynamic approach since any changes to the mapping require redeploying at least that part of the software. However mapper changes should be pretty rare, and modern environments make it easy to redeploy part of an application - and you can easily keep the generated mapper classes in a separately deployable binary.

Reflective programming often suffers in speed, although the problem here depends very much on the actual environment you are using. Also the reflection is being done in the context of a SQL call, and as such the slower speed of using reflection may not make that much difference considering the slow speed of the remote call.

Both approaches can be a little awkward to debug, the comparison between them depends very much on how used developers are to generated and reflective code.

One of the challenges of metadata is that although a simple metadata scheme often works well 90% of the time, there's often special cases that make life much more tricky. Often to handle these minority

cases you have to add a lot of complexity to metadata. A useful alternative is to handle special cases by overriding the generic code with subclasses where the special code is handwritten. Such special case subclasses would be subclasses of either the generated code, or the reflective routines. Since these special cases are... well... special; it isn't easy to describe define in general how you arrange things to support the overriding. My advice is that you handle them on a case by case basis. As you need the overriding, alter the generated/reflective code to isolate a single method that should be overridden and then override it in your special case.

## When to Use it

*Metadata Mapping* can greatly reduce the amount of work you need to do in handling database mapping. However there is some setup work required to prepare the framework to handle the *Metadata Mapping*. Also while it's often easy to handle most cases with *Metadata Mapping*, you often find exceptions that can really tangle the metadata.

It's no surprise that the commercial object-relational mapping tools use *Metadata Mapping*, since when selling a product it's always worth the effort of producing a sophisticated *Metadata Mapping*.

If you're building your own system, you should evaluate the trade-offs yourself. Compare the amount in adding new mappings using hand-written code and using *Metadata Mapping*. If you use reflection, look into the consequences for performance, sometimes reflection causes performance issues, but often it doesn't. Your own measurements will reveal whether it's an issue for you.

The extra work of hand coding can be greatly reduced by creating a good *Layer Supertype* that handles all the common behavior. That way you should only have a few hook routines to add in for each mapping. But usually *Metadata Mapping* can reduce this further.

*Metadata Mapping* can interfere with some refactoring, particularly if you're using automated tools. If you change the name of a private field, then this can break an application unexpectedly. Even automated refactoring tools won't be able to find the field name hidden in a XML data file of a map. Using code generation is a little easier, since search mechanisms are able to find the usage. However the automated update will get lost when you re-generate the code. So a tool can warn you of a problem, but it's up to you to change the metadata yourself. If you use reflection, you won't even get the warning.

## Example: Using metadata and reflection (Java)

The above examples, like most in this book, use explicit code. While that's the easiest to follow, it does lead to pretty tedious programming - and tedious programming is a sign that something is wrong. You can remove a lot of tedious programming by using metadata. The metadata can be used to generate code or it can be used in reflection. Here's a reflective example.

### Holding the Metadata

The first question to ask about metadata is how it's going to be kept. Here I'm keeping the metadata in two classes. The data map corresponds to the mapping of one class to one table. This is a simple mapping, but it will do for illustration.

```
class DataMap...
private Class domainClass;
private String tableName;
private List columnMaps = new ArrayList();
```

The data map contains a collection of column maps that map columns in the table to fields.

```
class ColumnMap...
private String columnName;
private String fieldName;
private Field field;
private DataMap dataMap;
```

This isn't a terribly sophisticated mapping. I'm just using the default Java type mappings, which means there's no type conversion between fields and columns. I'm also forcing a one to one relationship between tables and classes.

These structures hold the mappings, the next question is how do they get populated? For this example I'm going to populate them with Java code in specific mapper class. While that may seem a little odd, it still buys most of the benefit of metadata - that of avoiding repetitive code.

```
class PersonMapper...
protected void loadDataMap(){
dataMap = new DataMap (Person.class, "people");
dataMap.addColumn ("lastname", "varchar", "lastName");
dataMap.addColumn ("firstname", "varchar", "firstName");
dataMap.addColumn ("number_of_dependents", "int", "numberOfDependents");
}
```

During construction of the column mapper, I build the link to the field. Strictly this is an optimization, so you may not have to do this, but calculating the fields reduces the subsequent accesses by an order of magnitude on my little laptop.

```
class ColumnMap...
public ColumnMap(String columnName, String fieldName, DataMap dataMap) {
this.columnName = columnName;
this.fieldName = fieldName;
this.dataMap = dataMap;
initField();
}
private void initField() {
try {
field = dataMap.getDomainClass().getDeclaredField(getFieldName());
field.setAccessible(true);
} catch (Exception e) {
throw new ApplicationException ("unable to set up field: " + fieldName, e);
}
}
```

It's not much of a challenge to see how I could write a routine to load the map from an XML file, or from a metadata database. Paltry that challenge may be, but I'll decline it and leave it to you.

Now the mappings are defined, I can make use of them. The strength of the metadata approach is that all of the code that actually manipulates things is done in a superclass, so I don't have to write the mapping code that I wrote in the explicit cases.

## Find by ID

I'll begin with the find by id method.

```
class Mapper...
public Object findObject (Long key) {
if (uow.isLoaded(key)) return uow.getObject(key);
String sql = "SELECT" + dataMap.columnList() + " FROM " +
dataMap.getTableName() + " WHERE ID = ?";
PreparedStatement stmt = null;
ResultSet rs = null;
DomainObject result = null;
try {
stmt = DB.prepare(sql);
stmt.setLong(1, key.longValue());
rs = stmt.executeQuery();
rs.next();
result = load(rs);
} catch (Exception e) {throw new ApplicationException (e);
} finally {DB.cleanUp(stmt, rs);
}
return result;
}
private UnitOfWork uow;
protected DataMap dataMap;
class DataMap...
public String columnList() {
StringBuffer result = new StringBuffer(" ID");
for (Iterator it = columnMaps.iterator(); it.hasNext();) {
result.append(",");
ColumnMap columnMap = (ColumnMap)it.next();
result.append(columnMap.getColumnName());
}
return result.toString();
}
public String getTableName() {
return tableName;
}
```

The select is built more dynamically than the other examples, but it's still worth preparing it in a way that allows the database session to cache it properly. If it's an issue the column list could be calculated during construction and cached, since there's no call for updating the columns during the life of the data map. For this example, unlike the others in this pattern, I'm using a [Unit of Work](#) to handle the database session. There's no particular reason to use that with metadata, I'm just illustrating how that would work.

As with other example I've separated the load from the find, so that we can use the same load method from other find methods.

```
class Mapper...
public DomainObject load(ResultSet rs)
throws InstantiationException, IllegalAccessException, SQLException
{
Long key = new Long(rs.getLong("ID"));
if (uow.isLoaded(key)) return uow.getObject(key);
DomainObject result = (DomainObject) dataMap.getDomainClass().newInstance();
result.setID(key);
uow.registerClean(result);
loadFields(rs, result);
return result;
}

private void loadFields(ResultSet rs, DomainObject result) throws
SQLException {
for (Iterator it = dataMap.getColumns(); it.hasNext();) {
ColumnMap columnMap = (ColumnMap)it.next();
Object columnValue = rs.getObject(columnMap.getColumnName());
columnMap.setField(result, columnValue);
}
}

class ColumnMap...
public void setField(Object result, Object columnValue) {
try {
field.set(result, columnValue);
} catch (Exception e) { throw new ApplicationException ("Error in setting " +
fieldName, e);
}
}
```

This is a classic reflected program, we go through each of the column maps and use them to load the field in the domain object. I separated the loadFields method to show how we might extend this for more complicated cases. If we had a class and table where the simple assumptions of the metadata don't hold, I can just override loadFields in a subclass mapper to put in arbitrarily complex code. This is a common technique to use with metadata - providing a hook to override for more wacky cases. It's usually a lot easier to override wacky cases with subclasses than it is to build metadata sophisticated enough to hold a few rare special cases.

Of course, if we have a subclass, we might as well use it to avoid downcasting.

```
class PersonMapper...
public Person find(Long key) {
return (Person) findObject(key);
}
```

## Writing to the database

For updates, I have a single update routine.

```
class Mapper...
public void update (DomainObject obj) {
String sql = "UPDATE " + dataMap.getTableName() + dataMap.updateList() + "
WHERE ID = ?";
PreparedStatement stmt = null;
try {
stmt = DB.prepare(sql);
```

```
int argCount = 1;
for (Iterator it = dataMap.getColumns(); it.hasNext()) {
ColumnMap col = (ColumnMap) it.next();
stmt.setObject(argCount++, col.getValue(obj));
}
stmt.setLong(argCount, obj.getID().longValue());
stmt.executeUpdate();
} catch (SQLException e) {throw new ApplicationException (e);
} finally {DB.cleanUp(stmt);
}
}

class DataMap...
public String updateList() {
StringBuffer result = new StringBuffer(" SET ");
for (Iterator it = columnMaps.iterator(); it.hasNext()) {
ColumnMap columnMap = (ColumnMap)it.next();
result.append(columnMap.getColumnName());
result.append("=?");
}
result.setLength(result.length() - 1);
return result.toString();
}
public Iterator getColumns() {
return Collections.unmodifiableCollection(columnMaps).iterator();
}
```

Inserts use a similar scheme.

```
class Mapper...
public Long insert (DomainObject obj) {
String sql = "INSERT INTO " + dataMap.getTableName() + " VALUES (?) " +
dataMap.insertList() + ")";
PreparedStatement stmt = null;
try {
stmt = DB.prepare(sql);
stmt.setObject(1, obj.getID());
int argCount = 2;
for (Iterator it = dataMap.getColumns(); it.hasNext()) {
ColumnMap col = (ColumnMap) it.next();
stmt.setObject(argCount++, col.getValue(obj));
}
stmt.executeUpdate();
} catch (SQLException e) {throw new ApplicationException (e);
} finally {DB.cleanUp(stmt);
}
return obj.getID();
}

class DataMap...
public String insertList() {
StringBuffer result = new StringBuffer();
for (int i = 0; i < columnMaps.size(); i++) {
result.append(",");
result.append("?");
}
return result.toString();
}
```

## Multi-object finds

To get back multiple objects with a query, there are a couple of routes you can take. If you want a generic query capability on the generic mapper, you can have a query that takes a SQL where clause as an argument.

```
class Mapper...
public Set findObjectsWhere (String whereClause) {
String sql = "SELECT" + dataMap.columnList() + " FROM " +
dataMap.getTableName() + " WHERE " + whereClause;
PreparedStatement stmt = null;
ResultSet rs = null;
Set result = new HashSet();
try {
stmt = DB.prepare(sql);
rs = stmt.executeQuery();
result = loadAll(rs);
} catch (Exception e) {
throw new ApplicationException (e);
} finally {DB.cleanUp(stmt, rs);
}
return result;
}

public Set loadAll(ResultSet rs) throws SQLException, InstantiationException,
IllegalAccessException {
Set result = new HashSet();
while (rs.next()) {
DomainObject newObj = (DomainObject) dataMap.getDomainClass().newInstance();
newObj = load (rs);
result.add(newObj);
}
return result;
}
```

Your alternative is to provide special case finders on the mapper subtypes.

```
class PersonMapper...
public Set findLastNamesLike (String pattern) {
String sql =
"SELECT" + dataMap.columnList() +
" FROM " + dataMap.getTableName() +
" WHERE UPPER(lastName) like UPPER(?)";
PreparedStatement stmt = null;
ResultSet rs = null;
try {
stmt = DB.prepare(sql);
stmt.setString(1, pattern);
rs = stmt.executeQuery();
return loadAll(rs);
} catch (Exception e) {throw new ApplicationException (e);
} finally {DB.cleanUp(stmt, rs);
}
}
```

The great advantage of the metadata approach is that I can now add new tables and classes to my data

mapping and all I have to do is to provide a loadMap method and any specialized finders that I may fancy.



---

© Copyright [Martin Fowler](#), all rights reserved

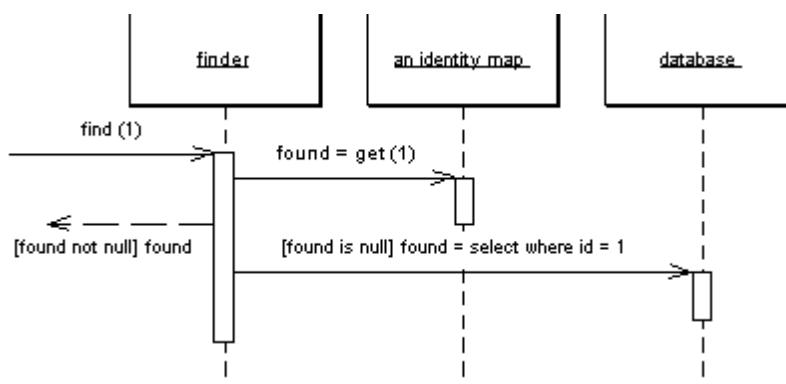
A blue rectangular banner with a white border. On the left side is a 3D-style cube with a rainbow gradient. The letter 'A' is on the top face and 'B' is on the front face. To the right of the cube, the text "ABC Amber CHM Converter Trial version" is displayed in red. Below that, in white text, is "Please register to remove this banner." At the bottom, the URL "http://www.processtext.com/abcchm.html" is shown in blue.

---

# Identity Map

---

*Ensure each object only gets loaded once by keeping every loaded object in a map. Lookup objects using the map when referring to them*



Some old proverb says that a man with two watches never knows what time it is. If two watches are confusing, you can get in an even bigger mess with loading objects from a database. If you aren't careful you can load the data from the same database record into two different objects. If you then update them both you'll have an interesting time writing the changes out to the database correctly.

Related to this is an obvious performance problem. If you load the same data more than once you're incurring an expensive cost in remote calls. So avoiding loading the same data twice doesn't just help correctness, it can also speed up your application.

A *Identity Map* keeps a record of all the objects that have been read from the database in a single business transaction. Whenever you want an object, you check the *Identity Map* first to see if you already have it.

## How it Works

The basic idea behind the identity map is to have a series of maps of objects that have been pulled from the database. In a simple case, with an isomorphic schema, you'll have one map per table in the database. When you load an object from the database, you first check the map. If there's an object in the map that corresponds to the one you're loading, then you return that. If not you go to the database, but as you load the objects you put them into the map for future reference.

There are a number of implementation choices to worry about. As well as these *Identity Maps* interact

with concurrency management, so you should consider [\*Optimistic Offline Lock\*](#).

## Choice of Keys

The first thing to consider is what the key should be for the map. The obvious choice is to use the primary key of the corresponding database table. This works well if the key is a single column and is immutable. Using a surrogate primary key fits in very well with this approach. You can then use the key as the key in the map. The key will usually be a simple data type so the comparison behavior will work nicely.

## Explicit or Generic

You have to choose whether to make the *Identity Map* explicit or generic. An explicit *Identity Map* is accessed with distinct methods for each kind of object you need: such as `findPerson(1)`. A generic map uses a single method for all kinds of objects, with a parameter to indicate which kind of object you need, such as `find("Person", 1)`. The obvious advantage is that you can support a generic map with a generic and reusable object. It's easy to construct a reusable [\*Registry\*](#) that can be used for all kinds of objects and doesn't need updating when you add a new map.

However I prefer an explicit *Identity Map*. For a start this gives you compile time checking in a strongly typed language. But more than that it has all the other advantages of an explicit interface: it's easier to see what maps are available and what they are called. It does mean adding a method each time you add a new map, but that is a small overhead for the virtues of explicitness.

Your type of key affects the choice. You can only use a generic map if all your objects have the same type of key. This is a good argument for encapsulating different kinds of database key behind a single key object - see [\*Identity Field\*](#) for details.

## How many

Here the decision varies between one map per class and one map for the whole session. A single map for the session only works if you have database-unique keys (see the discussion in [\*Identity Field\*](#) for the trade-offs on that.) Once you have one *Identity Map*, the benefit is that you only have one place to go to and no awkward decisions about inheritance.

If you have multiple maps then the obvious route is to have one map per class or per table. This works well if your database schema and object models are the same. If they look different then it's usually easier to base the maps on your objects rather than your tables, as the objects shouldn't really know about the intricacies of the mapping.

Inheritance rears an ugly head here. If you have cars as a subtype of vehicle, do you have one map or separate maps? Keeping them separate can make polymorphic references much more awkward, since any lookup needs to know to look in both maps. As a result I prefer to use a single map for each inheritance tree, but that means that you should also make your keys unique across the inheritance trees, which can be awkward if you use [\*Concrete Table Inheritance\*](#)

An advantage of a single map, is that you don't have to add new identity maps when you add database tables. However if you tie your maps to your [\*Data Mapper\*s](#) (see below) it won't be any extra burden.

## Where to put them

*Identity Maps* need to be put somewhere where they are easy to find. They are also tied to the process context that you're working in. You need to ensure that each session gets its own instance of martin, one that's isolated from any other session's instance. As such you need to put the *Identity Map* on a session specific object. If you are using [\*Unit of Work\*](#) then that's by far the best place for the *Identity Maps* since the [\*Unit of Work\*](#) is the main place for keeping track of data coming in or out of the database.

If you don't have a [\*Unit of Work\*](#) then the best bet is a [\*Registry\*](#) that's tied to the session.

As I've implied here, you usually see a single *Identity Map* for a session, otherwise you need to provide transactional protection for your *Identity Map*, which is more work than any sane developer would try to do. However there are a couple of exceptions. The biggest one is to use an object database as a transactional cache, even if you use a relational database for record data. While I haven't seen any independent performance studies, the possibilities suggest it's worth taking a look at and many people I respect are big fans of this approach to improve performance.

The second exception is for objects that are read-only in all cases. If an object can never be modified, there's no need to worry about it being shared across sessions. In performance intensive systems it can be very beneficial to load in all read-only data once and have available to whole process. In this case you would have your read-only *Identity Maps* held in a process context and your updatable *Identity Maps* at a session context. This would also apply to objects that aren't completely read-only, but are updated so rarely that you don't mind flushing the process wide *Identity Map* and potentially bouncing the server when it happens.

Even if you're inclined to have only one *Identity Map* you could split it into two along read-only and updatable lines. You can avoid clients having to know which is which by providing an interface that checks both maps.

## When to Use it

In general you need to use an *Identity Map* to manage any object that is brought from a database and modified. The key reason you need it is because you don't want a situation where you have two in-memory objects that correspond to a single database record, where you might modify the two records inconsistently and thus confuse the database mapping.

Another value in *Identity Map* is that it acts as a cache for database reads, which means you can avoid going to the database each time you need some data.

One case where you may not need an *Identity Map* is for immutable objects. If you can't change an object, then you don't have to worry about modification anomalies. But *Identity Map* still have advantages. The most important of these is the performance advantages of the cache. Another is that it

helps to avoid problems where people use the wrong form of equality test, a problem that's prevalent in Java where you can't override `==`.

Another case where you don't need a *Identity Map* is for a dependent object. Since their persistence is controlled by their parent, there is no need for a map to maintain identity. However although you don't need a map, you may want to provide one if there's a need to access the object through a database key. In this case the map is merely an index, so it's arguable whether it really counts as a map at all.

## Example: Methods for an *Identity Map* (Java)

For each *Identity Map* we have a map field and accessors.

```
private Map people = new HashMap();

public static void addPerson(Person arg) {
    soleInstance.people.put(arg.getID(), arg);
}

public static Person getPerson(Long key) {
    return (Person) soleInstance.people.get(key);
}

public static Person getPerson(long key) {
    return getPerson(new Long(key));
}
```

One of the annoyances of Java is the fact that long isn't an object, therefore you can't use a long as an index for a map. This isn't as much of a pain as it can be, since we don't actually do any arithmetic on the index. The one place where it is irritating is when you want to retrieve an object with a literal. You hardly ever need to do that in production code, but you often do in test code. So I've included a getting method that takes a long to make testing easier.



---

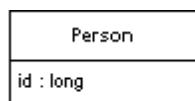
© Copyright [Martin Fowler](#), all rights reserved



# Identity Field

---

*Save a database id field in an object to maintain identity between an in-memory object and a database row.*



Relational databases tell one row from another by using keys, in particular the primary key. However in-memory objects don't need such a key, as the object system ensures the correct identity under the covers (or in C++'s case with raw memory locations). Reading data from a database is all very well, but in order to write data back, you need to tie the in-memory object system to the database.

In essence *Identity Field* is mind-numbingly simple. All you do is store the primary key of the relational database table in fields of the object.

## How it Works

Although the basic notion of *Identity Field* is very simple, there are oodles of complicated issues that come up.

## Choosing your Key

The first issue is what kind of key to choose in your database? Often, of course, this isn't a choice since you are often dealing with an existing database that already has its key structures in place. There's also a lot of discussion and material on this in the database community. But mapping to objects does add some concerns in the choice.

The first question is whether to use meaningful or meaningless keys. A **meaningful key** is something like the US social security number to identify a person. A **meaningless key** is essentially a random number that the database dreams up that's never intended for human use. The danger with a meaningful key is that while in theory they make good keys, in practice they don't. To work at all keys need to be unique. To work well keys need to be immutable. While assigned numbers are often supposed to be unique and immutable, human error often makes them neither. If you mistype my SSN for my wife the resulting record is neither unique nor immutable (assuming you'd like to fix the mistake.) While the database

should detect the uniqueness problem it can only do that after my SSN goes into the system, and of course there's a 50% chance that that won't happen. As a result meaningful keys should be distrusted. For small systems and/or very stable cases you may get away with it, but usually you should take a rare stand on the side of meaninglessness.

The next issue is between simple and compound keys. A **simple key** is only one database field, a **compound key** uses more than one field. The advantage of a compound key is that it's often easier to use when one table makes sense in the context of another. A good example is orders and line items. A compound key of the order number and a sequence number makes a good key for a line item. While compound keys often make sense, there is a lot to be said for the sheer uniformity of simple keys. If you use simple keys everywhere you can use the same code for all key manipulation. Compound keys require special handling in concrete classes. (With code generation this isn't a problem). Compound keys also carry a bit of meaning, so be careful about the uniqueness and particularly the immutability rule with them.

You have to choose the type of the key. The most common operation you'll do with a key is checking equality. So you want a type with a fast equality operation. The other important operation is getting the next key. As a result a long integer type is often the best bet for keys. Strings can also work, but equality checking may be slower and incrementing them is a bit more painful. Your DBA's preferences may well decide the issue.

(Beware about using dates or times in keys. Not just are they meaningful, they also lead to problems with portability and consistency. Dates in particular are vulnerable to this as they are often stored to some fractional second precision, which can easily get out of sync and lead to identity problems.)

You can have keys that are unique to the table, or database wide unique keys. A **table-unique key** ensures that key is unique across the table, which is what you need for a key in any case. A **database-unique key** is a key that is unique across every row in every table in the database. A table-unique key is usually fine, but a database-unique key is often easier to do, and allows you to use a single [\*Identity Map\*](#). Modern values being what they are, it's pretty unlikely you'll run out of numbers for new keys. If you really insist you can reclaim keys from deleted objects with a simple database script that compacts the key space - although running this script will require you to take the application offline. But if you use 64 bit keys (and you might as well) you're unlikely to need it.

If you use table-unique keys, be wary of inheritance. If you are using [\*Concrete Table Inheritance\*](#) or [\*Class Table Inheritance\*](#) life is much easier if you use keys that are unique to the hierarchy rather than unique to each table. I still use the term table-unique, even if it should strictly be something like "inheritance graph unique".

The size of your key may effect performance, particularly with indexes. This is very dependent on both your database system and how many rows you have, but it's worth doing a crude check before you get fixed into your decision.

## Representing the *Identity Field* in an object

The simplest form of *Identity Field* is a simple field which matches the type of the key in the database. So if you use a simple integral key, an integral field will work very nicely.

Compound keys are more problematic. If you have compound keys the best bet is to make a key class.

A generic key class can store a sequence of objects that act as the elements of the key. The key behavior (I have a quota of puns per book to fill) is equality. It's also useful to get parts of the key when you are doing the mapping to the database.

A key object is then easy to use with the association mappings and with [Identity Map](#). As such you might decide to avoid a key object when you are using a [Transaction Script](#) and [Row Data Gateway](#) architecture.

If you use the same basic structure for all keys, you can do all of the key handling in a [Layer Supertype](#). This works particularly well if you can use an integral key field for all database tables. If you start getting any compound keys you should use a key object in the [Layer Supertype](#) instead. Again you can put default behavior which will work for most cases in the [Layer Supertype](#) and extend it for the exceptional cases in the particular subtypes.

You can either have a single key class, which takes a generic list of key objects, or you can have a key class for each domain class with explicit fields for each part of the key. I usually prefer to be explicit, but in this case I'm not sure it buys very much. You end up with lots of small classes that don't do anything interesting. The main benefit is that you can avoid errors due to people putting the elements of the key in the wrong order, but that doesn't seem to be a big problem in practice.

If you are likely to import data between different database instances, then you need to remember that you'll get key collisions unless you come up with some scheme to separate the keys between different databases. You can solve this with some kind of key migration on the imports, but this can easily get very messy.

## Getting a new key

To create an object, you'll need to get a key. This sounds like a simple matter, but in practice it's often quite a problem. You have three basic choices: get the database to auto-generate, use a GUID, or generate your own.

The auto generate route should be the easiest. Each time you insert data for the database the database takes care of generating a unique primary key. You don't have to do anything - it sounds too good to be true and sadly it often is. Not all databases do this the same way, and many of those that do handle it in such a way that causes problems for object-relational mapping.

The most common auto-generation method is that of declaring one field to be an **auto-generated field**. In this case whenever you insert a row, this field is incremented to a new value. The problem with this scheme is that you can't easily determine what value got generated as the key. If you want to insert an order and several line items you need the key of the new order so you can put the value in the foreign key for the line items. You also need this before the transaction commits so you can save everything within the transaction. Sadly databases usually don't give you this information, so you usually can't use this kind of auto generation on any table where you need to insert connected objects.

An alternative approach to auto-generation is a **database counter**, Oracle does this with its sequence. An Oracle sequence works by sending a select statement that references a sequence, the database then returns a SQL record set consisting of the next value in the sequence. You can set a sequence to increment by any integer, which allows you get multiple keys at once. The sequence query is automatically carried out in a separate transaction, so that a accessing the sequence won't lock out other

transactions inserting at the same time. A database counter like this is perfect for our needs, but it's non-standard and not available in all databases.

A **GUID** (Globally Unique IDentifier) is a number generated on one machine that is guaranteed to be unique across all machines in space and time. Often platforms give you the API to generate one. The algorithm is an interesting one involving ethernet card addresses, time of the day in nanoseconds, chip id numbers, and probably the number of hairs on your left wrist. All that matters is that the resulting number is completely unique and thus a safe key. The only disadvantage is that the resulting key string is big, and that can be an equally big problem. There's always times when someone needs to type in a key to window or SQL expression, and long keys are hard both to type and to read. Large keys may also lead to performance problems, particularly with indexes.

The last option is rolling your own. A simple staple for small systems is to use a **table scan** using the SQL max function to find the largest key in the table, then add one to use it. Sadly this read locks the entire table while you are doing it. As a result it works fine if inserts are rare, but if you have inserts running concurrently with updates on the same table your performance will be toasted. You also have to ensure you have complete isolation between transactions otherwise you can end up with multiple transactions getting the same id value.

A better approach is to use a separate **key table**. This table is typically a table with two columns: a name and next available value. If you use database-unique keys, then you'll have just one row in this table. If you use table-unique keys then you'll have one row for each table in the database. To use the key table, all you need to do is read that one row note the number, increment it and write it back to the row. You can grab many keys at a time by adding a suitable number when you update the key table. This cuts down on expensive database calls as well as reducing contention on the key table.

If you use a key table, it's a good idea to design it so the access to the key table is done in a separate transaction to the one that updates the table you are inserting into. Say I'm inserting an order into the orders table. To do this I'll need to lock the orders row on the key table with a write lock (since I'm updating). That lock will last for the entire transaction that I'm in, locking out anyone else who wants a key. For table-unique keys, this means anyone inserting into the orders table. For database-unique keys this means anyone inserting anywhere.

By putting the access to the key table in a separate transaction, you only lock the row for that, much shorter, transaction. The downside is that if you rollback on your insert to the orders, then the key you got from the key table access is lost to everyone. Fortunately numbers are cheap, so that's not a big issue. Using a separate transaction also allows you to get the id as soon as you create the in-memory object, which is often some before you open the transaction to commit the business transaction.

Using a key table affects the choice on whether to use database-unique or table-unique keys. If you use a table-unique key, you have to add a row to the key table every time you add a table to the database. This is more effort, but it reduces contention on the row. If you keep your key table accesses in a different transaction, then contention is not so much of a problem, especially if you get multiple keys in a single call. But if you can't force the key table update to be in a separate transaction, that's a strong reason against using database-unique keys.

It's good to separate the code for getting a new key into its own class, as that makes it easier to build a [Service Stub](#) for testing purposes.

## When to Use it

You need to use *Identity Field* when there is a mapping between objects in memory and rows in a database. The usual cases for this is where you are using [Domain Model](#) or a [Row Data Gateway](#). Conversely you don't need this if you are using [Transaction Script](#), [Table Module](#), or a [Table Data Gateway](#)

A small object with value semantics, such as a money or date range object, will not have its own table. It's better to use [Embedded Value](#). For a complex graph of objects that doesn't need to be queried within the relational database [Serialized LOB](#) is usually easier to write and gives faster performance.

One alternative to *Identity Field* is to use an [Identity Map](#) to maintain the correspondence. This can be used for systems where you don't want to store an *Identity Field* in the in-memory object. The [Identity Map](#) needs to look up both ways: give me a key for an object or an object for a key. I don't see this done so often because usually it's easier to store the key in the object.

## Example: Integral Key (C#)

The simplest form of *Identity Field* is a integral field in the database that maps to an integral field in an in-memory object.

```
class DomainObject...
public const long PLACEHOLDER_ID = -1;
public long Id = PLACEHOLDER_ID;
public Boolean isNew() {return Id == PLACEHOLDER_ID;}
```

An object that's been created in memory but not saved to the database will not have a value for its key. For a .NET value object, this is a problem since .NET values cannot be null. Hence the placeholder value.

The key is becomes important in two places: finding and insertion. For finding you need to form a query using a key in a where clause. In .NET you may load many rows into a data set and then select a particular one with a find operation.

```
class CricketerMapper...
public Cricketer Find(long id) {
return (Cricketer) AbstractFind(id);
}
class Mapper...
protected DomainObject AbstractFind(long id) {
DataRow row = FindRow(id);
return (row == null) ? null : Find(row);
}
protected DataRow FindRow(long id) {
String filter = String.Format("id = {0}", id);
DataRow[] results = table.Select(filter);
return (results.Length == 0) ? null : results[0];
}
public DomainObject Find (DataRow row) {
DomainObject result = CreateDomainObject();
Load(result, row);
```

```
return result;
}
abstract protected DomainObject CreateDomainObject();
```

Most of this behavior can live on the [Layer Supertype](#), but you'll often need to define the find on the concrete class just to encapsulate the downcast. Naturally you can avoid this in a language that doesn't use compile-time typing.

With a simple integral *Identity Field* the insertion behavior can also be held at the [Layer Supertype](#).

```
class Mapper...
public virtual long Insert (DomainObject arg) {
    DataRow row = table.NewRow();
    arg.Id = GetNextID();
    row["id"] = arg.Id;
    Save (arg, row);
    table.Rows.Add(row);
    return arg.Id;
}
```

Essentially insertion involves creating the new row and using the next key for that new row. Once you have the new row you can save the in-memory object's data to this new row.

## Example: Using a Key Table (Java)

by Matt Foemmel and Martin Fowler

If your database supports a database counter and you're not worried about being dependent on database specific SQL, then you should use the counter. Even if you are worried about being dependent on a database you should still consider it - as long as your key generation code is nicely encapsulated you can always change it to a portable algorithm later. You could even use a [strategy](#) to use counters where you have them and roll your own when you don't.

But for the moment lets assume we have to this the hard way. The first thing we need is a key table in the database.

```
CREATE TABLE keys (name varchar primary key, nextID int)
INSERT INTO keys VALUES ('orders', 1)
```

The key table contains one row for each counter that's in the database. In this case I've initialized the key to 1. If you are pre-loading data in the database you'll need to set the counter to a suitable number. If you want database-unique keys you'll only need the one row. If you have table-unique keys then you'll have one row per table.

You can wrap all of your key generation code into its own class, that way it's easier to use it more widely around one or more applications. It also makes it easier to put key reservation into its own transaction.

We construct a key generator with its own database connection, together with information on how many keys to take from the database at once.

```
class KeyGenerator...
private Connection conn;
private String keyName;
private long nextId;
private long maxId;
private int incrementBy;

public KeyGenerator(Connection conn, String keyName, int incrementBy) {
    this.conn = conn;
    this.keyName = keyName;
    this.incrementBy = incrementBy;

    nextId = maxId = 0;

    try {
        conn.setAutoCommit(false);
    } catch(SQLException exc) {
        throw new ApplicationException("Unable to turn off autocommit", exc);
    }
}
```

We need to ensure that no auto-commit is going on since we absolutely have to have the select and update operating in one transaction.

```
class KeyGenerator...
public synchronized Long nextKey() {
    if (nextId == maxId) {
        reserveIds();
    }

    return new Long(nextId++);
}
```

When we ask for a new key, the generator first looks to see if it has one cached, rather than going to the database.

If the generator hasn't got one cached, then it needs to go to the database.

```
class KeyGenerator...
private void reserveIds() {
    PreparedStatement stmt = null;
    ResultSet rs = null;

    long newNextId;

    try {
        stmt = conn.prepareStatement("SELECT nextID FROM keys WHERE name = ? FOR
UPDATE");
        stmt.setString(1, keyName);
        rs = stmt.executeQuery();
        rs.next();
        newNextId = rs.getLong(1);
    }
    catch (SQLException exc) {
        throw new ApplicationException("Unable to generate ids", exc);
    }
}
```

```
finally {
DB.cleanUp(stmt, rs);
}

long newMaxId = newNextId + incrementBy;

stmt = null;

try {
stmt = conn.prepareStatement("UPDATE keys SET nextID = ? WHERE name = ? ");
stmt.setLong(1, newMaxId);
stmt.setString(2, keyName);
stmt.executeUpdate();

conn.commit();

nextId = newNextId;
maxId = newMaxId;
}
catch (SQLException exc) {
throw new ApplicationException("Unable to generate ids", exc);
}
finally {
DB.cleanUp(stmt);
}
}
```

In this case we use SELECT... FOR UPDATE to tell the database to hold a write lock on the keys table. This is an Oracle specific statement, so your mileage will vary if you're using something else. If you can't write lock on the select you run the risk of the transaction failing should another one get in there before you. In this case, however you can pretty safely just rerun reserveIds until you get a pristine set of keys.

## Example: Using a compound key (Java)

Using a simple integral key is a good simple solution, but often you need to use other types or compound keys.

### A Key Class

As soon as you need to use something else it's worth using putting together a key class. A key class needs to be able to store multiple elements of the key and to be able to tell if two keys are equal.

```
class Key...
private Object[] fields;

public boolean equals(Object obj) {
if (!(obj instanceof Key)) return false;
Key otherKey = (Key) obj;
if (this.fields.length != otherKey.fields.length) return false;
for (int i = 0; i < fields.length; i++)
if (!this.fields[i].equals(otherKey.fields[i])) return false;
```

```
return true;
}
```

The most elemental way to create a key is with an array parameter.

```
class Key...
public Key(Object[] fields) {
checkKeyNotNull(fields);
this.fields = fields;
}

private void checkKeyNotNull(Object[] fields) {
if (fields == null) throw new IllegalArgumentException("Cannot have a null
key");
for (int i = 0; i < fields.length; i++)
if (fields[i] == null)
throw new IllegalArgumentException("Cannot have a null element of key");
}
```

If you find there are common cases when you create keys with certain elements, you can add convenience constructors. The exact convenience constructors will depend on what kinds of keys your application has.

```
class Key...
public Key(long arg) {
this.fields = new Object[1];
this.fields[0] = new Long(arg);
}

public Key(Object field) {
if (field == null) throw new IllegalArgumentException("Cannot have a null
key");
this.fields = new Object[1];
this.fields[0] = field;
}

public Key(Object arg1, Object arg2) {
this.fields = new Object[2];
this.fields[0] = arg1;
this.fields[1] = arg2;
checkKeyNotNull(fields);
}
```

Don't be afraid to add these convenience methods, after all convenience is important to everyone using the keys.

Similarly you can add accessor functions to get parts of key. The application will need to do this for the mappings.

```
class Key...
public Object value(int i) {
return fields[i];
}

public Object value() {
```

```
checkSingleKey();
return fields[0];
}

private void checkSingleKey() {
if (fields.length > 1)
throw new IllegalStateException("Cannot take value on composite key");
}

public long longValue() {
checkSingleKey();
return longValue(0);
}

public long longValue(int i) {
if (!(fields[i] instanceof Long))
throw new IllegalStateException("Cannot take longValue on non long key");
return ((Long) fields[i]).longValue();
}
```

In this example we'll map to an order and line item tables. The order table has a simple integral primary key, the line item primary key is a compound of the order's primary key and a sequence number.

```
CREATE TABLE orders (ID int primary key, customer  varchar)
CREATE TABLE line_items (orderID int, seq int, amount int, product varchar,
primary key (orderID, seq))
```

The *Layer Supertype* for domain objects needs to take have a key field.

```
class DomainObjectWithKey...
private Key key;

protected DomainObjectWithKey(Key ID) {
this.key = ID;
}

protected DomainObjectWithKey() {
}

public Key getKey() {
return key;
}

public void setKey(Key key) {
this.key = key;
}
```

## Reading

As with other examples in this book I've split the behavior into find (which gets to the right row in the database) and load (which loads data from that row into the domain object). Both responsibilities are affected by the use of a key object.

The primary difference between them and the other examples in this book (which use simple integral

keys) is that we have to factor out certain pieces of behavior that are overridden by those classes that have more complex keys. For this example I'm assuming that most tables use simple integral keys but some use something else. So as a result I've made the default case the simple integral and embedded the behavior for that in the mapper [Layer Supertype](#). The order class is one of those simple cases. Here's the code for the find behavior.

```
class OrderMapper...
public Order find(Key key) {
return (Order) abstractFind(key);
}

public Order find(Long id) {
return find(new Key(id));
}

protected String findStatementString() {
return "SELECT id, customer from orders WHERE id = ?";
}

class AbstractMapper...
abstract protected String findStatementString();
protected Map loadedMap = new HashMap();

public DomainObjectWithKey abstractFind(Key key) {
DomainObjectWithKey result = (DomainObjectWithKey) loadedMap.get(key);
if (result != null) return result;
ResultSet rs = null;
PreparedStatement findStatement = null;
try {
findStatement = DB.prepare(findStatementString());
loadFindStatement(key, findStatement);
rs = findStatement.executeQuery();
rs.next();
if (rs.isAfterLast()) return null;
result = load(rs);
return result;
} catch (SQLException e) {
throw new ApplicationException(e);
} finally {
DB.cleanUp(findStatement, rs);
}
}

// hook method for keys that aren't simple integral
protected void loadFindStatement(Key key, PreparedStatement finder) throws
SQLException {
finder.setLong(1, key.longValue());
}
```

For the find behavior I've extracted out the building of the find statement, since that will require different parameters to be passed into the prepared statement. The line item is a compound key, so needs to override that method.

```
class LineItemMapper...
public LineItem find(long orderID, long seq) {
Key key = new Key(new Long(orderID), new Long(seq));
return (LineItem) abstractFind(key);
}
```

```

public LineItem find(Key key) {
    return (LineItem) abstractFind(key);
}

protected String findStatementString() {
    return
    "SELECT orderID, seq, amount, product " +
    "FROM line_items " +
    "WHERE (orderID = ?) AND (seq = ?)";
}

// hook methods overriden for the composite key
protected void loadFindStatement(Key key, PreparedStatement finder) throws
SQLException {
    finder.setLong(1, orderID(key));
    finder.setLong(2, sequenceNumber(key));
}

//helpers to extract appropriate values from line item's key
private static long orderID(Key key) {
    return key.longValue(0);
}

private static long sequenceNumber(Key key) {
    return key.longValue(1);
}

```

As well as defining the interface for the find methods and providing a SQL string for the find statement, the subclass needs to override the hook method to allow two parameters to go into the SQL statement. I've also written two helper methods to extract the parts of the key information. This makes for clearer code than just putting explicit accessors with numeric indices from the key. Such literal indices are a bad smell.

The load behavior shows a similar structure, default behavior in the [Layer Supertype](#) for simple integral keys, overridden for the more complex cases. In this case the order's load behavior looks like this.

```

class AbstractMapper...
protected DomainObjectWithKey load(ResultSet rs) throws SQLException {
    Key key = createKey(rs);
    if (loadedMap.containsKey(key)) return (DomainObjectWithKey)
        loadedMap.get(key);
    DomainObjectWithKey result = doLoad(key, rs);
    loadedMap.put(key, result);
    return result;
}

abstract protected DomainObjectWithKey doLoad(Key id, ResultSet rs) throws
SQLException;

// hook method for keys that aren't simple integral
protected Key createKey(ResultSet rs) throws SQLException {
    return new Key(rs.getLong(1));
}
class OrderMapper...
protected DomainObjectWithKey doLoad(Key key, ResultSet rs) throws
SQLException {
    String customer = rs.getString("customer");
}

```

```
Order result = new Order(key, customer);
MapperRegistry.lineItem().loadAllLineItemsFor(result);
return result;
}
```

The line item needs to override the hook to create a key based on two fields.

```
class LineItemMapper...
protected DomainObjectWithKey doLoad(Key key, ResultSet rs) throws
SQLException {
Order theOrder = MapperRegistry.order().find(orderID(key));
return doLoad(key, rs, theOrder);
}

protected DomainObjectWithKey doLoad(Key key, ResultSet rs, Order order)
throws SQLException
{
LineItem result;
int amount = rs.getInt("amount");
String product = rs.getString("product");
result = new LineItem(key, amount, product);
order.addLineItem(result); //links to the order
return result;
}

//overrides the default case
protected Key createKey(ResultSet rs) throws SQLException {
Key key = new Key(new Long(rs.getLong("orderID")), new
Long(rs.getLong("seq")));
return key;
}
```

The line item also has a separate load method for use when loading all the lines for the order.

```
class LineItemMapper...
public void loadAllLineItemsFor(Order arg) {
PreparedStatement stmt = null;
ResultSet rs = null;
try {
stmt = DB.prepare(findForOrderString);
stmt.setLong(1, arg.getKey().longValue());
rs = stmt.executeQuery();
while (rs.next())
load(rs, arg);
} catch (SQLException e) {
throw new ApplicationException(e);
} finally { DB.cleanUp(stmt, rs);
}
}

private final static String findForOrderString =
"SELECT orderID, seq, amount, product " +
"FROM line_items " +
"WHERE orderID = ?";

protected DomainObjectWithKey load(ResultSet rs, Order order) throws
SQLException {
Key key = createKey(rs);
if (loadedMap.containsKey(key)) return (DomainObjectWithKey)
```

```
loadedMap.get(key);
DomainObjectWithKey result = doLoad(key, rs, order);
loadedMap.put(key, result);
return result;
}
```

You need the special handling because the order object isn't put into the order's *Identity Map* until after it's been created. Creating an empty object and inserting it directly into the *Identity Field* would avoid the need for this, see [\[here\]](#)

## Insertion

In a similar way to reading data the insertion behavior has a default action for a simple integral key and the hooks to override this for more interesting keys.

In the mapper supertype I provide an operation to act as the interface together with a template method to do the work of the insertion.

```
class AbstractMapper...
public Key insert(DomainObjectWithKey subject) {
try {
return performInsert(subject, findNextDatabaseKeyObject());
} catch (SQLException e) {
throw new ApplicationException(e);
}
}

protected Key performInsert(DomainObjectWithKey subject, Key key) throws
SQLException {
subject.setKey(key);
PreparedStatement stmt = DB.prepare(insertStatementString());
insertKey(subject, stmt);
insertData(subject, stmt);
stmt.execute();
loadedMap.put(subject.getKey(), subject);
return subject.getKey();
}
abstract protected String insertStatementString();
class OrderMapper...
protected String insertStatementString() {
return "INSERT INTO orders VALUES(?,?)";
}
```

The data from the object goes into the insert statement through two methods separating the data from the key from the basic data of the object. I do this because I can provide a default implementation for the key, which will work for any class, like order, that uses the default simple integral key.

```
class AbstractMapper...
protected void insertKey(DomainObjectWithKey subject, PreparedStatement stmt)
throws SQLException
{
stmt.setLong(1, subject.getKey().longValue());
}
```

The rest of the data for the insert statement is dependent on the particular subclass, so this behavior is abstract on the superclass

```
class AbstractMapper...
abstract protected void insertData(DomainObjectWithKey subject,
PreparedStatement stmt)
throws SQLException;
class OrderMapper...
protected void insertData(DomainObjectWithKey abstractSubject,
PreparedStatement stmt) {
try {
Order subject = (Order) abstractSubject;
stmt.setString(2, subject.getCustomer());
} catch (SQLException e) {
throw new ApplicationException(e);
}
}
```

The line item overrides both of these methods. Line item pulls two values out for key.

```
class LineItemMapper...
protected String insertStatementString() {
return "INSERT INTO line_items VALUES (?, ?, ?, ?, ?)";
}

protected void insertKey(DomainObjectWithKey subject, PreparedStatement stmt)
throws SQLException
{
stmt.setLong(1, orderID(subject.getKey()));
stmt.setLong(2, sequenceNumber(subject.getKey()));
}
```

And provides its own implementation of the save statement for the rest of the data

```
class LineItemMapper...
protected void insertData(DomainObjectWithKey subject, PreparedStatement stmt)
throws SQLException
{
LineItem item = (LineItem) subject;
stmt.setInt(3, item.getAmount());
stmt.setString(4, item.getProduct());
}
```

Separating out the loading of the data into the insert statement like this is only worthwhile if most classes use the same single field for the key. If there is more variation for handling the key, then having just one command to insert the information is probably easier.

Coming up with the next database key is also something that I can separate into a default and an overridden case. For the default case I can use the key table scheme that I talked about earlier on. For the line item, we run into a problem. The line item's key uses the key of the order for the line item as part of its composite key. However there is no reference from the line item class to the order class. So it's impossible to tell a line item to insert itself into the database without providing the correct order as well. This leads to the always messy approach of implementing the superclass method with an unsupported

operation exception.

```
class LineItemMapper...
public Key insert(DomainObjectWithKey subject) {
throw new UnsupportedOperationException
("Must supply an order when inserting a line item");
}

public Key insert(LineItem item, Order order) {
try {
Key key = new Key(order.getKey().value(), getNextSequenceNumber(order));
return performInsert(item, key);
} catch (SQLException e) {
throw new ApplicationException(e);
}
}
```

Of course we can avoid this by having a back link from the line item to the order, effectively turning the association between the two into a bidirectional association. I've chosen not to do it here to illustrate what to do when you don't have that link.

