Software Design and Architecture

Subject Notes for SWEN90007

The School of Computing
and
Information Systems

The University of Melbourne
Second Semester, 2021

Aims, assessment, and subject outline

Subject Overview

The primary focus of this subject is software design and architecture for enterprise systems. The principles in this subject are not restricted to enterprise systems — they can be applied to other types of systems — however, most of the topics are taken from areas that are commonly found in enterprise systems engineering.

The topics in this subject include:

An introduction to enterprise systems The first part of the subject is an overview of enterprise systems, and introduces general methods and concepts found in enterprise software architecture and design.

Design principles and patterns This is the main part of the subject. In this section, we will look at common design principles and patterns for enterprise system components, principles and patterns for bringing together these components, and the trade-offs to be made in these designs.

Design for non-functional requirements This part of the subject will look at design principles and patterns for non-functional requirements. We will look at only two non-functional requirements: security and performance. This section serves as a more general demonstration of how design patterns and principles can apply to non-functional requirements.

Enterprise system integration The final part of the subject will look at design principles and patterns for integrating different enterprise systems with each other. This is a particular challenging task because different enterprise systems are typically built at different times using different technologies, but will have to work together.

Staff details

Eduardo Oliveira
Email: `eduardo.oliveira@unimelb.edu.au`
Level 3, Room 3332, Melbourne Connect

Maria Rodriguez Read
Email: `maria.read@student.unimelb.edu.au`
Level 2, Room 2332, Melbourne Connect

Learning outcomes

On completion of this subject, students should be able to:

- Analyse large scale and distributed enterprise systems to select appropriate architectures for them.
- Evaluate architectural designs for enterprise systems.
- Make suitable trade-offs between different architectures.

This subject also aims to reinforce the following generic skills:

- Ability to apply knowledge of science and engineering fundamentals.
- Ability to undertake problem identification, formulation, and solution.
- Ability to utilise a systems approach to complex problems and to design an operational performance.
- Proficiency in engineering design.
- Ability to manage information and documentation.
- Capacity for creativity and innovation.
- Ability to function effectively as an individual and in multidisciplinary and multicultural teams, as a team leader or manager as well as an effective team member.
- Capacity for lifelong learning and professional development.

Assumed background

- A solid understanding of the "Gang of Four" design patterns (SWEN30006).
- Ability to design a medium-scale system.
- Ability to apply UML to the documentation of designs.
- Practical experience with Java and related tools.

Assessment

One semester-long project, broken into four parts, totalling 100%.

Contact hours

- Lectures: One two-hour lecture each week.
- Workshop: A single one-hour workshop each week.

It is expected that you will attend lectures and workshops regularly.

Resources

The main resources used for communicating between staff and students are:

- Canvas - https://canvas.lms.unimelb.edu.au
- Ed Discussion - https://edstem.org/courses/6488/discussion/
- GitHub Classroom - https://github.com/SWEN900072021/Resources

Detailed topics list

A. Introduction
   1. Introduction to enterprise systems

B. Design principles and patterns for enterprise systems
   2. Organising domain logic
   3. Mapping to the data source layer
   4. The presentation layer
   5. Sessions
   6. Concurrency
   7. Distribution

C. Design principles and patterns for non-functional requirements
   8. Security
   9. Performance

D. Integrating enterprise systems
   10. Enterprise system integration
   11. Enterprise messaging

Contents

Chapter 1

Introduction

In this chapter, we define what we mean by "enterprise systems" and by "software architecture". We give some examples of enterprise systems, and discuss a general approach for software architecture of enterprise systems.

Learning outcomes A person familiar with the material in this chapter should be able to:

1. describe the properties of enterprise systems, and categorise examples as enterprise systems or not.
2. understand the difference between architecture, design, patterns, and principles.
3. derive a template for describing/documenting a software architectural design.
4. understand the concept of *layering* in software architecture, and explain the relevant layers commonly found in architectures for enterprise systems.
5. describe the concepts of transactions, including the *ACID* properties; and
6. explain the differences between business transactions and system transactions.

1.1 What is an enterprise system?

In the software engineering courses at the University of Melbourne, you will learn to engineer different types of systems and software. In this subject, we will focus on architectural patterns and principles within *enterprise systems* (or *enterprise applications*).

So, the first thing we need to do is define what an enterprise system is, and how an enterprise system differs from other types of software inclusive systems.