By supplying the order, it's easy to get the order's part of the key. The next problem is to come up with a sequence number for the order line. To find that out we need to find out what the next available sequence number is for an order. We can do this either with a max query in SQL, or by looking at the line items on the order in memory. For this example I'll do the latter.

```
class LineItemMapper...
private Long getNextSequenceNumber(Order order) {
loadAllLineItemsFor(order);
Iterator it = order.getItems().iterator();
LineItem candidate = (LineItem) it.next();
while (it.hasNext()) {
LineItem thisItem = (LineItem) it.next();
if (thisItem.getKey() == null) continue;
if (sequenceNumber(thisItem) > sequenceNumber(candidate)) candidate =
thisItem;
}
return new Long(sequenceNumber(candidate) + 1);
}

private static long sequenceNumber(LineItem li) {
return sequenceNumber(li.getKey());
}
//comparator doesn't work well here due to unsaved null keys
protected String keyTableRow() {
throw new UnsupportedOperationException();
}
```

This algorithm would be much nicer if I used the Collections.max method, but since we may (and indeed will) have at least one null key the max method would fail.

## Updates and Deletes

After all of that, updates and deletes are mostly harmless. Again we have an abstract method for the

assumed usual case, and an override for the special cases.

Updates work like this

```
class AbstractMapper...
public void update(DomainObjectWithKey subject) {
PreparedStatement stmt = null;
try {
stmt = DB.prepare(updateStatementString());
loadUpdateStatement(subject, stmt);
stmt.execute();
} catch (SQLException e) {
throw new ApplicationException(e);
} finally {
DB.cleanUp(stmt);
}
}

abstract protected String updateStatementString();
abstract protected void loadUpdateStatement(DomainObjectWithKey subject,
PreparedStatement stmt)
throws SQLException;
class OrderMapper...
protected void loadUpdateStatement(DomainObjectWithKey subject,
PreparedStatement stmt)
throws SQLException
{
Order order = (Order) subject;
stmt.setString(1, order.getCustomer());
stmt.setLong(2, order.getKey().longValue());
}

protected String updateStatementString() {
return "UPDATE orders SET customer = ? WHERE id = ?";
}
class LineItemMapper...
protected String updateStatementString() {
return
"UPDATE line_items " +
"SET amount = ?, product = ? " +
"WHERE orderId = ? AND seq = ?";
}

protected void loadUpdateStatement(DomainObjectWithKey subject,
PreparedStatement stmt)
throws SQLException
{
stmt.setLong(3, orderID(subject.getKey()));
stmt.setLong(4, sequenceNumber(subject.getKey()));
LineItem li = (LineItem) subject;
stmt.setInt(1, li.getAmount());
stmt.setString(2, li.getProduct());
}
```

Deletes work like this

```
class AbstractMapper...
public void delete(DomainObjectWithKey subject) {
PreparedStatement stmt = null;
try {
```

```
stmt = DB.prepareStatement(deleteStatementString());
loadDeleteStatement(subject, stmt);
stmt.execute();
} catch (SQLException e) {
throw new ApplicationException(e);
} finally {
DB.cleanUp(stmt);
}
}

abstract protected String deleteStatementString();

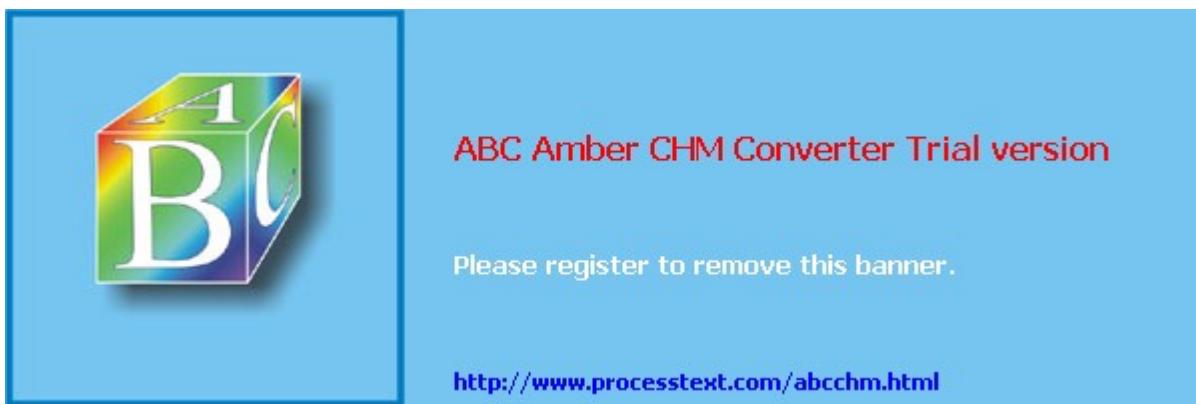
protected void loadDeleteStatement(DomainObjectWithKey subject,
PreparedStatement stmt)
throws SQLException
{
stmt.setLong(1, subject.getKey().longValue());
}
class OrderMapper...
protected String deleteStatementString() {
return "DELETE FROM orders WHERE id = ?";
}
class LineItemMapper...
protected String deleteStatementString() {
return "DELETE FROM line_items WHERE orderid = ? AND seq = ?";
}

protected void loadDeleteStatement(DomainObjectWithKey subject,
PreparedStatement stmt)
throws SQLException
{
stmt.setLong(1, orderID(subject.getKey()));
stmt.setLong(2, sequenceNumber(subject.getKey()));
}
```



---

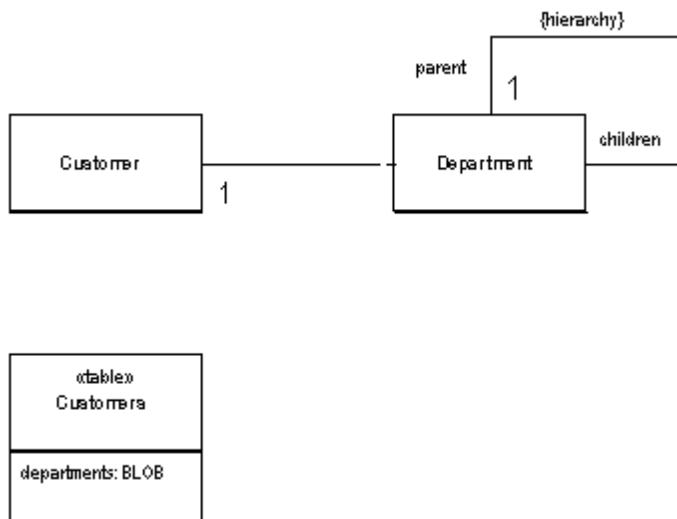
© Copyright [Martin Fowler](#), all rights reserved



# Serialized LOB

---

*Save a graph of objects as by serializing them into a single large object (LOB) and store the LOB in a database field.*



Object models often contain complicated graphs of small objects. Much of the information in these structures is not the objects but the links between the objects. Consider storing the organization hierarchy for all your customers. An object model would quite naturally show the composition pattern to represent organizational hierarchies. You can easily add methods that allow you to get ancestors, siblings, descendants and other common relationships

Putting all this into a relational schema, however, gets more awkward. The basic schema is simple, an organization table with a parent foreign key. However manipulation of the schema requires many joins, which is both slow and awkward.

Objects don't have to be persisted as table rows related to each other. Another form of persistence is serialization, where a whole graph of objects is written out as a single field in a table, this field then becomes a form of [memento](#).

## How it Works

There are two ways you can do the serialization: a binary (**BLOB**) or as textual characters (**CLOB**).

The BLOB is often the simplest to create since many platforms include the ability to automatically serialize an object graph. Saving the graph is then simply a matter of applying the serialization into a buffer and saving that buffer into the relevant field.

The advantages of the BLOB is that it's simple to program (if your platform supports it), and it uses the minimum of space. Your database, of course, must support a binary data type to support this. Also you have to live with the fact that you can't reconstruct the graph without the object, so the field is utterly impenetrable to casual viewing. The most serious problem to concern with, however, is versioning. If you change the department class, you may not be able to read all the previous serializations of that class; since data can live in the database for a long time, this is a serious problem.

The alternative is a CLOB. In this case you serialize the department graph into a text string which carries all the information you need. The text string can be read easily by a human viewing the row - which helps in casual browsing of the database. However the text approach will usually need more space and you may need to create your own parser for the textual format you use. It's also likely to be slower than a binary serialization.

Much of these disadvantages for CLOBs can be overcome by using XML. XML parsers are commonly available, so you don't have to write your own. Furthermore since XML is a widely supported standard, you can take advantage of tools that become available to do further manipulations. The disadvantage that XML does not help with is the matter of space. Indeed it makes the space issue much worse since XML is a very verbose format. One way to deal with that is to use a zipped XML format as your BLOB. That loses the direct human readability, but is an option if space really is an issue.

When you're using *Serialized LOB* beware of identity problems. Say you wanted to use *Serialized LOB* for the customer details on an order. For this don't put the customer LOB in the order table, otherwise the customer data will be copied on every order, which makes updates a problem. (This is actually a good thing, however, if you want to store a snapshot of the customer data as it was at the placing of the order - avoiding temporal relationships.) If you want your customer data to be updated for each order in the classical relational sense, you need to put the customer LOB in a customer table so many orders can link to it. There's nothing wrong with a table that just has an id and a single LOB field for its data.

In general you need to be careful of duplicating data when using *Serialized LOB*. Often it's not a whole *Serialized LOB* that gets duplicated, but part of a *Serialized LOB* that overlaps with another one. The thing to do is to pay careful attention to the data that's stored in the *Serialized LOB* and be sure that this data cannot be reached from anywhere but a single object that acts as the owner of the *Serialized LOB*.

## When to Use it

*Serialized LOB* isn't considered as often as it might be. Using XML makes it much more attractive since it yields an easy to implement textual approach.

The main disadvantage of the LOB is that you can't query the structure using SQL. SQL extensions are appearing to get at XML data within a field, but that's still not the same (or portable).

*Serialized LOB* works best when you can chop a piece of the object model out and represent the LOB. As such think of a LOB of a way to take a bunch of objects that aren't likely to be queried from any SQL route outside the application. This graph can then be hooked into the SQL schema.

*Serialized LOB* works poorly when you have references from objects outside the LOB into objects that are buried in the LOB. To handle this you have to come up with some form of referencing scheme that can support references to objects inside a LOB. While this is by no means impossible, it is awkward. Awkward enough to usually be not worth doing. Again XML, or rather XPATH, reduces this awkwardness somewhat.

If you are using a separate database for reporting and all other SQL goes against that database, then you can transform the LOB into a suitable table structure. The fact that a reporting database is usually denormalized means that structures that are suitable for *Serialized LOB* are often also suitable for a separate reporting database.

## Example: Serializing a department hierarchy in XML (Java)

For this example we'll take the notion of customers and departments and show how you might serialize all the departments into an XML CLOB. The object model of the sketch turns into the following class structures

```
class Customer...
private String name;
private List departments = new ArrayList();
class Department...
private String name;
private List subsidiaries = new ArrayList();
```

The database for this has only one table.

```
create table customers (ID int primary key, name varchar, departments varchar)
```

We'll treat the customer as an *Active Record* and illustrate writing the data with the insert behavior.

```
class Customer...
public Long insert() {
PreparedStatement insertStatement = null;
try {
insertStatement = DB.prepare(insertStatementString);
setID(findNextDatabaseId());
insertStatement.setInt(1, getID().intValue());
insertStatement.setString(2, name);
insertStatement.setString(3, XmlStringer.write(departmentsToXmlElement()));
insertStatement.execute();
Registry.addCustomer(this);
return getID();
} catch (SQLException e) {
throw new ApplicationException(e);
} finally {DB.cleanUp(insertStatement);
}
}

public Element departmentsToXmlElement() {
```

```
Element root = new Element("departmentList");
Iterator i = departments.iterator();
while (i.hasNext()) {
    Department dep = (Department) i.next();
    root.addContent(dep.toXmlElement());
}
return root;
}

class Department...
Element toXmlElement() {
    Element root = new Element("department");
    root.setAttribute("name", name);
    Iterator i = subsidiaries.iterator();
    while (i.hasNext()) {
        Department dep = (Department) i.next();
        root.addContent(dep.toXmlElement());
    }
    return root;
}
```

The customer has a method for serializing its departments field into a single XML DOM. Each department also has a method for serializing itself (and its subsidiaries recursively) into a DOM. The insert method then just takes the DOM of the departments, converts it into a string (via a utility class) and puts it into the database. We aren't particularly concerned with the structure of the string. It's human readable, but we aren't going to look at it on a regular basis.

```
<?xml version="1.0" encoding="UTF-8"?>
<departmentList><department name="US"><department name="New
England"><department name="Boston" />
<department name="Vermont" /></department><department name="California" />
<department name="Mid-West" /></department><department name="Europe"
/></departmentList>
```

Reading back is a fairly simple reversal of the process.

```
class Customer...
public static Customer load(ResultSet rs) throws SQLException {
    Long id = new Long(rs.getLong("id"));
    Customer result = (Customer) Registry.getCustomer(id);
    if (result != null) return result;
    String name = rs.getString("name");
    String departmentLob = rs.getString("departments");
    result = new Customer(name);
    result.readDepartments(XmlStringer.read(departmentLob));
    return result;
}

void readDepartments(Element source) {
    List result = new ArrayList();
    Iterator it = source.getChildren("department").iterator();
    while (it.hasNext())
        addDepartment(Department.readXml((Element) it.next()));
}

class Department...
static Department readXml(Element source) {
```

```
String name = source.getAttributeValue("name");
Department result = new Department(name);
Iterator it = source.getChildren("department").iterator();
while (it.hasNext())
result.addSubsidiary(readXml((Element) it.next()));
return result;
}
```

The load code is obviously a mirror image of the insert code. The department knows how to create itself (and its subsidiaries) from an XML element, and the customer knows how to create the list of departments from an XML element. The load method uses a utility class to turn the string from the database into a utility element.

An obvious danger here is that someone may try to edit the XML by hand in the database and mess up the XML making it unreadable by the load routine. More sophisticated tools that would support adding a DTD or XML Schema to a field as validation would obviously help with that.



---

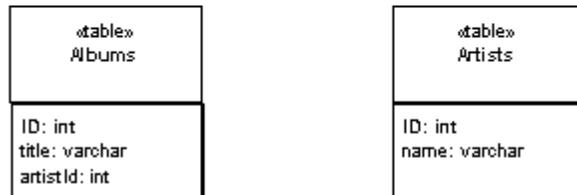
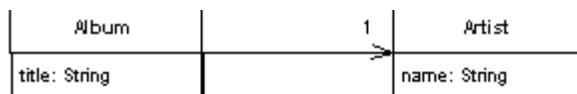
© Copyright [Martin Fowler](#), all rights reserved



# Foreign Key Mapping

---

*Map an association between objects to a foreign key reference between tables*



Objects can refer to each other directly by object references. Even the simplest object oriented system will contain a bevy of objects connected to each other in all sorts of interesting ways. To save these objects to a database, it's vital to save these references. However since the data in these references is specific to the specific instance of the running program, you can't just save raw data values.

This is further complicated by the fact that objects can easily hold collections of references to other objects. Such a structure violates the first normal form of relational databases.

A *Foreign Key Mapping* maps an object reference to a foreign key in the database.

## How it Works

The obvious key to this problem is [Identity Field](#). Each object contains the database key from the appropriate database table. If two objects are linked together with an association, this association can be replaced by a foreign key in the database. In the simple form of this, when you save an album to the database, you save the ID of the artist that the album is linked to in the album record, as in Figure 1.



Figure 1: Mapping a simple reference to a foreign key

This is the simple case. A more complicated case turns up when you have a collection of objects. You can't save a collection of objects in the database, so you have to reverse the direction of the reference. So if you have a collection of tracks in the album, you use put a the foreign key of the album in the track record as in Figure 2.

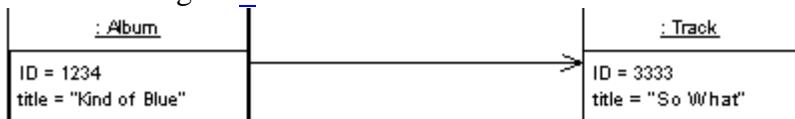


Figure 2: Mapping a simple reference to a foreign key

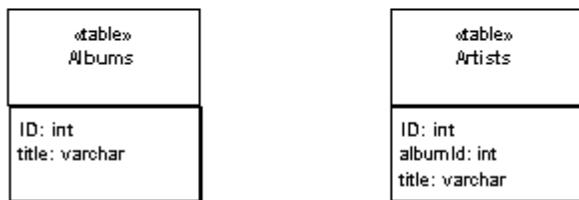
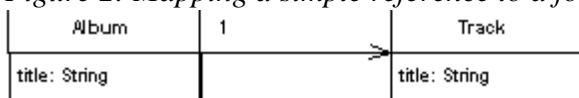


Figure 3: Classes and tables for a multi-valued reference

The complicated part of this occurs when you have an update. Updating implies that tracks can be added and removed to the collection within an album. How can you tell what the alterations are to put them to the database? Essentially you have three options, delete and insert, add a back pointer, or diff the collection.

With delete and insert you delete all the tracks in the database that link to the album, and then insert all the ones currently on the album. At first glance this sounds pretty appalling, especially if you haven't changed any. But the logic is easy to implement and as such it works pretty well compared to the alternatives. The limitation is that you can only do this if tracks are [Dependent Mapping](#)s. This means they must be owned by the album and can't be referred to outside the album.

Adding a back pointer puts a link from the track back to the album, effectively converting the association into a bidirectional association. This changes the object model, but now you can handle the update using the simple technique for single-valued fields on the other side.

If neither of those appeals, you can do a diff. There are two possibilities here, either diff with the current state of the database or diff with what you read the first time. Differing with the database involves re-reading the collection back from the database. You then compare the collection you read from the collection in the album. Anything on the database that isn't in the album was clearly removed, anything in the album that isn't on the disk is clearly a new item to be added. You then have to look at the logic of the application to decide what to do with each item.

Differing with what you read in the first place means you have to keep what you read. It's better as it avoids another database read. You can also diff what you first read with the database to help spot any concurrency problems.

In the general case anything that's added to the collection needs first to be checked to see if it's a new object, which you can do by seeing if it has a key. If there's no key, it needs to be added to the database. This step is made a lot easier by using [Unit of Work](#) because that way any new object will be inserted first automatically. In either case you then find the linked row in the database and update its foreign key to point to the current album.

For removal you have to know whether the track was moved to another album, has no album, or is deleted altogether. If it's moved to another album it should be updated when you update the other album. If it has no album then you need to null the foreign key. If it's deleted then it should be deleted as things get deleted. Handling deletes is much easier if the back link is mandatory, as it is here where every track must be on an album. That way you don't have to worry about detecting items removed from the collection, since they will be updated when you process the album they've been added to.

If the link is immutable, meaning you can't change a track's album, then adding always means insertion and removing always means deletion, which makes things simpler still. Usually in this case you can make the track a [Dependent Mapping](#).

One thing to watch out for is cycles in your links. Say you need to load an order, which has a link to a customer (which you load). The customer has a set of payments (which you load). Each payment has orders that it's paying for, which might include the original order you were trying to load. So you load the order (go back to the beginning of this paragraph.)

To avoid getting lost in recursive cycles you have two choices, which boil down to how you create your objects. Usually it's a good idea to have a creation method that includes data in the creation method that gives you a fully formed object. If you do that then you'll need to place [Lazy Load](#) at appropriate points to break the cycles. If you miss one you'll get a stack overflow, but if you're testing is good enough you can manage that burden.

The other is to create empty objects and immediately put them in an [Identity Map](#). That way when you

cycle back around the object is already loaded and you'll end the cycle. The objects you create aren't then fully formed, but they should be fully formed by the end of the load procedure. This avoids having to make special case decisions about the use of [Lazy Load](#) just to do a correct load.

## When to Use it

A *Foreign Key Mapping* can be used for most associations between classes. The most common case where it isn't possible to use *Foreign Key Mapping* is with many-to-many associations. Foreign keys are single values and first normal form means you can't store multiple foreign keys in a single field. So instead you need to use [Association Table Mapping](#)

If you have a collection field with no back pointer you should consider whether the many side should be a [Dependent Mapping](#), if so that can simplify your handling of the collection.

If the related object is a [Value Object](#) then you should use [Embedded Value](#).

## Example: Single Valued Reference (Java)

This is the simplest case, where an album has a single reference to an artist.

```
class Artist...
private String name;

public Artist(Long ID, String name) {
super(ID);
this.name = name;
}

public String getName() {
return name;
}

public void setName(String name) {
this.name = name;
}

class Album...
private String title;
private Artist artist;

public Album(Long ID, String title, Artist artist) {
super(ID);
this.title = title;
this.artist = artist;
}

public String getTitle() {
return title;
}

public void setTitle(String title) {
this.title = title;
}
```

```

}

public Artist getArtist() {
    return artist;
}

public void setArtist(Artist artist) {
    this.artist = artist;
}

```

Figure 4 shows how you can load an album. When an album mapper is told to load a particular album it queries the database and pulls back the result set for that object. It then queries the result set for each foreign key field and finds that object. Then it can create the album with the appropriate found objects. If the artist object was already in memory it would be fetched from the cache, otherwise it would be loaded from the database in the same way.

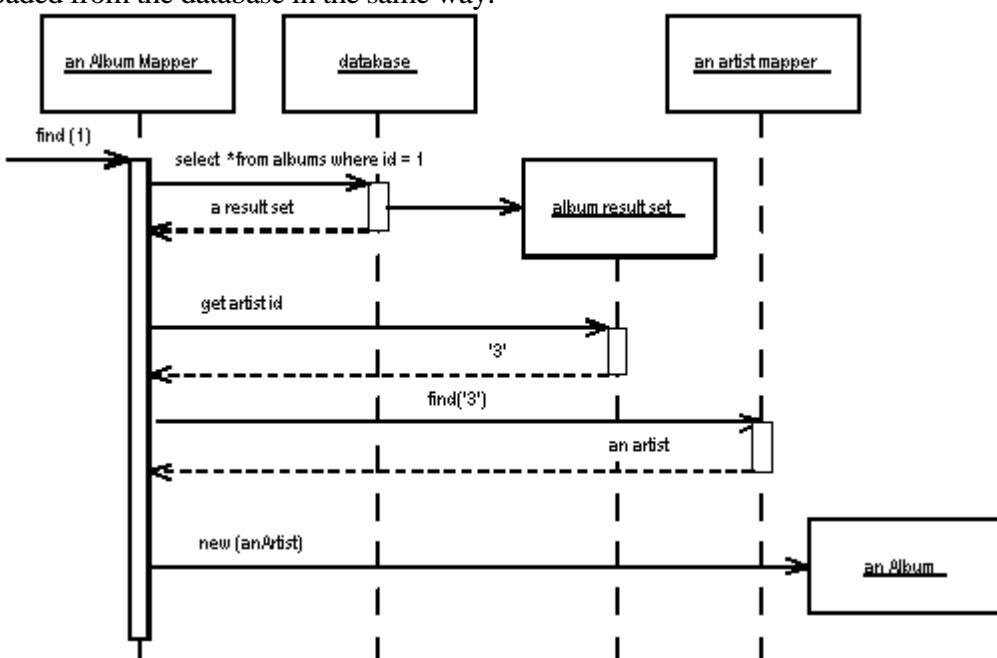


Figure 4: Sequence for loading a single valued field

The `find` operation uses abstract behavior to manipulate an [Identity Map](#)

```

class AlbumMapper...
public Album find(Long id) {
    return (Album) abstractFind(id);
}
protected String findStatement() {
    return "SELECT ID, title, artistID FROM albums WHERE ID = ?";
}

class AbstractMapper...
abstract protected String findStatement();

protected DomainObject abstractFind(Long id) {
    DomainObject result = (DomainObject) loadedMap.get(id);
    if (result != null) return result;
    PreparedStatement stmt = null;
    ResultSet rs = null;
    try {

```

```
stmt = DB.prepareStatement(findStatement());
stmt.setLong(1, id.longValue());
rs = stmt.executeQuery();
rs.next();
result = load(rs);
return result;
} catch (SQLException e) {
throw new ApplicationException(e);
} finally {cleanUp(stmt, rs);}
}

private Map loadedMap = new HashMap();
```

The find operation calls a load operation to actually load the data into the album.

```
class AbstractMapper...
protected DomainObject load(ResultSet rs) throws SQLException {
Long id = new Long(rs.getLong(1));
if (loadedMap.containsKey(id)) return (DomainObject) loadedMap.get(id);
DomainObject result = doLoad(id, rs);
doRegister(id, result);
return result;
}

protected void doRegister(Long id, DomainObject result) {
Assert.isFalse(loadedMap.containsKey(id));
loadedMap.put(id, result);
}

abstract protected DomainObject doLoad(Long id, ResultSet rs) throws
SQLException;
class AlbumMapper...
protected DomainObject doLoad(Long id, ResultSet rs) throws SQLException {
String title = rs.getString(2);
long artistID = rs.getLong(3);
Artist artist = MapperRegistry.artist().find(artistID);
Album result = new Album(id, title, artist);
return result;
}
```

To update an album the foreign key value is taken from the linked artist object.

```
class AbstractMapper...
abstract public void update(DomainObject arg);
class AlbumMapper...
public void update(DomainObject arg) {
PreparedStatement statement = null;
try {
statement = DB.prepare(
"UPDATE albums SET title = ?, artistID = ? WHERE id = ?");
statement.setLong(3, arg.getID().longValue());
Album album = (Album) arg;
statement.setString(1, album.getTitle());
statement.setLong(2, album.getArtist().getID().longValue());
statement.execute();
} catch (SQLException e) {
throw new ApplicationException(e);
} finally {
cleanUp(statement);
```

```
}
```

## Example: Multi-Table Find (Java)

While it's conceptually clean to issue one query per table, it's often inefficient - since SQL are remote calls and remote calls are slow. So it's often worth finding ways to gather information from multiple tables in a single query. So I can modify the above example to use a single query to get both the album and artist information with a single SQL call. The first alteration is that of the SQL for the find statement.

```
class AlbumMapper...
public Album find(Long id) {
    return (Album) abstractFind(id);
}
protected String findStatement() {
    return "SELECT a.ID, a.title, a.artistID, r.name " +
        " from albums a, artists r " +
        " WHERE ID = ? and a.artistID = r.ID";
}
```

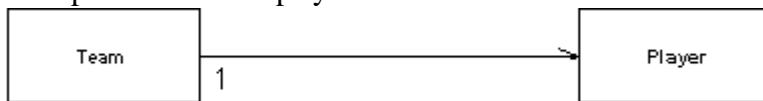
I then use a different load method that loads both the album and artist information together

```
class AlbumMapper...
protected DomainObject doLoad(Long id, ResultSet rs) throws SQLException {
    String title = rs.getString(2);
    long artistID = rs.getLong(3);
    ArtistMapper artistMapper = MapperRegistry.artist();
    Artist artist;
    if (artistMapper.isLoaded(artistID))
        artist = artistMapper.find(artistID);
    else
        artist = loadArtist(artistID, rs);
    Album result = new Album(id, title, artist);
    return result;
}
private Artist loadArtist(long id, ResultSet rs) throws SQLException {
    String name = rs.getString(4);
    Artist result = new Artist(new Long(id), name);
    MapperRegistry.artist().register(result.getID(), result);
    return result;
}
```

There's a tension here about where to put the method that maps the SQL result into the artist object. On the one hand it's better to put it in the artist's mapper, since it's the class that usually loads the artist. But on the other the load method is closely coupled to the SQL and thus should stay with the SQL query. In this case I've voted for the latter.

## Example: Collection of References (C#)

The case for a collection of references occurs when you have a field that is a collection. Here I'll use an example of teams and players where we'll assume we can't make player a [Dependent Mapping](#).



*Figure 5: A team with multiple players*

```

class Team...
public String Name;
public IList Players {
get {return ArrayList.ReadOnly(playersData);}
set {playersData = new ArrayList(value);}
}
public void AddPlayer(Player arg) {
playersData.Add(arg);
}
private IList playersData = new ArrayList();
  
```

In the database this will be handled with the player record having a foreign key to the team.



*Figure 6: Database structure for a team with multiple players*

```

class TeamMapper...
public Team Find(long id) {
return (Team) AbstractFind(id);
}
class AbstractMapper...
protected DomainObject AbstractFind(long id) {
Assert.True (id != DomainObject.PLACEHOLDER_ID);
DataRow row = FindRow(id);
return (row == null) ? null : Load(row);
}
protected DataRow FindRow(long id) {
String filter = String.Format("id = {0}", id);
DataRow[] results = table.Select(filter);
return (results.Length == 0) ? null : results[0];
}
protected DataTable table {
get {return dsh.Data.Tables[TableName];}
}
public DataSetHolder dsh;
abstract protected String TableName {get;}
class TeamMapper...
protected override String TableName {
get {return "Teams";}
}
  
```

The data set holder is a class that holds onto the data set in use, together with the adapters needed to

update it to the database.

```
class DataSetHolder...
public DataSet Data = new DataSet();
private Hashtable DataAdapters = new Hashtable();
```

For this example, we'll assume it has already been populated by some appropriate queries.

The find method calls a load to actually load the data into the new object.

```
class AbstractMapper...
protected DomainObject Load (DataRow row) {
long id = (int) row ["id"];
if (identityMap[id] != null) return (DomainObject) identityMap[id];
else {
DomainObject result = CreateDomainObject();
result.Id = id;
identityMap.Add(result.Id, result);
doLoad(result, row);
return result;
}
}
abstract protected DomainObject CreateDomainObject();
private IDictionary identityMap = new Hashtable();
abstract protected void doLoad (DomainObject obj, DataRow row);
class TeamMapper...
protected override void doLoad (DomainObject obj, DataRow row) {
Team team = (Team) obj;
team.Name = (String) row["name"];
team.Players = MapperRegistry.Player.FindForTeam(team.Id);
}
```

To bring in the players, I execute a specialized finder on the player mapper.

```
class PlayerMapper...
public IList FindForTeam(long id) {
String filter = String.Format("teamID = {0}", id);
DataRow[] rows = table.Select(filter);
IList result = new ArrayList();
foreach (DataRow row in rows) {
result.Add(Load (row));
}
return result;
}
```

To update, the team saves its own data and delegates to the player mapper to save the data into the player table.

```
class AbstractMapper...
public virtual void Update (DomainObject arg) {
Save (arg, FindRow(arg.Id));
}
abstract protected void Save (DomainObject arg, DataRow row);
class TeamMapper...
protected override void Save (DomainObject obj, DataRow row) {
Team team = (Team) obj;
row["name"] = team.Name;
savePlayers(team);
```

```
}

private void savePlayers(Team team){
foreach (Player p in team.Players) {
MapperRegistry.Player.LinkTeam(p, team.Id);
}
}

class PlayerMapper...
public void LinkTeam (Player player, long teamID) {
DataRow row = FindRow(player.Id);
row["teamID"] = teamID;
}
```



---

© Copyright [Martin Fowler](#), all rights reserved

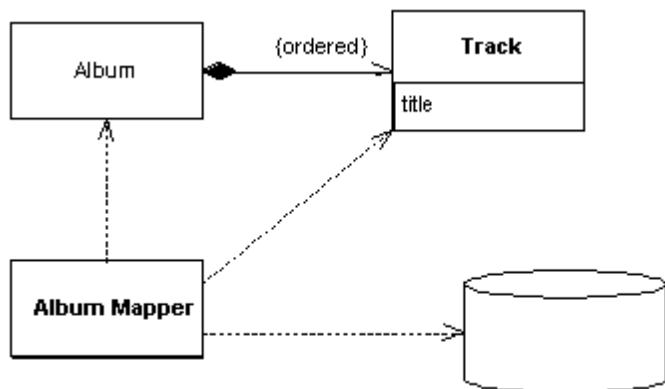


---

# Dependent Mapping

---

*Have one class perform the mapping for another*



Some objects naturally appear in the context of other objects. Tracks on an album may be loaded or saved whenever the underlying album is loaded or saved. If these tracks are not referenced to by any other table in the database, you can simplify the mapping procedure by having the album mapper perform the mapping for tracks as well - treating the mapping of the tracks as a *Dependent Mapping*.

## How it Works

The basic idea behind *Dependent Mapping* is that one class (the **dependent**) relies upon some other class (the **owner**) for its database persistence. Each dependent can have only one owner and must have one owner.

This manifests itself in terms of the classes that do the mapping. In the case of [Data Mapper](#) mapper class for the track. For an [Active Record](#) the track class won't contain any database mapping code, it will all be in the owning album. A [Row Data Gateway](#) will have no database code in the dependent. In a [Table Data Gateway](#) there will typically be no dependent class at all, all the handling of the dependent is done in the owner. In most cases every time you load an owner, the dependents are loaded too. If the dependents are expensive to load and infrequently used you can use a [Lazy Load](#) to avoid loading the dependents until you need them.

An important property of a dependent is that it does not have an [Identity Field](#) and therefore doesn't get stored in a [Identity Map](#). It therefore cannot be loaded by a find method that looks up an id. Indeed there is no finder for a dependent, all finds are done with the owner.

A dependent may itself be the owner of another dependent. In this case the owner of the first dependent is also responsible for the persistence of the second dependent. Indeed you can have a whole hierarchy of dependents controlled by a single primary owner.

It's usually easier for the primary key on the database to be a composite key that includes the owner's primary key. No other table should have a foreign key into the dependent's table, unless that object is dependent on that dependent. As a result no in-memory object, other than the owner, should have a reference to a dependent. Strictly speaking you can relax that rule providing that the reference isn't persisted to the database, but having a nonpersistent reference is itself a good source of confusion.

In a UML model, it's appropriate to use composition to show the relationship between an owner and its dependents.

Since the writing and saving of dependents is left to the owner, and there are no outside references, this allows updates to the dependents to be handled through deletion and insertion. So if you wish to update the collection of dependents you can safely delete all rows that link to the owner and then reinsert all the dependents. This saves you from having to do an analysis of what objects got added or removed from the owner's collection.

Dependents are in many ways rather like [Value Objects](#), although they often do not need the full mechanics that you use in making something a [Value Object](#) (such as overriding equals). The main difference is that from a purely in-memory point of view there is nothing special about them. The dependent nature of the objects is only really due to the database mapping behavior.

Using *Dependent Mapping* complicates tracking whether the owner has changed. Any change to a dependent needs to mark the owner as changed so that the owner will write the changes out to the database. You can simplify this considerably by making the dependent immutable, so that any change to a dependent needs to be done by deleting the old one and creating a new one. Although this can make the in-memory model harder to work with, it does simplify the database mapping. While in theory the in-memory and database mapping should be independent when you're using [Data Mapper](#), in practice you have to make the occasional compromise.

## When to Use it

You use *Dependent Mapping* when you have an object that is only referred to by one other object. Usually this occurs when you have one object having a collection of dependents. As such *Dependent Mapping* is a good way of dealing with the awkward situation where the owner has a collection of references to its dependents, but there is no back-pointer. Providing the many objects don't have any need for their own identity, using *Dependent Mapping* makes it easier to manage their persistence.

For *Dependent Mapping* to work there are a number of pre-conditions.

- A dependent must have exactly one owner.
- There must be no references from any object other than the owner to the dependent

There is a school of OO design which uses the notion of entity objects and dependent objects when designing a [Domain Model](#). I tend to think of *Dependent Mapping* as a particular technique which I use to simplify database mapping, rather than as a fundamental OO design medium. In particular I avoid

large graphs of dependents. The problem with these large graphs is that it's impossible to refer to a dependent from outside the graph, which often leads to complex lookup schemes based around the root owner.

I don't recommend using *Dependent Mapping* if you're using [Unit of Work](#). The delete and re-insert strategy doesn't help at all if you have a [Unit of Work](#) keeping track of things. It can also lead to problems since the [Unit of Work](#) isn't controlling the dependents. Mike Rettig told me about an application where a [Unit of Work](#) would keep track rows inserted for testing and then delete them all when done. As it didn't track dependents, orphan rows would appear causing bugs in the test runs.

## Example: Albums and Tracks (Java)

In this domain model an album holds a collection of tracks. Since this uselessly simple application does not need anything else to refer to a track, it's an obvious candidate for *Dependent Mapping*. (Indeed anyone would think the example is deliberately constructed for the pattern.)

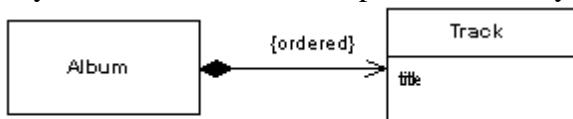


Figure 1: An album with tracks which can be handled using Dependent Mapping

This track just has a title. I've defined it as an immutable class.

```
class Track...
private final String title;

public Track(String title) {
this.title = title;
}

public String getTitle() {
return title;
}
```

The tracks are held in the album class.

```
class Album...
private List tracks = new ArrayList();

public void addTrack(Track arg) {
tracks.add(arg);
}

public void removeTrack(Track arg) {
tracks.remove(arg);
};

public void removeTrack(int i) {
tracks.remove(i);
}

public Track[] getTracks() {
return (Track[]) tracks.toArray(new Track[tracks.size()]);
}
```

}

The album mapper class handles all the SQL for tracks and thus defines the SQL statements that access the tracks table.

```
class AlbumMapper...
protected String findStatement() {
return
"SELECT ID, a.title, t.title as trackTitle" +
" FROM albums a, tracks t" +
"WHERE a.ID = ? AND t.albumID = a.ID" +
"ORDER BY t.seq";
}
```

The tracks are loaded into the album whenever the album is loaded.

```
class AlbumMapper...
protected DomainObject doLoad(Long id, ResultSet rs) throws SQLException {
String title = rs.getString(2);
Album result = new Album(id, title);
loadTracks(result, rs);
return result;
}

public void loadTracks(Album arg, ResultSet rs) throws SQLException {
arg.addTrack(newTrack(rs));
while (rs.next()) {
arg.addTrack(newTrack(rs));
}
}

private Track newTrack(ResultSet rs) throws SQLException {
String title = rs.getString(3);
Track newTrack = new Track (title);
return newTrack;
}
```

When the album is updated all the tracks are deleted and reinserted.

```
class AlbumMapper...
public void update(DomainObject arg) {
PreparedStatement updateStatement = null;
try {
updateStatement = DB.prepare("UPDATE albums SET title = ? WHERE id = ?");
updateStatement.setLong(2, arg.getID().longValue());
Album album = (Album) arg;
updateStatement.setString(1, album.getTitle());
updateStatement.execute();
updateTracks(album);
} catch (SQLException e) {
throw new ApplicationException(e);
} finally {DB.cleanUp(updateStatement);
}

public void updateTracks(Album arg) throws SQLException {
PreparedStatement deleteTracksStatement = null;
```

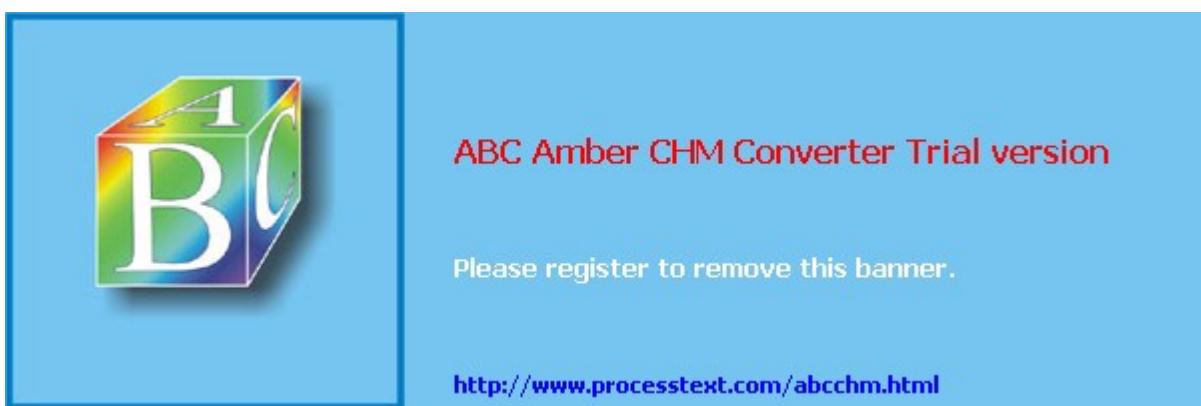
```
try {
deleteTracksStatement = DB.prepare("DELETE from tracks WHERE albumID = ?");
deleteTracksStatement.setLong(1, arg.getID().longValue());
deleteTracksStatement.execute();
for (int i = 0; i < arg.getTracks().length; i++) {
Track track = arg.getTracks()[i];
insertTrack(track, i + 1, arg);
}
} finally {DB.cleanUp(deleteTracksStatement);
}
}

public void insertTrack(Track track, int seq, Album album) throws
SQLException {
PreparedStatement insertTracksStatement = null;
try {
insertTracksStatement =
DB.prepare("INSERT INTO tracks (seq, albumID, title) VALUES (?, ?, ?)");
insertTracksStatement.setInt(1, seq);
insertTracksStatement.setLong(2, album.getID().longValue());
insertTracksStatement.setString(3, track.getTitle());
insertTracksStatement.execute();
} finally {DB.cleanUp(insertTracksStatement);
}
}
```



---

© Copyright [Martin Fowler](#), all rights reserved

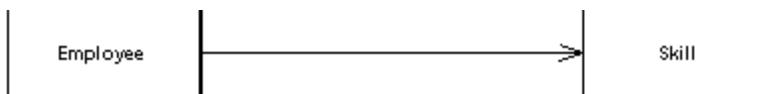


---

# Association Table Mapping

---

*Save an association as a table with foreign keys to the tables that are linked by the association.*



Objects can handle multi-valued fields quite easily by using collections as field values. Relational databases don't have this feature and are constrained to single-valued fields only. When you're mapping a one-to-many association you can handle this using [Foreign Key Mapping](#), essentially to use a foreign key for the single-valued end of the association. But a many-to-many association can't do this because there isn't a single-valued end to hold the foreign key.

The answer is the classic resolution used by relational data people for decades: create an extra table to record the relationship. We can then use *Association Table Mapping* to map the multi-valued field to this link table.

## How it Works

The basic idea behind *Association Table Mapping* is to use a link table to store the association. The link table has only the foreign key ids for the two tables that are linked together. It has one row for each pair of objects that are associated.

The link table has no corresponding in-memory object. As a result it has no ID. The primary key of the link table is the compound of the two primary keys of the tables that are associated together.

To load data from the link table you, in simple terms, perform two queries. Consider loading the skills for an employee. In this case you do, at least conceptually, queries in two stages. The first stage queries the skillsEmployees table to find all the rows that link to the employee you want. Then for each row in the

link table you find the skill object for the related id.

If all this information is already in memory, this scheme works fine. If it isn't, this scheme can be horribly expensive in queries, since you'll do a query for each skill that's in the link table. You can avoid this cost by joining the skills table to the link table which allows you to get all the data in a single query, albeit at the cost of making the mapping a bit more complicated.

Updating the link data involves many of the same issues as updating a many valued field. Fortunately the matter is made much easier since you can in many ways treat the link table in a similar way to [Dependent Mappings](#). No other table should refer to the link table, so you can freely create and destroy links as you need them.

## When to Use it

The canonical case for *Association Table Mapping* is a many-to-many association, since there's not really any alternatives for that situation.

*Association Table Mapping* can also be used for any other form of association. It's more complex to use than [Foreign Key Mapping](#), and involves an extra join - so it's not usually the right choice.

However *Association Table Mapping* is an option if you have tables that you need to link but can't be altered, or if that's the way the tables are in an existing database schema.

## Example: Employees and Skills (C#)

Here's a simple example using the sketch's model. We have an employee class with a collection of skills where each skill can appear for more than one employee.

```
class Employee...
public IList Skills {
get {return ArrayList.ReadOnly.skillsData;}
set {skillsData = new ArrayList(value);}
}
public void AddSkill (Skill arg) {
skillsData.Add(arg);
}
public void RemoveSkill (Skill arg) {
skillsData.Remove(arg);
}
private IList skillsData = new ArrayList();
```

To load an employee from the database, we need to pull in the skills using an employee mapper. Each employee mapper class has a find method that creates an employee object. All mappers are subclasses of the abstract mapper class that pulls together common services for the mappers.

```
class EmployeeMapper...
public Employee Find(long id) {
return (Employee) AbstractFind(id);
}
class AbstractMapper...
```

```
protected DomainObject AbstractFind(long id) {
    Assert.True (id != DomainObject.PLACEHOLDER_ID);
    DataRow row = FindRow(id);
    return (row == null) ? null : Load(row);
}
protected DataRow FindRow(long id) {
    String filter = String.Format("id = {0}", id);
    DataRow[] results = table.Select(filter);
    return (results.Length == 0) ? null : results[0];
}
protected DataTable table {
    get {return dsh.Data.Tables[TableName];}
}
public DataSetHolder dsh;
abstract protected String TableName {get;}
class EmployeeMapper...
protected override String TableName {
    get {return "Employees";}
}
```

The data set holder is a simple object that contains an ADO.NET data set and the relevant adaptors to save it to the database.

```
class DataSetHolder...
public DataSet Data = new DataSet();
private Hashtable DataAdapters = new Hashtable();
```

To make this example simple, indeed simplistic, we'll assume the data set has already been loaded with all the data we might need.

The find method calls load methods to load data for the employee.

```
class AbstractMapper...
protected DomainObject Load (DataRow row) {
    long id = (int) row ["id"];
    if (identityMap[id] != null) return (DomainObject) identityMap[id];
    else {
        DomainObject result = CreateDomainObject();
        result.Id = id;
        identityMap.Add(result.Id, result);
        doLoad(result, row);
        return result;
    }
}
abstract protected DomainObject CreateDomainObject();
private IDictionary identityMap = new Hashtable();
abstract protected void doLoad (DomainObject obj, DataRow row);
class EmployeeMapper...
protected override void doLoad (DomainObject obj, DataRow row) {
    Employee emp = (Employee) obj;
    emp.Name = (String) row["name"];
    loadSkills(emp);
}
```

Loading the skills is sufficiently awkward to demand a separate method to do the work.

```
class EmployeeMapper...
```

```
private IList loadSkills (Employee emp) {
    DataRow[] rows = skillLinkRows(emp);
    IList result = new ArrayList();
    foreach (DataRow row in rows) {
        long skillID = (int)row["skillID"];
        emp.AddSkill(MapperRegistry.Skill.Find(skillID));
    }
    return result;
}
private DataRow[] skillLinkRows(Employee emp) {
    String filter = String.Format("employeeID = {0}", emp.Id);
    return skillLinkTable.Select(filter);
}
private DataTable skillLinkTable {
    get {return dsh.Data.Tables["skillEmployees"]; }
}
```

To handle changes in skills information there is an update method on the abstract mapper.

```
class AbstractMapper...
public virtual void Update (DomainObject arg) {
    Save (arg, FindRow(arg.Id));
}
abstract protected void Save (DomainObject arg, DataRow row);
```

The update method calls a save method in the subclass.

```
class EmployeeMapper...
protected override void Save (DomainObject obj, DataRow row) {
    Employee emp = (Employee) obj;
    row["name"] = emp.Name;
    saveSkills(emp);
}
```

Again I've made a separate method for saving the skills.

```
class EmployeeMapper...
private void saveSkills(Employee emp) {
    deleteSkills(emp);
    foreach (Skill s in emp.Skills) {
        DataRow row = skillLinkTable.NewRow();
        row["employeeID"] = emp.Id;
        row["skillID"] = s.Id;
        skillLinkTable.Rows.Add(row);
    }
}
private void deleteSkills(Employee emp) {
    DataRow[] skillRows = skillLinkRows(emp);
    foreach (DataRow r in skillRows) r.Delete();
}
```

The logic here does the simple thing of deleting all existing link table rows and creating new ones. This saves me having to figure out which ones have been added and deleted.

## Example: Albums, tracks and performers

# (Java)

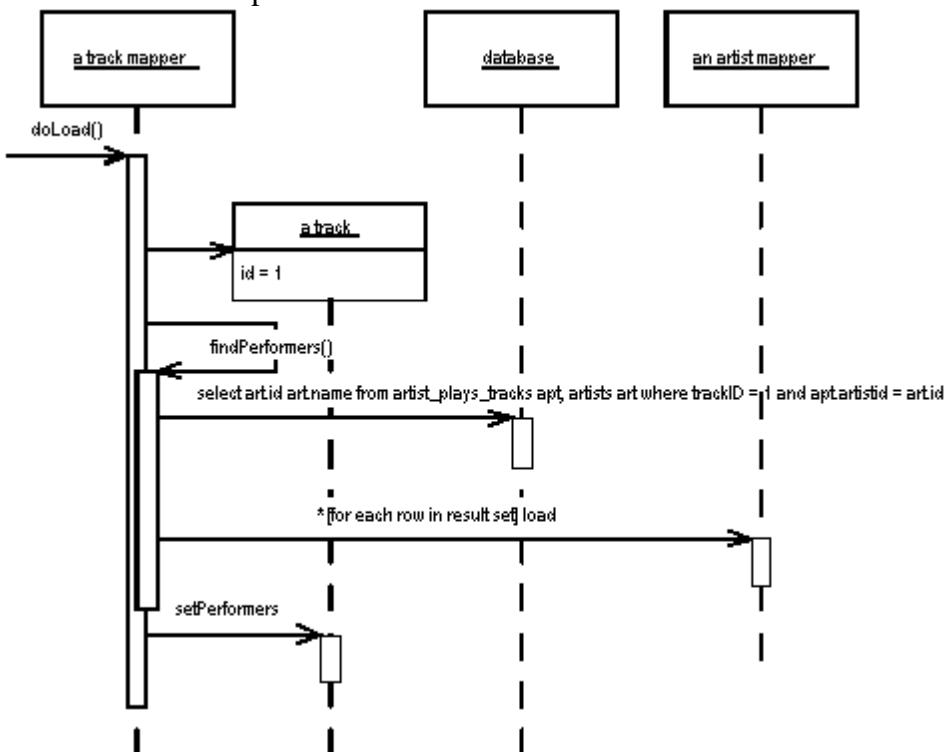
One of the nice things about ADO.NET is that it allows me to discuss the basics of an object-relational mapping without getting into the sticky details of minimizing queries. With other relational mapping schemes you are closer to the SQL and have to take much of that into account.

For this example we'll look at reading in data for the classes in Figure 1.



*Figure 1:*

If we want to load an album, we want to load all the tracks on that album. As we load each track, we want to load all of the performers for that track.



*Figure 2: Finding the performers for an album with a single SQL call*

I'll begin by looking at the logic for finding an album.

```

class AlbumMapper...
public Album find(Long id) {
return (Album) abstractFind(id);
}

protected String findStatement() {
return "select ID, title, artistID from albums where ID = ?";
}

class AbstractMapper...

```

```
abstract protected String findStatement();
protected Map loadedMap = new HashMap();

protected DomainObject abstractFind(Long id) {
DomainObject result = (DomainObject) loadedMap.get(id);
if (result != null) return result;
PreparedStatement stmt = null;
ResultSet rs = null;
try {
stmt = DB.prepare(findStatement());
stmt.setLong(1, id.longValue());
rs = stmt.executeQuery();
rs.next();
result = load(rs);
return result;
} catch (SQLException e) {
throw new ApplicationException(e);
} finally {cleanUp(stmt, rs);
}
}
```

The album loads its data and also loads the tracks.

```
class AbstractMapper...
protected DomainObject load(ResultSet rs) throws SQLException {
Long id = new Long(rs.getLong("id"));
if (loadedMap.containsKey(id)) return (DomainObject) loadedMap.get(id);
DomainObject result = doLoad(id, rs);
loadedMap.put(id, result);
return result;
}

abstract protected DomainObject doLoad(Long id, ResultSet rs) throws
SQLException;
class AlbumMapper...
protected DomainObject doLoad(Long id, ResultSet rs) throws SQLException {
String title = rs.getString(2);
long artistID = rs.getLong(3);
Artist artist = FinderRegistry.artist().find(artistID);
Album result = new Album(id, title, artist);
result.setTracks(FinderRegistry.track().findForAlbum(id));
return result;
}
```

To load the tracks it uses a particular finder to find the tracks for the album.

```
class TrackMapper...
public static final String findForAlbumStatement =
"SELECT ID, seq, albumID, title " +
"FROM tracks " +
"WHERE albumID = ? ORDER BY seq";

public List findForAlbum(Long albumID) {
PreparedStatement stmt = null;
ResultSet rs = null;
try {
stmt = DB.prepare(findForAlbumStatement);
stmt.setLong(1, albumID.longValue());
```

```
rs = stmt.executeQuery();
List result = new ArrayList();
while (rs.next())
result.add(load(rs));
return result;
} catch (SQLException e) {
throw new ApplicationException(e);
} finally {cleanUp(stmt, rs);
}
}
```

Loading the track data occurs in the load method.

```
class TrackMapper...
protected DomainObject doLoad(Long id, ResultSet rs) throws SQLException {
String title = rs.getString("title");
Track result = new Track(id, title);
result.setPerformers(findPerformers(id));
return result;
}
```

There's a separate method for loading the performers. This invokes the query on the link table.

```
class TrackMapper...
private static final String findPerformersStatement =
"SELECT " + ArtistMapper.COLUMN_LIST +
" FROM artist_plays_tracks apt, artists art " +
" WHERE apt.trackID = ? AND apt.artistId = art.id " +
" ORDER BY apt.artistID";

public List findPerformers(Long trackID) {
PreparedStatement stmt = null;
ResultSet rs = null;
try {
stmt = DB.prepare(findPerformersStatement);
stmt.setLong(1, trackID.longValue());
rs = stmt.executeQuery();
List result = new ArrayList();
while (rs.next()) {
ArtistMapper artistMapper = (ArtistMapper) FinderRegistry.artist();
result.add(artistMapper.load(rs));
}
return result;
} catch (SQLException e) {
throw new ApplicationException(e);
} finally {cleanUp(stmt, rs);
}
}
```

Since this method invokes the load method on the artist mapper, it gets the list of columns to return from the artist mapper class. That way the columns and the load method stay in sync.





ABC Amber CHM Converter Trial version

Please register to remove this banner.

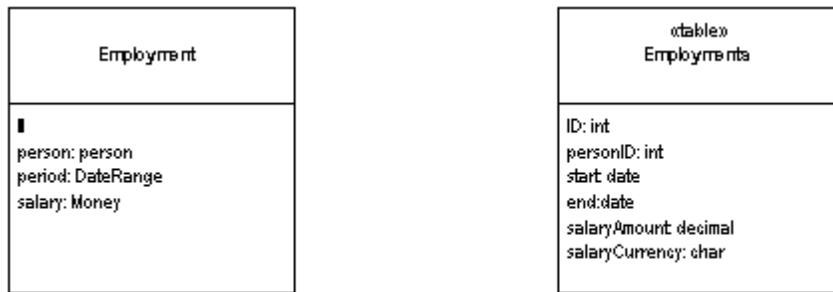
<http://www.processtext.com/abcchm.html>

---

# Embedded Value

---

*Map an object into several fields of another object's table*



There are many small objects that make sense in an OO system, that don't make sense as tables in a database. Examples include currency aware money objects, date ranges and the like. So although the default thinking is to save an object as a table, no sane person would want a table of money values.

An *Embedded Value* maps the values of an object to fields in the record of the object's owner. So in the sketch, we have an employment object with links to a date range object and a money object. In the resulting table the fields in those objects are mapped to fields in the employment table, rather than making new objects themselves.

## How it Works

Carrying out this exercise is actually quite simple. When the owning object (the employment) is loaded or saved, the dependent objects (the date range and the money) are loaded and saved at the same time. The dependent classes won't have their own persistence methods, all persistence is done by the owner.

## When to Use it

This is one of these patterns where the doing of it is very straightforward, but knowing when to use it becomes much more complicated.

The simplest and straightforward cases for this are the clear simple *Value Objects* like money and date range. Since *Value Objects* don't have identity you can create and destroy these easily without worrying

about such things as [Identity Map](#)s to keep them all in sync. Indeed all [Value Object](#)s should be persisted as *Embedded Value*, since there you'd never want a table for them.

The grey line comes on whether it's worth storing reference objects using *Embedded Value*, such as an order and shipping object. The principal question here is whether the shipping data has any relevance outside the context of the order. One issue is the loading and saving. If you only load the shipping data into memory when you load the order, then that's an argument for saving both into the same table. Another question is whether you'll want to access the shipping data separately through SQL. This can be an important issue if you are using reporting through SQL and aren't using a separate database for reporting.

In most cases you'll only use *Embedded Value* on a reference object when the association between them is single valued at both ends (a one-to-one association). Occasionally you may use it if there are multiple candidate dependents if their number is small and fixed. Then you'll have numbered fields for each value. This is messy table design, and is horrible to query in SQL, but it may have performance benefits. Usually if this is the case, however, [Serialized LOB](#) is the better choice.

Since so much of the logic for deciding when to use *Embedded Value* is the same as for [Serialized LOB](#), there's the obvious matter of choosing between the two. The great advantage of *Embedded Value* is that it allows SQL queries to be made against the values in the dependent object. Although using XML as the serialization, together with XML based query add-ons to SQL may alter that in the future, at the moment you really need to use *Embedded Value* if you want to use dependent values in a query. This may be important for separate reporting mechanisms on the database

*Embedded Value* can only be used for fairly simple dependents. A solitary dependent, or a few separated dependents works well. [Serialized LOB](#) however works with more complex structures, including potentially large object sub-graphs.

## Further Reading

*Embedded Value* has been called a couple of different names in its history. TOPLink refers to this pattern as *aggregate mapping*. Visual Age refers to this as *composer*.

## Example: Simple value object (Java)

This is the classic example of a value object mapped with *Embedded Value*. We'll begin with a simple product offering class with the following fields

```
class ProductOffering...
private Product product;
private Money baseCost;
private Integer ID;
```

In these fields the ID is an [Identity Field](#), and the product is a regular record mapping. We'll map the baseCost using *Embedded Value*. We'll do the overall mapping with [Active Record](#) to help keep things simple.

Since we're using *Active Record* we need save and load routines. These routines are on the product offering class, since it's the owner. The money class has no persistence behavior at all. These routines are quite simple. Here is the load method.

```
class ProductOffering...
public static ProductOffering load(ResultSet rs) {
try {
Integer id = (Integer) rs.getObject("ID");
BigDecimal baseCostAmount = rs.getBigDecimal("base_cost_amount");
Currency baseCostCurrency =
Registry.getCurrency(rs.getString("base_cost_currency"));
Money baseCost = new Money(baseCostAmount, baseCostCurrency);
Integer productID = (Integer) rs.getObject("product");
Product product = Product.find((Integer) rs.getObject("product"));
return new ProductOffering(id, product, baseCost);
} catch (SQLException e) {
throw new ApplicationException(e);
}
}
```

Here's the update behavior. Again it's a simple variation on the updates.

```
class ProductOffering...
public void update() {
PreparedStatement stmt = null;
try {
stmt = DB.prepare(updateStatementString);
stmt.setBigDecimal(1, baseCost.amount());
stmt.setString(2, baseCost.currency().code());
stmt.setInt(3, ID.intValue());
stmt.execute();
} catch (Exception e) {
throw new ApplicationException(e);
} finally {DB.cleanUp(stmt);}
}

private String updateStatementString =
"UPDATE product_offerings" +
"SET base_cost_amount = ?, base_cost_currency = ? " +
"WHERE id = ?";
```





ABC Amber CHM Converter Trial version

Please register to remove this banner.

<http://www.processtext.com/abcchm.html>

# Lazy Load

---

*An object that doesn't contain all of the data you need, but knows how to get it.*

```
class Supplier...
public List getProducts() {
    if (products == null) products = Product.findForSupplier(getID());
    return products;
}
```

As you load data from a database into memory it's handy to design things so that as you load an object of interest, you also load the objects that are related to that object. This makes loading easier on the developer using the object, who otherwise has to explicitly load all the objects they need themselves.

However if you take this to its logical conclusion, you reach the point where loading one object can have the effect of loading a huge amount of other related objects into the system, something that hurts performance when only a few of the objects are actually needed.

A *Lazy Load* interrupts this loading process for the moment, leaving a marker in the object structure so that if the data is needed it can be loaded only then. As many people know, if you're lazy about doing things you'll win when it turns out you don't need to do them at all.

## How it Works

There are four main ways you can implement *Lazy Load*: lazy initialization, virtual proxy, value holder, and ghost.

**Lazy initialization** [[Beck-patterns](#)] is the simplest one to do. The basic idea is that every access to the field checks first to see if it's null. If it's null it calculates the value of the field before returning the field. To make this work you have to ensure that the field is self-encapsulated; meaning that all access to the field, even from within the class, is done through a getting method.

Using a null to signal a field that hasn't been loaded yet works well, unless null is a legal value for the field. In this case you either need something else to signal the field hasn't been loaded, or to use a [Special Case](#) for the null value.

Using lazy initialization is simple, but it does tend to force a dependency between the object concerned and the database so it works best for [Active Record](#), [Table Data Gateway](#), and [Row Data Gateway](#).

If you're using [\*Data Mapper\*](#) you'll need an additional layer of indirection. You can obtain this by using a **virtual proxy**[[Gang of Four](#)]. A virtual proxy is an object that looks like the object that should be in the field, but actually doesn't contain anything. Only when one of its methods is called does it load the correct object from the database.

The good thing about a virtual proxy is that looks exactly like the object that's supposed to be there. However it isn't so you can easily run into a nasty identity problem. Often the virtual proxy is a different object to the real object. Furthermore you can have more than one virtual proxy for the same real object. All of these will have different object identities, yet they represent the same conceptual object. At the very least you have to override the equality method and remember to use it instead of an identity method. But without that, and discipline, you'll run into some very hard to track bugs.

In some environments the other problem with a virtual proxy is that you end up having to create lots of them, one for each class you are proxying. You can usually avoid this in dynamically typed languages, but with static type languages things usually get messy. Even when the platform provides handy facilities, such as Java's proxies, they introduce other inconveniences.

These problems don't hit you if you only use virtual proxies for collections classes, such as lists. Since collections are [\*Value Objects\*](#), their identity doesn't matter. Additionally you only have a few collection classes to write virtual collections for.

With domain classes you can avoid these problems by using a **value holder**. The value holder concept, which I first came across in Smalltalk, is an object that wraps some other object. To get the underlying object you ask the value holder for its value. Only on the first access does it pull the data from the database. The disadvantages of the value holder is that the class needs to know there is a value holder present, and you lose the explicitness of strong typing. You can avoid identity problems by ensuring that the value holder is never passed out beyond its owning class.

A **ghost** is the real object, but not in its full state. When you load the object from the database it contains just its ID. Whenever you try to access a field it loads its full state from the database. You can think of a ghost as an object where every field is lazy initialized in one fell swoop, or as a virtual proxy where the object is its own virtual proxy. Of course there's no need to load all the data in one go, you may group the data into groups that are commonly used together. If you use a ghost you can put it immediately in its [\*Identity Map\*](#). By doing this not just will you maintain identity, you'll also avoid all problems due to cyclic references when reading in data.

When you use a virtual proxy or a ghost the proxy/ghost doesn't need to be completely devoid of data. If you have some data which is quick to get hold of and commonly used, it may make sense to load this data when you load the proxy or ghost. (This is sometimes referred to as a "light object".)

Inheritance often poses a problem with *Lazy Load*. If you're going to use ghosts, you'll need to know what type of ghost to create, which you often can't tell without loading the thing properly. Virtual proxies can suffer from the same problem in static typed languages.

One danger with *Lazy Load* is that it can easily cause more database accesses than you need. A good example of this **ripple loading** is if you fill a collection with *Lazy Loads* and then look at them one at a time. This will cause you to go to the database once for each object, instead of reading them all in at once. I've seen ripple loading cripple the performance of an application. One way to avoid ripple loading is not to have a collection of *Lazy Loads*, rather make the collection itself a *Lazy Load* and when you load it load all the contents. The limitation of this tactic is when the collection is very large, such as all the IP addresses in the world. In practice these aren't usually linked up through associations in the object

model, so that doesn't happen very often. When it does you'll need a Value List Handler [missing reference].

*Lazy Load* is a good candidate for aspect-oriented programming. You can put the *Lazy Load* behavior into a separate aspect, which allows you change the lazy load strategy separately as well as freeing the domain developers from having to deal with lazy load issues. I've also seen a project post-process Java bytecode to implement *Lazy Load* in a transparent way.

Often you'll run into situations where different use cases work best with a different variety of laziness. Some use cases need one subset of the object graph, others use a different subset. For maximum efficiency you want to load the right sub-graph for the right use case.

The way to deal with this is to have separate database interaction objects for the different use cases. So if you use [Data Mapper](#) you may have two order mapper objects, one that loads the line items immediately, and one that loads the line items lazily. The application code chooses the appropriate mapper depending on the use case. A variation on this is to have the same basic loader object, but defer to a strategy object to decide the loading pattern. It's a bit more sophisticated but can be a better way to factor the behavior.

In theory, you might want a range of different degrees of laziness, but in practice you really only need two: a complete load and enough of a load for identification purposes in a list. Adding more usually adds more complexity than is worth while.

## When to Use it

Deciding when to use *Lazy Load* is all about deciding how much you want to pull back from the database as you load an object, and how many database calls it will require. It's pointless to use *Lazy Load* on a field that's stored in the same row as the rest of the object, because it doesn't cost any more to bring back extra data in a call, even if the data field is quite large: such as a [Serialized LOB](#). So it's only worth considering *Lazy Load* if the field requires an extra database call to access.

In performance terms it's then about deciding when you want to take the hit of bringing back the data. Often it's a good idea to bring everything you'll need in one call so you have it in place, particularly if it corresponds to a single interaction with a UI. The best time to use *Lazy Load* is when not just does it involve an extra call, it's also data that often isn't used when the main object is used.

Adding *Lazy Load* does add a little complexity to the program. So my preference is to not use *Lazy Load* unless I actively think I'll need it.

## Example: Lazy Initialization (Java)

The essence of lazy initialization is code like this.

```
class Supplier...
public List getProducts() {
if (products == null) products = Product.findForSupplier(getID());
return products;
```

}

In this way the first access of the products field causes the data to be loaded from the database.

## Example: Virtual Proxy (Java)

The key to the virtual proxy is to provide a class that looks like the actual class you would normally use, but actually holds a simple wrapper around the real class. So a list of products for a supplier would be held with a regular list field.

```
class SupplierVL...
private List products;
```

The most complicated thing about producing a list proxy like this is setting it up so that you can provide an underlying list that's only created when it's accessed. To do this we have to pass the code that's needed to create the list into the virtual list when it's instantiated.

The best way to do this in Java is to define an interface for the loading behavior.

```
public interface VirtualListLoader {
List load();
}
```

Then we can instantiate the virtual list with a loader that calls the appropriate mapper method.

```
class SupplierMapper...
public static class ProductLoader implements VirtualListLoader {
private Long id;

public ProductLoader(Long id) {
this.id = id;
}

public List load() {
return ProductMapper.create().findForSupplier(id);
}
}
```

During the load method we assign the product loader to the list field

```
class SupplierMapper...
protected DomainObject doLoad(Long id, ResultSet rs) throws SQLException {
String nameArg = rs.getString(2);
SupplierVL result = new SupplierVL(id, nameArg);
result.setProducts(new VirtualList(new ProductLoader(id)));
return result;
}
```

The virtual list's source list is self encapsulated and evaluates the loader on first reference

```
class VirtualList...
private List source;
private VirtualListLoader loader;

public VirtualList(VirtualListLoader loader) {
this.loader = loader;
}

private List getSource() {
if (source == null) source = loader.load();
return source;
}
```

I then implement the regular list methods to delegate to the source list.

```
class VirtualList...
public int size() {
return getSource().size();
}

public boolean isEmpty() {
return getSource().isEmpty();
}

//... and so on for rest of list methods
```

This way the domain class knows nothing about how the mapper class does the *Lazy Load*. Indeed the domain class isn't even aware that there is a *Lazy Load*.

## Example: Using a Value Holder (Java)

A value holder can be used as a generic *Lazy Load*. In this case the domain type is aware that something is afoot, since the product field is typed as a value holder. This fact can be hidden from clients of the supplier by the getting method.

```
class SupplierVH...
private ValueHolder products;

public List getProducts() {
return (List) products.getValue();
}
```

The value holder itself does the *Lazy Load* behavior. It needs to be passed the necessary code to load its value when it's accessed. We can do this by defining a loader interface.

```
class ValueHolder...
private Object value;
private ValueLoader loader;

public ValueHolder(ValueLoader loader) {
this.loader = loader;
}
```

```
public Object getValue() {  
    if (value == null) value = loader.load();  
    return value;  
}
```

```
public interface ValueLoader {  
    Object load();  
}
```

A mapper can set up the value holder by creating an implementation of the loader and putting it into the supplier object

```
class SupplierMapper...  
protected DomainObject doLoad(Long id, ResultSet rs) throws SQLException {  
    String nameArg = rs.getString(2);  
    SupplierVH result = new SupplierVH(id, nameArg);  
    result.setProducts(new ValueHolder(new ProductLoader(id)));  
    return result;  
}  
  
public static class ProductLoader implements ValueLoader {  
    private Long id;  
  
    public ProductLoader(Long id) {  
        this.id = id;  
    }  
  
    public Object load() {  
        return ProductMapper.create().findForSupplier(id);  
    }  
}
```

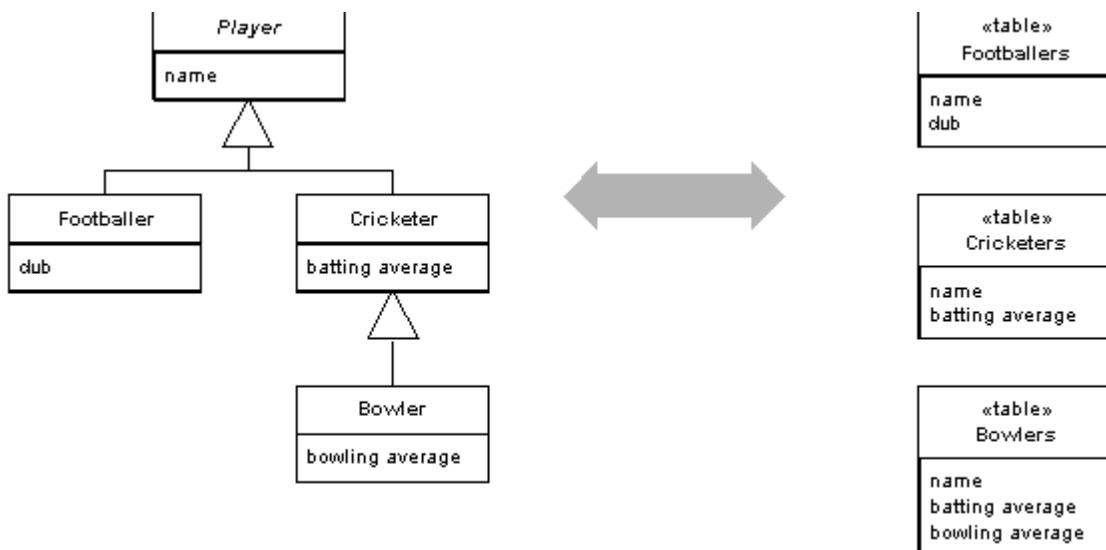


© Copyright [Martin Fowler](#), all rights reserved



# Concrete Table Inheritance

*Represent an inheritance hierarchy of classes with one table per concrete class in the hierarchy.*



## How it Works

*Concrete Table Inheritance* uses one database table for each concrete class in the hierarchy. Each table contains columns for the concrete class and all its ancestors. As a result any fields in a superclass are duplicated across the tables of the subclasses.

As with all of these inheritance schemes the basic behavior uses [\*Inheritance Mappers\*](#).

You need to pay attention to the keys with this scheme. Punningingly the key thing to do is to ensure that keys are not just unique to a table, but also unique to all the tables in a hierarchy. The classic case of where you need this is if you have a collection of players and you are using [\*Identity Field\*](#). If keys can be duplicated between concrete tables you'll get multiple rows for a particular key value. You thus need a key allocation system that keeps track of key usage across the tables, it also means you can't rely on the database's primary key uniqueness mechanism.

This becomes particularly awkward if you are hooking up to databases that are used by other systems. In many of these cases you can't guarantee key uniqueness across tables. In this situation you either have to avoid using superclass fields or do a compound key that involves a table identifier.

An example of avoiding superclass fields is to use separate footballer, cricketer, and bowler collections as fields instead of a player collection. You can still have a player collection in an interface, but you have to concatenate the fields together to do this.

For compound keys you need a special key object to use as your id field for [Identity Field](#). This key would use both the primary key of the table and the table name to determine uniqueness.

Related to this is problems with using referential integrity in the database. Consider an object model like Figure 1. To implement this in the database you need a link table that contains foreign key columns for the charity function and the player. The problem is that there's no table for the player. As a result you can't put together a referential integrity constraint for the foreign key field that takes either footballers or cricketers. You either have to ignore the referential integrity, or use multiple link tables, one for each of the actual tables in the database. On top of this there's also problems if you can't guarantee key uniqueness.

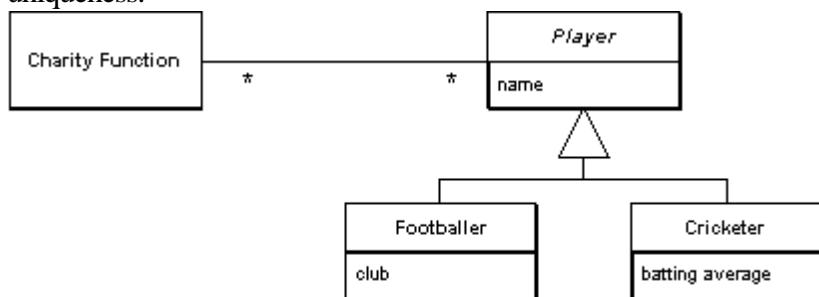


Figure 1:

If you are searching for players with a select statement, you need to look at all concrete tables to see which ones contain the appropriate value. This leads to multiple queries to pull back a single row of data. While this is easy to program, it can hurt performance. You don't suffer the performance hit when you know the class you need, but you do have to prefer to use the concrete class to improve performance.

The pattern is often referred to as something along the lines of **leaf table inheritance**. Some people prefer a variation where you have one table per leaf class, instead of one table per concrete class. If you don't have any concrete superclasses in the hierarchy this ends up as the same thing. Even if you do have concrete superclasses the difference is pretty minor.

## When to Use it

When figuring out how to map inheritance, [Concrete Table Inheritance](#), [Class Table Inheritance](#) and [Single Table Inheritance](#) are the alternatives.

The strengths of [Concrete Table Inheritance](#) are:

- Each table is self contained and doesn't have any irrelevant fields. As a result it makes good sense when used by other applications that aren't using the objects.
- There's no joins to do when reading the data from the concrete mappers.
- Each table is only accessed when that class is accessed, which can spread the access load.

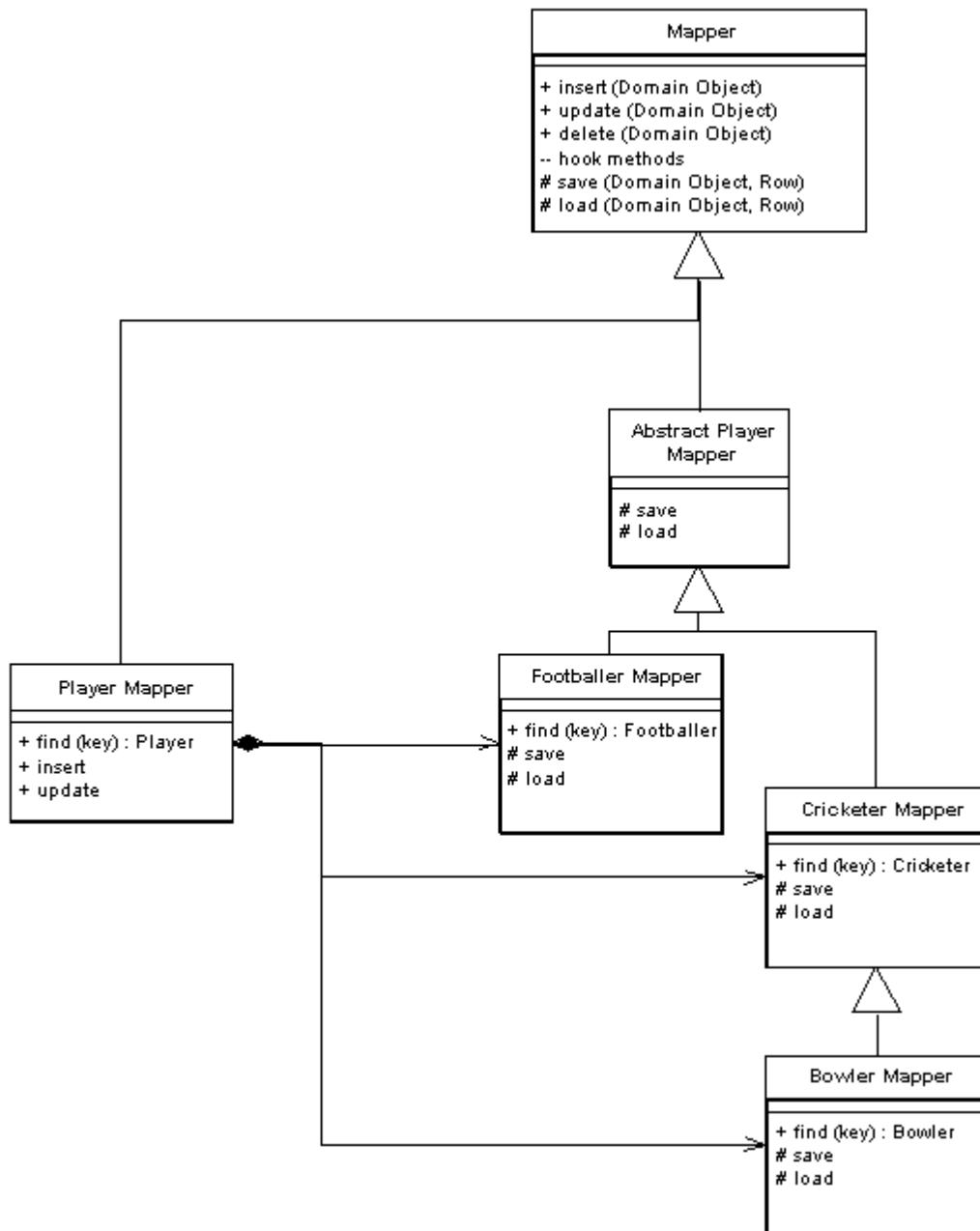
The weaknesses of *Concrete Table Inheritance* are

- Primary keys can be a pain to handle.
- Can't enforce database relationships to abstract classes
- If the fields on the domain classes are pushed up or down the hierarchy, you have to alter the table definitions. You don't have to do as much alteration as with [\*Class Table Inheritance\*](#) but you can't ignore this as you can with [\*Single Table Inheritance\*](#).
- If a superclass field changes, you need to change each table that has this field, since the superclass fields are duplicated across the tables.
- A find on the superclass forces you to check all the tables, which leads to multiple database accesses (or a weird join.)

Remember that the trio of inheritance patterns can coexist in a single hierarchy. So you might use *Concrete Table Inheritance* for one or two subclasses and [\*Single Table Inheritance\*](#) for the rest.

## Example: Concrete Players (C#)

Here I'll show and implementation for the sketch. As with all the inheritance examples in this chapter, I'm using the basic design of classes from [\*Inheritance Mappers\*](#). Figure 2 shows the basic design.



*Figure 2: The generic class diagram of Inheritance Mappers*

Each mapper is linked to the database table that is the source of the data. In ADO.NET a data set holds the data table.

```

class Mapper...
public Gateway Gateway;
private IDictionary identityMap = new Hashtable();
public Mapper (Gateway gateway) {
this.Gateway = gateway;
}
private DataTable table {
get {return Gateway.Data.Tables[TableName];}
}
abstract public String TableName {get;}

```

The gateway class holds onto the data set with in its data property. The data can be loaded up by

supplying suitable queries.

```
class Gateway...
public DataSet Data = new DataSet();
```

Each concrete mapper needs to define what is the name of the table that holds its data.

```
class CricketerMapper...
public override String TableName {
get {return "Cricketers";}
}
```

The player mapper has fields for each concrete mapper.

```
class PlayerMapper...
private BowlerMapper bmapper;
private CricketerMapper cmapper;
private FootballerMapper fmapper;
public PlayerMapper (Gateway gateway) : base (gateway) {
bmapper = new BowlerMapper(Gateway);
cmapper = new CricketerMapper(Gateway);
fmapper = new FootballerMapper(Gateway);
}
```

## Loading an object from the database

Each concrete mapper class has a find method that returns an object given a key value.

```
class CricketerMapper...
public Cricketer Find(long id) {
return (Cricketer) AbstractFind(id);
}
```

The abstract behavior on the superclass finds the right database row for the id, creates a new domain object of the correct type, and uses the load method to load it up (I'll describe the load in a moment.)

```
class Mapper...
public DomainObject AbstractFind(long id) {
DataRow row = FindRow(id);
if (row == null) return null;
else {
DomainObject result = CreateDomainObject();
Load(result, row);
return result;
}
}
private DataRow FindRow(long id) {
String filter = String.Format("id = {0}", id);
DataRow[] results = table.Select(filter);
if (results.Length == 0) return null;
else return results[0];
}
protected abstract DomainObject CreateDomainObject();
class CricketerMapper...
protected override DomainObject CreateDomainObject(){
return new Cricketer();
}
```

The actual loading of data from the database is done by the load method, or rather by several load methods - one for the mapper class and all its superclasses.

```
class CricketerMapper...
protected override void Load(DomainObject obj, DataRow row) {
base.Load(obj, row);
Cricketer cricketer = (Cricketer) obj;
cricketer.battingAverage = (double)row["battingAverage"];
}

class AbstractPlayerMapper...
protected override void Load(DomainObject obj, DataRow row) {
base.Load(obj, row);
Player player = (Player) obj;
player.name = (String)row["name"];
class Mapper...
protected virtual void Load(DomainObject obj, DataRow row) {
obj.Id = (int) row ["id"];
}
```

This logic is the logic for finding an object using a mapper for a concrete class. You can also use a mapper for the superclass: the player mapper. It needs to find an object from whichever table it is living in. Since all the data is already in memory in the data set, I can do it like this.

```
class PlayerMapper...
public Player Find (long key) {
Player result;
result = fmapper.Find(key);
if (result != null) return result;
result = bmapper.Find(key);
if (result != null) return result;
result = cmapper.Find(key);
if (result != null) return result;
return null;
}
```

Remember this is reasonable only because the data is already in memory. If you need to go to the database three times (or more for more subclasses) this will be slow. One way that may help is to do a join across all the concrete tables. This allows you to access the data in one database call. However large joins are often slow in their own right, you'll need to do some benchmarks with your own application to find out what works and what doesn't. Also this will be an outer join, and as well as slow the syntax for this is non-portable and often cryptic.

## Updating an object

The update method can be defined on the mapper superclass.

```
class Mapper...
public virtual void Update (DomainObject arg) {
Save (arg, FindRow(arg.Id));
}
```

Similar to loading, we use a sequence of save methods for each mapper class.

```
class CricketerMapper...
protected override void Save(DomainObject obj, DataRow row) {
base.Save(obj, row);
Cricketer cricketer = (Cricketer) obj;
row["battingAverage"] = cricketer.battingAverage;
}
class AbstractPlayerMapper...
protected override void Save(DomainObject obj, DataRow row) {
Player player = (Player) obj;
row["name"] = player.name;
}
```

The player mapper needs to find the correct concrete mapper to use and then delegate the update call.

```
class PlayerMapper...
public override void Update (DomainObject obj) {
MapperFor(obj).Update(obj);
}
private Mapper MapperFor(DomainObject obj) {
if (obj is Footballer)
return fmapper;
if (obj is Bowler)
return bmapper;
if (obj is Cricketer)
return cmapper;
throw new Exception( "No mapper available");
}
```

## Inserting an object

Insertion is a variation on updating, the extra behavior is creating the new row - this can be done on the superclass.

```
class Mapper...
public virtual long Insert (DomainObject arg) {
DataRow row = table.NewRow();
arg.Id = GetNextID();
row["id"] = arg.Id;
Save (arg, row);
table.Rows.Add(row);
return arg.Id;
}
```

Again the player class delegates to the appropriate mapper

```
class PlayerMapper...
public override long Insert (DomainObject obj) {
return MapperFor(obj).Insert(obj);
}
```

## Deleting an object

Deletion is very straightforward. Again we have a method defined on the superclass

```
class Mapper...
public virtual void Delete(DomainObject obj) {
    DataRow row = FindRow(obj.Id);
    row.Delete();
}
```

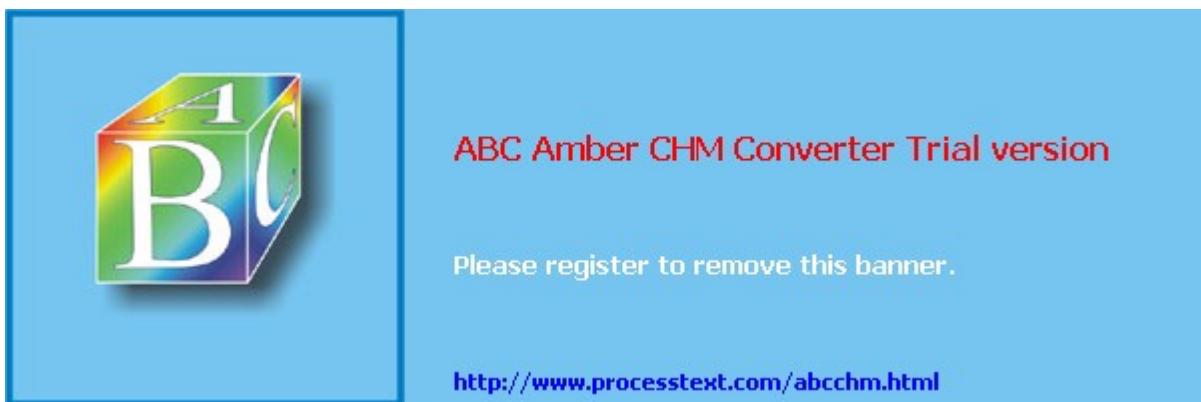
And a delegating method on the player mapper.

```
class PlayerMapper...
public override void Delete (DomainObject obj) {
    MapperFor(obj).Delete(obj);
}
```



---

© Copyright [Martin Fowler](#), all rights reserved

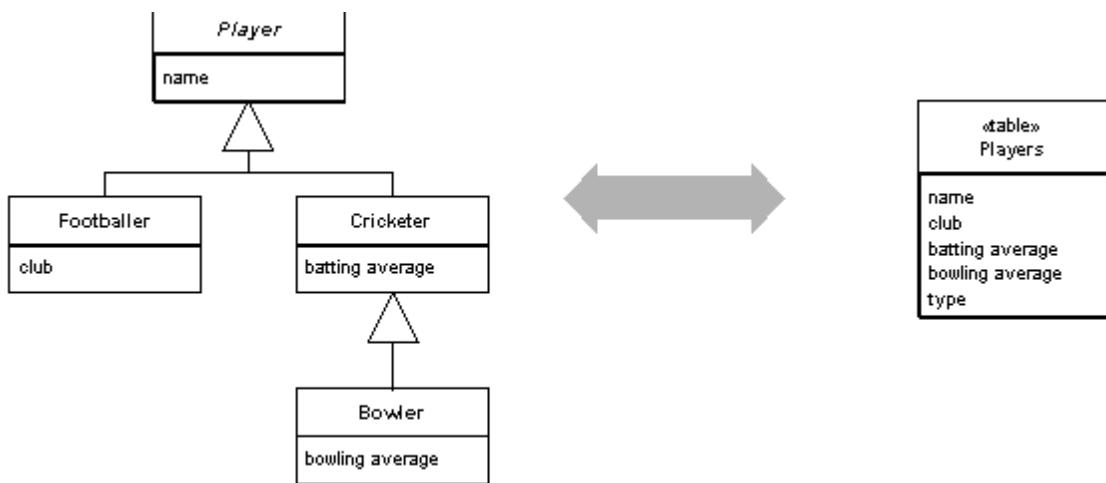


---

# Single Table Inheritance

---

*Represent an inheritance hierarchy of classes as a single table which has fields for all the fields of the various classes*



## How it Works

In this inheritance mapping scheme we have one table that contains all the data for all the classes in the inheritance hierarchy. Each class stores the data that's relevant for that class into one row of the table. Any columns in the database that aren't relevant for the appropriate class are left empty.

The basic mapping behavior follows the general scheme of [Inheritance Mappers](#).

When loading an object into memory you need to know which class to instantiate. To do this you have a field in the table that indicates which class should be used. This can be the name of the class or a code field. A code field needs to be interpreted by some code to map it to the relevant class. This code needs to be extended when a class is added to the hierarchy. If you embed the class name into the table you can just use it directly to instantiate an instance. The class name, however, will take up more space and maybe less easy to process by those using the database table structure directly, as well as more closely coupling the class structure to the database schema.

When loading data, you read the code first to figure out which subclass to instantiate. On saving the data the code needs to be written out by the superclass in the hierarchy.

# When to Use it

*Single Table Inheritance* is one of the options for mapping the fields in an inheritance hierarchy to a relational database. The alternatives are [Class Table Inheritance](#) and [Concrete Table Inheritance](#)

The strengths of *Single Table Inheritance* are:

- A single table to worry about on the database
- No joins in retrieving data
- Any refactoring that pushes fields up or down the hierarchy doesn't require you to change the database.

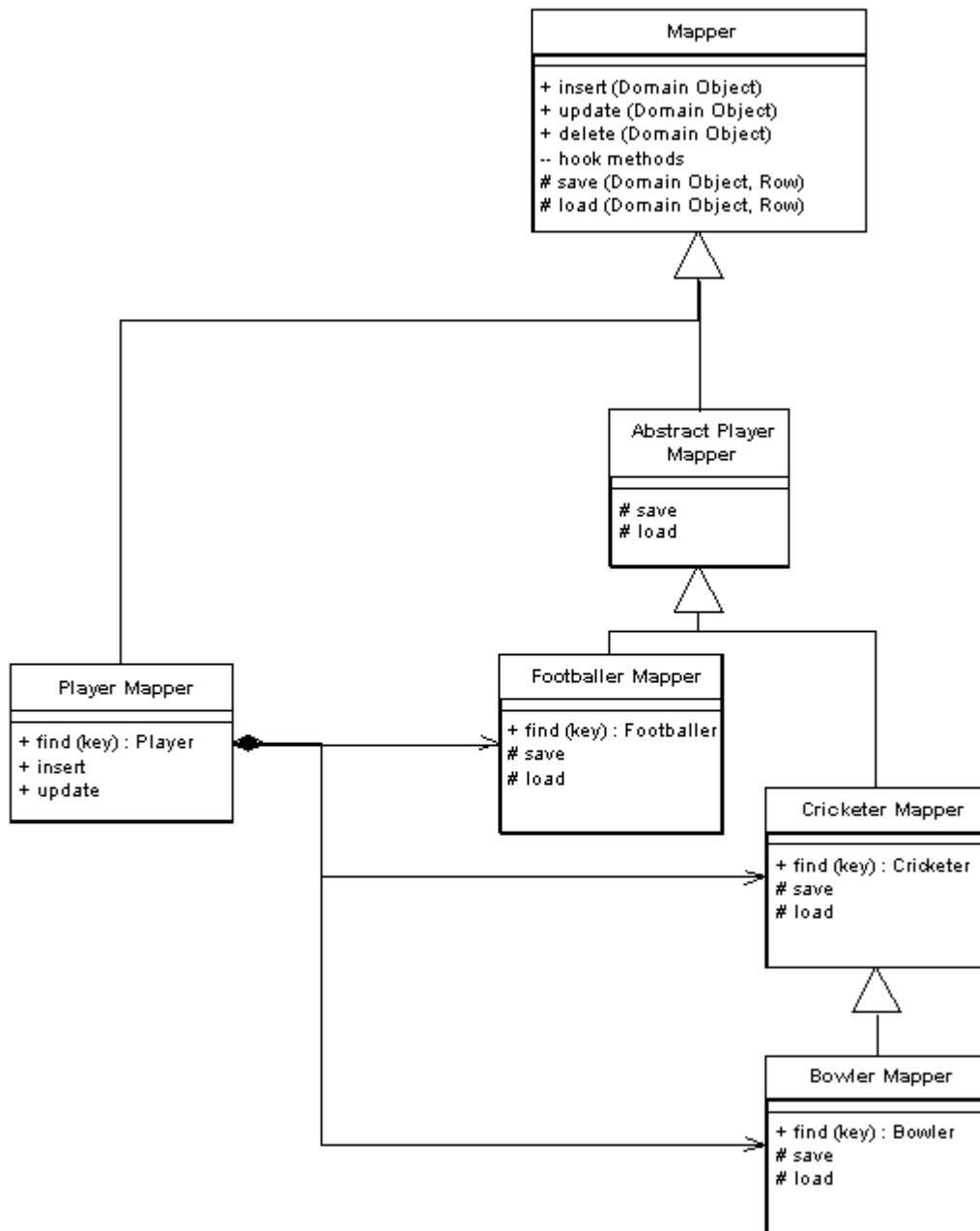
The weaknesses of *Single Table Inheritance* are:

- Fields are sometimes relevant and sometimes not, which can be confusing to people using the tables directly.
- Columns that are only used by some subclasses lead to wasted space in the database. The degree of how much this is actually a problem depends on the specific data characteristics and how well the database compresses empty columns. Oracle, for example, is very efficient about trimming wasted space particularly if you keep your optional columns to the right hand side of the database table. Each database will have its own tricks for this.
- The single table may end up being too large with many indexes and frequent locking. This may hurt performance. You can avoid this by having separate index tables that either list keys of rows that have a certain property, or copy a subset of fields relevant to an index.
- You only have a single name space for fields, so you have to be sure that you don't want to use the same name for different fields. You can do this by using compound names that use the name of the class as a prefix or suffix

It's important to remember that you don't need to use one form of inheritance mapping for your whole hierarchy. It's perfectly fine to map half a dozen similar classes in a single table, but to use [Concrete Table Inheritance](#) for a couple of classes that have a lot of specific data.

## Example: A single table for players (Java)

I've based this code example, like the other inheritance examples, on [Inheritance Mappers](#)



*Figure 1: The generic class diagram of Inheritance Mappers*

Each mapper needs to be linked to a data table in an ADO.NET data set. This link can be made generically in the mapper superclass. The gateway's data property is a data set which can be loaded by a query.

```

class Mapper...
protected DataTable table {
get {return Gateway.Data.Tables[TableName];}
}
protected Gateway Gateway;
abstract protected String TableName {get;}
  
```

Since there is only one table, this can be defined by the abstract player mapper.

```

class AbstractPlayerMapper...
protected override String TableName {
get {return "Players";}
}
  
```

```
}
```

Each class needs a type code to help the mapper code figure out what kind of player it has to deal with. The type code is defined on the superclass and implemented in the subclasses.

```
class AbstractPlayerMapper...
abstract public String TypeCode {get;}
class CricketerMapper...
public const String TYPE_CODE = "C";
public override String TypeCode {
get {return TYPE_CODE;}
}
```

The player mapper has fields for each of the three concrete mapper classes

```
class PlayerMapper...
private BowlerMapper bmapper;
private CricketerMapper cmapper;
private FootballerMapper fmapper;
public PlayerMapper (Gateway gateway) : base (gateway) {
bmapper = new BowlerMapper(Gateway);
cmapper = new CricketerMapper(Gateway);
fmapper = new FootballerMapper(Gateway);
}
```

## Loading an object from the database

Each concrete mapper class has a find method to get an object from the data.

```
class CricketerMapper...
public Cricketer Find(long id) {
return (Cricketer) AbstractFind(id);
}
```

This calls generic behavior to find an object

```
class Mapper...
protected DomainObject AbstractFind(long id) {
DataRow row = FindRow(id);
return (row == null) ? null : Find(row);
}
protected DataRow FindRow(long id) {
String filter = String.Format("id = {0}", id);
DataRow[] results = table.Select(filter);
return (results.Length == 0) ? null : results[0];
}
public DomainObject Find (DataRow row) {
DomainObject result = CreateDomainObject();
Load(result, row);
return result;
}
abstract protected DomainObject CreateDomainObject();
class CricketerMapper...
protected override DomainObject CreateDomainObject() {
return new Cricketer();
}
```

I load the data into the new object with a series of load methods, one on each class in the hierarchy.

```
class CricketerMapper...
protected override void Load(DomainObject obj, DataRow row) {
base.Load(obj, row);
Cricketer cricketer = (Cricketer) obj;
cricketer.battingAverage = (double)row["battingAverage"];
}

class AbstractPlayerMapper...
protected override void Load(DomainObject obj, DataRow row) {
base.Load(obj, row);
Player player = (Player) obj;
player.name = (String)row["name"];
}

class Mapper...
protected virtual void Load(DomainObject obj, DataRow row) {
obj.Id = (int) row ["id"];
}
```

I can also load a player through the player mapper. It needs to read the data and use the type code to determine which concrete mapper to use.

```
class PlayerMapper...
public Player Find (long key) {
DataRow row = FindRow(key);
if (row == null) return null;
else {
String typecode = (String) row["type"];
switch (typecode){
case BowlerMapper.TYPE_CODE:
return (Player) bmapper.Find(row);
case CricketerMapper.TYPE_CODE:
return (Player) cmapper.Find(row);
case FootballerMapper.TYPE_CODE:
return (Player) fmapper.Find(row);
default:
throw new Exception( "unknown type" );
}
}
}
```

## Updating an object

The basic operation for updating an object is the same for all objects, so I can define the operation on the mapper superclass.

```
class Mapper...
public virtual void Update (DomainObject arg) {
Save (arg, FindRow(arg.Id));
}
```

The save method is similar to the load, each class defines it to save the data in that class.

```
class CricketerMapper...
protected override void Save(DomainObject obj, DataRow row) {
```

```
base.Save(obj, row);
Cricketer cricketer = (Cricketer) obj;
row["battingAverage"] = cricketer.battingAverage;
}
class AbstractPlayerMapper...
protected override void Save(DomainObject obj, DataRow row) {
Player player = (Player) obj;
row["name"] = player.name;
row["type"] = TypeCode;
}
```

The player mapper forwards to the appropriate concrete mapper

```
class PlayerMapper...
public override void Update (DomainObject obj) {
MapperFor(obj).Update(obj);
}
private Mapper MapperFor(DomainObject obj) {
if (obj is Footballer)
return fmapper;
if (obj is Bowler)
return bmapper;
if (obj is Cricketer)
return cmapper;
throw new Exception( "No mapper available");
}
```

## Inserting an object

Insertions are similar to updates, the only real difference is that a new row needs to be made in the table before saving

```
class Mapper...
public virtual long Insert (DomainObject arg) {
DataRow row = table.NewRow();
arg.Id = GetNextID();
row["id"] = arg.Id;
Save (arg, row);
table.Rows.Add(row);
return arg.Id;
}
class PlayerMapper...
public override long Insert (DomainObject obj) {
return MapperFor(obj).Insert(obj);
}
```

## Deleting an object

Deletes are pretty simple, defined at the abstract mapper level or in the player wrapper.

```
class Mapper...
public virtual void Delete(DomainObject obj) {
DataRow row = FindRow(obj.Id);
row.Delete();
}
```

```
class PlayerMapper...
public override void Delete (DomainObject obj) {
MapperFor(obj).Delete(obj);
}
```



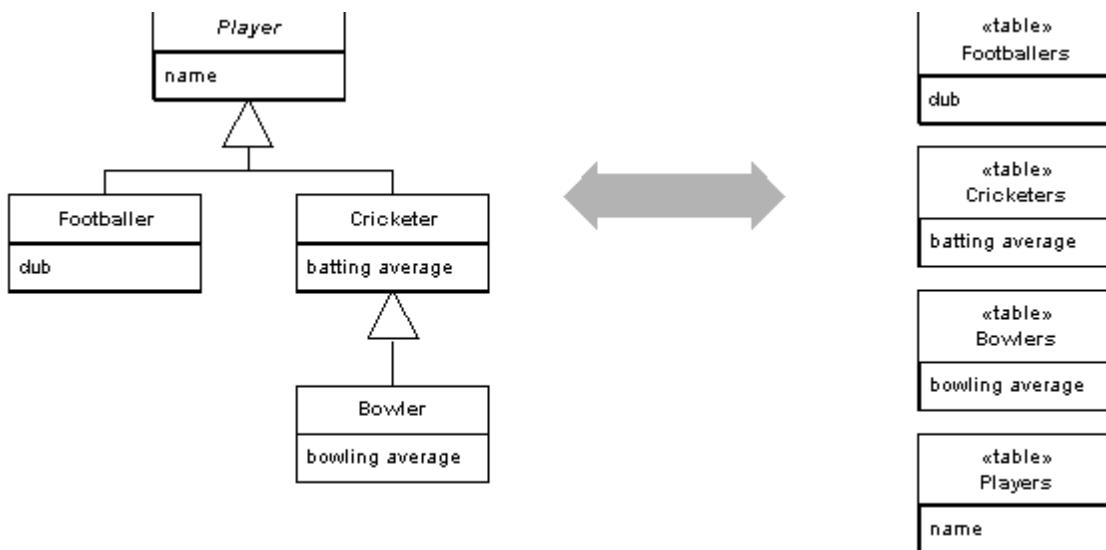
---

© Copyright [Martin Fowler](#), all rights reserved



# Class Table Inheritance

*Represent an inheritance hierarchy of classes with one table for each class*



## How it Works

The straightforward thing about *Class Table Inheritance* is that it has one table per class in the domain model. The fields in the domain class map directly to fields in the corresponding tables.

As with the other inheritance mappings the fundamental approach of [\*Inheritance Mappers\*](#) applies.

One issue is how to link together the corresponding rows of the database tables. One scheme is to use a common primary value, so the row of key 101 in the footballers table and the row of key 101 in the players table correspond to the same domain object. Since the superclass table has a row for each row in the other tables, the primary keys are going to be unique across the tables if you use this scheme. An alternative is to let each table have its own primary and use foreign keys into the superclass table to tie the rows together.

The biggest implementation issue with *Class Table Inheritance* is how to bring the data back from multiple tables in an efficient manner. Obviously making a call for each table isn't nice since you have multiple calls to the database. You can avoid this by doing a join across the various component tables. However joins for more than three or four tables tend to be slow due to the way databases do their

optimizations. You also need to often query the root table first to find what kind of object you need to load and then do a second query to get the full collection of data. To do this well you need to store a type code in the root table, without a type code you need to search each table to get the value.

## When to Use it

*Class Table Inheritance*, [\*Single Table Inheritance\*](#) and [\*Concrete Table Inheritance\*](#) are the three alternatives to consider for inheritance mapping.

The strengths of *Class Table Inheritance* are:

- All columns are relevant for every row so tables are easier to understand and don't waste space
- The relationship between the domain model and the database is very straightforward

The weaknesses of *Class Table Inheritance* are:

- To load an object you need to touch multiple tables, which means a join or multiple queries and sewing in memory
- Any movement of fields up or down the hierarchy causes database changes
- The supertype tables may become a bottleneck as they have to be accessed frequently
- The high normalization may make it hard to understand for ad hoc queries

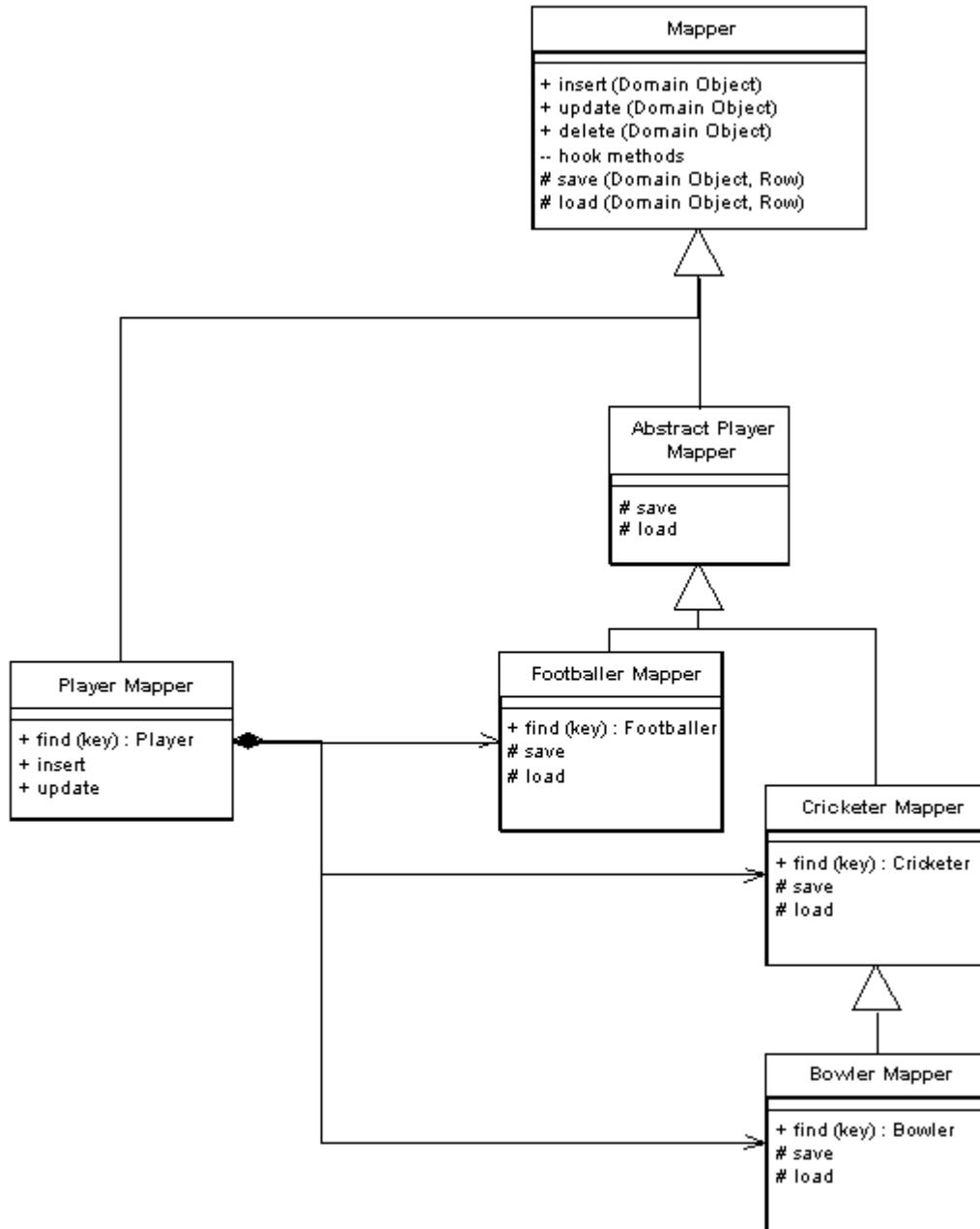
You don't have to choose just one inheritance mapping pattern for one class hierarchy. You could use *Class Table Inheritance* for the classes at the top of the hierarchy and a bunch of [\*Concrete Table Inheritance\*](#) for those lower down.

## Further Reading

A number of IBM texts refer to this as Root-Leaf mapping [\[Brown et al\]](#)

## Example: Players and their kin (C#)

Here's an implementation for the sketch. Again I'll follow the familiar (if perhaps a little tedious) theme of players and the like using [\*Inheritance Mappers\*](#)



*Figure 1: The generic class diagram of Inheritance Mappers*

Each class needs to define the table that holds the data for the class and a type code for the class.

```

class AbstractPlayerMapper...
abstract public String TypeCode {get;}
protected static String TABLENAME = "Players";
class FootballerMapper...
public override String TypeCode {
get {return "F";}
}
protected new static String TABLENAME = "Footballers";

```

Unlike the other inheritance examples, this one does not have a overridden table name, since we have to have the table name for this class even when the instance is an instance of the subclass.

## Loading an object

If you've been reading the other mappings, you'll know the first step is the find method on the concrete mappers.

```
class FootballerMapper...
public Footballer Find(long id) {
return (Footballer) AbstractFind (id, TABLENAME);
}
```

The abstract find method looks for a row matching the key and, if successful, creates a domain object and calls the load method on that object.

```
class Mapper...
public DomainObject AbstractFind(long id, String tablename) {
DataRow row = FindRow (id, tableFor(tablename));
if (row == null) return null;
else {
DomainObject result = CreateDomainObject();
result.Id = id;
Load(result);
return result;
}
protected DataTable tableFor(String name) {
return Gateway.Data.Tables[name];
}
protected DataRow FindRow(long id, DataTable table) {
String filter = String.Format("id = {0}", id);
DataRow[] results = table.Select(filter);
return (results.Length == 0) ? null : results[0];
}
protected DataRow FindRow (long id, String tablename) {
return FindRow(id, tableFor(tablename));
}
protected abstract DomainObject CreateDomainObject();
class FootballerMapper...
protected override DomainObject CreateDomainObject(){
return new Footballer();
}
```

There is one load method for each class which loads the data defined by that class.

```
class FootballerMapper...
protected override void Load(DomainObject obj) {
base.Load(obj);
DataRow row = FindRow (obj.Id, tableFor(TABLENAME));
Footballer footballer = (Footballer) obj;
footballer.club = (String)row["club"];
}
class AbstractPlayerMapper...
protected override void Load(DomainObject obj) {
DataRow row = FindRow (obj.Id, tableFor(TABLENAME));
Player player = (Player) obj;
player.name = (String)row["name"];
}
```

As with the other sample code, but more noticeably in this case, I'm relying on the fact that the ADO.NET data set has brought the data from the database and cached it into memory. This allows me to make several accesses to the table based data structure without a high performance cost. If you are going directly to the database, you'll need to reduce that load. For this example you might do this by creating a join across all the tables and manipulating that.

The player mapper determines which kind of player it needs to find, and then delegates the correct concrete mapper.

```
class PlayerMapper...
public Player Find (long key) {
    DataRow row = FindRow(key, tableFor(TABLENAME));
    if (row == null) return null;
    else {
        String typecode = (String) row["type"];
        if (typecode == bmapper.TypeCode)
            return bmapper.Find(key);
        if (typecode == cmapper.TypeCode)
            return cmapper.Find(key);
        if (typecode == fmapper.TypeCode)
            return fmapper.Find(key);
        throw new Exception("unknown type");
    }
}
protected static String TABLENAME = "Players";
```

## Updating an object

The update method appears on the mapper superclass

```
class Mapper...
public virtual void Update (DomainObject arg) {
    Save (arg);
}
```

It's implemented through a series of save methods, one for each class in the hierarchy.

```
class FootballerMapper...
protected override void Save(DomainObject obj) {
    base.Save(obj);
    DataRow row = FindRow (obj.Id, tableFor(TABLENAME));
    Footballer footballer = (Footballer) obj;
    row["club"] = footballer.club;
}

class AbstractPlayerMapper...
protected override void Save(DomainObject obj) {
    DataRow row = FindRow (obj.Id, tableFor(TABLENAME));
    Player player = (Player) obj;
    row["name"] = player.name;
    row["type"] = TypeCode;
}
```

The player mapper's update method overrides the general method to forward to the correct concrete mapper.

```
class PlayerMapper...
public override void Update (DomainObject obj) {
MapperFor(obj).Update(obj);
}
private Mapper MapperFor(DomainObject obj) {
if (obj is Footballer)
return fmapper;
if (obj is Bowler)
return bmapper;
if (obj is Cricketer)
return cmapper;
throw new Exception( "No mapper available");
}
```

## Inserting an object

The method for inserting an object is declared on the mapper superclass. It has two stages, first creating new database rows to hold the data, secondly using the save methods to update these blank rows with the necessary data.

```
class Mapper...
public virtual void Update (DomainObject arg) {
Save (arg);
}
```

Each class inserts a row into its table

```
class FootballerMapper...
protected override void AddRow (DomainObject obj) {
base.AddRow(obj);
InsertRow (obj, tableFor(TABLENAME));
}

class AbstractPlayerMapper...
protected override void AddRow (DomainObject obj) {
InsertRow (obj, tableFor(TABLENAME));
}

class Mapper...
abstract protected void AddRow (DomainObject obj);
protected virtual void InsertRow (DomainObject arg, DataTable table) {
DataRow row = table.NewRow();
row["id"] = arg.Id;
table.Rows.Add(row);
}
```

The player mapper delegates to the appropriate concrete mapper

```
class PlayerMapper...
public override long Insert (DomainObject obj) {
return MapperFor(obj).Insert(obj);
}
```

## Deleting an object

To delete an object, each class deletes a row from the corresponding table in the database.

```
class FootballerMapper...
public override void Delete(DomainObject obj) {
base.Delete(obj);
DataRow row = FindRow(obj.Id, TABLENAME);
row.Delete();
}
class AbstractPlayerMapper...
public override void Delete(DomainObject obj) {
DataRow row = FindRow(obj.Id, tableFor(TABLENAME));
row.Delete();
}
class Mapper...
public abstract void Delete(DomainObject obj);
```

The player mapper, again wimps out of all the hard work and just delegates to the concrete mapper.

```
class PlayerMapper...
override public void Delete(DomainObject obj) {
MapperFor(obj).Delete(obj);
}
```



---

© Copyright [Martin Fowler](#), all rights reserved

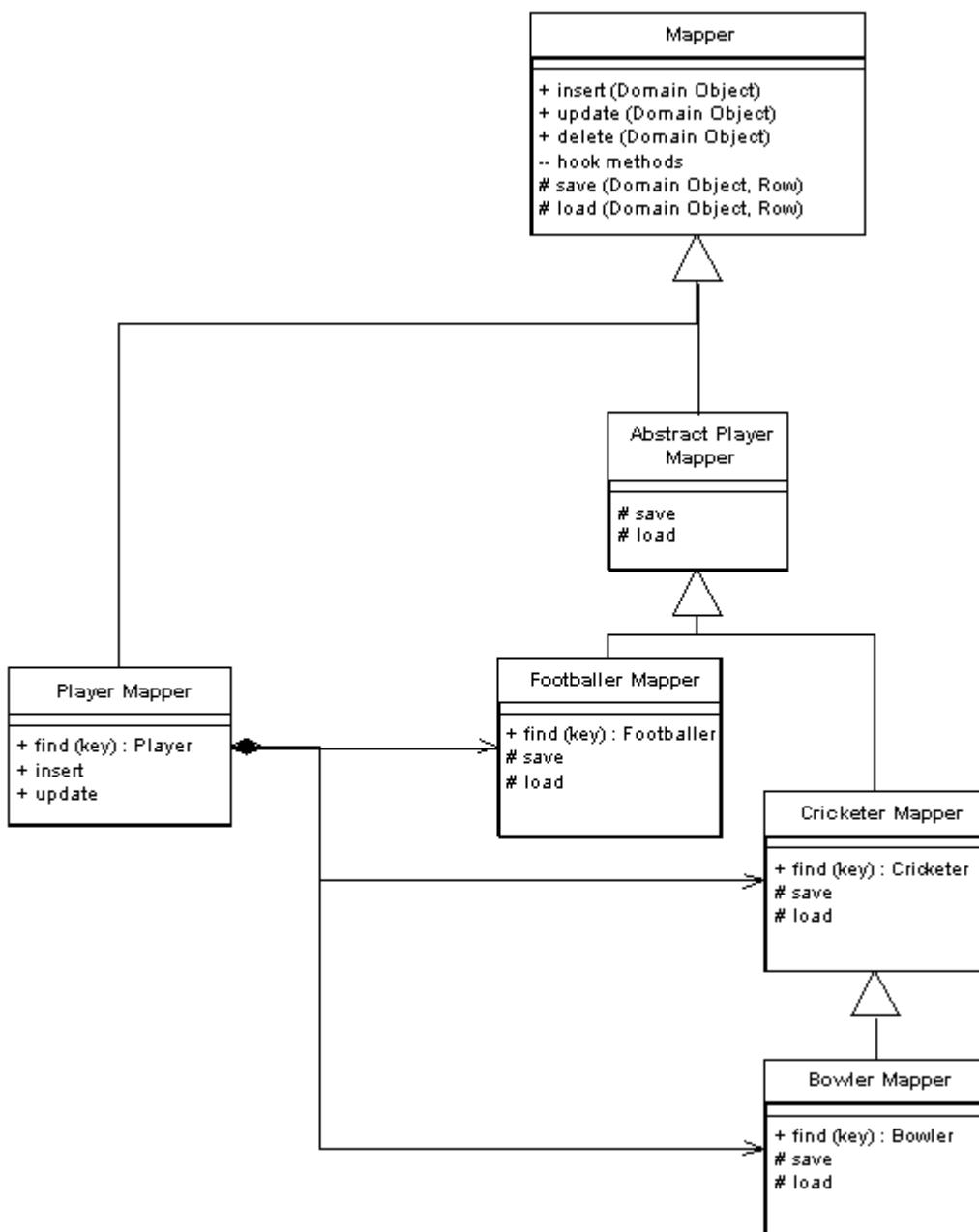


---

# Inheritance Mappers

---

*A set of mappers to handle inheritance hierarchies*



When you are mapping from an object-oriented inheritance hierarchy in memory to a relational database

you need to minimize the amount of code needed to save and load the data to database. You also want to provide both abstract and concrete mapping behavior that allows you to save or load a superclass or a subclass.

Although the details of this behavior varies with your inheritance mapping scheme ([Single Table Inheritance](#), [Class Table Inheritance](#), [Concrete Table Inheritance](#)) the general structure works the same for all of them.

## How it Works

You can organize the mappers using a hierarchy so that each domain class has a mapper that saves and loads the data for that domain class. That way when the mapping changes, you have one point where you can change the mapping. This approach works well for concrete mappers that know how to map the concrete objects in the hierarchy. There are times, however, where you also need mappers for the abstract classes in the hierarchy. These can be implemented with mappers that are actually outside of the basic hierarchy but delegate to the appropriate concrete mappers.

To best understand how this works, I'll start with the concrete mappers. In the sketch the concrete mappers are the mappers for footballer, cricketer, and bowler. The basic behavior of the mappers include the find, insert, update, and delete operations.

The finder methods are declared on the concrete subclasses because they will return a concrete class, so the find method on BowlerMapper should return a Bowler not an abstract class. Common OO languages cannot let you change the declared subtype of a method, so it's not possible to inherit the find operation and still declare a specific return type. You can, of course, return an abstract type but that forces the user of the class to downcast - which is best to avoid. (A language with run time typing avoids this problem.)

The basic behavior of the find method is to find the appropriate row in the database, instantiate an object of the correct type (a decision that's made by the subclass) and then load the object with data from the database. The load method is implemented by each mapper in the hierarchy and the mapper loads the behavior for its corresponding domain object. So the bowler mapper's load method loads the data specific to the bowler class, calls the superclass method to load the data specific to the cricketer, which calls its superclass method, etc.

The insert and update methods operate in a similar way using a save method. Here you can define the interface on the superclass, indeed on a [Layer Supertype](#). The insert method creates a new row and then saves the data from the domain object using the save hook methods. The update method just saves the data using the save hook methods. The save hook methods operate similarly to the load hook methods, with each class storing its specific data and calling the superclass save method.

This scheme makes it quite easy to write the appropriate mappers to save the specific information needed for a particular part of the hierarchy. The next step is to support loading and saving an abstract class, in this example a player. While a first thought is to put appropriate methods on the superclass mapper, that actually gets awkward because while concrete mapper classes can just use the abstract mapper's insert and update methods, the player mapper's insert and update needs to override these to call a concrete mapper instead. The result is one of those combinations of generalization and composition that twists your brain cells into an appalling knot.

So I prefer to separate them into two classes. The abstract player mapper is the one whose responsibility is to load and save the specific player data to the database. This is a class that is abstract and whose behavior is just used by the concrete mapper objects. A separate player mapper class is used for its interface. It provides a find method and overrides the insert and update methods. For all of these its responsibility is to figure out which concrete mapper should handle the task and delegate to it.

Although this broad scheme makes sense for each type of inheritance mapping, the details do vary with the exact mapping scheme, so it's not possible to show a code example for this case. You can find good examples in each of the inheritance mapping patterns: [Single Table Inheritance](#), [Class Table Inheritance](#), and [Concrete Table Inheritance](#).

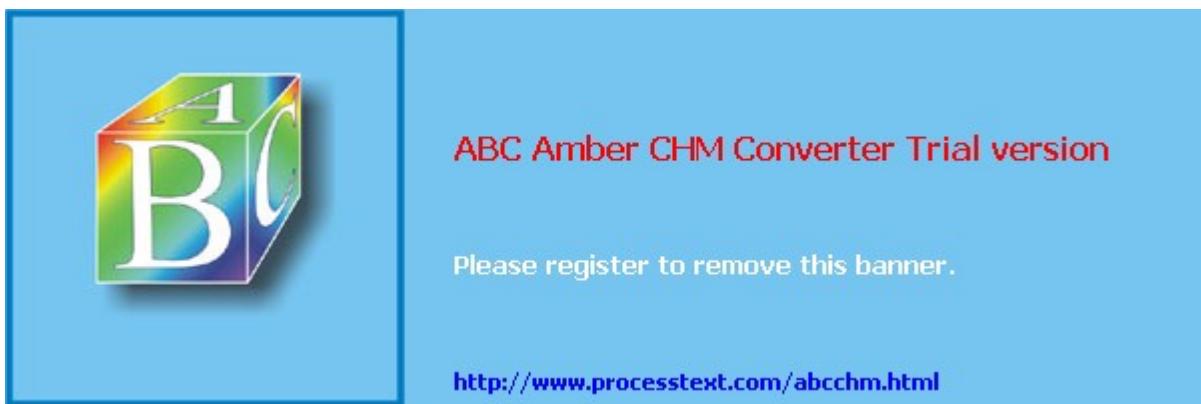
## When to Use it

This general scheme makes sense for any inheritance based database mapping. The alternatives involve such things as duplicating superclass mapping code amongst the concrete mappers or folding the player's interface into the abstract player mapper class. The former is a heinous crime and latter is possible but leads to a messy and confusing player mapper class. So on the whole its hard to think of a good alternative to this pattern.



---

© Copyright [Martin Fowler](#), all rights reserved

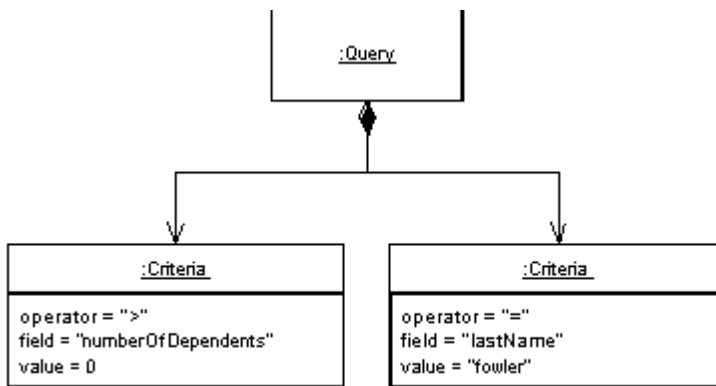


---

# Query Object

---

*An object that represents a database query*



SQL can be an involved language, and many developers are not particularly familiar with it. Furthermore to form queries, you need to know what the database schema looks like. You can avoid this by creating specialized finder methods that hide the SQL inside parameterized methods, but that makes it difficult to form more ad hoc queries. It also leads to duplication in the SQL statements, should the database schema change.

A *Query Object* is an [interpreter](#) that is a structure of objects, but is able to form itself into a SQL query. You can create this query by referring to classes and their fields, rather than tables and columns. That way those who write the queries can do so independently of the database schema and changes to the schema can be localized into a single place.

## How it Works

A *Query Object* is an application of the [interpreter](#) geared to represent a SQL query. The primary roles of the *Query Object* is to allow a client to form queries of various kinds, and then to turn that object structure into the appropriate SQL string.

In order to represent any query, you need quite a flexible *Query Object*. Often, however, applications can make do with a lot less than the full power of SQL, in which case you build a simpler *Query Object*. While it won't be able to represent anything, it can satisfy your particular needs and if you need to enhance it, it's usually no more work to enhance when you need more capability than it is to create a fully capable *Query Object* right from the beginning. As a result you should only create a minimally functional

*Query Object* for your current needs, and evolve it as your needs grow.

A common feature of *Query Object* is that they can represent queries in the language of the in-memory objects rather than the database schema. So instead of using table and column names, you can use object and field names. While this isn't important if your objects and database have the same structure, it can be very useful if you get variations between the two. In order to perform this change of view, the *Query Object* needs to know how the database structure maps to the object structure, so this capability really needs [Metadata Mapping](#).

If you have multiple databases to work with, you can design your *Query Object* so that it can produce different SQL depending on which database the query is running against. At its simplest level, this can take into account the annoying differences in SQL syntax that keep cropping up. At a more ambitious level this can use different mappings to cope with the same classes being stored in different database schemas.

A particularly sophisticated use of *Query Object* is using it to eliminate redundant queries against a database. At its simplest level you can look to see if you have run the same query earlier on in the session. If so then you can use the query to select objects from the [Identity Map](#) and avoid a trip to the database. A more sophisticated approach can detect whether one query is a particular case of an earlier query, such as a query that is the same as an earlier one but with an additional clause linked with an AND.

Exactly how to do these more sophisticated features is beyond the scope of this book. But these are the kind of features that OR mapping tools may provide.

A variation on the *Query Object* that Cocobase does, is to allow a query to be specified by an example domain object. So you might have a Person object whose last name is set to "Fowler" but all other attributes are set to null. You can then treat this domain object as a query by example that's processed in a similar way to the interpreter style *Query Object*, that would return all people in the database whose last name is "Fowler".

## When to Use it

*Query Objects* are a pretty sophisticated pattern to put together, so most projects don't use them if they have a hand built data source layer. You only really need them when you are using [Domain Model](#) and [Data Mapper](#), you also really need [Metadata Mapping](#) to make real use of them.

Even then they aren't always necessary. Many developers are comfortable with SQL. You can hide much of the details of the database schema behind specific finder methods.

The advantages of *Query Object* come with more sophisticated needs: keeping database schema encapsulated, supporting multiple databases, supporting multiple schemas, and optimizing to avoid multiple queries. Although some projects with a particularly sophisticated data source team might want to build these themselves, most people that use *Query Object* do so with a commercial tool. My inclination is that if you want these capabilities, you almost always better off buying a tool.

Having said all that, you may find that limited *Query Object* may fulfill your needs and not be that difficult to build on a project that wouldn't justify a fully featured *Query Object*. The trick to this is paring

down the functionality of the *Query Object* and being determined to not build more than you actually use.

## Example: A simple *Query Object* (Java)

This is a simple example of a *Query Object*, rather less than would be useful for most situations, but enough to give you an idea of what a *Query Object* is about.

This *Query Object* is able to query a single table based on set of criteria that are'ded together, (or in slightly more technical language it handles a conjunction of elementary predicates.)

The *Query Object* is set up using the language of domain objects rather than that of the table structure. So a query knows the class the query is for and a collection of criteria that'll correspond to the clauses of a where clause.

```
class QueryObject...
private Class klass;
private List criteria = new ArrayList();
```

A simple form of criteria is one that takes a field, a value, and a SQL operator to compare them.

```
class Criteria...
private String sqlOperator;
protected String field;
protected Object value;
```

To make it easier to create the right criteria, I can provide an appropriate creation method.

```
class Criteria...
public static Criteria greaterThan(String fieldName, int value)  {
return Criteria.greaterThan(fieldName, new Integer(value));
}
public static Criteria greaterThan(String fieldName, Object value)  {
return new Criteria(" > ", fieldName, value);
}
private Criteria(String sql, String field, Object value) {
this.sqlOperator = sql;
this.field = field;
this.value = value;
}
```

This allows me to find everyone with dependents by forming a query such as

```
class Criteria...
public static Criteria greaterThan(String fieldName, int value)  {
return Criteria.greaterThan(fieldName, new Integer(value));
}
public static Criteria greaterThan(String fieldName, Object value)  {
return new Criteria(" > ", fieldName, value);
}
private Criteria(String sql, String field, Object value) {
this.sqlOperator = sql;
this.field = field;
this.value = value;
```

```
}
```

So if I have a person object such as this

```
class Person...
private String lastName;
private String firstName;
private int numberOfDependents;
```

I can form a query that asks for all people with dependents by creating a query for person and adding a criteria

```
QueryObject query = new QueryObject(Person.class);
query.addCriteria(Criteria.greaterThan("numberOfDependents", 0));
```

That's enough for to describe the query, the query then needs to execute by turning itself into a SQL select. In this case I assume my mapper class supports a method that will find objects based on a string which is a where clause.

```
class QueryObject...
public Set execute(UnitOfWork uow) {
this.uow = uow;
return uow.getMapper(klass).findObjectsWhere(generateWhereClause());
}

class Mapper...
public Set findObjectsWhere (String whereClause) {
String sql = "SELECT" + dataMap.columnList() + " FROM " +
dataMap.getTableName() + " WHERE " + whereClause;
PreparedStatement stmt = null;
ResultSet rs = null;
Set result = new HashSet();
try {
stmt = DB.prepare(sql);
rs = stmt.executeQuery();
result = loadAll(rs);
} catch (Exception e) {
throw new ApplicationException (e);
} finally {DB.cleanUp(stmt, rs);
}
return result;
}
```

Here I'm using a [Unit of Work](#) that holds mappers indexed by the class and a mapper that uses [Metadata Mapping](#)

To generate the where clause, the query iterates through the criteria and gets each one to print itself out and tied them together with ANDs

```
class QueryObject...
private String generateWhereClause() {
StringBuffer result = new StringBuffer();
for (Iterator it = criteria.iterator(); it.hasNext(); ) {
Criteria c = (Criteria)it.next();
if (result.length() != 0)
```

```
result.append(" AND ");
result.append(c.generateSql(uow.getMapper(klass).getDataMap()));
}
return result.toString();
}
class Criteria...
public String generateSql(DataMap dataMap) {
    return dataMap.getColumnForField(field) + sqlOperator + value;
}
```

As well as criteria with simple SQL operators, we can create more complex criteria classes that do a little more. So consider a case insensitive pattern match query, such as finding all people whose last names start with 'f'. We can form a query object for all people with dependents whose name starts with 'f'.

```
QueryObject query = new QueryObject(Person.class);
query.addCriteria(Criteria.greaterThan("numberOfDependents", 0));
query.addCriteria(Criteria.matches("lastName", "f%"));
```

This uses a different criteria class that forms a more complex clause into the where statement.

```
class Criteria...
public static Criteria matches(String fieldName, String pattern){
    return new MatchCriteria(fieldName, pattern);
}
class MatchCriteria extends Criteria...
public String generateSql(DataMap dataMap) {
return "UPPER(" + dataMap.getColumnForField(field) + ") LIKE UPPER('" +
value + "'")";
}
```



---

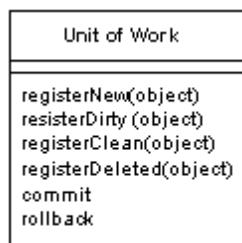
© Copyright [Martin Fowler](#), all rights reserved



# Unit of Work

---

*Maintains a list of objects that are affected by a business transaction and coordinates the writing out of changes and resolution of concurrency problems.*



When you're pulling data in and out of a database, it's important to keep track of what you've changed, otherwise you won't get stuff written back into the database. Similarly you have to insert new objects you create and remove any object you've deleted.

You could change the database with each change to your object model, but this can lead to lots of very small calls to the database, which ends up being very slow. Furthermore it requires you to have a transaction open for the whole interaction - which is impractical if you have a business transaction that spans multiple requests. This situation is even worse if you need to keep track of objects you've read so you can avoid inconsistent reads.

A *Unit of Work* keeps track of everything you do during a business transaction that can have ramifications to the database. When you are done, it figures out all everything that needs to be done to alter the database as a result of all the work.

## How it Works

The obvious things that cause you to deal with the database are changes: new object created and existing ones updated or deleted. *Unit of Work* is an object that keeps track of these things. As soon as you start doing something that may affect a database, you create a *Unit of Work* to keep track of the changes. Every time you create, change, or delete an object you tell the *Unit of Work*. You can also let the *Unit of Work* know about objects you've read. This way the *Unit of Work* can check for inconsistent reads by verifying that none of the objects changed on the database during the business transaction.

The key thing about *Unit of Work* is that when it comes time to commit, the *Unit of Work* decides

what to do. It opens a transaction, does any concurrency checking (using [Pessimistic Offline Lock](#) or [Optimistic Offline Lock](#)) and writes changes out to the database. Application programmers never explicitly call methods to update the database. This way they don't have to keep track of what's changed, nor do they need to worry about how referential integrity affects the order in which they need to do things.

Of course for this to work the *Unit of Work* needs to know what objects it should keep track of. You can do this either by the caller doing it or by getting the object to tell the *Unit of Work*.

With **caller registration** the user of an object needs to remember to register the object with the *Unit of Work* for changes. Any objects that aren't registered won't get written out on commit. Although this allows forgetfulness to cause trouble, it does give flexibility in allowing people to make in-memory changes that they don't want written out - although we would argue that is going to cause far more confusion than would be worthwhile. It's better to make an explicit copy for that purpose.

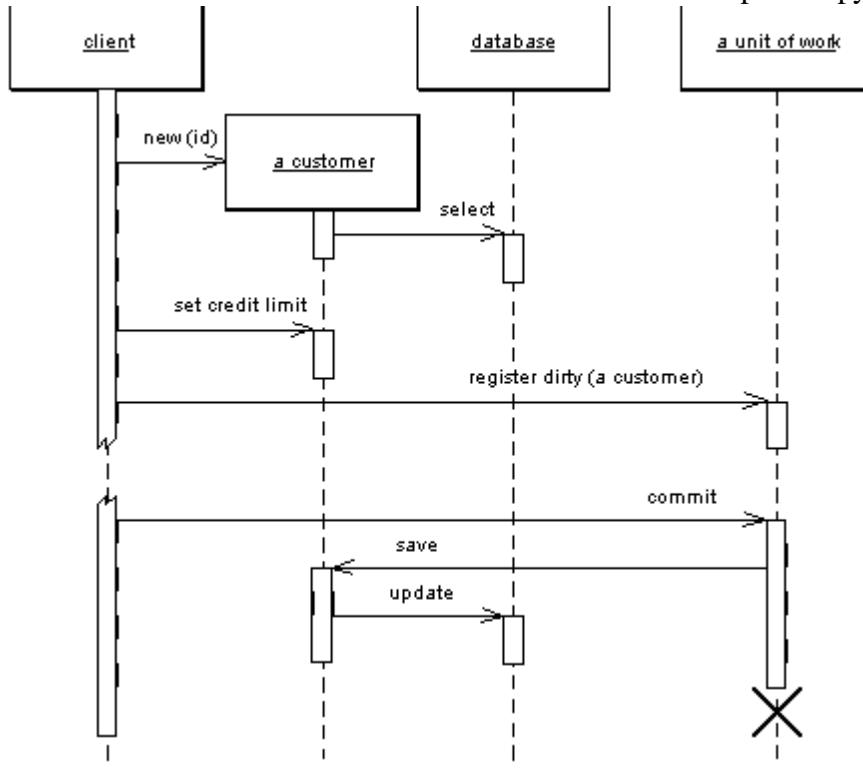


Figure 1: Having the caller register a changed object

With **object registration** the onus is removed from the caller. The usual trick here is to place registration methods in object methods. Loading an object from the database registers the object as a clean object, the setting methods cause the object to be registered as dirty. For this scheme to work the *Unit of Work* needs to either be passed to the object or to be in a well known place. Passing the *Unit of Work* around is tedious, but it's usually no problem to have the *Unit of Work* present in some kind of session object.

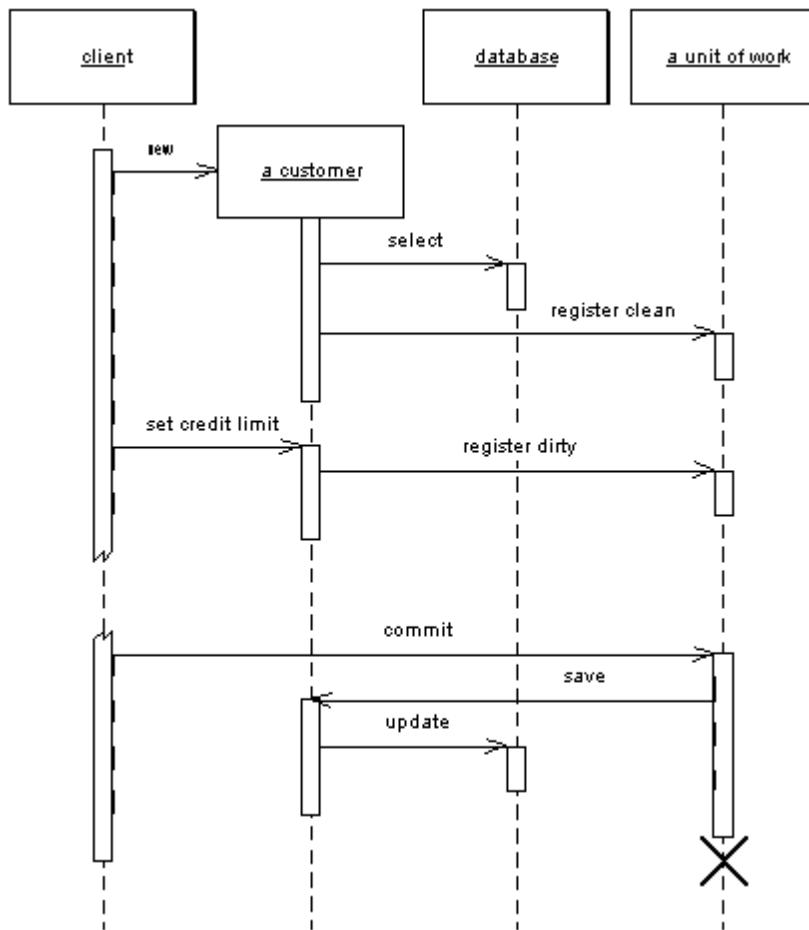


Figure 2: Getting the receiver object to register itself

Even object registration leaves something to remember, that is the developer of the object needs to remember to add a registration call in the right places. The consistency becomes habitual, but is still an awkward bug when it's missed.

This is a natural place for code generation to generate appropriate calls, but that only works when you can clearly separate generated and non-generated code. This turns out to be particularly suited to aspect-oriented programming. I've also come across post-processing of the object files to pull this off. In this example a post-processor examined all the Java .class files, looked for the appropriate methods and inserted registration calls into the byte code. This kind of finicking around feels dirty, but it separates the database code from the regular code. Aspect-oriented approach will do this more cleanly with source code, and as aspect oriented programming tools become more commonplace I would expect to see this strategy being used.

Another technique I've seen is **unit of work controller**, which the TOPLink product uses. Here the *Unit of Work* handles all reads from the database, and registers clean objects whenever they are read. Rather than marking objects as dirty the *Unit of Work* takes a copy at read time, and then compares the object at commit time. Although this adds overhead to the commit process, it allows a selective update of only those fields that were actually changed, as well as avoiding registration calls in the domain objects. A hybrid approach is to take copies only of objects that are changed, while this requires registration, it supports selective update and greatly reduces the overhead of the copy if there many more reads than updates.

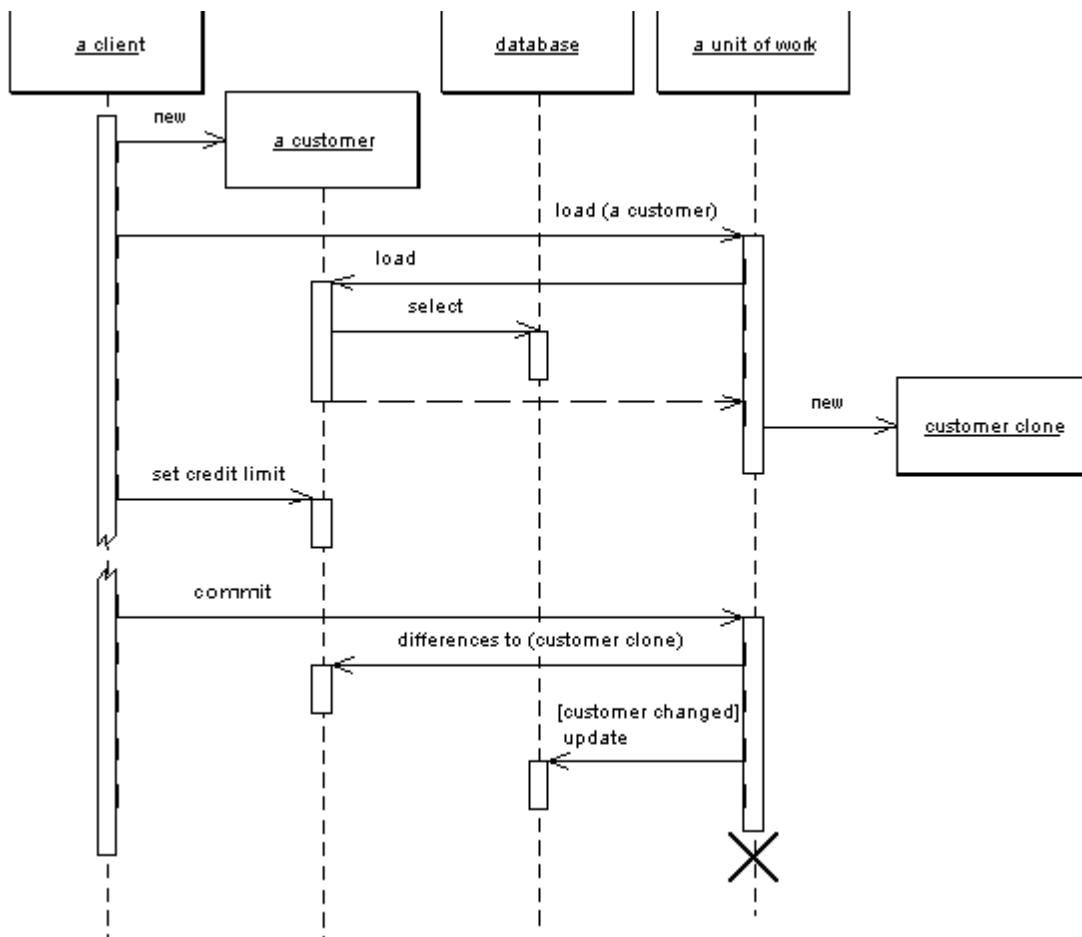


Figure 3: Using the Unit of Work as the controller for the database access

Creating an object is often a special time to consider caller registration. It's not uncommon for people to want to create objects that are only supposed to transient. A good example of this is in testing domain objects where the tests will run much faster without database writes. Caller registration can make this apparent. However there are other solutions, such as providing a transient constructor that doesn't register with the unit or work, or better still providing a [Special Case Unit of Work](#) that does nothing with a commit.

Another area where a *Unit of Work* can be helpful is in update order when a database uses referential integrity. Most of the time you can avoid this issue by ensuring that the database only checks referential integrity when the transaction commits, rather than with each SQL call. Most databases allow it, and if available there's no good reason not to do it. If you can't then the *Unit of Work* is the natural place to sort out the update order. In smaller systems this can be done with explicit code that contains details about which tables to write first based on the foreign key dependencies. In a larger application it's better to use metadata to figure out which order to write to the database. How you do that is beyond the scope of this book, and a common reason to use a commercial tool. If you have to do it yourself I'm told the key to the puzzle is a topological sort.

You can use a similar technique to minimize deadlocks. If every transaction uses the same sequence of tables to edit, then you greatly reduce the chances of getting deadlocks. The *Unit of Work* is an ideal place to hold a fixed sequence of table writes so that you always touch the tables in the same order.

Objects need to be able to find their current *Unit of Work*. A good way to do this is to use a thread scoped [Registry](#). Another approach is to pass the *Unit of Work* around to objects that need it, either in

method calls or by passing in the *Unit of Work* when you create an object. In either case make sure you can't get more than one thread getting access to a *Unit of Work*, there lies the way to madness.

*Unit of Work* makes an obvious point to handle batch updates. The idea behind a **batch update** is to send multiple SQL commands as a single unit so that they can be processed in a single remote call. This is particularly important for updates where many updates, inserts, and deletes are often sent in rapid succession. Different environments provide different levels of support for batch updates. JDBC has a batch update facility which allows you to add individual statements to another statement to batch them up. If you don't have this you can mimic this capability by building up a string that has multiple SQL statements and then submitting that string as one statement, [missing reference] describes an example of this for the Microsoft platforms. However if you do this, check to see if it interferes with statement pre-compilation.

*Unit of Work* works with any transactional resource, not just databases, so you can also use it to coordinate with message queues and transaction monitors.

## .NET

The *Unit of Work* in .NET is done by the disconnected data set, which is a slightly different *Unit of Work* to the classical variety. Most *Unit of Work* I've come across register and track changes to objects. .NET reads data from the database into a data set, which is a series of objects arranged like the tables, rows and columns of the database. The data set is essentially an in-memory mirror image of the result of one or SQL queries. Each DataRow has the concept of a version (current, original, proposed) and a state (unchanged, added, delete, modified). This information, together with the fact that the data set mimics the database structure, makes it straightforward to write changes out to the database.

## When to Use it

The fundamental problem that *Unit of Work* deals with is keeping track of the various objects that you've manipulated, so that you know which objects you need to consider to synchronize your in-memory data with the database. If you are able to do all your work within a system transaction, then the only objects you need to worry about are those you alter. Although *Unit of Work* is generally the best way of doing this, there are alternatives.

Perhaps the simplest alternative is to explicitly save any object whenever you alter it. The problem here is that you may get many more database calls than you would like, since if you alter one object in three different points in your work, then you get three calls rather than one call in its final state.

To avoid multiple database calls, you can leave all your updates to the end. To do this you need to keep track of all the objects that have changed. You can do this with variables in your code, but this soon becomes unmanageable once you have more than a few. Such variables often can work fine with a [\*Transaction Script\*](#), but is very difficult with a [\*Domain Model\*](#).

Rather than keep objects in variables you can give each object a dirty flag which you set when the object changes. Then you need to find all the dirty objects at the end of your transaction and write them out. The value of this technique hinges on how easy it is to find them. If all your objects are in a single

hierarchy, then you can traverse the hierarchy and write out any object that's been changed. But a more general object network, such as a [Domain Model](#), is harder to traverse.

The great strength of *Unit of Work* is that it keeps all this information in one place. Once you have *Unit of Work* working for you, you really don't have to remember to do much in order to keep track of your changes. *Unit of Work* also is a firm platform for more complicated situations, such as handling business transactions that span several system transactions using [Optimistic Offline Lock](#) and [Pessimistic Offline Lock](#).

## Example: *Unit of Work* with Object Registration (Java)

by David Rice

Here's a unit of work that can track all changes for a given business transaction and then commit them to the database when instructed to do so. Our domain layer has a [Layer Supertype](#), DomainObject, with which the unit of work will interact. To store the change set we will use three lists to store new, dirty, and removed domain objects.

```
class UnitOfWork...
private List newObjects = new ArrayList();
private List dirtyObjects = new ArrayList();
private List removedObjects = new ArrayList();
```

The registration methods maintain the state of these lists. These methods must perform basic assertions such as checking that an id is not null or that a dirty object is not being registered as new.

```
class UnitOfWork...
public void registerNew(DomainObject obj) {
    Assert.notNull("id not null", obj.getId());
    Assert.isTrue("object not dirty", !dirtyObjects.contains(obj));
    Assert.isTrue("object not removed", !removedObjects.contains(obj));
    Assert.isTrue("object not already registered new", !newObjects.contains(obj));
    newObjects.add(obj);
}

public void registerDirty(DomainObject obj) {
    Assert.notNull("id not null", obj.getId());
    Assert.isTrue("object not removed", !removedObjects.contains(obj));
    if (!dirtyObjects.contains(obj) && !newObjects.contains(obj)) {
        dirtyObjects.add(obj);
    }
}

public void registerRemoved(DomainObject obj) {
    Assert.notNull("id not null", obj.getId());
    if (newObjects.remove(obj)) return;
    dirtyObjects.remove(obj);
    if (!removedObjects.contains(obj)) {
        removedObjects.add(obj);
    }
}
```

```
public void registerClean(DomainObject obj) {
    Assert.notNull("id not null", obj.getId());
}
```

Notice that registerClean() doesn't do anything here. A common practice is to place an [Identity Map](#) within a unit of work. An [Identity Map](#) is necessary most anytime you are storing domain object state in memory, such as in this unit of work, as multiple copies of the same object will result in undefined behavior. Were an [Identity Map](#) in place registerClean() would put the registered object into the map. Likewise registerNew() would put a new object in the map and registerRemoved() would remove a deleted object from the map. Without the [Identity Map](#) you have the option of not including registerClean() in your *Unit of Work*. I've seen implementations of registerClean() that remove changed objects from the dirty list but partially rolling back changes is always tricky. Be careful when reversing any state in the change set.

commit() will locate the [Data Mapper](#) for each object and invoke the appropriate mapping method. updateDirty() and deleteRemoved() are not shown but they would behave in similar fashion to insertNew(), exactly as would expect.

```
class UnitOfWork...
public void commit() {
    insertNew();
    updateDirty();
    deleteRemoved();
}

private void insertNew() {
    for (Iterator objects = newObjects.iterator(); objects.hasNext();) {
        DomainObject obj = (DomainObject) objects.next();
        MapperRegistry.getMapper(obj.getClass()).insert(obj);
    }
}
```

Not included in this *Unit of Work* is tracking of any objects we've read and would like to check for inconsistent read errors upon commit. This is addressed in [Optimistic Offline Lock](#)

Next, we need to facilitate object registration. First, each domain object needs to find the *Unit of Work* serving the current business transaction. Since the *Unit of Work* will be needed by the entire domain model passing it around as a parameter is probably unreasonable. As each business transaction executes within a single thread we can associate the *Unit of Work* with the currently executing thread using the java.lang.ThreadLocal class. Keeping things simple we'll add this functionality by using static methods on our *Unit of Work* class. If we already had some sort of session object associated to the business transaction execution thread we should place the current *Unit of Work* on that session object rather than add the management overhead of another thread mapping. Besides, the *Unit of Work* would logically belong to the session anyway.

```
class UnitOfWork...
private static ThreadLocal current = new ThreadLocal();

public static void newCurrent() {
    setCurrent(new UnitOfWork());
}

public static void setCurrent(UnitOfWork uow) {
```

```
current.set(uow);
}

public static UnitOfWork getCurrent() {
return (UnitOfWork) current.get();
}
```

We can now provide our abstract domain object marking methods to register itself with the current *Unit of Work*:

```
class DomainObject...
protected void markNew() {
UnitOfWork.getCurrent().registerNew(this);
}

protected void markClean() {
UnitOfWork.getCurrent().registerClean(this);
}

protected void markDirty() {
UnitOfWork.getCurrent().registerDirty(this);
}

protected void markRemoved() {
UnitOfWork.getCurrent().registerRemoved(this);
}
```

Concrete domain objects then need to remember to mark themselves new and dirty where appropriate.

```
class Album...
public static Album create(String name) {
Album obj = new Album(IdGenerator.nextId(), name);
obj.markNew();
return obj;
}

public void setTitle(String title) {
this.title = title;
markDirty();
}
```

Not shown is that registration of removed objects can be handled by a remove() method on the abstract domain object and if you've implemented registerClean() your [Data Mapper](#) will need to register any newly loaded object as clean.

The final piece is to register and commit the *Unit of Work* where appropriate. This can be done in either explicit or implicit fashion. Here's what explicit *Unit of Work* management looks like:

```
class EditAlbumScript...
public static void updateTitle(Long albumId, String title) {
UnitOfWork.newCurrent();
Mapper mapper = MapperRegistry.getMapper(Album.class);
Album album = (Album) mapper.find(albumId);
album.setTitle(title);
UnitOfWork.getCurrent().commit();
}
```

Beyond the simplest of applications implicit *Unit of Work* management is more appropriate to avoid repetitive, tedious coding. Here's a servlet [\*Layer Supertype\*](#) that registers and commits the *Unit of Work* for its concrete subtypes. Subtypes will implement handleGet() rather than override doGet(). Any code executing within handleGet() will have a *Unit of Work* with which to work.

```
class UnitOfWorkServlet...
final protected void doGet(HttpServletRequest request, HttpServletResponse
response)
throws ServletException, IOException {
try {
UnitOfWork.newCurrent();
handleGet(request, response);
UnitOfWork.getCurrent().commit();
} finally {
UnitOfWork.setCurrent(null);
}
}

abstract void handleGet(HttpServletRequest request, HttpServletResponse
response)
throws ServletException, IOException;
```

The above servlet example is obviously a bit simplistic in that system transaction control is skipped. And if you were using [\*Front Controller\*](#) you would be more likely to wrap *Unit of Work* management around your commands rather than doGet(). Similar wrapping approaches can be taken with just about any execution context.



---

© Copyright [Martin Fowler](#), all rights reserved



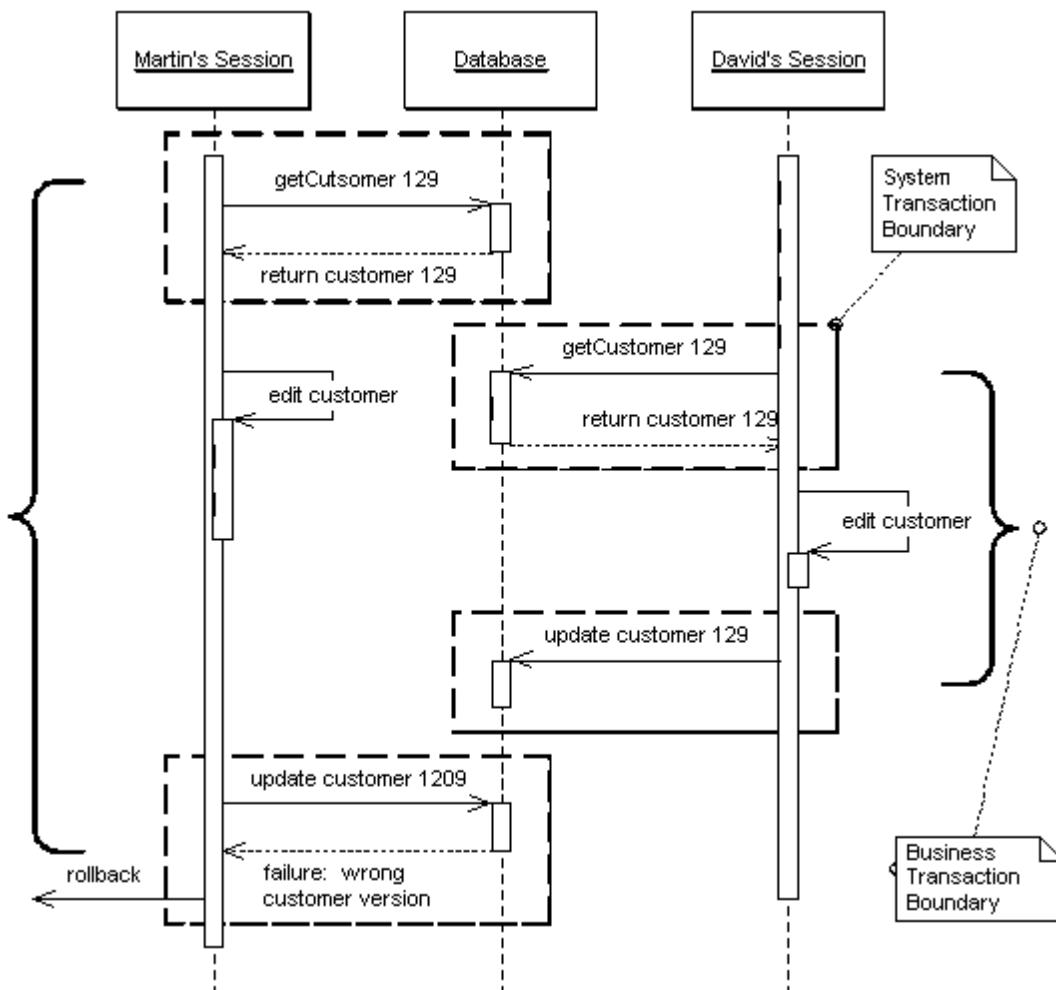
---

# Optimistic Offline Lock

---

by David Rice

*Prevent conflicts between concurrent business transactions, by detecting a conflict and rolling back the transaction.*



Often a business transaction executes across a series of system transactions. Once outside the confines of a single system transaction, we cannot depend upon our database manager alone to ensure that the business transaction will leave the record data in a consistent state. Data integrity is at risk once two sessions begin to work on the same records. With two sessions editing the same data lost updates are quite possible. With one session editing data that another is reading an inconsistent read becomes likely.

*Optimistic Offline Lock* solves this problem by validating that the changes about to be committed by one session don't conflict with the changes of another session. A successful pre-commit validation is, in a sense, obtaining a lock indicating it's OK to go ahead with the changes to the record data. So long as the validation and the updates to the record data occur within a single system transaction the business transaction will display consistency.

Whereas Pessimistic Offline Lock assumes that the chance of session conflict is high and therefore limits the system's concurrency, *Optimistic Offline Lock* assumes that the chance of conflict is rather low. The expectation that session conflict isn't likely allows multiple users to work with the same data at the same time.

## How it Works

An *Optimistic Offline Lock* is obtained by validating that in the time since a session loaded a record another session has not altered that record. An *Optimistic Offline Lock* can be acquired at any time but is valid only during the system transaction in which it is obtained. So in order that a business transaction not corrupt record data it must acquire an *Optimistic Offline Lock* for each member of its change set during the system transaction in which it applies changes to the database.

The most common implementation is to associate a version number to each record in your system. When a record is loaded that version number is maintained by the session along with all other session state. Getting the *Optimistic Offline Lock* is a matter of comparing the version stored in your session data to the current version in the record data. Once the verification succeeds, all changes, including an increment of the version, can be committed. The version increment is what prevents inconsistent record data. A session with an old version cannot acquire the lock.

With an RDBMS data store the verification is a matter of adding the version number to the criteria of any SQL statements used to update or delete a record. A single SQL statement can both acquire the lock and update the record data. The final step is for the business transaction to inspect the row count returned by the SQL execution. A row count of 1 indicates success. A row count of 0 indicates that the record has been changed or deleted. With a row count of 0 the business transaction must rollback the system transaction to prevent any changes from entering the record data. At this point the business transaction must either abort or attempt to resolve the conflict and retry.

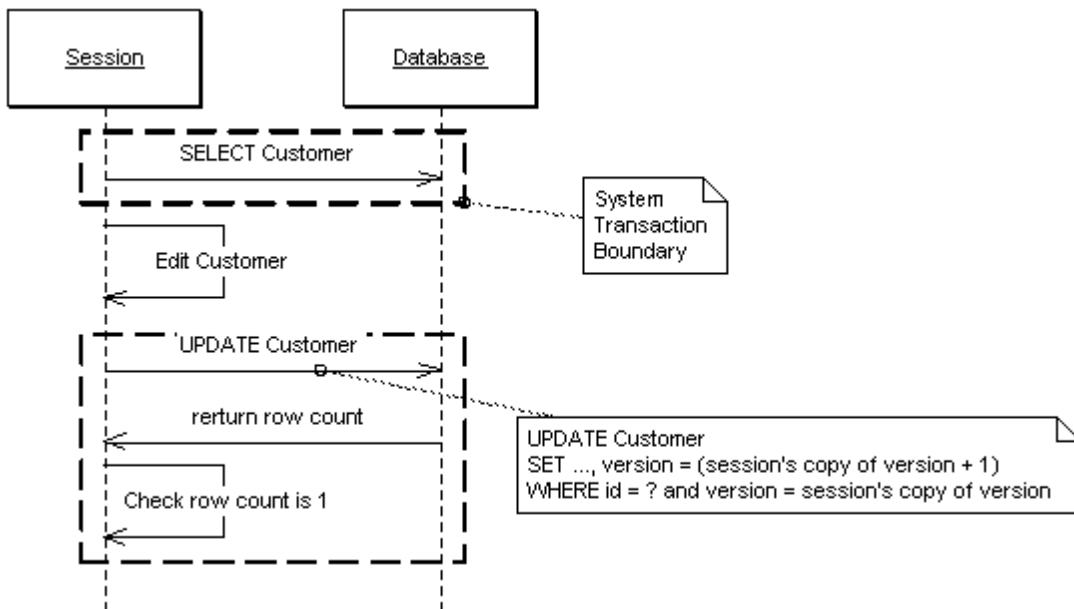


Figure 1: UPDATE Optimistic Check

In addition to a version number for each record, storing information as to who last modified a record and when can be quite useful when managing concurrency conflicts. When informing a user of a failed update due to a concurrency violation a proper application will provide a message as to who altered the record in question and when. Note that it is a bad idea to use the modification timestamp rather than a version count for your optimistic checks. System clocks are simply too unreliable. Even more so if you are coordinating across multiple servers.

An alternative implementation that sometimes comes up is for the where clause in the update to include every field in the row. The advantage of this is that you can use this without using some form of version field, which can be handy if you can't alter the database tables to add a version field. The problem is that this complicates the update statement with a potentially rather large where clause. This may also be a performance impact, although this depends on how clever the database is about using the primary key index.

Often, implementation of *Optimistic Offline Lock* is left at including the version in UPDATE and DELETE statements. This fails to address the problem of an inconsistent read. Suppose a billing system that creates charges and calculates appropriate sales tax. A session creates the charge and then looks up the customer's address to calculate the tax on that charge. But while the charge generation session is in progress a separate customer maintenance session edits the customer's address. As local tax rates are dependent upon location the tax rate calculated by the charge generation session might be invalid. But since the charge generation session did not make any changes to the address the conflict will not be detected.

There is no reason why *Optimistic Offline Lock* cannot be used to detect an inconsistent read. In the example above the charge generation session needs to recognize that its correctness is dependent upon the value of the customer's address. The session should perform a version check on the address as well. This could be done by adding the address to your change set or maintaining a separate list of items on which to perform a version check. The latter requires a bit more work to setup but results in code that more clearly states its intent. If you are checking for a consistent read simply by re-reading the version rather than an artificial update be especially aware of your system transaction isolation level. The version

re-read will only work with repeatable read or stronger isolation. Anything weaker requires an increment of the version.

A version check might be overkill for certain inconsistent read problems. Often a transaction will be dependent only upon the presence of a record or maybe the value of only one of its fields. In such a case it may improve your system's liveness to check these types of conditions rather than the version, as fewer concurrent updates will result in the failure of competing business transactions. The better you understand your concurrency issues the better you can manage them in your code.

The *Coarse-Grained Lock* can help with certain inconsistent read conundrums by treating a group of objects as a single lockable item. A final option is to simply execute all of the steps of the problematic business within a long running transaction. The ease of implementation might prove worth the resource hit of using a few long transactions here and there.

Where detection of an inconsistent read gets a bit difficult is when your transaction is dependent upon the results of a dynamic query rather than the reading of specific records. It's possible that you could save the initial results and compare them to the results of the same query at commit time as a means of obtaining an *Optimistic Offline Lock*.

As with all locking schemes *Optimistic Offline Lock* by itself does not provide adequate solutions for some of the trickier concurrency and temporal issues in a business application. We cannot stress enough that in a business application concurrency management is as much a domain issue as it is a technical one. Is the customer address scenario above really a conflict? It might be OK that I calculated the sales tax with an older version of the customer. Which version should I actually be using? This is a business issue. Or consider a collection. What if two sessions simultaneously add items to a collection. The typical *Optimistic Offline Lock* scheme will not prevent this. But this might very well be a violation of business rules.

A system using *Optimistic Offline Lock* that we all should be familiar with is a source code management system. When an SCM system detects a conflict between programmers it usually can figure out the correct merge and retry the commit. A quality merge strategy makes the use of *Optimistic Offline Lock* very powerful. Not only is the system's concurrency quite high but users rarely have to redo any work. Of course, the big difference between an SCM system and an enterprise business application is that an SCM system must implement only one type of merge while an enterprise business system might have hundreds of merges to implement. Some might be of such complexity that they are not worth the cost of coding. Others might be of such value to the business that the merge should by all means be coded. Despite rarely being done the merging of business objects is quite possible. Merging business data is a pattern unto its own so I will leave it at that rather than butcher the topic, but do understand the power that it adds to using the *Optimistic Offline Lock*.

*Optimistic Offline Lock* only lets us know if a business transaction will commit during the last system transaction but it's occasionally useful to know earlier on if a conflict has occurred. To do this you can provide a checkCurrent method that looks to see if anyone else has updated the data. Such a check is not a guarantee that you won't get a conflict, but it may be worthwhile to stop a complicated process if you can tell in advance that it certainly won't commit.

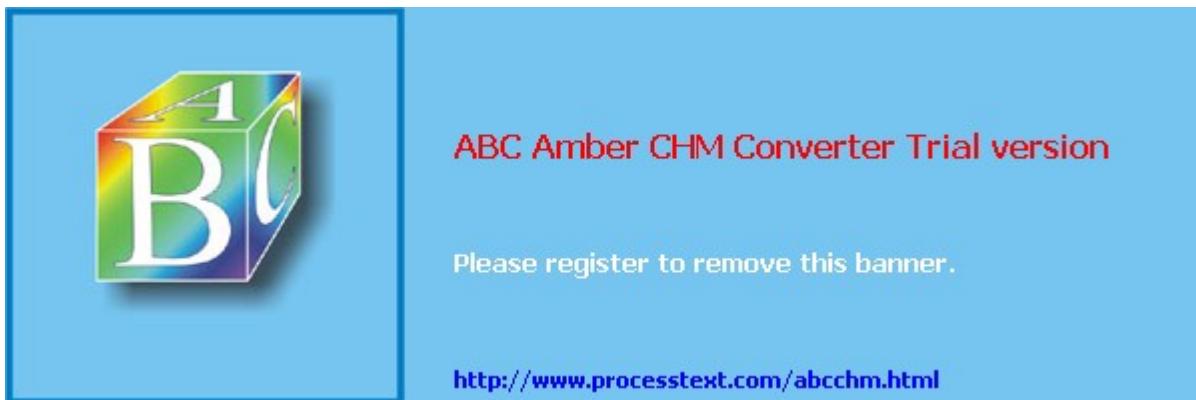
# When to Use it

Optimistic concurrency management is appropriate when the chance of conflict between any two business transactions in a system is low. If conflicts are likely it is not user-friendly to announce a conflict only when the user has finished his work and is ready to commit. Eventually the user will assume the failure of business transactions and stop using the system. [Pessimistic Offline Lock](#) is more appropriate when the chance of conflict is high or the expense of a conflict is unacceptable.

As optimistic locking is much easier to implement and not prone to the same defects and runtime errors as a [Pessimistic Offline Lock](#), consider using it as the default approach to business transaction conflict management in any system you build. Where needed, [Pessimistic Offline Lock](#) works well as a complement to [Optimistic Offline Lock](#). So rather than asking when to use an optimistic approach to conflict avoidance, ask when is the optimistic approach alone is not good enough. The correct approach to concurrency management will maximize concurrent access to data while minimizing the number of conflicts.



© Copyright [Martin Fowler](#), all rights reserved

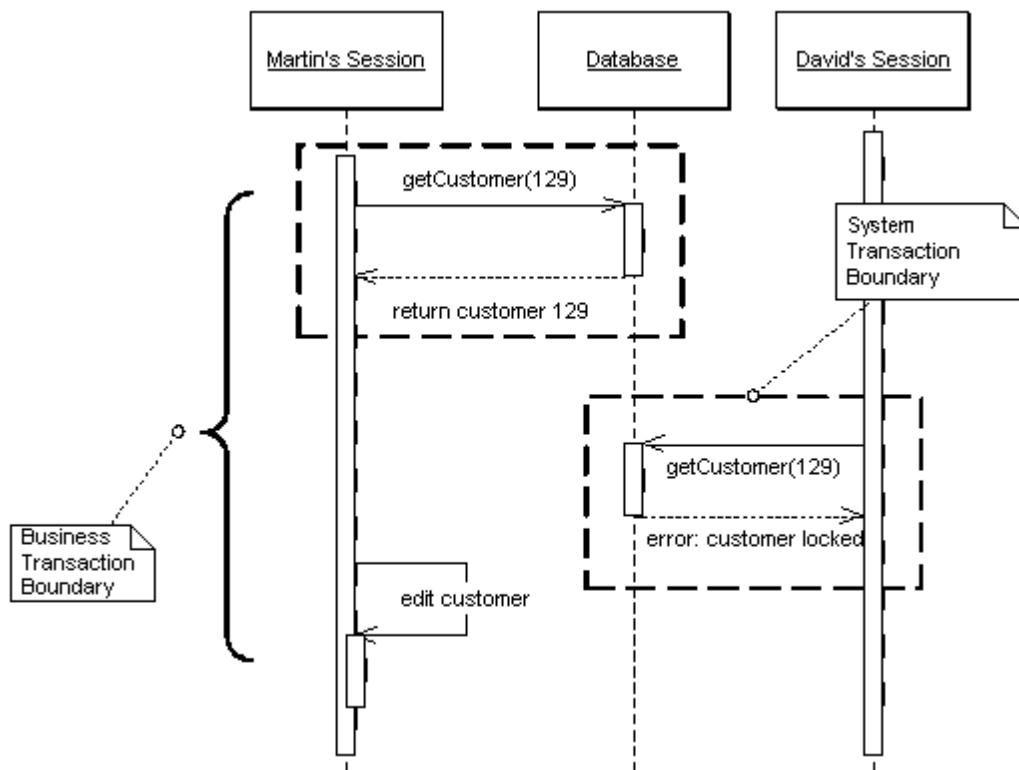


---

# Pessimistic Offline Lock

by David Rice

*Prevent conflicts between concurrent business transactions by allowing only one business transaction to access data at once.*



Offline concurrency involves manipulating data for a business transaction that spans multiple requests. The simplest approach to use is to have a system transaction open for the whole business transaction. Sadly this often does not work well because transaction systems are not geared to work with these long transactions. When you can't use long transactions you have to use multiple system transactions. At that point you're left to your own devices to manage concurrent access to your data.

The first approach to try is [Optimistic Offline Lock](#). However [Optimistic Offline Lock](#) has its problems. If several people access the same data within a business transaction, then one of them will commit with no problems but the others will conflict and fail. Since the conflict is only detected at the end of the business transaction, the victims will do all the work of the transaction only to find the whole thing fails - wasting their time. If this happens a lot on lengthy business transactions the system can soon become very unpopular.

*Pessimistic Offline Lock* prevents conflicts by attempting to avoid them altogether. *Pessimistic Offline Lock* forces a business transaction to acquire a lock on a piece of data before it starts to use that data. This way most of the time, once you begin a business transaction you can be pretty sure you'll complete it without being bounced by concurrency control.

## How it Works

Implementing *Pessimistic Offline Lock* involves three phases: determining what type of locks you need, building a lock manager, and defining procedures for a business transaction to use locks. Additionally, if using *Pessimistic Offline Lock* as a complement to [\*Optimistic Offline Lock\*](#) it becomes necessary to determine which record types to lock.

Looking at lock types, the first option is to have an **exclusive write lock**. That is, require only that a business transaction acquire a lock in order to edit session data. This avoids conflict by not allowing two business transactions to simultaneously make changes to the same record. What this locking scheme ignores is the reading of data. If it is not critical that a view session have the absolute most recent data this strategy will suffice.

If it becomes critical that a business transaction always have the most recent data regardless of its intention to edit then use the **exclusive read lock**. Using an exclusive read lock requires that a business transaction acquire a lock to simply load the record. This strategy clearly has the potential of severely restricting the concurrency of your system. For most enterprise systems the exclusive write lock will afford much more concurrent record access than this strategy.

There is a third strategy combining more specialized forms of read and write locks that will provide the more restrictive locking of the exclusive read lock while also providing the increased concurrency of the exclusive write lock. This third strategy, the **read/write lock**, is a bit more complicated than the first two. The relationship of the read and write locks is the key to getting the best of both worlds:

- Read and write locks are mutually exclusive. A record cannot be write locked if any other business transaction owns a read lock on that record. A record cannot be read locked if any other business transaction owns a write lock on that record.
- Concurrent read locks are acceptable. The existence of a single read lock prevents any business transaction from editing the record, so there is no harm in allowing any number of sessions as readers once one is allowed.

Allowing multiple read locks is what increases the concurrency of the system. The downside of this scheme is that it's a bit nasty to implement and presents more of a challenge for domain experts to wrap their heads around when they are modeling the system.

In choosing the correct lock type you are looking to maximize system concurrency, meet business needs, and minimize code complexity. In addition, the locking strategy must be understood by the domain modelers and analysts. Locking is not just a technical problem as the wrong lock type, simply locking every record, or locking the wrong types of records can result in an ineffective *Pessimistic Offline Lock*.

strategy. An ineffective *Pessimistic Offline Lock* strategy is one that doesn't prevent conflict at the onset of the business transaction or degrades the concurrency of your system such that your multi-user system seems more like single-user system. The wrong locking strategy cannot be saved by a proper technical implementation. In fact, it's not a bad idea to include *Pessimistic Offline Lock* in your domain model.

Once you have decided upon your lock type define your lock manager. The lock manager's job is to grant or deny any request by a business transaction to acquire or release a lock. For a lock manager to do its job it needs to know about what's being locked as well as the intended owner of the lock, the business transaction. It's quite possible that your concept of a business transaction isn't some *thing* that can be uniquely identified. This makes it a bit difficult to pass a business transaction to the lock manager. In this case take a look at your concept of a session as you are more likely to have a session object at your disposal. So long as business transactions execute serially within a session the session will serve as a fine *Pessimistic Offline Lock* owner. The terms session and business transaction are fairly interchangeable. The code example should shed some light on the idea of a session as lock owner.

The lock manager shouldn't consist of much more than a table that maps locks to owners. A simple lock manager might wrap an in-memory hash table. Another option is to use a database table to store your locks. You must have one and only one lock table, so if it's in memory be sure to use a [singleton](#) lock manager. If your application server is clustered an in-memory lock table will not work unless it is pinned to a single server instance. The database-based lock manager is probably more appropriate once you are in a clustered application server environment.

The lock, whether implemented as an object or as SQL against a database table, should remain private to the lock manager. Business transactions should interact only with the lock manager, never with a lock object.

It is now time to define the protocol by which a business transaction must use the lock manager. This protocol must include what to lock, when to lock, when to release a lock, and how to act when a lock cannot be acquired.

The answer to what depends upon when so let's look first at when. Generally, the business transaction should acquire a lock before loading the data. There is not much point in acquiring a lock without a guarantee that you will have the latest version of the item once it's locked. But, since we are acquiring locks within a system transaction there are circumstances where the order of the lock and load won't matter. Depending upon your lock type if you are using serializable or repeatable read transactions the order in which you load objects and acquire locks might not matter. Another option might be to perform an optimistic check on an item after you acquire the *Pessimistic Offline Lock*. The rule is that you should be very sure that you have the latest version of an object after you have locked it. This usually translates to acquiring the lock before loading the data.

So, what are we locking? While we are locking objects or records or just about anything, what we usually lock is the id, or primary key, that we use to find those objects. This allows us to obtain the lock before loading the object. Locking the object works fine so long as this doesn't force you to break the rule that an object must be current after you acquire its lock.

The simplest rule for releasing locks is to release them when the business transaction completes. Releasing a lock prior to completion of a business transaction might be allowable, dependent upon your lock type and your intention to use that object again within the transaction. Unless you have a very specific reason to release the lock, such as a particularly nasty system liveness issue, stick to releasing all locks upon completion of the business transaction.

The easiest course of action for a business transaction when it is unable to acquire a lock is to abort. The user should find this acceptable since using *Pessimistic Offline Lock* should result in failure happening rather early in the transaction. The developer and designer can certainly help the situation by not waiting until late in the transaction before acquiring a particularly contentious lock. If at all possible acquire all of your locks before the user begins work.

For any given item that you intend to lock access to the lock table must be serialized. With an in-memory lock table it's easiest to simply serialize access to the entire lock manager with whatever constructs your programming language provides. If you need concurrency greater than this affords be aware you are entering complex territory.

If the lock table is stored in a database the first rule is, of course, interact with the lock table within a system transaction. Take full advantage of the serialization capabilities that a database provides. With the exclusive read and exclusive write lock types serialization is simply a matter of having the database enforce a uniqueness constraint on the column storing the lockable item's id. Storing read/write locks in a database makes things a bit more difficult since read/write lock logic requires reads of the lock table in addition to inserts. Thus, it becomes imperative to avoid inconsistent reads. A system transaction with isolation level of serializable will provide ultimate safety as we are guaranteed not to have inconsistent reads. But using serializable transactions throughout our system might get us into performance trouble. Using a separate serializable system transaction for lock acquisition and a less strict isolation level for other work might ease the performance problem. Still another option is to investigate whether a stored procedure might help with lock management. Concurrency management can be tough so don't be afraid to defer to your database at key moments.

The serial nature of lock management screams performance bottleneck. A large consideration is lock granularity. The fewer locks required the less of a bottleneck you will have. A [\*Coarse-Grained Lock\*](#) can address lock table contention.

When using a system transaction pessimistic locking scheme, such as 'SELECT FOR UPDATE...' or entity EJBs, deadlock is a distinct possibility. Deadlock is a possibility because these locking mechanisms will wait until a lock becomes available. Suppose two users need resources A and B. If one gets the lock on A and the other on B both transactions might sit and wait forever for the other lock. Given that we're spanning multiple system transactions waiting for a lock does not make much sense. A business transaction might take 20 minutes. Nobody wants to wait for those locks. This is good. Coding for a wait involves timeouts and quickly gets complicated. So simply have your lock manager throw an exception as soon as a lock is unavailable. This removes the burden of coping with deadlock.

A final requirement is managing lock timeouts for lost sessions. If a client machine crashes in the middle of a transaction the lost business transaction is unable to complete and release any owned locks. This is an even bigger deal for a web application where sessions are regularly abandoned by users. Ideally you will have a mechanism managed by your application server rather than your application available for handling timeouts. Web application servers provide an HTTP session that can be utilized for timeout purposes. Timeouts can be implemented by registering a utility object that releases all locks for the session when the HTTP session becomes invalid. Another option would be to associate a timestamp with each lock and consider any lock older than a certain age invalid.

## When to Use it

*Pessimistic Offline Lock* is appropriate when the chance of conflict between concurrent sessions is high. A user should never have to throw away work. Locking is also quite appropriate when the cost of a concurrency conflict is too high, regardless of the likelihood of that conflict. Locking every entity in a system will almost surely create tremendous data contention problems so remember that *Pessimistic Offline Lock* is very complementary to [\*Optimistic Offline Lock\*](#) and only use *Pessimistic Offline Lock* where it is truly required.

If you have to use *Pessimistic Offline Lock* you should also consider a long transaction. Although long transactions are never a good thing, in some situations they may work out to be no more damaging than *Pessimistic Offline Lock*, and they are much easier to program. Do some load testing before you choose.

Do not use these techniques if your business transactions fit within a single system transaction. Many system transaction pessimistic locking techniques ship with the application and database servers you are already using. Among these techniques are the 'SELECT FOR UPDATE' SQL statement for database locking and the entity EJB for application server locking. Why worry about timeouts, lock visibility and such when there is no need. Understanding these types of locking can certainly add a lot of value to your implementation of *Pessimistic Offline Lock*. But understand that the inverse is not true! What you have read here has not prepared you to write a database manager or transaction monitor. The offline locking techniques presented in this book are all dependent upon your system having a real transaction monitor.





ABC Amber CHM Converter Trial version

Please register to remove this banner.

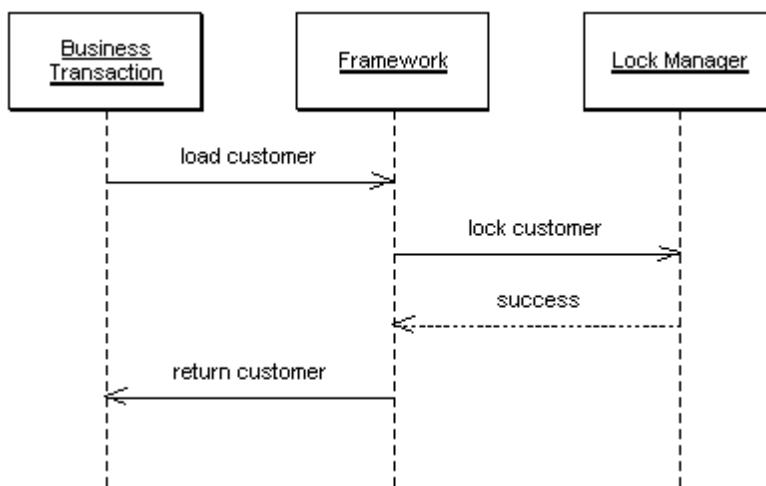
<http://www.processtext.com/abcchm.html>

# Implicit Lock

---

by David Rice

*Allow framework or layer supertype code to acquire offline locks.*



Key to any locking scheme is that there are no gaps in its use. A developer's forgetting to write a single line of code that acquires a lock can render an entire offline locking scheme useless. Failing to retrieve a read lock where other transactions use write locks means you might not get up-to-date session data. Failing to use a version count properly can result in unknowingly writing over someone's changes. Generally, if an item might be locked *anywhere* it must be locked *everywhere*. Ignoring its application's locking strategy allows a business transaction to create inconsistent data. Not releasing locks will not corrupt your record data but will eventually bring productivity to a halt. As offline concurrency management is a difficult concept to test such errors might go undetected by all of your test suites.

One solution is to not allow developers to make such a mistake. Locking tasks that cannot be overlooked should be handled implicitly by the application rather than explicitly by developers. The fact that most enterprise applications make use of some combination of framework, [Layer Supertype](#), and code generation provides us with ample opportunity to facilitate *Implicit Lock*.

## How it Works

Implementing *Implicit Lock* is a matter of factoring your code such that any locking mechanics that

absolutely cannot be skipped can be carried out by your application framework. For lack of a better word we'll use the word framework to mean any combination of [\*Layer Supertype\*](#), framework classes, and any other 'plumbing' code. Code generation tools are another avenue to enforce proper use of a locking strategy. OK, this is by no means a ground-breaking idea. You are very likely to head down this path once you have coded the same locking mechanics a few times over for in your application. But we have seen it done poorly often enough that it merits a quick look.

The first step is to assemble a list of what tasks are mandatory for a business transaction to work within your locking strategy. For [\*Optimistic Offline Lock\*](#) the list will include items such as storing a version count for each record, including the version in update SQL criteria, and storing an incremented version when changing the record. The [\*Pessimistic Offline Lock\*](#) list will include items along the lines of acquiring any lock necessary to load a piece of data, typically the exclusive read lock or the read portion of the read/write lock, and the release of all locks when the business transaction or session completes.

Note that the [\*Pessimistic Offline Lock\*](#) list did not include acquiring any lock only necessary for editing a piece of data. This is the exclusive write lock and the write portion of the read/write lock. Yes, this is indeed mandatory should your business transaction wish to edit the data. However, implicitly acquiring these locks would present a couple of difficulties. First, the only points where we might implicitly acquire a write lock, such as the registration of a dirty object within a [\*Unit of Work\*](#), offer us no promise that the transaction will abort as soon as the user begins to work should the locks be unavailable. The application cannot figure out on its own when is a good time to acquire these locks. A transaction not failing rapidly conflicts with an intent of [\*Pessimistic Offline Lock\*](#): that a user not have to perform work twice. Second, and just as important, is that these are the types of locks that most greatly limit the system's concurrency. Avoiding [\*Implicit Lock\*](#) here helps us think about how we are impacting concurrency by forcing the issue out of the technical arena and into the business domain. But still we must enforce that locks necessary for writing are acquired before changes are committed. What your framework can do is assure that a write lock has already been obtained before committing any changes. Not having acquired the lock by commit time is a programmer error and the code should at least throw an assertion failure. We'd advise skipping the assertion and throwing a concurrency exception here as you really don't want any such errors in your production system when assertions are turned off.

A word of caution with using the [\*Implicit Lock\*](#). While [\*Implicit Lock\*](#) allows developers to ignore a large part of locking mechanics it does not allow them to ignore consequences. For example, if using [\*Implicit Lock\*](#) with a pessimistic locking scheme that waits for locks the developers still need to think about deadlock possibilities. The danger with [\*Implicit Lock\*](#) is that business transactions can fail in unexpected fashion once developers stop thinking about locking.

Making it work is a matter of determining the best way to get your framework to implicitly carry out locking mechanics. Please see [\*Optimistic Offline Lock\*](#) for samples of implicit handling of that lock type. Below is an example of an implicit [\*Pessimistic Offline Lock\*](#). The possibilities for a quality [\*Implicit Lock\*](#) implementation are far too numerous to demonstrate here.

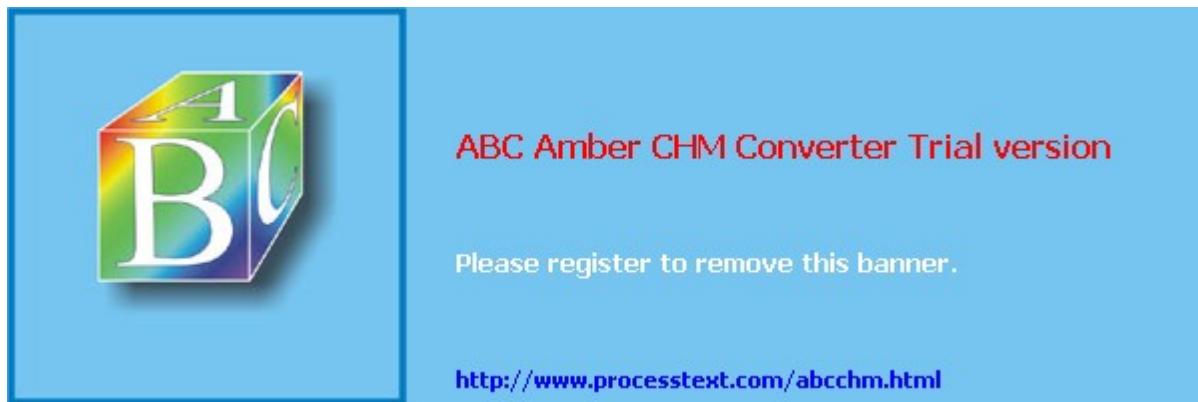
## When to Use it

*Implicit Lock* should be used except in the simplest of applications that have no concept of framework. The risk of a single forgotten lock is too great.



---

© Copyright [Martin Fowler](#), all rights reserved

A blue rectangular banner with a white border. On the left side is a 3D-style cube with faces colored red, green, blue, and yellow, with the letters 'A', 'B', and 'C' visible on its faces. To the right of the cube, the text "ABC Amber CHM Converter Trial version" is displayed in red. Below this, in a smaller black font, is the instruction "Please register to remove this banner." At the bottom right of the banner is the URL "http://www.processtext.com/abcchm.html".

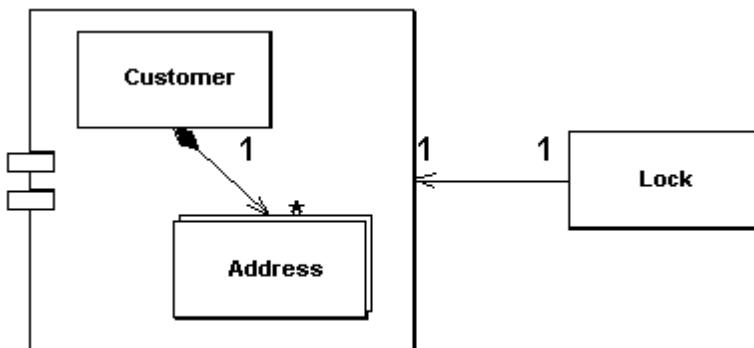
---

# Coarse-Grained Lock

---

by David Rice and Matt Foemmel

*Lock a set of related objects with a single lock*



Often groups of objects can be edited as a group. Perhaps you have a customer and its set of addresses. When using the application it makes sense that if you want to lock any one of these items, you should lock all of them. Having a separate lock for each individual object presents a number of challenges. The first problem is that anyone manipulating these objects has to write code that can find all the objects in the group in order to lock them. This is easy enough for a customer and its addresses but gets tricky as you have more locking groups. And what if the groups get complicated? And where is this behavior when your framework is managing lock acquisition? Then, if your locking strategy requires that an object be loaded in order that it be locked, such as with [\*Optimistic Offline Lock\*](#), locking a large group will present a performance problem. And if using [\*Pessimistic Offline Lock\*](#) a large lock set is a management headache and increases lock table contention.

A *Coarse-Grained Lock* is a single lock that covers many objects. Not just does this simplify the locking action itself, it also means that you don't have load all the members of a group in order to lock them.

## How it Works

The first step to implementing *Coarse-Grained Lock* is to create a single point of contention for locking a group of objects. This allows that only one lock be necessary for locking the entire set. Then provide the shortest path possible to finding that single lock point in order to minimize the group members which

must be identified and possibly loaded into memory in the process of obtaining that lock.

With the [\*Optimistic Offline Lock\*](#), having each item in a group share a version creates the single point of contention. And this means sharing the *same* version, not an *equal* version. Incrementing this version will lock the entire group with a **shared lock**. Setup your model to point every member of the group at the shared version and you have certainly minimized the path to the point of contention.

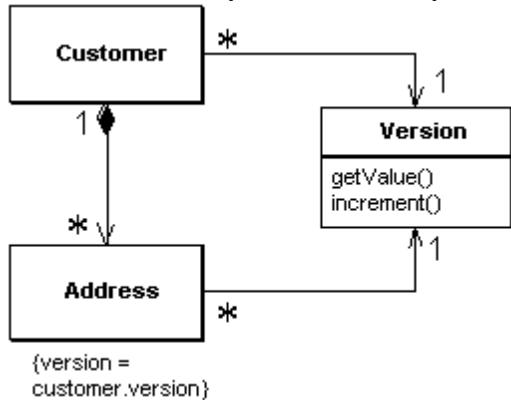


Figure 1: Sharing a version

Using a shared [\*Pessimistic Offline Lock\*](#) will require that each member of the group share some sort of lockable token. The [\*Pessimistic Offline Lock\*](#) must then be acquired on this token. As [\*Pessimistic Offline Lock\*](#) is often used as a complement to [\*Optimistic Offline Lock\*](#) a shared version object makes an excellent candidate for the role of lockable token.

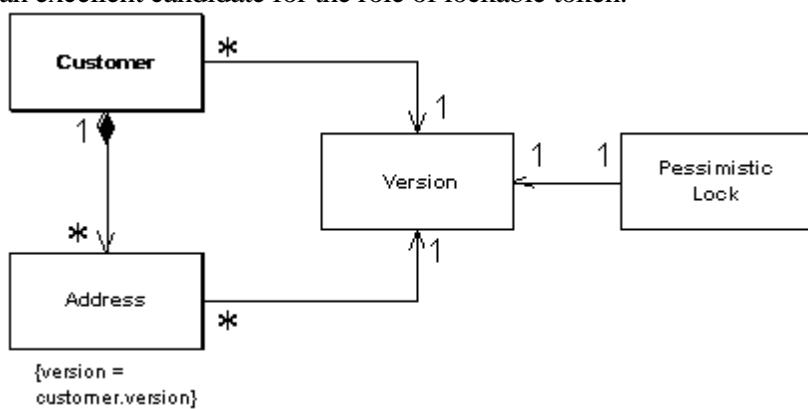


Figure 2: Locking a shared version

Eric Evans and David Siegel[missing reference] define an **aggregate** as a cluster of associated objects that we treat as a unit for data changes. Each aggregate has a **root** that provides the only access point to members of the set and a **boundary** that defines what gets included in the set. This aggregate has the characteristics that call for a *Coarse-Grained Lock* since working with any member of the aggregate requires locking all members. Locking an aggregate yields an alternative to a shared lock that we'll call a **root lock**. This works by locking the root and making all lock usage within the aggregate use that root lock. This approach gives us a single point of contention.

Using a root lock as a *Coarse-Grained Lock* makes it necessary to implement navigation to the root in your object graph. This allows a locking mechanism, upon receiving a request to lock any object in the aggregate, to navigate to the root and lock it instead. This navigation can be accomplished in a couple of

fashions. You can maintain a direct navigation to the root for each object in the aggregate, or you can use a sequence of intermediate relationships. For example, in a hierarchy, the obvious root is the top level parent. You can link the descendants to the top level parent directly, or you can give each node a link to its immediate parent and navigate that structure to reach the root. In a large graph the later strategy might cause performance problems as each parent must be loaded in order to determine whether it has a parent. Be sure to use a [Lazy Load](#) when loading the objects that make up the path to your root. This will not only prevent objects from being loaded before they are needed but will prevent an infinite mapping loop should you be mapping a bidirectional relationship. Be wary of the fact that if [Lazy Load](#) for a single aggregate occur across multiple system transactions you may end up with an aggregate built from inconsistent parts. That, of course, is not good.

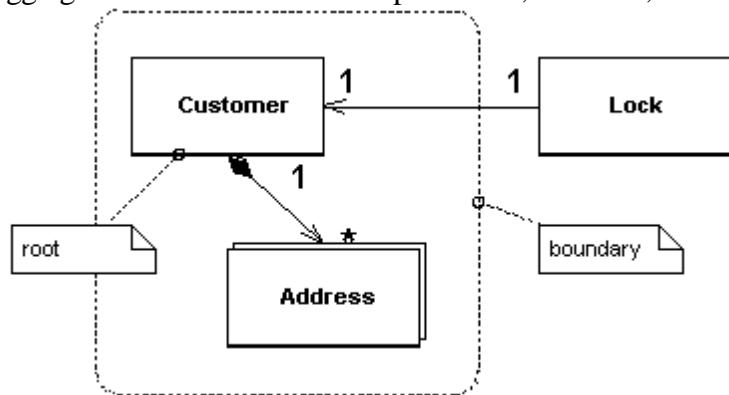


Figure 3: Locking the root

Note that a shared lock also works for locking an aggregate as locking any object in the aggregate will simultaneously lock the root.

Both the shared lock and root lock implementation of *Coarse-Grained Lock* have their trade-offs. When using a relational database the shared lock carries the burden that most all of your selects will require a join to the version table. On the other hand, loading objects while navigating to the root could be a performance hit as well. The root lock and [Pessimistic Offline Lock](#) perhaps make an odd combination. By the time you navigate to the root and lock it you may need to reload a few objects to guarantee their freshness. And, as always, building a system against a legacy data store will place numerous constraints on your implementation choice. Locking implementations abound. The number of subtleties even more numerous. Be sure to arrive at an implementation that suits your needs.

## When to Use it

The most obvious place to use a *Coarse-Grained Lock* is to satisfy business requirements. This is the case when locking an aggregate. Consider a lease object that owns a collection of assets. It probably does not make business sense that one user edits the lease while another user simultaneously edits an asset. Locking either the asset or the lease ought to result in the lease and all of its assets being locked.

A very positive outcome of using *Coarse-Grained Lock* is the decreased cost of acquiring and releasing locks. This is certainly a legitimate motivation for using *Coarse-Grained Lock*. The shared lock can be used beyond the concept of Evans' aggregate. But be careful when working from non-functional

requirements such as performance. Beware of creating unnatural object relationships in order to facilitate *Coarse-Grained Lock*.



---

© Copyright [Martin Fowler](#), all rights reserved

A blue rectangular banner with a thin black border. On the left side, there is a 3D-style cube with faces colored red, green, blue, and yellow, with the letters 'A', 'B', and 'C' visible on its faces. To the right of the cube, the text "ABC Amber CHM Converter Trial version" is displayed in red. Below this, in black text, is the instruction "Please register to remove this banner." At the bottom, the URL "http://www.processtext.com/abcchm.html" is shown in blue.

ABC Amber CHM Converter Trial version

Please register to remove this banner.

<http://www.processtext.com/abcchm.html>

# Client Session State

---

*Store session state on the client*

## How it Works

Even the most server-oriented designs need at least a little *Client Session State*, if only to hold a session identifier. For some applications you can consider putting all of the session data on the client. In this case the client sends the full set of session data with each request, and the server sends back the full session state with each response. This allows the server to be completely stateless.

Most of the time you'll want to use [\*Data Transfer Object\*](#) to handle the data transfer. The [\*Data Transfer Object\*](#) can serialize itself over the wire and thus allow even complex data to be transmitted.

The client also needs to store the data. If our client is a rich client application - it can do this within its own structures. These could be the fields in the rich client interface - although I would drink Budweiser rather than do that. A set of non-visual objects often makes a better bet. This could be the [\*Data Transfer Object\*](#) itself, or a domain model. Either way it's not usually a big problem.

If we have a HTML interface then things get a bit more complicated. There are three common ways to do it: URL parameters, hidden fields, and cookies.

**URL Parameters** are the easiest to work with for a small amount of data. Essentially it means that all the URLs on any response page add the session state as parameters to the URL. The clear limit to doing this is that the size of an URL is limited. However if you only have a couple of data items this works well, so it's a popular choice for something like a session id. Some platforms will do automatic URL rewriting to add a session id. Changing the URL may be a problem with bookmarks, so that's an argument against using it for consumer sites.

A **hidden field** is a field sent to the browser that isn't displayed on the web page. You get it by using a tag of the form <INPUT type = "hidden">. To make hidden fields work you serialize your session state into the hidden field when you make a response and read it back in again on each request. You'll need to make a format for putting the data in the hidden field. XML is an obvious standard choice, but of course is rather wordy. You can also encode the data into some text based encoding scheme. Remember that a hidden field is only hidden from the displayed page, anyone can look at the data by looking at the page source.

Beware if you have a mixed site that has some older or fixed web pages. You can lose all the session

data if you navigate to these pages.

The last, and sometimes controversial choice, is **cookies**. Cookies are sent back and forth automatically. Just like a hidden field you can use them by serializing the session state into the cookie. You are limited in size to how big the cookie can be. Another issue is that many people don't like cookies, as a result they turn them off. If they turn them off then your site will stop working. More and more sites are dependent on cookies now, so that will happen less often, and certainly isn't a problem for a purely in-house system. Cookies also are no more secure than anything else, so assume prying of all kinds can happen.

Cookies also only work within a single domain name, so if your site is separated into different domain names the cookies won't travel between them.

Some platforms can detect whether cookies are enabled, and if not they can use URL rewriting. This can make it very easy for very small amounts of data.

## When to Use it

*Client Session State* contains a number of advantages. In particular it reacts well in supporting stateless server objects with maximal clustering and fail-over resiliency. Of course if the client fails, all is lost; but often the user would expect that anyway.

The arguments against *Client Session State* vary exponentially with the amount of data involved. With just a few fields, everything works nicely. With large amounts of data then the issues of where to store the data, and the time cost of transferring everything with every request start becoming prohibitive. This is especially true if your stars include an http client.

There's also the security issue. Any data sent to the client is vulnerable to being looked at and altered. Encryption is the only way to stop this, but encrypting and decrypting with each request will add some performance burden. Without encryption you have to be sure you aren't sending anything you would rather hide from prying eyes. Fingers can pry too, so don't assume that what got sent out is the same as what gets sent back. Any data coming back will need to be completely revalidated.

You'll almost always have to use *Client Session State* for session identification. Fortunately this should be just one number, which won't burden any of the above schemes. You should still be concerned about session stealing, which is what happens when a malicious user changes his session id to see if he can snag someone else's session.





ABC Amber CHM Converter Trial version

Please register to remove this banner.

<http://www.processtext.com/abcchm.html>

# Database Session State

---

*Store session date as committed data in the database.*

## How it Works

When a call occurs from the client to the server, the server object will first pull the data required for the request from the database. The server object then does the work it needs to do and saves all the data required back to the database.

In order to pull information from the database, the server object will need some information about the session - requiring at least a session ID number to be stored on the client. Usually, however, this information is nothing more than the appropriate set of keys that's needed to find the appropriate amount of data in the database.

The data that's involved will typically involve a mix of session data, that's only local to the current interaction, and committed data that's relevant to all interactions.

One of the key issues to consider here is how to deal with the fact that session data is usually considered local to the session and shouldn't affect other parts of the system until the session as a whole is committed. So if you are working on an order in a session and you want to save its intermediate state to the database, you usually need to handle it differently to an order that is confirmed at the end of a session. This is because that often, you don't want pending orders to appear in queries that are run against the database for such things as book availability, revenue that day, and the like.

So how do you separate the session data? One route is to simply add a field to each database row that may have session data. The simplest form of this is to just have a boolean field `isPending`. However a better way is to store a session id in this field. This makes it much easier to find all the data for a particular session. All queries that want only record data now need to be modified with a clause `sessionID is not NULL`, or use a view that filters out that data.

Using a session ID field ends up being a very invasive solution because all applications that touch the record database need to know about the meaning of the field to avoid getting session data. Views will sometimes do the trick and remove the invasiveness, but they often impose other costs of their own.

A second alternative is to use a separate set of pending tables. So if you have an orders and an order lines tables in your database, you would add tables for pending orders and pending order lines. While session data is pending you save it to the pending table and when it becomes record data you save it to

the real tables. This removes much of the invasiveness problem, but you'll need to add the appropriate table selection logic to your database mapping code, which will certainly add some complications.

Often the record data will have integrity rules that aren't really true for pending data. In this case the pending tables allow you to not have the rules when you don't want them, but to enforce them when you do. Validation rules also will typically not be applied when saving pending data. You may face different validation rules depending on where you are in the session - but this will typically appear in server object logic.

If you use pending tables, they should be pretty much exact clones of the real tables. That way you can keep your mapping logic as similar as possible. So use the same field names between the two tables. However do add a session id field to the pending tables so that you can easily find all the data for a session.

You'll need a mechanism to clean out the session data if a session is cancelled or abandoned. Using a session ID makes this very straightforward as you can find all data with that session ID and delete it. If users abandon the session without telling you, you'll need some kind of timeout mechanism. A daemon that runs every few minutes can look for old session data. This implies a table in the database that keeps a track of the time of the last interaction with the session.

Rollback is made much more complicated by updates. If you update an existing order in a session that allows a rollback of the whole session, how do you perform the rollback? One option is to not allow cancellation of a session like this. Any updates to existing record data become part of the record data at the end of the request. This is simple and often fits the users' view of the world. The alternative is awkward whether you use pending fields or pending tables. Pending tables can be easier as you can copy all the data that may be modified into pending tables, modify it there, and commit it back to the record tables at the end of the session. You can do a similar approach with a pending field, but only if the session ID becomes part of the key so that you can keep the old and new IDs in the same table at the same time - which can get very messy.

If you are going to use separate pending tables that are only read by objects that handle a session, then there may be little point in tabularizing the data. Then it's better to use a [Serialized LOB](#). At this point we've crossed the boundary into a [Server Session State](#)

You can avoid all of the hassles of pending data by not having any. If you design your system so that all data is considered to be record data, then you avoid all issues of pending data. This isn't always possible, of course, and sometimes it's done in such an awkward way that the designers would be better off with thinking about explicit pending data. But if you have the option it makes *Database Session State* a lot easier to work with.

## When to Use it

*Database Session State* is one alternative to handling session state and should be compared with [Server Session State](#) and [Client Session State](#).

The first aspect to consider with this pattern is performance. You'll gain by allowing yourself to use stateless objects on the server, thus enabling pooling and easy clustering. However the cost you'll pay is the time to pull the data in and out of the database with each request. You can reduce this cost by

caching the server object. This will save you having to read the data out of the database whenever the cache is hit. You'll still pay the write costs.

The second main issue is that programming effort, most of which centers around handling session state. If you have no session state and are able to save all your data as record data in each request - then this pattern is an obvious choice as you lose nothing in either effort or performance (if you cache your server objects).

In a choice between *Database Session State* and [\*Server Session State\*](#) the biggest issue may be in how easy it is to support clustering and fail over with [\*Server Session State\*](#) in your particular application server. Getting clustering and fail over to work with *Database Session State* is usually more straightforward, at least with the regular solutions.



© Copyright [Martin Fowler](#), all rights reserved

A blue rectangular banner with a white border. On the left side is a 3D cube graphic with faces colored red, green, blue, and yellow, with the letters 'A', 'B', and 'C' on its visible faces. To the right of the cube, the text "ABC Amber CHM Converter Trial version" is displayed in red. Below this, in smaller black text, is "Please register to remove this banner." At the bottom right, the URL "http://www.processtext.com/abcchm.html" is shown in blue.

# Server Session State

---

*Keep the session state in an active process in memory that can be serialized into a memento if the memory resources are needed or the session migrates.*

## How it Works

The simplest form of this pattern occurs when a session object is held in memory on an application server. In this case you can simply have some kind of map in memory which holds these session objects keyed by a session id - and then all the client needs to do is to give the session id and the session object can be retrieved from the map to process the request.

This simple scenario assumes, of course, that the application server carries enough memory to be able to perform this task. It also assumes that there is only one application server, i.e. no clustering, and that if the application server fails then it is appropriate for the session to be abandoned and all work so far in the session lost the great bit-bucket in the sky.

For many application this set of assumptions is actually not a problem. However for others it may be problematic. But there are ways of dealing with cases when the assumptions are no longer valid, and these introduce common variations that add complexity to this essentially simple pattern.

The first issue is that of dealing with memory resources held by the session objects - which indeed is the common objection to *Server Session State*. The answer, of course, is to not keep them in memory but instead serialize all the session state to a [memento](#). This presents two questions: in what form do you persist the *Server Session State* and where do you persist the *Server Session State*.

The form to use is usually as simple a form as possible, since the accent of *Server Session State* is its simplicity in programming. Several platforms provide a simple binary serialization mechanism that allows you serialize a graph of objects quite easily. Another route is to serialize into another form. A common one to consider these days is a textual form, fashionably as an XML file.

The binary one is usually easier, since it requires little programming, while the textual form will usually require at least a little code. Binary serializations also require less disk space - and although total disk space will rarely be a concern, large serialized graphs will take longer to activate into memory.

There are two common issues with binary serialization. Firstly the serialized form is not human readable - which is a problem iff humans want to read it. Secondly there may be problems with versioning. If you modify a class, by say adding a field, after you've serialized it; you may not be able to read it back. Of

course not many sessions are likely to span an upgrade of the server software - unless of course it's a 7/24 server where you may have a cluster of machines running: some upgraded and some not.

This brings us to question of where to store the *Server Session State*. An obvious possibility is on the application server itself, either in the file system or in a local database. This is the simple route - but may not support efficient clustering or support fail over. In order to support these the passivated *Server Session State* needs to be somewhere generally accessible, such as shared server. This will now support clustering and fail over, at the cost of a longer time to activate the server - although caching may well eliminate much of this cost.

This line of reasoning may lead to the ironic route of storing the serialized *Server Session State* in the database using a session table indexed by the session id. This table would require a [\*Serialized LOB\*](#) to hold the serialized *Server Session State*. Database performance varies when it comes to handling large objects, so the performance aspects of this one is very database dependent.

At this point we are right at the boundary between *Server Session State* and [\*Database Session State\*](#). The boundary between these patterns is completely arbitrary - but I've drawn the line at the point where you convert the data in the *Server Session State* into tabular form.

If you're storing the *Server Session State* in a database, then you'll have to worry about handling sessions that have gone away, especially in a consumer application. One route is to have a daemon that looks for aged sessions and deletes them, but this can lead to a lot of contention on the session table. Kai Yu told about an approach he used with success. In this case they partitioned the session table into twelve database segments. Every two hours they would rotate the segments, deleting everything in the oldest segment and directing all inserts to the newly empty segment. While this did mean that any session that was active for twenty-four hours got unceremoniously dumped, that's sufficiently rare to not be a problem.

Although all of these variations get more and more effort to do, the good news is that increasingly application servers are supporting these capabilities automatically. So it may well be that these are things that the application server vendors can worry their ugly little heads about.

## **Java**

The two most common techniques for *Server Session State* are using the http session and using a stateful session bean. The http session is a simple route and leads to the session data being stored by the web server. In most cases this leads to server affinity and can't cope with fail over. Some vendors are implementing a shared http session capability which allows you to store http session data in a database that's available to all application servers. (You can also do this manually, of course.)

The other common route is to use a stateful session bean, which requires an EJB server. The EJB container handles all persistence and passivation, so this makes it very easy to program to. The main disadvantage is that the specification doesn't ask application servers to avoid server affinity. However some application servers do provide this kind of capability, for example IBM's WebSphere can serialize a stateful session bean into a BLOB in DB2, which allows multiple application servers to get at its state.

Another alternative is to use an entity bean. On the whole I've been pretty dismissive of entity beans, but you can use an entity bean to store a [\*Serialized LOB\*](#) of session data. This is pretty simple and is less likely to run into many of the issues that usually surround entity beans.

## .NET

*Server Session State* are easy to implement with the built in session state capability. By default .NET stores session data in the server process itself. You can also adjust the storage using a state service, which can reside on the local machine or on any other machine on the network. By using a separate state service you can reset the web server and still retain the session state. You make the change between in-process state and a state service in the configuration file, so you don't have to change the application.

## When to Use it

The great appeal of *Server Session State* is its simplicity. In a number of cases you don't have to do any programming at all to make this work. Whether you can get away with that depends on if you can get away with the in-memory implementation, and if not how much help your application server platform gives you.

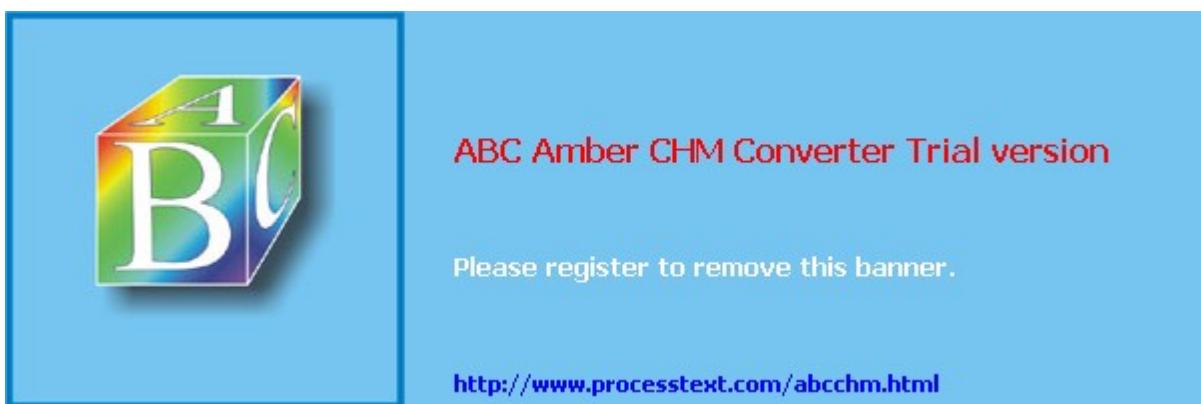
Even without that you may well find that the effort you do need is little. Serializing a BLOB to a database table may turn out to be much less effort than converting the server objects to tabular form.

Where the programming effort does come into play is in the session maintenance. Particularly if you have to roll your own support to enable clustering and fail-over that may work out to be more trouble than your other options, particularly if you don't have much session data to deal with, or if your session data is easily converted to tabular form.



---

© Copyright [Martin Fowler](#), all rights reserved

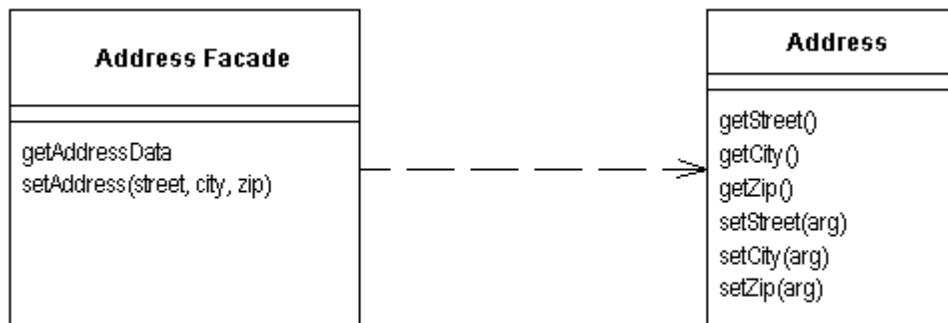


---

# Remote Facade

---

*Provides a coarse grained facade on fine grained objects to improve efficiency over a network.*



In an object-oriented model, you do best with small objects that have small methods. This gives you lots of opportunity for control and substitution of behavior, and to use good intention revealing naming to help an application be easier to understand. So if you have an address object you would store and retrieve each piece of information individually. You have separate fields for the city, state, and zip code. You have separate getting methods to get this data, and separate setting methods to update this information. This allows the class to control updates easily, deciding to do special validation on a zip or determining a city automatically if the address only has a zip code. Other behaviors, such as determining if two addresses are within a few miles of each other, are also done on a fine grained basis. This allows clients to clearly ask for exactly the information and behavior they want.

One of the consequences of this fine grained behavior is that there is usually a lot of interaction between objects, and that interaction usually requires lots of method invocations. Within a single address space, this is no great issue. Method calls are cheap these days, and clever compilers and VMs can eliminate even this small cost. In the very few cases where it does make a difference, it's a small job to optimize.

But this happy state does not exist when you make calls between processes. Remote calls are much more expensive because there is a lot more to do. Data may have to be marshaled, security may need to be checked, packets may need to be routed through switches. If the two processes are running on machines on opposite sides of the globe, the speed of light may be a factor. The brutal truth is that any inter-process call is orders of magnitude more expensive than an in-process call - even if both processes are on the same machine. Such a performance effect cannot be ignored, even for us believers in lazy optimization.

As a result any object that's intended to be used as a remote object needs a coarse grained interface that minimizes the amount of calls needed to get something done. If you want to change the city, state, and zip of a remote address; you want to do this in one call. Not just does this affect your method calls, it

also affects your objects. Rather than ask for an order and its order lines individually, you need to access and update the order and order lines in a single call. This affects your entire object structure.

This coarse grained interfaces come with a price. You give up the clear intention and fine-grained control that you get with small objects and small methods. Programming becomes more difficult and your productivity slows.

You can't rely on clever middleware to solve your problem. I've seen many sellers of distributed objects say that you can just take a regular object model and add CORBA (for instance) to make the whole thing distributed in a simple step. They point out that with their powerful software you only pay the price of a remote call if the call is actually remote. So you can program fine grained objects and only pay a performance cost should you actually use them remotely.

Frankly, that doesn't wash. The point is that if you use a fine grained object remotely, its performance will suck. If you use coarse-grained objects inside a process you lose productivity. There's no point taking a fine grained object and making it remotable just in case you want to call it remotely. Either it's a remote object or it isn't. The fundamental issue is that you need to have fine grained objects within a process and coarse grained objects between processes.

A *Remote Facade* is a coarse-grained facade over a web of fine-grained objects. None of the fine-grained objects have a remote interface and the *Remote Facade* does not contain any domain logic.

## How it Works

*Remote Facade* tackles the distribution problem by the standard OO approach of separating distinct responsibilities into different objects, and as a result has become the standard pattern for this problem. Firstly I recognize that fine-grained objects are the right answer for complex logic - so I ensure that any complex logic is placed in fine-grained objects which are designed to collaborate within a single process. To allow them to be accessed efficiently remotely I make a separate facade object to act as a remote interface. The facade, as the name implies, is merely a thin skin that switches from a coarse grained to a fine grained interface.

In a simple case, like an address object, a *Remote Facade* would replace all the getting and setting methods of the regular address object with one getter and one setter: often referred to as bulk accessors. When a client calls a bulk setting method, the address facade reads the data from the setting method and calls the individual accessors on the real address object (see Figure 1). The facade does nothing more than this. This way all the logic of validation and computation stays on the address object where it can factored cleanly and be used by other fine-grained objects.

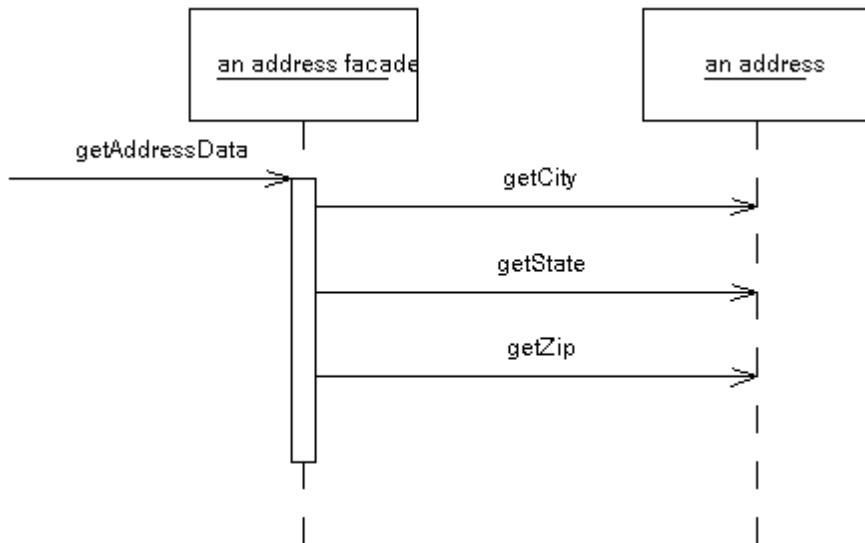


Figure 1: One call to a facade causes several calls from the facade to the domain object

In a more complex case a single *Remote Facade* may act as a remote gateway for many fine-grained objects. An order facade may be used to get and update information for an order, all its order lines, and maybe even some customer data as well.

In transferring information in bulk like this, you need to transfer this information in a form that can easily be moved over the wire. If your fine-grained classes are present on both sides of the connection, and these objects are serializable, then you can transfer the objects directly by making a copy. In this case a `getAddressData` method would create a copy of the original address object. The `setAddressData` would receive an address object and use this address to update the data of the actual address object. (This assumes that the original address object needs to preserve its identity and thus can't be just replaced with the new address.)

Often, however you can't do this. You may not want to duplicate your domain classes on multiple processes. Often it's difficult to serialize a segment of a domain model due to its complicated relationship structure. The client may not want the whole model, just a simplified subset of it. In these cases it makes sense to use a [Data Transfer Object](#) as the basis of the transfer.

In the sketch, I've shown a *Remote Facade* that corresponds to a single domain object. While this is not uncommon and is easy to understand, it isn't the most usual case. A single *Remote Facade* would have a number of methods each designed to pass on information from several objects. So `getAddressData` and `setAddressData` would be methods defined on a class like `CustomerService` which would also have methods along the lines of `getPurchasingHistory` and `updateCreditData`.

The question of granularity is one of the most tricky questions that comes up with *Remote Facade*. Some people like to make fairly small *Remote Facades*, such as one per use case. I prefer a coarser grained structure with much fewer *Remote Facades*. For even a moderate sized application I might have just a single *Remote Facade*. Even for a large application I may only have half a dozen *Remote Facade*s. This means that each *Remote Facade* has a lot of methods, but since these methods are small I don't see this as a problem.

You design a *Remote Facade* based on the needs of a particular client usage. The most common example would be the need to view and update information through a user interface. In this case you

might have a single *Remote Facade* for a family of screens with one bulk accessor method for loading and saving the data for each screen. Pressing buttons on a screen, to change a order's status, would invoke command methods on the facade. Quite often you'll have different methods on the *Remote Facade* that do pretty much the same thing on the underlying objects. This is common and reasonable. The facade is designed to make life simpler for external users, not the internal system, so if the client process thinks of it as a different command, it is a different command - even if it all goes to the same internal command.

*Remote Facade* can be stateful or stateless. A stateless *Remote Facade* can be pooled, and this can improve resource usage and improve performance, especially in a B2C situation. However if the interaction involves state across a session, then it needs to store session state somewhere using [Client Session State](#) or [Database Session State](#) or a some implementation of [Server Session State](#). A stateful *Remote Facade* can hold onto its own state, which makes for an easy implementation of [Server Session State](#). This may lead to performance issues when you have thousands of simultaneous users.

As well as providing a coarse-grained interface, several other responsibilities naturally go to the *Remote Facade*. The methods of a *Remote Facade* are a natural point to apply security. An access control list can say which users can invoke calls on which methods of a *Remote Facade*. The *Remote Facade* methods also are a natural point to apply transactional control. A *Remote Facade* method can start a transaction, do all the internal work and then commit the transaction at the end. Each call makes a good transaction since you don't want a transaction open when return goes back to the client, since transactions aren't built to be efficient for such long running cases.

## When to Use it

Use *Remote Facade* whenever you need remote access to a fine-grained object model. By using *Remote Facade* you gain the advantages of a coarse grained interface while still keeping the advantage of fine grained objects. This gives you the best of both worlds.

The most common case of using this is between a presentation and a [Domain Model](#) where they may run on different processes. You'll get this between a swing UI and server domain model, or with a servlet and a server object model if the application and web servers are different processes.

If all your access is within a single process, then you don't need this kind of conversion. So I wouldn't use this pattern to communicate between a client [Domain Model](#) and its presentation nor between a CGI script and [Domain Model](#) that are running all in one web server. You don't tend to see *Remote Facade* used with a [Transaction Script](#), since a [Transaction Script](#) is inherently a coarser grained structure.

*Remote Facade*'s imply a synchronous, that is a remote procedure call style of distribution. Often you can greatly improve the responsiveness of an application by going with an asynchronous, message based style of remote communication. Indeed there are many compelling advantages for an asynchronous approach. Sadly discussion of asynchronous patterns is outside the scope of this book.

## Further Reading

[Alur, Crupi, and Malks] discusses Session Facade in detail in the context of J2EE. Also take a look at [Kyle Brown's Session Facade](#) paper.

## Example: Using a Java Session Bean as a Remote Facade (Java)

If you are working with the Enterprise Java platform, then a good choice for a distributed facade is a session bean. Session beans are remote objects, they may be stateful or stateless. In this example I'll run a bunch of plain old Java objects inside an EJB container and access them remotely through a session bean that's designed as a *Remote Facade*. Session Bean's aren't particularly complicated, so this example should make sense even if you haven't done any work with them before.

I feel the need for a couple of side notes here. Firstly I've been surprised by how many people seem to believe that you can't run plain objects inside an EJB container in Java. I hear the question "are the domain objects Entity Beans?". They can be entity beans (that's a separate question), but they don't have to be. Simple Java objects work just fine - as in this example.

My other side note is just to point out that this is not the only way to use a session bean. It can be used to host [Transaction Script](#)s. I'll look at that usage elsewhere, in this case I'm using it as a *Remote Facade*.

In this example I'll look at remote interfaces to accessing information about music albums. The [Domain Model](#) consists of fine grained objects that represent an artist, and album, and tracks. Surrounding this are several other packages that provide the data sources for the application Figure 2.

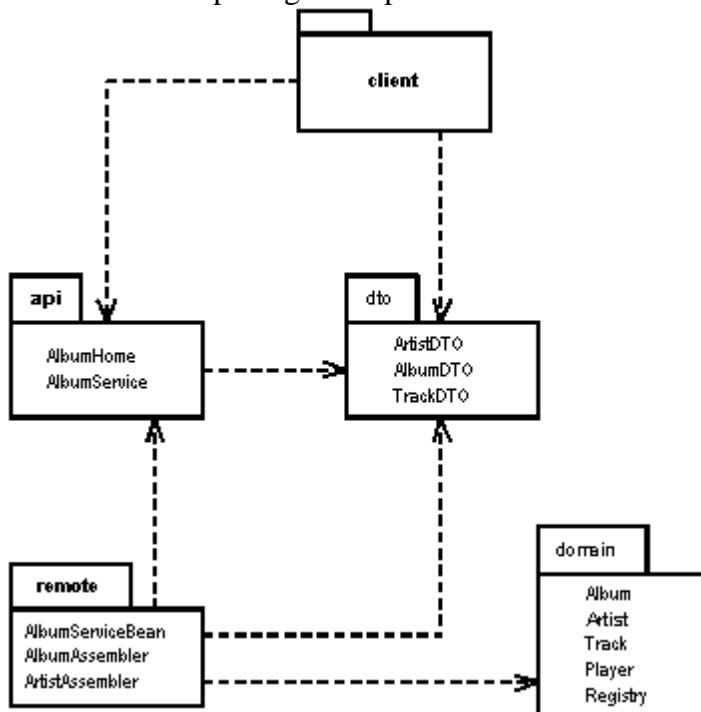


Figure 2: Packages the remote interfaces

The dto package contains [Data Transfer Objects](#) that help us move data over the wire to the client. They have simple accessor behavior and also the ability to serialize themselves in binary or XML textual formats. In the remote package are assembler objects that will move data between the domain objects and the [Data Transfer Objects](#). If you're interested in how this works see the discussion of [Data Transfer Object](#).

To explain the facade I'll assume I can move data back and forth into [Data Transfer Objects](#) and concentrate on the remote interfaces. A single logical Java session bean has three actual classes. Two of these make up the remote API (and in fact are Java interfaces) the other is the class that implements the API. The two interfaces are the AlbumService itself and the home object: AlbumHome. The home object is used by the naming service to get access to the distributed facade, that's a detail on EJB that I'll skip over here. Our interest is in the *Remote Facade* itself: which is AlbumService. Its interface is declared in the API package to be used by the client, and is just a list of methods.

```
class AlbumService...
String play(String id) throws RemoteException;

String getAlbumXml(String id) throws RemoteException;

AlbumDTO getAlbum(String id) throws RemoteException;

void createAlbum(String id, String xml) throws RemoteException;

void createAlbum(String id, AlbumDTO dto) throws RemoteException;

void updateAlbum(String id, String xml) throws RemoteException;

void updateAlbum(String id, AlbumDTO dto) throws RemoteException;

void addArtistNamed(String id, String name) throws RemoteException;

void addArtist(String id, String xml) throws RemoteException;

void addArtist(String id, ArtistDTO dto) throws RemoteException;

ArtistDTO getArtist(String id) throws RemoteException;
```

You'll notice that even in this short example, I see methods for a two different classes in the [Domain Model](#): artist and album. I also see minor variations on the same method. Methods have variants that use either the [Data Transfer Object](#) or an XML string to move data into the remote service. This allows the client to choose which form to use depending on the nature of the client and of the connection. As you can see, for even a small application this can lead to many methods on AlbumService.

Fortunately the methods themselves are very simple. Here are the ones for manipulating albums.

```
class AlbumServiceBean...
public AlbumDTO getAlbum(String id) throws RemoteException {
return new AlbumAssembler().writeDTO(Registry.findAlbum(id));
}

public String getAlbumXml(String id) throws RemoteException {
AlbumDTO dto = new AlbumAssembler().writeDTO(Registry.findAlbum(id));
return dto.toXmlString();
}

public void createAlbum(String id, AlbumDTO dto) throws RemoteException {
new AlbumAssembler().createAlbum(id, dto);
```

```
}
```

```
public void createAlbum(String id, String xml) throws RemoteException {
    AlbumDTO dto = AlbumDTO.readXmlString(xml);
    new AlbumAssembler().createAlbum(id, dto);
}
```

```
public void updateAlbum(String id, AlbumDTO dto) throws RemoteException {
    new AlbumAssembler().updateAlbum(id, dto);
}
```

```
public void updateAlbum(String id, String xml) throws RemoteException {
    AlbumDTO dto = AlbumDTO.readXmlString(xml);
    new AlbumAssembler().updateAlbum(id, dto);
}
```

As you can see, each method really does nothing more than delegate to another object, so each method is only a line or two in length (ignoring the noise of the try/catch blocks). This small example illustrates pretty nicely what a distributed facade should look like: a long list of very short methods with very little logic in them. The facade then is nothing more than a packaging mechanism - which is as it should be.

We'll just finish with a few words on testing. It's very useful to be able to do as much testing as possible in a single process. In this case I can do this by writing tests for the session bean implementation directly: these can be run without deploying to the EJB container.

```
class XmlTester...
private AlbumDTO kob;
private AlbumDTO newkob;
private AlbumServiceBean facade = new AlbumServiceBean();

protected void setUp() throws Exception {
    facade.initializeForTesting();
    kob = facade.getAlbum("kob");
    Writer buffer = new StringWriter();
    kob.toXmlString(buffer);
    newkob = AlbumDTO.readXmlString(new StringReader(buffer.toString()));
}

public void testArtist() {
    assertEquals(kob.getArtist(), newkob.getArtist());
}
```

This is one of the jUnit tests to be run in-memory. It shows how I can create an instance of the session bean outside the container and run tests on it. This allows for a faster testing turn-around time.

## Example: Web Service (.NET)

I was talking over this book with Mike Hendrickson, my editor at Addison-Wesley. Ever alert to the latest buzzwords, he asked me if I'd got anything about web services in it. I'm actually loath to rush to every fashion in these books, after due to the languid pace of book publishing any latest fashion that I

write about will seem quaint by the time you read it. But it's a good example of how core patterns so often keep their value even with the latest technological flip-flops.

At its heart a web service is nothing more than an interface for remote usage. As such the basic advice of *Remote Facade* still holds: build your functionality in a fine-grained manner, and then layer a *Remote Facade* over the fine-grained model in order to handle web services.

For the example, I'll use the same basic problem as I described above, concentrating just on the request for information about a single album. Figure 3 shows the various classes that take part. They fall into the familiar groups: Album Service is the *Remote Facade*, two *Data Transfer Objects*, three objects in a *Domain Model* and an assembler to pull data from the *Domain Model* into the *Data Transfer Objects*.

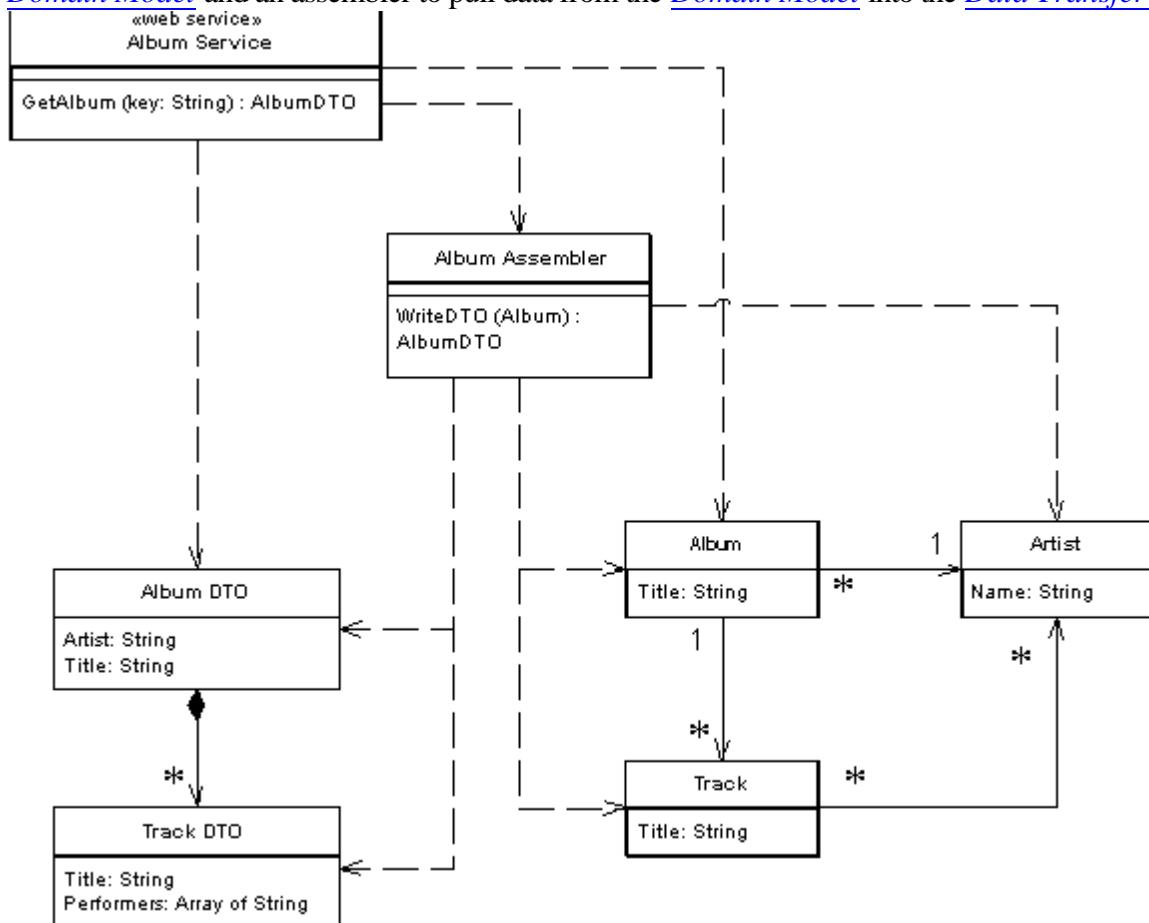


Figure 3: Classes for the album web service

The *Domain Model* is absurdly simple, indeed for this kind of problem you'd be better off using a *Table Data Gateway* to create the *Data Transfer Objects* directly - but that would rather spoil the example of showing a *Remote Facade* over a domain model.

```

class Album...
public String Title;
public Artist Artist;
public IList Tracks {
get {return ArrayList.ReadOnly(tracksData);}
}
public void AddTrack (Track arg) {
tracksData.Add(arg);
}
public void RemoveTrack (Track arg) {
tracksData.Remove(arg);
}
  
```

```
}
```

```
private IList tracksData = new ArrayList();
```

```
    class Artist...
```

```
public String Name;
```

```
    class Track...
```

```
public String Title;
```

```
public IList Performers {
```

```
get {return ArrayList.ReadOnly(performersData);}
```

```
}
```

```
public void AddPerformer (Artist arg) {
```

```
performersData.Add(arg);
```

```
}
```

```
public void RemovePerformer (Artist arg) {
```

```
performersData.Remove(arg);
```

```
}
```

```
private IList performersData = new ArrayList();
```

I use [Data Transfer Objects](#) for passing the data over the wire. These are just data holders which flatten the structure for the purposes of the web service.

```
class AlbumDTO...
```

```
public String Title;
```

```
public String Artist;
```

```
public TrackDTO[] Tracks;
```

```
    class TrackDTO...
```

```
public String Title;
```

```
public String[] Performers;
```

Since this is .NET, I don't need to write any code to serialize and restore into XML. The .NET framework comes with the appropriate serializer class to do the job.

Since this is a web service, I also need to declare the structure of the [Data Transfer Objects](#) in WSDL. The Visual Studio tools will generate the WSDL for me, and I'm a lazy kind of guy, so I'll let it do that. Here's the XML Schema definition that corresponds to the [Data Transfer Objects](#).

```
<s:complexType name="AlbumDTO">
```

```
  <s:sequence>
```

```
    <s:element minOccurs="1" maxOccurs="1" name="Title" nillable="true"
```

```
      type="s:string" />
```

```
    <s:element minOccurs="1" maxOccurs="1" name="Artist" nillable="true"
```

```
      type="s:string" />
```

```
    <s:element minOccurs="1" maxOccurs="1" name="Tracks" nillable="true"
```

```
      type="s0:ArrayOfTrackDTO" />
```

```
  </s:sequence>
```

```
</s:complexType>
```

```
<s:complexType name="ArrayOfTrackDTO">
```

```
  <s:sequence>
```

```
    <s:element minOccurs="0" maxOccurs="unbounded" name="TrackDTO"
```

```
      nillable="true" type="s0:TrackDTO" />
```

```
  </s:sequence>
```

```
</s:complexType>
```

```
<s:complexType name="TrackDTO">
```

```
  <s:sequence>
```

```
    <s:element minOccurs="1" maxOccurs="1" name="Title" nillable="true"
```

```
      type="s:string" />
```

```
    <s:element minOccurs="1" maxOccurs="1" name="Performers" nillable="true"
```

```
type="s0:ArrayOfString" />
  </s:sequence>
</s:complexType>
<s:complexType name="ArrayOfString">
  <s:sequence>
    <s:element minOccurs="0" maxOccurs="unbounded" name="string"
nillable="true" type="s:string" />
  </s:sequence>
</s:complexType>
```

Being XML, it's a particularly verbose form of data structure definition, but it does the job.

To get the data from the [Domain Model](#) to the [Data Transfer Object](#) I need an assembler.

```
class AlbumAssembler...
public AlbumDTO WriteDTO (Album subject) {
AlbumDTO result = new AlbumDTO();
result.Artist = subject.Artist.Name;
result.Title = subject.Title;
ArrayList trackList = new ArrayList();
foreach (Track t in subject.Tracks)
trackList.Add (WriteTrack(t));
result.Tracks = (TrackDTO[]) trackList.ToArray(typeof(TrackDTO));
return result;
}
public TrackDTO WriteTrack (Track subject) {
TrackDTO result = new TrackDTO();
result.Title = subject.Title;
result.Performers = new String[subject.Performers.Count];
ArrayList performerList = new ArrayList();
foreach (Artist a in subject.Performers)
performerList.Add (a.Name);
result.Performers = (String[]) performerList.ToArray(typeof (String));
return result;
}
```

The last piece we need is the service definition itself. This comes first from the C# class.

```
class AlbumService...
[ WebMethod ]
public AlbumDTO GetAlbum(String key) {
Album result = new AlbumFinder()[key];
if (result == null)
throw new SoapException ("unable to find album with key: " +
key, SoapException.ClientFaultCode);
else return new AlbumAssembler().WriteDTO(result);
}
```

Of course, this isn't the real interface definition - that comes from the WSDL file. Here are the relevant bits

```
<portType name="AlbumServiceSoap">
<operation name="GetAlbum">
<input message="s0:GetAlbumSoapIn" />
<output message="s0:GetAlbumSoapOut" />
</operation>
</portType>
```

```
<message name="GetAlbumSoapIn">
<part name="parameters" element="s0:GetAlbum" />
</message>
<message name="GetAlbumSoapOut">
<part name="parameters" element="s0:GetAlbumResponse" />
</message>
<s:element name="GetAlbum">
<s:complexType>
<s:sequence>
<s:element minOccurs="1" maxOccurs="1" name="key" nillable="true"
type="s:string" />
</s:sequence>
</s:complexType>
</s:element>
<s:element name="GetAlbumResponse">
<s:complexType>
<s:sequence>
<s:element minOccurs="1" maxOccurs="1" name="GetAlbumResult"
nillable="true" type="s0:AlbumDTO" />
</s:sequence>
</s:complexType>
</s:element>
```

As expected, WSDL is rather more garrulous than your average politician, but unlike so many of them - it does get the job done. I can now invoke the service by sending a SOAP message of the form

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
<soap:Body>
<GetAlbum xmlns="http://martinfowler.com">
<key>aKeyString</key>
</GetAlbum>
</soap:Body>
</soap:Envelope>
```

The important thing to remember about this example isn't the cool gyrations with SOAP and .NET, but the fundamental layering approach. Design an application without distribution, then layer the distribution ability on top of that with *Remote Facades* and [\*Data Transfer Objects\*](#).





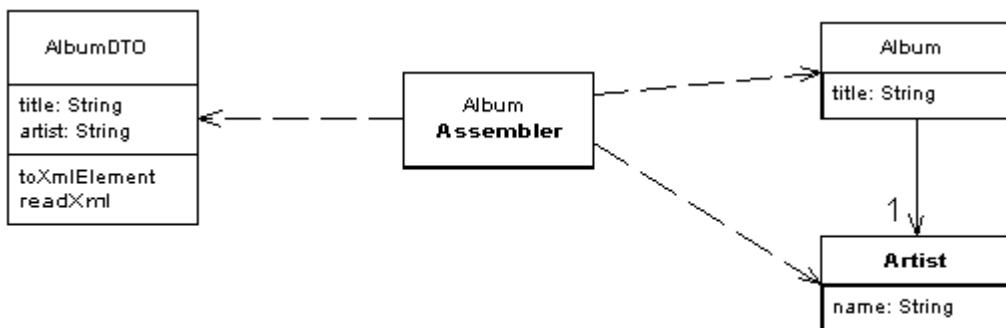
ABC Amber CHM Converter Trial version

Please register to remove this banner.

<http://www.processtext.com/abcchm.html>

# Data Transfer Object

An object which acts purely as a carrier of data in order to reduce method calls.



When you're working with a remote interface, such as a [Remote Facade](#), you need to be able to send and receive a large amount of information in a single call. One way to do this is to use lots of parameters. However this is often awkward to program and indeed it's often impossible, such as with languages like Java that only return a single return value.

So the way to deal with this is to create a *Data Transfer Object* which can hold all the data for the call. This object usually needs to be serializable to go across the connection.

(Many people in the Sun community use the term **Value Object** for this pattern. I use [Value Object](#) to mean something else, see the discussion there for the background to this.)

## How it Works

In many ways, a *Data Transfer Object* is one of those objects our mothers told us never to write. A *Data Transfer Object* is often little more than a bunch of fields and getters and setters for these fields. The value of this usually hateful beast is that it allows you to move several pieces of information over a network in a single call - a trick that's essential for distributed systems.

When using a *Data Transfer Object* whenever a remote object needs some data, it asks for a suitable *Data Transfer Object*. The *Data Transfer Object* will usually carry much more data than the specific item that the remote object wants from this request, but should carry all the data that the remote object will need for a while. Due to the latency costs of remote calls, its better to err on the side of sending too much data, rather than run into a situation where multiple calls are needed.

A single *Data Transfer Object* will usually contain more than just a single server object. A *Data Transfer Object* will usually aggregate data from all the server objects that the remote object is likely to want data from. So if a remote object requests data about an order object, the returned *Data Transfer Object* will contain data from the order, customer, the line items, the products on the line items, the delivery information, all sorts of stuff.

As a result it usually makes sense to design the *Data Transfer Object* around the needs of a particular client. As a result you'll often see *Data Transfer Objects* corresponding to web pages or GUI screens. Similarly you may see multiple *Data Transfer Objects* for an order, depending on the particular screen, of course if different presentations require similar data, then it makes sense to use a single *Data Transfer Object* to handle them all.

A related question to consider is whether to use a single *Data Transfer Object* for a whole interaction, or to use different *Data Transfer Objects* for each request and response. Using different *Data Transfer Objects* makes it easier to see what data is transferred in each call, but leads to a lot of *Data Transfer Objects*. The single *Data Transfer Object* is less work to write, but makes it harder to see how each call transfers information. I'm inclined to use a single *Data Transfer Object* if there's a lot of commonality over the data, but I don't hesitate to use different *Data Transfer Objects* if a particular request suggests it. It's one of those things where you can't really make a blanket rule, so I might use one *Data Transfer Object* for most of the interaction and use different ones for a couple of requests and responses.

A similar question is whether to have a single *Data Transfer Object* for both request and response, or separate *Data Transfer Objects* for each. Again there's no blanket rule. If the data in each case is pretty similar, then I'd use one. If they are very different I'd use two.

The fields in a *Data Transfer Object* are usually fairly simple. They are either primitives, simple classes like strings and dates, or other *Data Transfer Objects*. This is for a couple of reasons. Firstly they have to be serializable into whatever transport format is being used. Secondly they need to be understood by both sides of the wire.

Some people like to make *Data Transfer Objects* immutable. In this scheme you receive one *Data Transfer Object* from the client and create and send back a different object, even if it's the same class. Others will alter the request *Data Transfer Object*. I don't have any strong opinions either way. On the whole I prefer a mutable *Data Transfer Object* because that way it's easier to put the data in gradually, even if you do make a new object for the response. Some arguments in favor of immutable *Data Transfer Object* are due to the naming confusion with [Value Object](#).

A common form for *Data Transfer Object* is that of a record set. A record set is a set of tabular records, exactly what you get back from a SQL query. Indeed a record set is the *Data Transfer Object* for a SQL database. Often architectures use this form further through the design. Many GUI tools are designed to work with record sets, so providing a record set to such tools makes a lot of sense. A number of platforms, in particular those with an origin in two-layer client server systems, provide the ability for other layers to create record sets. In this way a domain model could generate a record set of data to transfer to a client, that client treats the record set as if it was coming directly from SQL. The record set can be entirely created by the domain logic, or more likely it's generated from a SQL query and modified by the domain logic before it's passed on to the presentation.

## Serializing the *Data Transfer Object*

Other than simple getters and setters, the *Data Transfer Object* is also usually responsible for serializing itself into some format that will go over the wire. Which form to choose depends on what runs on both sides of the connection, what can run over the connection itself, and the ease of doing the serialization. A number of platforms provide built in serialization for simple objects. For example Java has a built-in binary serialization and .NET has built in binary and XML serializations. If there is a built in serialization this usually works right out of the box as *Data Transfer Object* are simple structures that don't need to deal with the kinds of complexities that you run into with objects in a domain model. As a result I always use the automatic mechanism if I can.

If you don't have an automatic mechanism already available, you can usually create one yourself. In several cases I've seen code generators that take a simple record descriptions and generate appropriate classes to hold the data, provide accessors, and to read and write serializations of that data. The important thing to remember is to make the generator only as complicated as you actually need: don't try to put in features that you think you may need. Often it's a good idea to write the first classes by hand, and use the hand written classes to help you write the generator.

You also have to choose a mechanism that both ends of the connection will work with. If you control both ends, then you can choose the easiest one that will work. If you don't have both ends, you may be able to provide a connector at the end you don't own. Then you can use a simple *Data Transfer Object* on both sides of the connection and use the connector to adapt to the foreign component.

One of the most common issues to face is whether to use a text or binary serialization form. Text serializations have the advantage that they are easy to read to see what's being communicated. XML is a popular choice since you can easily get tools to create and parse XML documents. The big disadvantage with text is that it more bandwidth to send the same data: something that is particularly true of XML.

An important factor for serialization is dealing with the synchronization of a the *Data Transfer Object* on each side of the wire. In theory whenever the server changes the definition of the *Data Transfer Object* the client will update as well. In practice, however, this may not happen. Accessing a server with an out of date client is always going to lead to some kinds of problem, but the serialization mechanism can make the problems more or less painful. With a pure binary serialization of a *Data Transfer Object* the result will be that communication of that *Data Transfer Object* will be entirely lost, since any change to the structure of the *Data Transfer Object* will usually cause the an error on deserialization. Even an innocuous change, such as adding an optional field, will have this effect. As a result direct binary serialization can introduce a lot of fragility into the communication lines.

Other serialization schemes can avoid this. The XML serialization can usually be written in such a way that makes the classes more tolerant of these kinds of changes. Another approach is to use a more tolerant binary approach. Serializing the data using a dictionary is one way to do this. Although I don't like using a dictionary as the *Data Transfer Object*, it can be a useful way of doing a binary serialization of the data, since that introduces some tolerance into the synchronization.

## Assembling a *Data Transfer Object* from domain objects

A *Data Transfer Object* does not know about how to connect with the domain objects. This is because the *Data Transfer Object* is something that should be deployed on both sides of the connection, so I don't want the *Data Transfer Object* to be dependent on the domain object. Similarly

I don't want the domain objects to be dependent of the *Data Transfer Object* since the structure of the *Data Transfer Object* will change when I alter interface formats. As a general rule I want to keep the domain model independent of the external interfaces.

As a result of this I like to have a separate assembler object that is responsible for creating a *Data Transfer Object* from the domain model and updating the domain model from a *Data Transfer Object*.

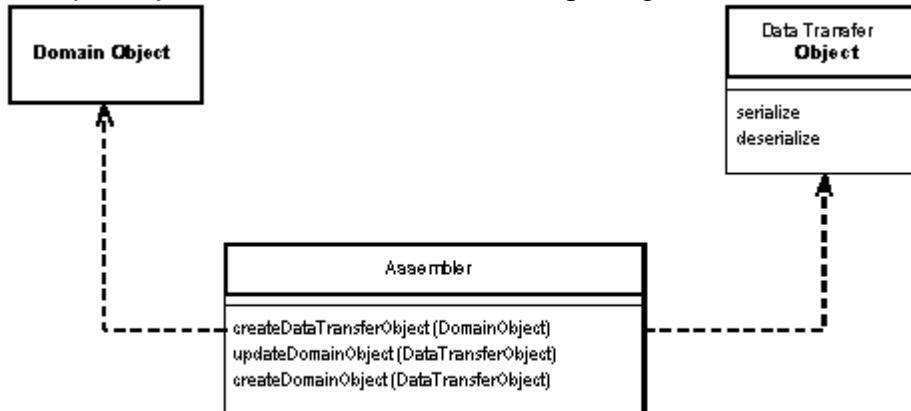


Figure 1: An assembler object can keep the domain model and data transfer objects independent of each other

The attributes of the data transfer object will either be simple value objects or other data transfer objects. Any structure between data transfer objects should be a very simple graph structure - usually a hierarchy - as opposed to the more complicated graph structures that you see in a [Domain Model](#)

When I choose a structure for a *Data Transfer Object* I prefer to base the structure on the needs of the client of the *Data Transfer Object*. So if I'm presenting information in a web page, I'll design the *Data Transfer Object* around the structure of the web page. This may well mean that you'll have multiple *Data Transfer Objects* for different presentations. This is quite reasonable. Whichever way I do it I need to do transformation from the structure of the domain model to the structure of the presentation - and the assembler is in the best position to do this because it is able to take advantage of the full richness of the domain model's associations and behavior to construct the *Data Transfer Object*. I make an exception for simple variations, so if one presentation shows a subset of data of another I would have them share the same *Data Transfer Object*.

I may also find multiple assemblers sharing the same *Data Transfer Object*. A common case for this is when you have different update semantics in different scenarios using the same data.

## When to Use it

Use a *Data Transfer Object* whenever you need to transfer multiple items of data between two processes in a single method call.

There are some alternatives to *Data Transfer Object*, although I'm not a fan of them. One is to not use an object at all, but simply to use a setting method with many arguments or a getting method with several pass by reference arguments. This runs into a few problems. Many languages, such as Java, only allow one object as a return value. So although this can be used for updates, it can't be used for retrieving

information without playing games with callbacks. Even when it is possible the *Data Transfer Object* makes an excellent boundary point for testing, allowing you run tests in one process without connecting to other processes. It also allows you to queue calls, rather like a command object, and encapsulates the serialization approach.

Another alternative is to use a collection class to hold the data. I've seen arrays used for this - but I discourage that because the array indices obscure the code. The best collection to use is a dictionary, since you can use meaningful strings as keys. The problem with a dictionary is you lose the advantage of an explicit interface and strong typing. It can be worth using for ad hoc cases when you don't have a generator to hand, as it's easier to manipulate a dictionary than to write a *Data Transfer Object* by hand. However with a generator, I think you're better off with an explicit interface, especially when you consider that it is being used as communication protocol between different components.

In particular it's worth creating a *Data Transfer Object* when you want to use XML to communicate between components. The XML DOM is a pain in the neck to manipulate, and it's much better to use a *Data Transfer Object* that encapsulates the DOM, especially since it's so easy to generate the *Data Transfer Object*.

Another common purpose for a *Data Transfer Object* is to act as a common source of data for various different components in different layers. Each component takes the *Data Transfer Object*, makes some changes to it, and then passes on to the next layer. The use of [Record Set](#) in COM and .NET is a good example of this, where each layer knows how to manipulate record set based data, whether it comes directly from a SQL database, or has been modified by other layers. .NET expands on this by providing a built in mechanism to serialize record sets into XML.

## Further Reading

[\[Alur, Crupi, and Malks\]](#) discuss this pattern under the name *value object*. Remember that the [\[Alur, Crupi, and Malks\]](#) is equivalent to my *Data Transfer Object*; my [Value Object](#) is a different pattern entirely. This is a name collision, many people have used value object in the sense that I use it. As far as I can tell the use of value object to mean what I call *Data Transfer Object* is only done within the J2EE community. As a result I've followed the more general usage.

## Example: Transferring information about albums (Java)

For this example I'll use the domain model in Figure 2. The data I want to transfer is the data about these linked objects and the structure for the data transfer objects is the one in Figure 3

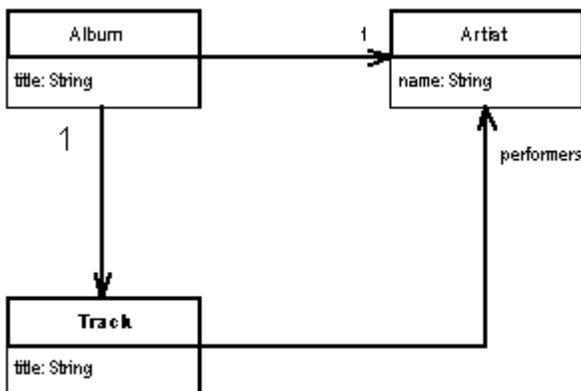


Figure 2: A class diagram of artists and albums

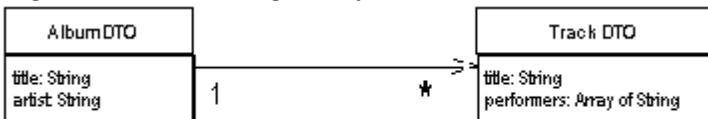


Figure 3: A class diagram the data transfer objects

The data transfer objects simply the structure a good bit. The name of the artist is collapsed into the album DTO and the performers for a track are represented as an array of strings. This is typical of the kind of collapsing of structure you see for a data transfer object. There are two data transfer objects present, one for the album and one for each track. In this case I don't need a transfer object for the artist as all the data is present on one of the other two. I only have the track as a transfer object because there are several of them in the album and each one can contain more than one data item.

Here's the code to write a *Data Transfer Object* from the domain model.

```
class AlbumAssembler...
```

```

public AlbumDTO writeDTO(Album subject) {
    AlbumDTO result = new AlbumDTO();
    result.setTitle(subject.getTitle());
    result.setArtist(subject.getArtist().getName());
    writeTracks(result, subject);
    return result;
}

private void writeTracks(AlbumDTO result, Album subject) {
    List newTracks = new ArrayList();
    Iterator it = subject.getTracks().iterator();
    while (it.hasNext()) {
        TrackDTO newDTO = new TrackDTO();
        Track thisTrack = (Track) it.next();
        newDTO.setTitle(thisTrack.getTitle());
        writePerformers(newDTO, thisTrack);
        newTracks.add(newDTO);
    }
    result.setTracks((TrackDTO[]) newTracks.toArray(new TrackDTO[0]));
}

private void writePerformers(TrackDTO dto, Track subject) {
    List result = new ArrayList();
    Iterator it = subject.getPerformers().iterator();
  
```

```
while (it.hasNext()) {
Artist each = (Artist) it.next();
result.add(each.getName());
}
dto.setPerformers((String[]) result.toArray(new String[0]));
}
```

Updating the model from the *Data Transfer Object* is usually a good bit more involved. For this example there is a difference between creating a new album and updating an existing one. Here's the creation code.

```
class AlbumAssembler...
public void createAlbum(String id, AlbumDTO source) {
Artist artist = Registry.findArtistNamed(source.getArtist());
if (artist == null)
throw new RuntimeException("No artist named " + source.getArtist());
Album album = new Album(source.getTitle(), artist);
createTracks(source.getTracks(), album);
Registry.addAlbum(id, album);
}

private void createTracks(TrackDTO[] tracks, Album album) {
for (int i = 0; i < tracks.length; i++) {
Track newTrack = new Track(tracks[i].getTitle());
album.addTrack(newTrack);
createPerformers(newTrack, tracks[i].getPerformers());
}
}

private void createPerformers(Track newTrack, String[] performerArray) {
for (int i = 0; i < performerArray.length; i++) {
Artist performer = Registry.findArtistNamed(performerArray[i]);
if (performer == null)
throw new RuntimeException("No artist named " + performerArray[i]);
newTrack.addPerformer(performer);
}
}
```

Reading the DTO involves quite a few decisions. The noticeable one here is how to deal with the artist names as they come in. In this case our requirements are that artists should already be in the registry when I create the album, so if I can't find an artist this is an error. A different create method might decide to create artists when they are mentioned in the *Data Transfer Object*.

For this example I have a different method for updating an existing album.

```
class AlbumAssembler...
public void updateAlbum(String id, AlbumDTO source) {
Album current = Registry.findAlbum(id);
if (current == null)
throw new RuntimeException("Album does not exist: " + source.getTitle());
if (source.getTitle() != current.getTitle())
current.setTitle(source.getTitle());
if (source.getArtist() != current.getArtist().getName()) {
Artist artist = Registry.findArtistNamed(source.getArtist());
if (artist == null)
throw new RuntimeException("No artist named " + source.getArtist());
current.setArtist(artist);
}
```

```
updateTracks(source, current);
}

private void updateTracks(AlbumDTO source, Album current) {
for (int i = 0; i < source.getTracks().length; i++) {
current.getTrack(i).setTitle(source.getTrackDTO(i).getTitle());
current.getTrack(i).clearPerformers();
createPerformers(current.getTrack(i), source.getTrackDTO(i).getPerformers());
}
}
```

When you do updates you can decide to either update the existing domain object or to destroy the existing domain object and replace it with a new one. The question here is whether you have other objects referring to the object you want to update. So in this code I'm updating the album since I have other objects referring to the existing album and tracks. However for the title and performers of a track I just replace the objects that are there.

Another question is that if an artist changes, is this changing the name of the existing artist, or changing the artist that the album is linked to. Again these questions have to be settled on a use case by use case basis, and I'm handling it by linking to a new artist.

In this example I've used the native binary serialization, this means I have to be careful that the *Data Transfer Object* classes on both sides of the wire are kept in sync. If I make a change to the data structure of the server *Data Transfer Object* and don't change the client, then I'll get errors in the transfer. I can make the transfer more tolerant by using a Map as my serialization.

```
class TrackDTO...
public Map writeMap() {
Map result = new HashMap();
result.put("title", title);
result.put("performers", performers);
return result;
}
public static TrackDTO readMap(Map arg) {
TrackDTO result = new TrackDTO();
result.title = (String) arg.get("title");
result.performers = (String[]) arg.get("performers");
return result;
}
```

Now if I add a field to the server and use the old client, then although the new field won't be picked up by the client, the rest of the data will transfer correctly.

Of course, writing the serialization and deserialization routines like this is tedious work. I can avoid much of this tedium by using a reflective routine such as this on the [Layer Supertype](#).

```
class DataTransferObject...
public Map writeMapReflect() {
Map result = null;
try {
Field[] fields = this.getClass().getDeclaredFields();
result = new HashMap();
for (int i = 0; i < fields.length; i++)
result.put(fields[i].getName(), fields[i].get(this));
} catch (Exception e) {throw new ApplicationException (e);
}
```

```
return result;
}
public static TrackDTO readMapReflect(Map arg) {
TrackDTO result = new TrackDTO();
try {
Field[] fields = result.getClass().getDeclaredFields();
for (int i = 0; i < fields.length; i++)
fields[i].set(result, arg.get(fields[i].getName()));
} catch (Exception e) {throw new ApplicationException (e);
}
return result;
}
```

This kind of routine will handle most cases pretty well (although you'll have to add extra code to handle primitives.)

## Example: Serializing using XML (Java)

Once I have the data structure for the *Data Transfer Object*, I need to decide how to serialize it. In Java you get binary serialization for free by simply using a marker interface. This works completely automatically for a *Data Transfer Object* so it's our first choice. However often it's necessary to use a text based serialization so I can easily send it over http. So for this example I'll use XML.

For this example, I'm using JDOM, since that makes working with XML much easier than using the W3C standard interfaces. For each *Data Transfer Object* class I write methods to read and write an XML element to represent that class.

```
class AlbumDTO...
Element toXmlElement() {
Element root = new Element("album");
root.setAttribute("title", title);
root.setAttribute("artist", artist);
for (int i = 0; i < tracks.length; i++)
root.addContent(tracks[i].toXmlElement());
return root;
}

static AlbumDTO readXml(Element source) {
AlbumDTO result = new AlbumDTO();
result.setTitle(source.getAttributeValue("title"));
result.setArtist(source.getAttributeValue("artist"));
List trackList = new ArrayList();
Iterator it = source.getChildren("track").iterator();
while (it.hasNext())
trackList.add(TrackDTO.readXml((Element) it.next()));
result.setTracks((TrackDTO[]) trackList.toArray(new TrackDTO[0])));
return result;
}

class TrackDTO...
Element toXmlElement() {
Element result = new Element("track");
result.setAttribute("title", title);
for (int i = 0; i < performers.length; i++) {
Element performerElement = new Element("performer");
```

```
performerElement.setAttribute("name", performers[i]);
result.addContent(performerElement);
}
return result;
}

static TrackDTO readXml(Element arg) {
TrackDTO result = new TrackDTO();
result.setTitle(arg.getAttributeValue("title"));
Iterator it = arg.getChildren("performer").iterator();
List buffer = new ArrayList();
while (it.hasNext()) {
Element eachElement = (Element) it.next();
buffer.add(eachElement.getAttributeValue("name"));
}
result.setPerformers((String[]) buffer.toArray(new String[0])));
return result;
}
```

Of course these only create the elements in the XML DOM. To perform the serialization I need to read and write text. Since the track is only transferred in the context of the album, I only need to write this code for the album.

```
class AlbumDTO...
public void toXmlString(Writer output) {
Element root = toXmlElement();
Document doc = new Document(root);
XMLOutputter writer = new XMLOutputter();
try {
writer.output(doc, output);
} catch (IOException e) {
e.printStackTrace();
}
}

public static AlbumDTO readXmlString(Reader input) {
try {
SAXBuilder builder = new SAXBuilder();
Document doc = builder.build(input);
Element root = doc.getRootElement();
AlbumDTO result = readXml(root);
return result;
} catch (Exception e) {
e.printStackTrace();
throw new RuntimeException();
}
}
```



ABC Amber CHM Converter Trial version

Please register to remove this banner.

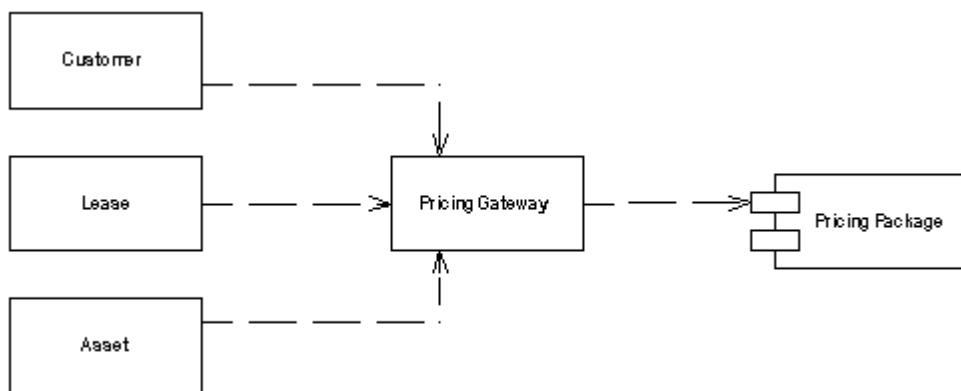
<http://www.processtext.com/abcchm.html>

---

# Gateway

---

*An object that encapsulates access to an external system or resource*



Interesting software rarely lives in isolation. Even the most carefully pure object-oriented system often has to deal with things that aren't objects: relation database tables, CICS transactions, XML data structures.

When accessing external resources like this, you'll usually get an API for that resource. However these APIs are naturally going to be somewhat complicated as they take the nature of the resource into mind. Anyone who needs to understand that resource needs to understand that API: whether it be JDBC and SQL for relational databases, or the w3c or JDOM APIs for XML. Not just does this make the software harder to understand, it also makes it much harder to change should you shift some data from a relational database to an XML message at some point in the future.

The answer is so common, that it's hardly worth stating. Wrap all the special API code into a class whose interface looks like a regular object. Other objects access the resource through this *Gateway*, and the *Gateway* object translates the simple method calls into the appropriate specialized API.

## How it Works

In reality this is a very simple wrapper pattern. Take the external resource. Consider what the application needs to do with this resource. Create a simple API for your usage and use the *Gateway* to translate to the external source.

One of the key uses for a *Gateway* is to make a good point to apply a [\*Service Stub\*](#). Often you'll find

you may alter the design of the *Gateway* to make it easier to apply a [\*Service Stub\*](#). Don't be afraid to do this - well placed [\*Service Stub\*](#)s can make a system much easier to test and that will make the system easier to write.

Keep a *Gateway* as simple as you can. Focus on the essential roles of adapting the external service and providing a good point for stubbing. Keep the *Gateway* as minimal as possible yet able to handle these tasks. Any more complex logic should be in clients of the *Gateway*.

Often it's a good idea to use code generation to create *Gateways*. From a definition of the structure of the external resource, you can generate a *Gateway* class to wrap it. You might use relational meta data to create a wrapper class for a relational table, or an XML schema or DTD to generate code for a *Gateway* for XML. The resulting *Gateway* are dumb but they do the trick, other objects can carry out more complicated manipulations.

Sometimes a good strategy is to build the *Gateway* in terms of more than one object. The obvious form is to use two objects: a back-end and a front-end. The back-end acts as a minimal overlay to the external resource and does not simplify the API of the external resource at all. The front end then transforms the awkward API into one that's more convenient for your application to use. Use this approach if the wrapping of the external service and the adaptation to your needs are both reasonably complicated. That way each responsibility is handled by a single class. Conversely if the wrapping of the external service is simple, then one class can handle that and any adaptation that's needed.

## When to Use it

You should consider this whenever you have an awkward interface to something that feels external. Rather than let the awkwardness spread through the whole system, use a *Gateway* to contain it. There's hardly any downside to making the *Gateway*, and the code elsewhere in the system becomes much easier to read.

Using *Gateway* usually makes a system easier to test by giving you a clear point to deploy [\*Service Stub\*](#)s. So even if the external system's interface is fine, a *Gateway* is useful as a first move to make in applying [\*Service Stub\*](#).

A clear benefit of *Gateway* is that it also makes it easier for you to swap out one kind of resource for another. Any change in resource means you only have to alter the *Gateway* class and the change doesn't ripple through the rest of the system. *Gateway* is a simple and powerful form of protected variation. In many cases reasoning about this flexibility is the focus of a discussion about whether to use *Gateway* or not. However don't forget that even if you don't think the resource is ever going to change, you can benefit simply because of the simplicity and testability that *Gateway* gives you.

When you have a couple of subsystems like this, then another choice to decouple them is to use a [\*mediator\*](#). However mediator is much more complicated to use than *Gateway*. As a result I find that the majority of external resource access is done using a *Gateway*.

I must admit that I've struggled a fair bit with whether to make this a new pattern as opposed to referencing existing patterns such as [\*facade and adaptor\*](#). I've separated from these other patterns because I think there's a useful distinction to be made.

- While *facade* also simplifies a more complex API, a facade is usually done by the writer of the

service for general use, while a *Gateway* is written by the client for their particular use. In addition a facade always implies a different interface to what it's covering, while a *Gateway* may copy the wrapped facade entirely, being used for substitution or testing purposes

- *Adapter* alters an implementation's interface to match another interface which you need to work with. With *Gateway* there usually isn't an existing interface, although you might use an adaptor to map an implementation to an existing *Gateway* interface. In this case the adaptor is part of the implementation of the *Gateway*.

## Example: A gateway to a proprietary messaging service (Java)

I was talking about this pattern with my colleague, Mike Rettig, and he described how he's used this pattern to handle interfaces with EAI (Enterprise Application Integration) software. We decided that this would be a good inspiration for an example of *Gateway*.

To keep things to the usual level of ludicrous simpleness we'll just build a gateway to an interface that just sends a message using the message service. The interface is just a single method.

```
int send(String messageType, Object[] args);
```

The first argument is a string indicating the type of the message, the second is the arguments of the message. The messaging system allows you to send any kind of message, so it needs a generic interface like this. When you configure the message system in your use of it, you specify the types of messages the system will send, and the number and types of arguments for the message. So we might configure the confirmation message with the string "CNFRM" and have arguments for an id number as a string, an integer amount and a string for the ticker code. The messaging system will check the types of the arguments for us and generate an error if we send a wrong message or the right message with the wrong arguments.

All this is laudable, and necessary, flexibility but the generic interface is awkward to use because it fails to be explicit. You can't tell by looking at the interface what the legal message types are, nor what arguments are needed for a certain message type. What we need instead is an interface with methods like this.

```
public void sendConfirmation(String orderId, int amount, String symbol);
```

That way if we want a domain object to send a message, it can do so like this

```
class Order...
public void confirm() {
if (isValid()) Environment.getMessageGateway().sendConfirmation(id, amount,
symbol);
}
```

Here the name of the method tells us what message we are sending, and the arguments are typed and given names. This is a much easier method to call than the generic method. This is the role of the gateway to make a more convenient interface. It does mean that every time we add or change a

message type in the messaging system we need to change the gateway class. But we would have to change the calling code anyway, and at least this way the compiler can help us find clients and check for errors.

There's another problem. When we get an error with this interface it tells us by giving us a return error code. A return code of 0 indicates success, anything else is an error. Different numbers indicate different errors. While this is a natural way for a C programmer to work, it isn't the way Java does things. In Java you throw an exception to indicate an error. So the gateway methods should throw exceptions rather than return error codes.

The full range of possible errors is something that we will naturally ignore. Instead I'll focus on just two: sending a message with an unknown message type, and sending a message where one of the arguments is null. The return codes are defined in the messaging system's interface.

```
public static final int NULL_PARAMETER = -1;
public static final int UNKNOWN_MESSAGE_TYPE = -2;
public static final int SUCCESS = 0;
```

The two errors have a significant difference. The unknown message type error indicates an error in the gateway class, since any client is only calling a fully explicit method, clients should never generate this error. Clients might pass in a null, however, and thus see the null parameter error. The null parameter error is not a checked exception since it indicates a programmer error - not something that you would write a specific handler for. The gateway could actually check for nulls itself, but if the messaging system is going to raise the same error it probably isn't worth it.

As a result the gateway has to do both the translation from the explicit interface to the generic interface, and translate the return codes into exceptions.

```
class MessageGateway...
protected static final String CONFIRM = "CNFRM";
private MessageSender sender;

public void sendConfirmation(String orderID, int amount, String symbol) {
Object[] args = new Object[]{orderID, new Integer(amount), symbol};
send(CONFIRM, args);
}

private void send(String msg, Object[] args) {
int returnCode = doSend(msg, args);
if (returnCode == MessageSender.NULL_PARAMETER)
throw new NullPointerException("Null Parameter bassed for msg type: " + msg);
if (returnCode != MessageSender.SUCCESS)
throw new IllegalStateException(
"Unexpected error from messaging system #: " + returnCode);
}

protected int doSend(String msg, Object[] args) {
Assert.notNull(sender);
return sender.send(msg, args);
}
```

So far, it's hard to see the point of the doSend method. It's there for another key role for a gateway - testing. We can test objects that use the gateway without the message sending service being present. To do this we need to create a message gateway stub. In this case the gateway stub is a subclass of the real

gateway and overrides doSend .

```
class MessageGatewayStub...
protected int doSend(String messageType, Object[] args) {
int returnCode = isMessageValid(messageType, args);
if (returnCode == MessageSender.SUCCESS) {
messagesSent++;
}
return returnCode;
}

private int isMessageValid(String messageType, Object[] args) {
if (shouldFailAllMessages) return -999;
if (!legalMessageTypes().contains(messageType))
return MessageSender.UNKNOWN_MESSAGE_TYPE;
for (int i = 0; i < args.length; i++) {
Object arg = args[i];
if (arg == null) {
return MessageSender.NULL_PARAMETER;
}
}
return MessageSender.SUCCESS;
}

public static List legalMessageTypes() {
List result = new ArrayList();
result.add(CONFIRM);
return result;
}

private boolean shouldFailAllMessages = false;

public void failAllMessages() {
shouldFailAllMessages = true;
}

public int getNumberOfMessagesSent() {
return messagesSent;
}
```

Capturing the number of messages sent is a simple way of helping us test that the gateway works correctly with tests like these.

```
class GatewayTester...
public void testSendNullArg() {
try {
gate().sendConfirmation(null, 5, "US");
fail("Didn't detect null argument");
} catch (NullPointerException expected) {
}
assertEquals(0, gate().getNumberOfMessagesSent());
}

private MessageGatewayStub gate() {
return (MessageGatewayStub) Environment.getMessageGateway();
}

protected void setUp() throws Exception {
Environment.testInit();
}
```

You usually set up the gateway so classes can find it from a well known place. Here I've used a static environment interface. Testing code initializes it to use the gateway stub while in regular use you use the real gateway.

In this case I've used a subclass of the gateway to stub the messaging service. Another route is to subclass (or reimplement) the service itself. For testing you'd then connect the gateway to the sending service stub. This works if reimplementation of the service is not too difficult. You always have the choice of stubbing the service or stubbing the gateway. In some cases it's even useful to stub both of them, using the stubbed gateway for testing clients of the gateway and the stubbed service to test the gateway itself.



---

© Copyright [Martin Fowler](#), all rights reserved

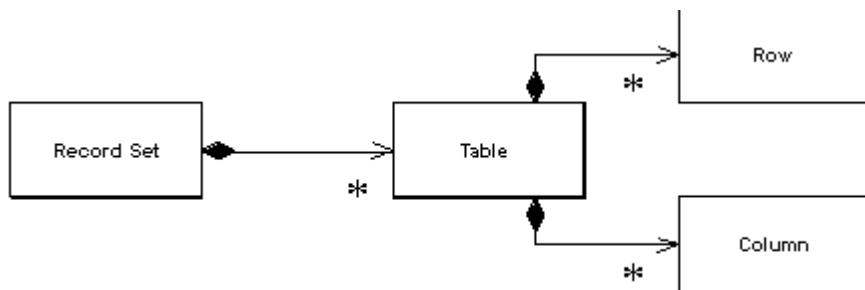


---

# Record Set

---

*An in-memory representation of tabular data*



In the last twenty years, the dominant way to represent data in a database is the tabular form of relational database. Backed by database companies big and small, and a fairly standard query language, almost every new development I see uses relational data.

On top of this came a wealth of tools to help build UI's quickly. These data aware UI frameworks rely on the fact that the underlying data is relational, and provide UI widgets of various kinds that make it easy to view and manipulate this data with hardly any programming.

The dark side of these environments, was the fact that while they made display and simple updates ridiculously easy, they had no real facilities to place any business logic. Any validations beyond the "is this a valid date", and any business rules or computations have no good place to go. Either they get jammed into the database as stored procedures or they get mingled with UI code.

The idea of the *Record Set* is to have your cake and eat it, by providing an in memory structure that looks exactly like the result of a SQL query, but can be generated and manipulated by other parts of the system.

## How it Works

A *Record Set* is usually something that you won't build yourself. Usually it's provided by a vendor of the software platform you are working with. Examples include the Data Set of ADO.NET and the row set of JDBC 2.0.

The first essential element of a *Record Set* is that it looks exactly the same as the result of a database query. This way you can use the classical two-tier approach of issuing a query and throwing the data

directly into a UI with all the ease that these two-tier tools give you. The second essential element is that you can easily build a *Record Set* yourself, or take a *Record Set* that is the result of a database query and easily manipulate it with domain logic code.

Although platforms often give you *Record Set* you can create them yourself. The problem is that there isn't that much point without the data-aware UI tools, which you would also need to create yourself. However it's fair to say that the approach of building a *Record Set* structure as a list of maps, which is common in dynamically typed scripting languages, is a fair example of *Record Set*.

The ability to disconnect the *Record Set* is very valuable. A disconnected *Record Set* is one that is separated from its link to the data source. This allows you to pass the *Record Set* around a network without having to worry about database connections. Furthermore if you can then easily serialize the *Record Set* it can also act as a [\*Data Transfer Object\*](#) for an application.

Disconnection raises the question of what happens when if you update the *Record Set*. Increasingly platforms are allowing the *Record Set* to be a form of [\*Unit of Work\*](#). In this way you can take the *Record Set*, modify it, and then return it to the data source to be committed. A data source can typically use [\*Optimistic Offline Lock\*](#) to see if there's any conflicts, and if not write the changes out to the database.

## When to Use it

To my mind the value of *Record Set* comes from having an environment that relies on it as a common way of manipulating data. If you have a lot of UI tools that use *Record Set*, then that's a compelling reason to use them yourself. If you have such an environment, then that's a big reason to use [\*Table Module\*](#) to organize your domain logic. In this style you get a *Record Set* from the database, pass it to a [\*Table Module\*](#) to calculate derived information, pass it to a UI for display and editing, pass it back to a [\*Table Module\*](#) for validation, and that commit updates to the database.

In many ways the appearance of the tools that make *Record Set* so valuable occurred because of the ever-presence of relational databases and SQL. There hasn't really been an alternative common structure and query language. But now, of course, there is XML which has a widely standardized structure and a query language in XPath. Thus I think it's likely that we'll tools appear that use a hierachic structure in the same way that tools now use *Record Set*. In this case this is perhaps really a particular case of a more generic pattern: something like *Generic Data Structure*. But I'll leave thinking about that pattern until then.





ABC Amber CHM Converter Trial version

Please register to remove this banner.

<http://www.processtext.com/abcchm.html>

# Value Object

---

*A small simple object, like a money or date range, whose equality is not based on identity.*

In dealing with object systems of various kinds, I've found it useful to distinguish between reference objects and *Value Objects*. Of these two a *Value Object* is usually a small object, similar to the primitive types present in many languages that are not purely object-oriented.

## How it Works

Defining the difference between a reference object and *Value Object* is often a tricky thing. In a broad sense we like to think of it that *Value Object* are small objects, such as a Money object or a date; while reference objects are larger things like orders and customers. Such a definition is handy, but annoyingly informal.

The key difference between a reference and value object lies in how they deal with equality. A reference object uses identity as the basis for equality. This may be the identity within the programming system, such as the built in identity of OO programming languages; or it may be some kind of ID number, such as the primary key in a relational database.

A *Value Object* bases its notion of equality on field values within the class. So two date objects may be the same if their day, month, and year values are the same.

The difference manifests itself in how you deal with them. Since *Value Objects* are small and easily created, they are often passed around by value instead of by reference. You don't really care about how many March 18 2001 objects there are in your system. Nor do you care if two objects share the same physical date object, or whether they have different yet equal copies.

Most languages don't have any special facility for value objects. In these cases for value objects to work properly it's a very good idea to make any *Value Object* immutable. That is once created, none of its fields should change. The reason for this is to avoid aliasing bugs. An aliasing bug occurs when two objects share the same value object and one of the owners changes the values in the value. So if Martin has a hire date of Mar 18 and we know Cindy was hired on the same day, we may set Cindy's hire date to be the same as Martin's. If Martin then changes the month in his hire date to May, Cindy's hire date changes too. Whether it's correct or not, it isn't what people would expect. Usually with small values like this people would expect to change a hire date by replacing the existing date object with a new object. Making *Value Objects* immutable fulfills that expectation.

## .NET

A pleasant exception to this is .NET which has a first class treatment of *Value Object*. In C# objects are marked as *Value Object* by declaring them as a struct instead as a class.

*Value Objects* shouldn't be persisted as complete records. Instead use [Embedded Value](#) or [Serialized LOB](#). Since *Value Objects* are small [Embedded Value](#) is usually the best choice, since it also allows SQL querying using the data in the *Value Object*.

If you are doing a lot of binary serializing you may find that optimizing the serialization of *Value Objects* can have a big impact on improving performance, particularly in languages like Java that don't have special treatment for *Value Objects*.

For an example of a *Value Object* see [Money](#).

## When to Use it

Treat something as a value object when you are basing equality on something other than an identity. It's worth considering this for any small object that is easy to create.

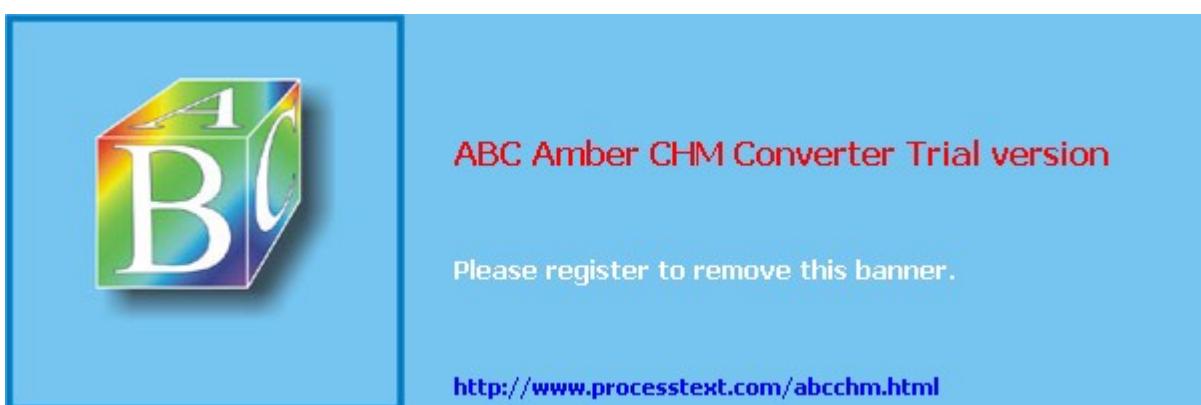
## Name Collisions

I've seen *Value Object* used in this style for quite a time. Sadly recently I've seen people use the term 'value object' to mean [Data Transfer Object](#), something that caused a storm in the teacup of the patterns community. This is just one of these clashes over names that happen all the time in this business. I continue to use *Value Object* in this way in this text. If nothing else this allows me to be consistent with my own previous writings!



---

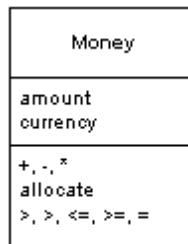
© Copyright [Martin Fowler](#), all rights reserved



# Money

---

*Represents a monetary value*



A pretty large proportion of computers in this world manipulate money. As such it's always puzzled me that Money isn't actually a first class data type in any mainstream programming language. That lack of a type causes problems. The most obvious problem surrounds currencies. If all your calculations are done in a single currency, then this isn't a huge problem, but once you involve multiple currencies then you wasn't avoid adding your dollars to your yen without taking the currencies into account. The more subtle problem is with rounding. Monetary calculations are often rounded to the smallest currency unit. When you do this it's easy to lose pennies (or your local equivalent) due to rounding errors.

The good thing about object-oriented programming is that you can fix these problems by creating a Money class that handles these issues. Of course it's still surprising that non of the mainstream base class libraries actually do this

## How it Works

The basic idea is to have a Money class with fields for the numeric amount and the currency. You can store the amount as either an integral type or a fixed decimal type. The decimal type is easier for some manipulations and the integral type for others. You should absolutely avoid any kind of floating point type, as that will introduce the kind of rounding problems that *Money* is intended to avoid. Most of the time people want monetary values rounded to the smallest complete unit, such as cents in the dollar. There are times, however, when fractional money units are needed. It's important to make it clear what kind of money you are working with, especially in an application where you use both kinds. It makes sense to have two different types for the two cases as they behave quite differently under arithmetic.

Money is a [\*Value Object\*](#), so it should have its equality and hash code operations overridden to be based on the currency and amount.

Money needs arithmetic operations so that you can use money objects as easily as you use numbers. But arithmetic operations for money have some important differences to money operations in numbers. Most obviously any addition or subtraction needs to be currency aware, so you can react if you try to add together monies of different currencies. The simplest, and most common, response is to treat adding of disparate currencies as an error. In some more sophisticated situations you can use Ward Cunningham's idea of a Money Bag. (A Money Bag is an object that contains monies of multiple currencies together in one object. This object can then participate in calculations just like any money object. You can also value a money bag into a currency.)

Multiplication and division end up being more complicated due to rounding problems. When you multiply money you'll do it with a scalar. If you want to add 5% tax to a bill you multiply by 0.05. So you'll see multiplication by regular numeric types.

The awkward complication comes with rounding, particularly when allocating money between different places. Here's Matt Foemmel's simple conundrum. Suppose I have a business rule that says that I have to allocate the whole amount of a sum of money to two accounts: 70% to one and 30% to another. I have 5 cents to allocate. If I do the math I end up with 3.5 cents and 1.5 cents. Whichever way I round these I get into trouble. If I do the usual rounding to nearest then 1.5 becomes 2 and 3.5 becomes 4. So I end up gaining a penny. Rounding down gives me 4 cents and rounding up gives me six cents. There's no general rounding scheme I can apply to both that will avoid losing or gaining a penny.

I've seen various approaches to this.

- Perhaps the most common is to ignore the problem, after all it's only a penny here and there. However this tends to make accountants understandably nervous.
- The simplest rule to follow is that when you are allocating you always do the last allocation by subtracting from what you've allocated so far. While this avoids losing pennies you can get a cumulative amount of pennies on the last allocation.
- You can allow the users of a money class to declare the rounding scheme when they call the method. This would allow a programmer to say that the 70% case rounds up and the 30% rounds down. This can get more complicated when you are allocating across ten accounts instead of two. You also have to remember to do this. To help encourage people to remember I've seen some money classes force a rounding parameter into the multiply operation. Not just does this force the programmer to think about what rounding they need, it also might remind people about what tests to write. But this gets messy if you have a lot tax calculations that all round the same way.
- My favorite solution is to have an allocator function on the Money. The parameter to the allocator is a list of numbers, representing the ratio to be allocated. (so it would look something like aMoney.allocate([7,3])). It then returns a list of monies. The allocator guarantees no pennies get dropped by scattering pennies across the allocated monies in a way that looks pseudo-random from the outside. The allocator is my favorite but has faults: you have to remember to use it and if you have precise rules about where the pennies go they are difficult to enforce.

The fundamental issue here is between using multiplication to determine proportional charge (such as a tax charge) and using multiplication to allocate a sum of money across multiple places. Multiplication works well for the former, but an allocator works better for the latter. The important thing is to consider your intent whenever you want to use multiplication or division on a monetary value.

You may want to convert from one currency to another with a method along the line of

aMoney.convertTo(Currency.DOLLARS). The obvious way to do this is to look up an exchange rate and to multiply by it. While this works in many situations, there are cases where it doesn't - again due to rounding. The Euro conversion rules between the fixed Euro currencies had specific roundings applied that made simple multiplication not work. As a result it's wise to have a convertor object to encapsulate the algorithm.

Comparison operations allow you to sort monies. Like the addition operation, conversions need to be currency aware. You can either choose to throw an exception if you compare different currencies, or you can do a conversion.

One of the most useful results of using a money object is that it can encapsulate the printing behavior. This makes it much easier to provide good display on user interfaces and reports. A money class can also parse a string to provide a currency aware input mechanism - which again is very useful for the user interface.

Storing a money in a database always raises a question, since databases also don't seem to understand that money is important (although their vendors do.) The obvious route to take is to use [Embedded Value](#). This results in storing a currency for every money. This can be overkill, for instance an account may have all its entries be in pounds. In this case you may store the currency on the account and alter the database mapping to pull the account's currency whenever you load entries.

## When to Use it

I pretty much use *Money* for all numeric calculation in object-oriented environments. The primary reason is to encapsulate the handling of rounding behavior, which helps reduce the problems of rounding errors. Another reason to use *Money* is to make multi-currency work much easier. The most common objection to using *Money* is performance, although I've only rarely heard of cases where using *Money* makes any noticeable difference - and even then the encapsulation often makes tuning easier.

## Example: A Money Class (Java)

by Matt Foemmel and Martin Fowler

The first decision is what data type to use for the amount. If anyone needs convincing that a floating point number is a bad idea, ask them to run this code.

```
double val = 0.00;
for (int i = 0; i < 10; i++) val += 0.10;
System.out.println(val == 1.00);
```

With floats safely disposed of, the choice lies between fixed point decimals and integers. In Java this boils down to BigDecimal, BigInteger and long. Using an integral value actually makes the internal math easier, and if we use long we get to use primitives and thus have readable math expressions.

```
class Money...
private long amount;
```

```
private Currency currency;
```

Since I'm using an integral amount, the integral is the amount of the smallest base unit, which I refer to as cents in the code since that is as good a name as any. By using a long we will get an overflow error if the number gets too big. If you give us \$92,233,720,368,547,758.09 we'll write you a version that uses BigInteger.

It's useful to provide constructors from various numeric types

```
public Money(double amount, Currency currency) {
    this.currency = currency;
    this.amount = Math.round(amount * centFactor());
}
public Money(long amount, Currency currency) {
    this.currency = currency;
    this.amount = amount * centFactor();
}
private static final int[] cents = new int[] { 1, 10, 100, 1000 };
private int centFactor() {
    return cents[currency.getDefaultFractionDigits()];
}
```

Different currencies have different fractional amounts. Java's 1.4 version has a currency class that will tell you the number of fractional digits in a class. We could determine how many minor units there are in a major unit by raising ten to the power, but that's such a pain to do in Java that the array is easier (and probably quicker). We're prepared to live with the fact that this code breaks if someone uses four fractional digits.

Although most of the time you'll want to use money operation directly, there are occasions when you'll need access to the underlying data

```
class Money...
public BigDecimal amount() {
    return BigDecimal.valueOf(amount, currency.getDefaultFractionDigits());
}

public Currency currency() {
    return currency;
}
```

You should always question yourself if you find yourself using the accessors, almost always there will be a better way that won't break encapsulation. One example that I couldn't avoid was database mapping, as in [Embedded Value](#)

If you use one currency very frequently for literal amounts, then it can be useful to provide a helper constructor.

```
class Money...
public static Money dollars(double amount) {
    return new Money(amount, Currency.USD);
}
```

As *Money* is a [Value Object](#) you'll need to define equals..

```
class Money...
public boolean equals(Object other) {
return (other instanceof Money) && equals((Money)other);
}
public boolean equals(Money other) {
return currency.equals(other.currency) && (amount == other.amount);
}
```

And wherever there's an equals there should be a hash

```
class Money...
public int hashCode() {
return (int) (amount ^ (amount >>> 32));
}
```

We'll start going through arithmetic with addition and subtraction.

```
class Money...
public Money add(Money other) {
assertSameCurrencyAs(other);
return newMoney(amount + other.amount);
}
private void assertSameCurrencyAs(Money arg) {
Assert.equals("money math mismatch", currency, arg.currency);
}
private Money newMoney(long amount) {
Money money = new Money();
money.currency = this.currency;
money.amount = amount;
return money;
}
```

Note the use of a private factory method here that doesn't do the usual conversion into the cent-based amount. We'll use that a few times inside the Money code itself.

With addition defined, subtraction is easy.

```
class Money...
public Money subtract(Money other) {
assertSameCurrencyAs(other);
return newMoney(amount - other.amount);
}
```

The base method for comparison is compare

```
class Money...
public int compareTo(Object other) {
return compareTo((Money)other);
}

public int compareTo(Money other) {
assertSameCurrencyAs(other);
if (amount < other.amount) return -1;
else if (amount == other.amount) return 0;
else return 1;
}
```

Although that's all you get on most Java classes these days, we find code is more readable with the others such as these.

```
class Money...
public boolean greaterThan(Money other) {
return (compareTo(other) > 0);
}
```

Now we feel ready to look at multiplication. We're providing a default rounding mode as well as allowing you to set one yourself.

```
class Money...
public Money multiply(double amount) {
return multiply(new BigDecimal(amount));
}
public Money multiply(BigDecimal amount) {
return multiply(amount, BigDecimal.ROUND_HALF_EVEN);
}
public Money multiply(BigDecimal amount, int roundingMode) {
return new Money(amount().multiply(amount), currency, roundingMode);
}
```

If you want to allocate a sum of money amongst many targets, and you don't want to lose cents, then you'll want an allocation method. The simplest one allocates the same amount (almost) amongst a number of targets.

```
class Money...
public Money[] allocate(int n) {
Money lowResult = new Money(amount / n);
Money highResult = new Money(lowResult.amount + 1);

Money[] results = new Money[n];
int remainder = (int) amount % n;
for (int i = 0; i < remainder; i++) results[i] = highResult;
for (int i = remainder; i < n; i++) results[i] = lowResult;

return results;
}
```

A more sophisticated allocation algorithm can handle any ratio.

```
class Money...
public Money[] allocate(long[] ratios) {
long total = 0;
for (int i = 0; i < ratios.length; i++) total += ratios[i];

long remainder = amount;
Money[] results = new Money[ratios.length];
for (int i = 0; i < results.length; i++) {
results[i] = new Money(amount * ratios[i] / total);
remainder -= results[i].amount;
}

for (int i = 0; i < remainder; i++) {
results[i].amount++;
}
```

```
return results;
}
```

You can use this to solve Foemmel's Conundrum.

```
class Money...
public void testAllocate2() {
long[] allocation = {3,7};
Money[] result = Money.dollars(0.05).allocate(allocation);
assertEquals(Money.dollars(0.02), result[0]);
assertEquals(Money.dollars(0.03), result[1]);
}
```



---

© Copyright [Martin Fowler](#), all rights reserved



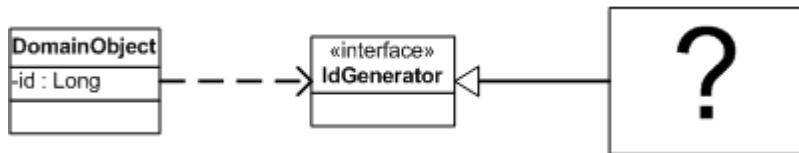
---

# Plugin

---

by David Rice and Matt Foemmel

*Link classes during configuration rather than compilation*



Consider these two seemingly unrelated scenarios that rear their heads regularly in the building of an OO system:

- Reversing the dependency rules between packages. Maybe you have a small bit of domain code that depends upon functionality that can only be implemented in the mapping layer. Yes, there are a handful of cases where this cannot be avoided.
- Switching object type as part of deployment configuration. Perhaps you want to use a cheaper means of key generation when unit testing rather than the database sequence selection you're using in production.

Both scenarios, of course, quickly point one towards defining some functionality in an interface or abstract class.

But there's a bit more to it than that. Both are best implemented by linking to the concrete implementation at configuration, or application startup, time rather than compile time.

In the first scenario the interface will be defined in the domain package and implemented in the mapping package. This avoids the explicit illegal package dependency. But use of a typical factory method to get the implementation just won't do as it doesn't remove hidden compile time dependencies. We need a means of linking after compilation time.

In the second case the interface will be implemented by a handful of classes, each appropriate for a different deployment environment. Here, the use of a factory method might not create the same compile time dependency problems, but we would end up with code like this:

```
class IdGeneratorFactory...  
public static IdGenerator getIdGenerator() {  
if (TESTING) {
```

```
return new TestIdGenerator();
} else {
return new OracleIdGenerator();
}
}
```

Use of a standard factory method to get the implementation requires that runtime configuration information be made available to the factory. Once an application is using more than a couple factories accounting for new configurations within each and every factory becomes quite unruly. Here we have a need to govern the loading of implementations via a central, external point of configuration. Also, despite the potential lack of dependency problems we're still better off linking and loading at startup rather than compilation as the definition of a new configuration ought not mandate the rebuilding or redeployment of an application.

## How it Works

In both cases we do still want something factory-like that can provide an object implementing a requested interface. The requirements for our factory are that its linking instructions be provided by an easily configurable external resource, and that the linking occur as part of configuration rather than compilation.

As the linking and loading of the implementing class occurs during configuration or even later -- just in time, and the type of the implementation is dependent upon the environment configuration we call this pattern *Plugin*.

Implementing a *Plugin* requires a mapping of interfaces to implementations and an object capable of constructing those implementations. Let's refer to an interface requiring implementation via *Plugin* as a plugin type, an implementing class as a plugin, and the object capable of constructing the implementation as a plugin factory.

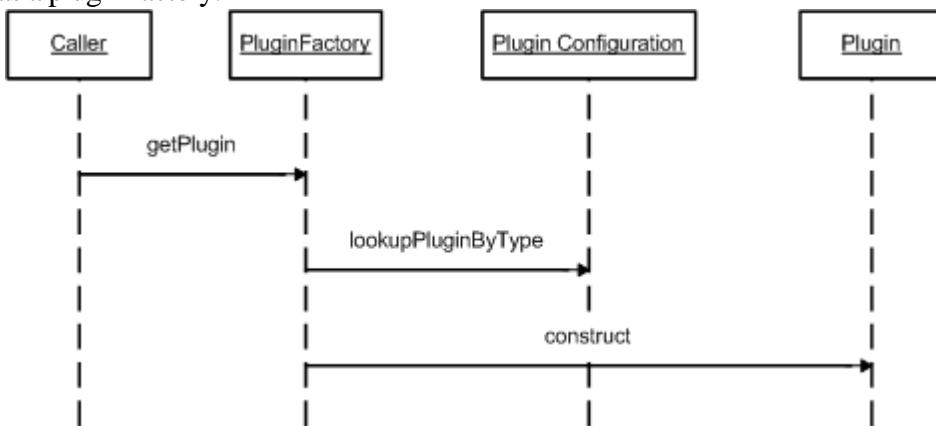


Figure 1:

*Plugin* works best in a language such as Java that has the ability to construct class instances by name via reflection. Using reflection as the means of plugin construction allows a single object to act as the plugin

factory for all plugin types within an application, the plugin factory class to live in a framework package, and the assembling of plugin configurations to be a matter of editing text files.

Yes, the construction of an object by class name rather than constructor is typically a bad smell, but given our requirements it seems rather appropriate. Reflection does have a time and place. The fact that this use reflection is far removed from business logic makes it even more acceptable.

Without reflection more classes and/or methods are required to implement *Plugin* but better code will still result. Linking must now be described in a class rather than a text file thus some compilation and redeployment will be necessary to define a new configuration. However, so long as this configuration class is deployed in its own package on which no application packages are dependent the primary objectives of *Plugin* can still be achieved.

If configuration beyond the implementation type is required, that is, once the correct implementation is constructed the object requires some runtime information such as a connection URL, stick to the basics of separation of concerns and steer clear of using *Plugin* to solve that problem as well.

## When to Use it

Use *Plugin* whenever you have a good reason to reverse a package, or layer, dependency rule. Perhaps you have a serial number generation algorithm that is clearly domain logic but you'd like to perform a uniqueness check before assigning the number to an asset. There's no way to get around the fact that the domain layer is now dependent on the mapping or data access layer. So you define an interface in the domain layer and implement it in the mapping layer. But you've read Martin's discussion on layering and you agree that there's a good reason for this dependency rule. The domain layer should not have a compile time dependency on the mapping layer. Use of *Plugin* will maintain proper layer dependencies while providing a single point at which to 'break' the rules.

The use of *Plugin* also makes sense when an implementation type is determined by environment rather than business logic. Key generation is a very common task in an OO application. In production a database sequence might provide keys. But in unit testing we like a simple in-memory counter to provide keys so as not to avoid the expense of connecting to a database.

A third reason to use *Plugin* is that framework classes frequently need to talk to application classes. This is often the case when using the [Registry](#) pattern. For example, the [Unit of Work](#), typically a framework class, requires access to a registry of [Data Mapper](#) instances in order to commit. Simply define a DatabaseMapperRegistry interface in the same framework package as the [Unit of Work](#) and use *Plugin* to locate the implementation at runtime.

## Example: An Id Generator (Java)

As discussed above, id generation is a task whose implementation might vary between deployment environments.

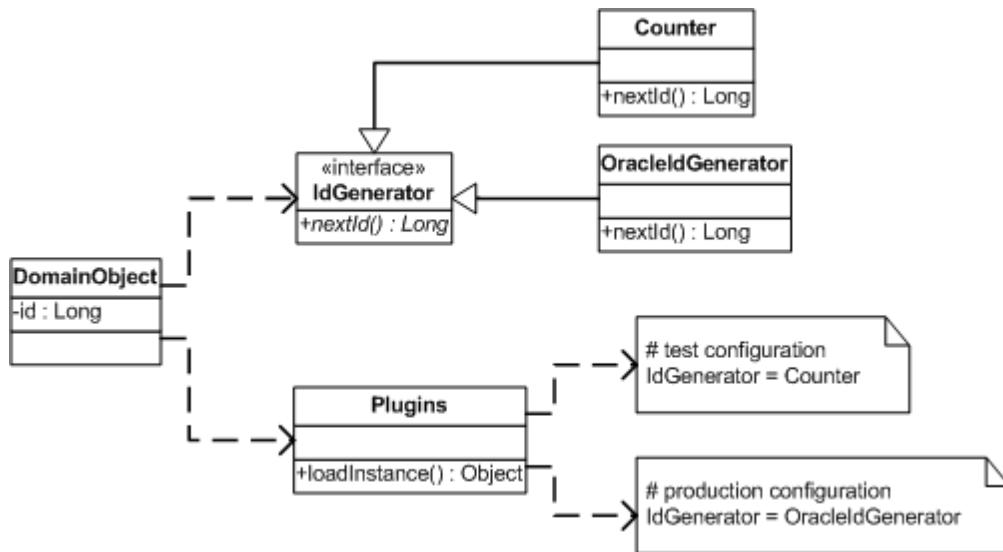


Figure 2:

First define the IdGenerator interface and its implementations

```

class IdGenerator...
public Long nextId();

class OracleIdGenerator...
public OracleIdGenerator() {
    this.sequence = Environment.getProperty("id.sequence");
    this.datasource = Environment.getProperty("id.source");
}
class TestIdGenerator...

public class TestIdGenerator implements IdGenerator {

    private long seed = System.currentTimeMillis();

    public Long nextId() {
        return new Long(seed++);
    }
}
  
```

Now that we have something to construct let's write the plugin factory that will read the interface to implementation mappings for the current environment and construct the implementations as needed.

```

class PluginFactory...
private static Properties props = new Properties();

static {
    try {
        String propsFile = System.getProperty("plugins");
        props.load(new FileInputStream(propsFile));
    } catch (Exception ex) {
        throw new ExceptionInInitializerError(ex);
    }
}
  
```

```
public static Object getPlugin(Class iface) {
String implName = props.getProperty(iface.getName());
if (implName == null) {
throw new RuntimeException("implementation not specified for " +
iface.getName() + " in PluginFactory properties.");
}
try {
return Class.forName(implName).newInstance();
} catch (Exception ex) {
throw new RuntimeException("factory unable to construct instance of " +
iface.getName());
}
}
```

Assemble your plugins.properties file that will map interfaces to their factories and place it in the application's working directory. Or set the VM system property 'plugins.properties' to point to the location of the file. This allows configuring plugins by editing the command that launches the VM for your application server or testing environment. Another option is writing the Plugins class to read its properties from the classpath in order that the properties be distributed in a jar along with the application's classes.

```
# test configuration
IdGenerator=TestIdGenerator

# live configuration
IdGenerator=OracleIdGenerator
```

Let's go back to the IdGenerator interface and add a static INSTANCE member that is set by a call to the plugin factory. This combines *Plugin* with the singleton pattern to provide an extremely simple, readable call to obtain on id.

```
class IdGenerator...
public static final IdGenerator INSTANCE =
(IdGenerator) PluginFactory.getPlugin(IdGenerator.class);
```

It is now possible to make a clean call to obtain an id, knowing that you'll get the right id for the right environment.

```
class TestDomainObject...
public TestDomainObject create(String name) {
Long newObjId = IdGenerator.INSTANCE.nextId();
TestDomainObject obj = new TestDomainObject(name, newObjId);
obj.markNew();
return obj;
}
```





ABC Amber CHM Converter Trial version

Please register to remove this banner.

<http://www.processtext.com/abcchm.html>

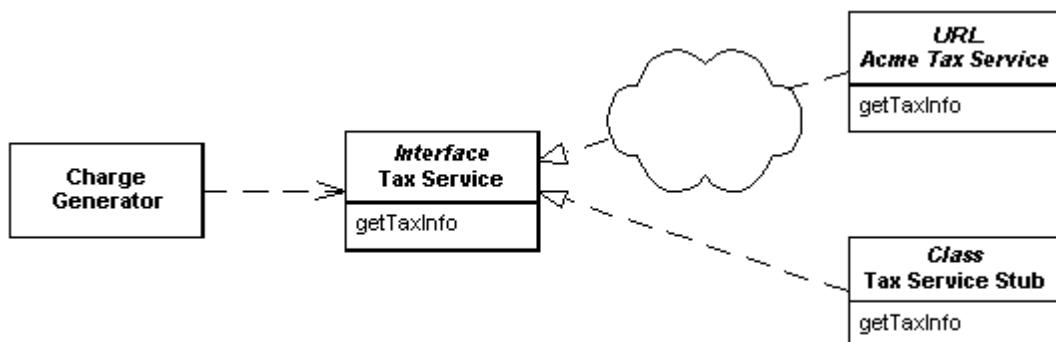
---

# Service Stub

---

by David Rice

*A stand-in implementation of an external service*



Business systems are quite often dependent upon access to third party services such as credit scoring, tax rate lookups, or a pricing engine. These services are often not only third party, but also provided by a remote or external resource. Any developer who has built such an application can speak to the frustration of being dependent upon resources completely out of his control. Feature delivery, reliability, and performance are all unpredictable.

At the very least such issues will slow the development process. Developers will sit around waiting for the service to come back on line or maybe put some hacks into the code to compensate for yet to be delivered features. Much worse, and quite likely, such dependencies will lead to periods in time when tests cannot execute. When tests cannot run the development process is broken.

What is needed is a stand-in implementation of this service. One whose development is completely within the control of the application team and one that runs locally, fast, and in-memory. This stand-in code is often referred to as a *Service Stub*.

## How it Works

The first step is to define access to this service with an interface. If your service provider has already provided an interface for your platform be sure to read the note at the bottom of this section on interface

design. Once the interface is defined and an implementation or two has been coded it's simply a matter of using the [Plugin](#) pattern to provide the proper implementations of services to your application at runtime.

As with most all design the key principal to stub implementation is keeping it simple. Since the primary reason behind writing a stub is to speed and ease the development process, writing a complex stub defeats its own purpose.

Let's walk through the thought process of stubbing a sales tax service. Code examples are provided below. This service provides state sales tax amount and rate, given an address, product type, and sales amount. The first question to ask when stubbing is 'What's the simplest possible means of providing this service?' The answer is to write a 2 or 3 line implementation that uses a flat tax rate for all requests.

But tax laws aren't that simple. Certain products are exempt from taxation in certain states. As such rules are part of tax law we're dependent upon our tax service for such information. And we have a lot of functionality that is dependent upon whether taxes are charged so we need to accommodate the possibility of tax exemption in our service. Now ask what is the simplest means of adding this to the stub? It's probably to add an if statement and simply for a specific combination of address and product. The number of lines of code in our stub can probably still be counted on one hand.

If you're building one of those fantastically large back office applications with 10,000+ functions points and as many unit tests you might even add some more dynamic functionality to your stub. We might wish to add a method to the service that allows tests to setup exemptions for themselves. We could even provide reset functionality for test cleanup.

Even with the addition of dynamic functionality the stub is incredibly simple. We've only added a map and some simple lookup code. If you've ever dealt with state and local sales tax laws you know they are beyond complicated. A switch to a flat tax would indeed put tens of thousands of accountants and lawyers out of work! Keep the stub simple enough so that it doesn't break under its own weight. If the stub is too simple to possibly work it's probably a good stub. Always remember that the primary purpose of *Service Stub* is to speed up development.

The last, more dynamic stub above brings up an interesting question regarding the dependency between test cases and a *Service Stub*. The stub relies upon a method for adding exemptions that is not in the service interface. This implies that the test case has some prior knowledge as to the type of the tax service that will be loading during its execution. This defeats the purpose of using the [Plugin](#) pattern to load the stub implementation.

Two solutions come to mind: The first is to add the test methods to the public interface. Throw assertions or system exceptions in the production implementation of these methods. If you're writing to a 3rd party interface you will need to extend the interface.

Another alternative is possible if your tests cases don't require that the stub be configured and reset

between each case. Simply have the stub configure itself. If a developer adds a test case she must make sure that all of the available stubs can provide appropriate behavior for this test case. This avoids compilation dependencies and seems a reasonable compromise to the logical dependency problem. There is no way around the fact that some coordination is required between your stubs and your tests.

A note on interface definition:

Perhaps the service provider supplies a well designed SDK for your particular platform and this work has already been done for you. Otherwise you'll have to do it yourself. I've seen instances where the application developers might someday -- 'in theory' -- switch vendors for a particular service, therefore despite the existence of a perfectly good interface supplied by a vendor the application team goes ahead and defines its own interface. The application is providing its own definition of the service. This is over-design. It's too much work and results in unreadable code and a debugging nightmare. Writing to the vendor interface is clean enough. A switch in vendors, if the services are in fact similar, will require merely a simple refactoring of the code that calls that interface.

## **When to Use it**

Deciding to use *Service Stub* is a simple task. Ask yourself these questions to determine whether to use *Service Stub*

- Is my dependence upon this remote or third party service slowing down my development process?
- Does this service make testing laborious?
- Can I write an extremely simple stand-in implementation of this service that would suffice for testing purposes?

If you find yourself answering yes by all means go ahead and write the stub. Just never forget that a stub is meant to make your life easier rather than more difficult.

## **Example: Sales Tax Service (Java)**

Our sales tax service is being used by the A/R subsystem of a lease management system. Within this system billing schedules store information as to when to generate charges for each leased asset. Each night a charge generation process inspects each billing schedule and generates appropriate charges. For each charge a call to the tax service is made in order to generate any tax charges.

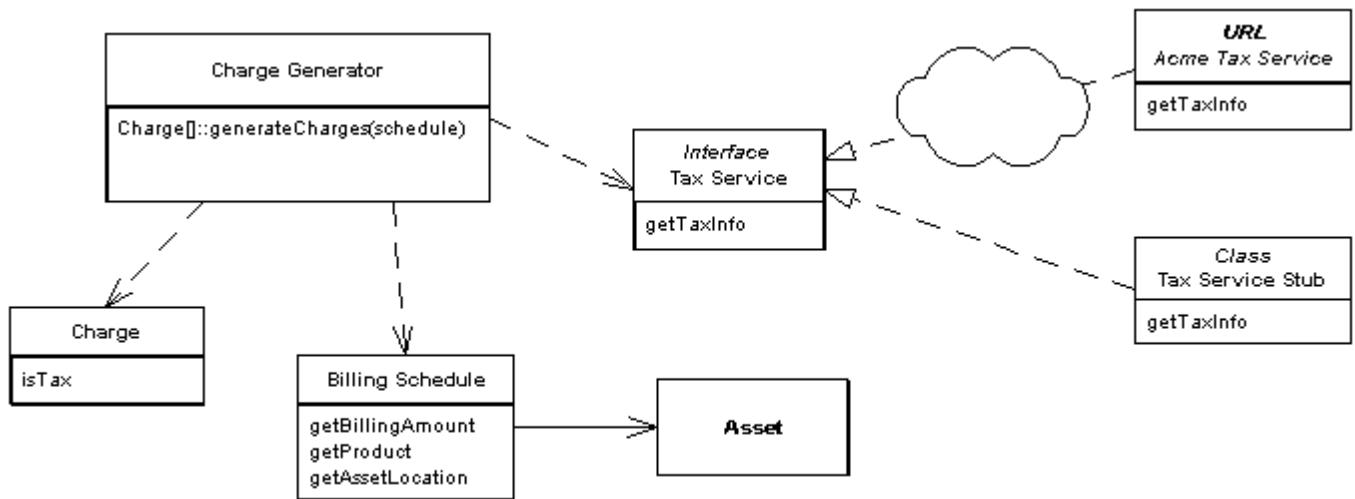


Figure 1:

What is the point of such a long winded description of a leasing A/R subsystem? It's that a *Service Stub* makes no sense unless you're actually testing functionality that is dependent upon that service. Here we have a charge generation service that is creating extra charges if taxes ought to be charged. Our A/R subsystem has lots of behavior dependent upon whether tax charges were created. But our tax system is a web service and we're having massive downtime problems. Sounds like an ideal candidate for a stub.

Since we've decided to use a tax service that has been deployed as a web service the first thing we need to do is write a Java interface to define the service. While we've been given a nice WSDL interface that's not something we want to write to from our domain layer so we'll define our own Java interface:

```

class SalesTaxService...

public interface SalesTaxService {

    public static final SalesTaxService INSTANCE = (SalesTaxService)
        PluginFactory.getPlugin(SalesTaxService.class);

    public TaxInfo getSalesTaxInfo(String productCode, Address addr, Money
        saleAmount);
}
  
```

Note the use of [Plugin](#) to load the service implementation.

The application code that we're working on here is charge generation and we want to test that the generator will return a single charge if a product is exempt from taxation and an additional state sales tax charge if the product is not exempt. We also want to test the returned charges are of the correct type. So let's write our test:

```

class Tester...

public void setUp() throws Exception {
    exemptBillingSchedule =
        new BillingSchedule(new Money(200, Currency.USD), "12300", CHICAGO_ILLINOIS);
    nonExemptBillingSchedule =
        new BillingSchedule(new Money(200, Currency.USD), "12305", CHICAGO_ILLINOIS);
  
```

```
}

public void testChargeGenerationExemptProduct() throws Exception {
    Charge[] charges = new
    ChargeGenerator().calculateCharges(exemptBillingSchedule);
    assertEquals("single charge", charges.length, 1);
    assertTrue("non tax charge", !charges[0].isTax());
    assertEquals("correct charge", charges[0].getAmount(),
        exemptBillingSchedule.getBillingAmount());
}

public void testChargeGenerationNonExemptProduct() throws Exception {
    Charge[] charges = new
    ChargeGenerator().calculateCharges(nonExemptBillingSchedule);
    assertEquals("two charges", charges.length, 2);
    assertTrue("non tax charge", !charges[0].isTax());
    assertTrue("tax charge", charges[1].isTax());
    assertEquals("correct charge", charges[0].getAmount(),
        nonExemptBillingSchedule.getBillingAmount());
    assertEquals("correct tax", charges[1].getAmount(), new Money(10,
        Currency.USD));
}
```

Given these tests we can use the 'middle of the road' stub approach from above and write a service implementation that looks for a particular product and location to determine exemption status:

```
class ExemptProductTaxService...
```

As we discussed above there are a few trade-offs to think about when approaching the test case to stub dependency issue. Here we have values hard coded in both the test case and the stub. There's no way around this when testing so just choose a sensible approach.

Here's what the simplest possible service would look like:

```
class FlatSalesTaxService...
public TaxInfo getSalesTaxInfo(String productCode, Address addr, Money
saleAmount) {
return new TaxInfo(FLAT_RATE, saleAmount.multiply(FLAT_RATE.doubleValue()));
}
```

Here's what the dynamic tax service might look like:

```
class TestTaxService...
```





ABC Amber CHM Converter Trial version

Please register to remove this banner.

<http://www.processtext.com/abcchm.html>

# Layer Supertype

---

*A type that acts as the supertype for all types in its layer*

Often all the objects in a layer will have some common methods that you don't want to have duplicated throughout the system. So you can move all of this behavior into a common *Layer Supertype*.

## How it Works

*Layer Supertype* is a simple idea that leads to a very short pattern. All you need is a superclass for all the objects in a layer. For example all the domain objects in a [Domain Model](#) might have a superclass called Domain Object. Common features, such as the storage and handling of [Identity Fields](#) can go there. Similarly all [Data Mappers](#) in the mapping layer may have a superclass which can rely on the fact that all the domain objects have a common superclass.

If you have more than one kind of object in a layer, then it's useful to have more than one *Layer Supertype*.

## When to Use it

Use *Layer Supertype* when you have common features from all objects in a layer. Often I do this automatically now, because I so often make use of common features.

## Example: Domain Object (Java)

Domain objects can have a common superclass for id handling.

```
class DomainObject...
private Long ID;

public Long getID() {
return ID;
}

public void setID(Long ID) {
```

```
Assert.notNull("Cannot set a null ID", ID);
this.ID = ID;
}

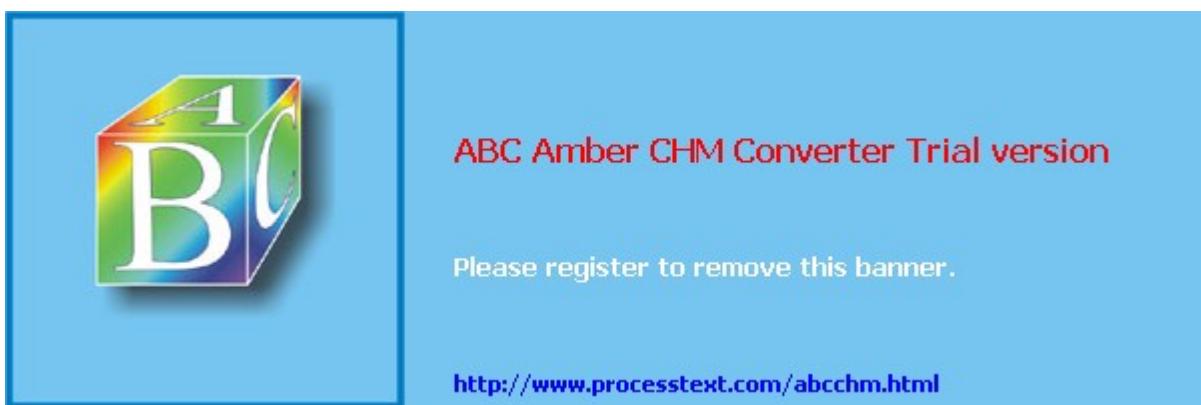
public DomainObject(Long ID) {
this.ID = ID;
}

public DomainObject() {
```



---

© Copyright [Martin Fowler](#), all rights reserved

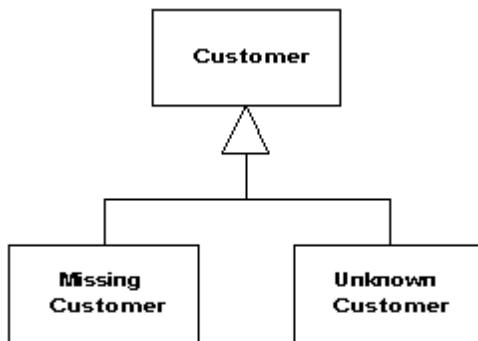


---

# Special Case

---

*A subclass which provides special behavior for particular cases.*



Null's make awkward things in object-oriented programs because they defeat polymorphism. Usually you can invoke foo freely on a variable reference of a given type without worrying about whether the item in the type is of the exact type or a subclass. With a strongly typed language you can even have the compiler check that the call is correct. However since a variable can contain null, you may run into a runtime error by invoking a message on null, which will get you a nice friendly stack trace.

If it's possible for a variable to be null, you have to remember to surround it with null test code so you remember do the right thing if a null is present. Often the right thing is same in lots of contexts, so you end up writing similar code in lots of places - which is the sin of duplicate code.

Nulls are a common problem of this kind, but others crop up regularly. In number systems you have to deal with infinity, which has special rules for things like addition that break the usual invariants of real numbers. One of my earliest examples in business software was a utility customer who wasn't fully known - referred to as "occupant". All of these imply altered behavior to the usual behavior of the type.

Instead of returning null, or some odd value, return a *Special Case* that has the same interface as what the caller expects.

## How it Works

The basic idea is to create a subclass to handle the *Special Case*. So if you have a customer object and you want to avoid null checks, you make a null customer object. Take all of the methods for customer and override them in the *Special Case* to provide some harmless behavior. Then whenever you would

have a null put in an instance of null customer instead.

There's usually not any reason to distinguish between different instances of null customer, so you can often implement a *Special Case* with a flyweight [\[Gang of Four\]](#). This isn't true all the time. For a utility you can accumulate charges against an occupant customer even you can't do much of the billing, so it's important to keep your occupants separate.

People often use a null to mean different things. A null customer may mean there isn't a customer or it may mean that there is a customer but we don't know who it is. Rather than just use a null customer, consider having separate *Special Cases* for missing customer and unknown customer.

A common way for a *Special Case* to override methods is to return another *Special Case*. So if you ask an unknown customer for his last bill, you may well get an unknown bill.

## When to Use it

Use *Special Case* whenever you have multiple places in the system that do the same behavior after a conditional check for a particular instance of this class, or same behavior after a null check.

## Further Reading

I haven't seen *Special Case* written up as a pattern yet, but Null Object has been written up in [\[Woolf\]](#). If you'll pardon the unresistable punnery I see Null Object as special case of *Special Case*.

## Example: A Simple Null Object (C#)

Here's a simple example of using *Special Case* for its common use as a null object.

We have a regular employee.

```
class Employee...
public virtual String Name {
get {return _name;}
set {_name = value;}
}
private String _name;
public virtual Decimal GrossToDate {
get {return calculateGrossFromPeriod(0);}
}
public virtual Contract Contract {
get {return _contract;}
}
private Contract _contract;
```

The features of the class could be overridden by a null employee

```
class NullEmployee : Employee, INull...
public override String Name {
get {return "Null Employee";}
set {}
}
public override Decimal GrossToDate {
get {return 0m;}
}
public override Contract Contract {
get {return Contract.NULL;}
}
```

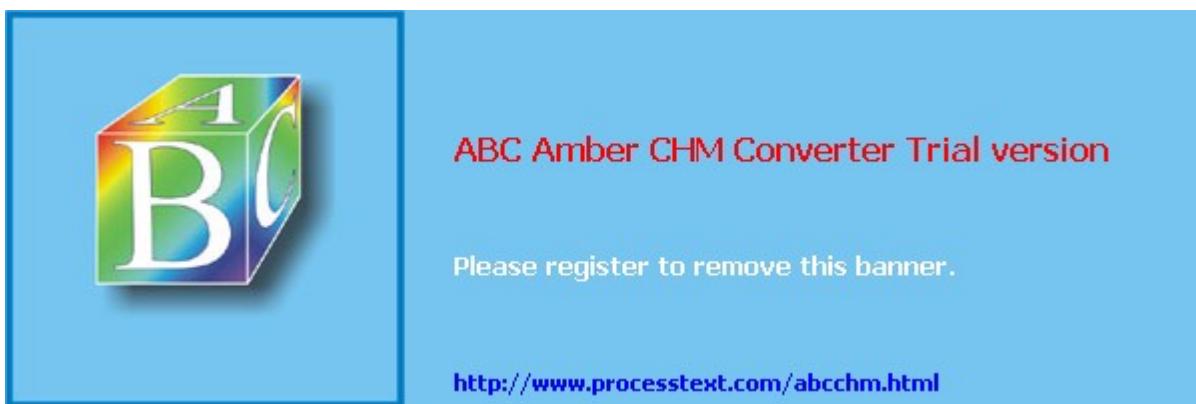
Notice how when you ask a null employee for its contract you get a null contract back.

The default values here avoid a lot of null tests if the null tests end up with the same null values. The repeated null values are handled by the null object by default. You can also test for nullness explicitly either by giving customer an isNull method or by using a type test for a marker interface.



---

© Copyright [Martin Fowler](#), all rights reserved



# Registry

---

*A well known object that other objects can use to find common objects and services*



Most of the movement between objects is done by following references in fields. If you want to find all the orders for a customer, you'll most often start with the customer object and use a method on the customer to get the orders. However in some cases you won't have an appropriate object to start with. You may know the customer's id number, but not have a reference to the customer. In this case you need to use some kind of lookup method - a finder. But the question just continues to ask itself - how do you get to the finder?

A *Registry* is essentially some kind of global object, or at least it looks like a global object - even if it isn't as global as it may appear.

## How it Works

As with any object, you have to think about the design of a *Registry* in terms of interface and implementation. And like many objects the two are quite different while people often make the mistake of thinking they should be the same.

The first thing to think of is the interface, and for *Registrys* my preferred interface is static methods. A static method on a class is easy to find anywhere in an application. Furthermore you can encapsulate any logic you like within the static method, including delegation to other methods which can either be static or instance methods.

However just because your methods are static does not mean your data should be in static fields. Indeed I almost never use static fields unless they are constants.

Before you decide on how to hold your data, you have to think about the scope of the data. The data for a *Registry* can vary with different execution contexts. Some data is global across an entire process, some global across a thread, some global across a session. Different scopes call for different implementations, however different scopes don't call for different interfaces. The application programmer

need not know whether a call to a static method yields process scoped or thread scoped data. You can have different *Registries* for different scopes, but you can also have a single *Registry* where different methods are at different scopes.

If your data is common to a whole process, then a static field is an option. However I rarely use static mutable data because they don't allow for substitution. It is often extremely useful to be able to substitute a *Registry* for a particular purpose, especially for testing ([Plugin](#) is a good way to do this.)

So for process scoped *Registry* the usual option is a singleton [[Gang of Four](#)]. The *Registry* class contains a single static field that holds an instance of the *Registry*. Often when people use a singleton they make the caller of the singleton explicitly access the underlying data (`Registry.getInstance.getFoo()`), but I prefer to have a static method that hides the singleton object from me (`Registry.getFoo()`). This works particularly well since C based languages allow static methods to access private instance data.

Singletons are quite widely used in single-threaded applications, but can be quite a problem for multi-threaded applications. This is because it's too easy to have multiple threads manipulating the same object in unpredictable ways. You may be able to solve this with synchronization, but the difficulty of writing the synchronization code is likely to drive you into an insane asylum before you get all the bugs out. So I don't recommend using a singleton for mutable data in a multi-threaded environment. A singleton does work well for immutable data, anything that can't change isn't going to run into thread clash problems. So something like a list of all states in the US makes a good candidate for a process scoped *Registry*. Such data can be loaded when a process starts up and then never needs changing, or may be updated rarely with some kind of process interrupt.

A common kind of *Registry* data is thread scoped data. A good example of this might be a database connection. In this case many environments give you some form of thread specific storage that you can use, such as Java's thread local. Another technique is to have a dictionary keyed by thread whose value is an appropriate data object. A request for a connection would then result in a lookup in that dictionary by the current thread.

The important thing to remember about using thread-scoped data is that it looks no different from using process-scoped data. I can still use a method such as `Registry.getDbConnection()`, which is just the same form as if I'm accessing process-scoped data.

A dictionary lookup is also a technique that you can use for session-scoped data. Here you need a session id, but this could be put into a thread scoped registry when a request begins. Any subsequent accesses for session data can then look the data up in a map that's keyed by session using the session id that's held in thread specific storage.

Some applications may have a single *Registry*, some may have several. *Registries* are usually divided up by system layer, or by execution context. My preference is to divide them up by how they are used, rather than how they are implemented.

## When to Use it

Despite the encapsulation of a method a *Registry* is still global data, and as such is something I'm uncomfortable using. I find I almost always see some form of *Registry* in an application, but I always try

to access objects through regular inter-object references rather than resorting to the *Registry*. So you should always use a *Registry* as a means of last resort.

There are alternatives to using a *Registry* at all. One alternative is to pass around any widely needed data in parameters. The problem with this is that it leads to parameters that are added to method calls where they aren't needed by the called method, only needed for some other method that's called several layers deep in the call tree. Passing a parameter round when it's not needed 90% of the time is the trigger that leads me to use a *Registry* instead.

Another alternative I've seen to a *Registry* is to add a reference to the common data to objects when they are created. Although this leads to an extra parameter in a constructor, at least it's only used by the constructor. It's still more trouble than it's often worth, but if you have data that's only used by a subset of classes, this technique does allow you to restrict things that way.

One of the problems with a *Registry* is that every time you add a new piece of data you have to modify the *Registry*. This leads to some people preferring to use a map as their holder of global data. I prefer the explicit class because it keeps the methods explicit - so there's no confusion about what key you use to find something. With an explicit class you can just look at the source code or generated documentation to see what's available, with a map you have to find places in the system where data is read or written to the map to find out what key is used - or rely on documentation which quickly becomes stale. An explicit class also allows you to keep type safety in a statically typed language as well as to encapsulate the structure of the *Registry*, which allows you to refactor it as the system grows. A bare map also is unencapsulated, which makes it harder to hide the implementation, which is particularly awkward if you have to change the execution scope of the data.

So there are times when it's right to use a *Registry* - but remember that any global data is always guilty until proven innocent.

## Example: A Singleton Registry (Java)

Consider an application that reads data from a database and then munges on the data to turn it into information. Well imagine a fairly simple system that uses [Row Data Gateways](#) to access the data. With such a system it's useful to have finder objects to encapsulate the database queries. These finders are best made as instances because that way we can substitute them to make a [Service Stub](#) for testing purposes. To get hold of the finders, we'll need a place to put them: a *Registry* is the obvious choice.

A singleton registry is a very simple example of the singleton pattern [\[Gang of Four\]](#). You have a static variable for the single instance.

```
class Registry...
private static Registry getInstance() {
    return soleInstance;
}
private static Registry soleInstance = new Registry();
```

Everything that's stored on the registry is stored on the instance.

```
class Registry...
protected PersonFinder personFinder = new PersonFinder();
```

To make access easier, however, I make the public methods static.

```
class Registry...
public static PersonFinder personFinder() {
    return getInstance().personFinder;
}
```

I can re-initialize the registry by simply creating a new sole instance.

```
class Registry...
public static void initialize() {
    soleInstance = new Registry();
}
```

If I want to use [Service Stub](#)s for testing, I use a subclass instead.

```
class RegistryStub extends Registry...
public RegistryStub() {
    personFinder = new PersonFinderStub();
}
```

The finder [Service Stub](#) just returns hard coded instances of the person [Row Data Gateway](#)

```
class PersonFinderStub...
public Person find(long id) {
    if (id == 1) {
        return new Person("Fowler", "Martin", 10);
    }
    throw new IllegalArgumentException("Can't find id: " + String.valueOf(id));
}
```

I put a method on the registry to initialize it in stub mode, but by keeping all the stub behavior in the subclass I can separate all the code required for testing.

```
class Registry...
public static void initializeStub() {
    soleInstance = new RegistryStub();
}
```

## Example: Thread Safe *Registry* (Java)

*Matt Foemmel and Martin Fowler*

The simple example above won't work for a multi-threaded application where different threads need their own registry. Java provides Thread Specific Storage [Schmidt](#) - variables that are local to a thread, helpfully called thread local variables. You can use these to create a registry that's unique for a thread.

```
class ThreadLocalRegistry...
private static ThreadLocal instances = new ThreadLocal();
public static ThreadLocalRegistry getInstance() {
    return (ThreadLocalRegistry) instances.get();
```

}

The *Registry* needs to be set up with methods to acquire and release the registry. You would typically do this on a transaction or session call boundary.

```
class ThreadLocalRegistry...
public static void begin() {
    Assert.isTrue(instances.get() == null);
    instances.set(new ThreadLocalRegistry());
}
public static void end() {
    Assert.notNull(getInstance());
    instances.set(null);
}
```

We can then store person finders just as before.

```
class ThreadLocalRegistry...
private PersonFinder personFinder = new PersonFinder();
public static PersonFinder personFinder() {
    return getInstance().personFinder();
}
```

Calls from the outside then wrap their use of a registry in the begin and end methods.

```
try {
    ThreadLocalRegistry.begin();
    PersonFinder f1 = ThreadLocalRegistry.personFinder();
    Person martin = Registry.personFinder().find(1);
    assertEquals("Fowler", martin.getLastName());
} finally {ThreadLocalRegistry.end();
}
```



---

© Copyright [Martin Fowler](#), all rights reserved