Definition 1.1 (Enterprise system)
An *enterprise system* is a software system that supports organisations in their goals to handle business processes, resource planning & management, service management, and business intelligence. □

Let's break this definition down a bit by looking at the properties here:

- Enterprise systems are software systems that are part of a larger organisational system. Therefore, they are typically not stand-alone packages that run on a desktop PC with a single user. Instead, they support multiple users and often a diverse set of features that are related to the enterprise.

- Enterprise systems are used in enterprises. Therefore, enterprise applications are used by businesses and government, and generally not used by individuals.

- Enterprise systems are used to support business in management of their key resources. This means that enterprise systems are typically business oriented. Word processors are not enterprise systems, even though they are used within organisations, because they do not support key businesses processes.

- Enterprise systems support enterprise goals. Therefore, enterprise systems are generally closely related to the domain in which they are deployed, and in many cases, to the organisation in which they are deployed.

Some examples

To help clarify how enterprise systems differ from other software systems, let's consider some examples.

Enterprise systems	**Non-enterprise systems**
Automated billing	Word processing
Patient health records	Air traffic control
Supply chain management	Railway switching
Accounting	Telecommunication systems
Order tracking	Operating systems
Online shopping/payment	Compilers & IDEs
Enterprise resource planning	Automobile braking & control
Enterprise forms automation	Control systems

Common properties of enterprise systems

Many enterprise systems have common properties, which is why there are people who specialise in building enterprise systems. The main ones from Fowler [3] are:

1. *Persistent data*: Data in the system must remain persistent between executions, and in many cases, must be consistent for decades.

2. *A lot of data*: Most enterprise systems have to handle millions of records, totalling many gigabytes, terabytes or even petabytes of data. Typically these are stored in relational databases, but older systems still in operation use indexed file structures. Some newer systems store bulk data (e.g. multimedia files) outside the database management systems too.

3. *Concurrent access*: Enterprise systems almost always have to support concurrent access. That is, multiple users will be trying to access the system and its data at the same time, often conflicting in the data they wish to read/write. In some organisations, this may be less than 100 people, but for customer-facing software running over the Internet, it can be orders of magnitude higher.

4. *Many user interfaces*: Many enterprise systems handle the same set of data for many different types of user. As such, the amount and variety of data that must be handled and the number of different types of user typically leads to enterprise systems having hundreds of different screens/windows.

5. *Build on business logic*: Although business is generally anything but logical, enterprise systems are built on the business logic and in the business domain of the organisation, usually as the form of business rules. These rules cannot be modified by the software engineers, no matter how illogical they seem, because they form the way to enterprise does business. It is the job of the software engineers and enterprise architects to design and build software that obeys these rules.

6. *Integrate with other enterprise systems*: It is common for enterprise systems to have to integrate with other enterprise systems. For example, to process payroll, a system may have to integrate with a system that handles human resource information. These systems are often built at different times using different technologies, so integrating them is a non-trivial challenge.

1.2 Architecture, design, principles, and patterns

"*Software architecture serves as the blueprint for both the system and the project developing it, defining the work assignments that must be carried out by design and implementation teams. The architecture is the primary carrier of system qualities such as performance, modifiability, and security, none of which can be achieved without a unifying architectural vision. Architecture is an artifact for early analysis to make sure that a design approach will yield an acceptable system. By building effective architecture, you can identify design risks and mitigate them early in the development process*" – Source unknown.

Software architecture

The term "architecture" comes from building construction, but has found its way into various fields of science and engineering. In the context of this subject, we will be discussing the architecture of enterprise software systems. In this context, we define an architecture as follows.

Definition 1.2 (Software architecture)
A *software architecture* is a high-level view of a software system's components, how those components behave, how they relate to each other, and how they relate to their environment, for the purpose of achieving their stated functionality. A software architecture must meet the needs of its intended users over an extended period of time and under different patterns of usage. □

It is important to note that an architecture is both an abstraction and an understanding. Such an understanding is subjective. While it may be an accurate abstraction, an architecture is not a tangible item, but instead if just a representation of a (possibly shared) understanding.

In addition, an architecture consists of different views. Some of these views are considered more important than others. In enterprise systems, the views considered important can be extracted directly from Definition 1.2:

1. *Structural*: A static view of the components and how they are related to each other.

2. *Behavioural*: A dynamic view of how the components behave; that is, a specification of the inputs and outputs of a component.

3. *Interaction*: A dynamic view of the flow of control and data between the components.

It is common in architectural documents to consider only the static structural view, while neglecting the the dynamic views. This is especially the case in student projects. However, the structural view is no more important than the dynamic views. In fact, due to its static nature, a structural view is often easier to extract from source code, and easier to understand, so from a model/documentation perspective, it is more important to capture the dynamic aspects of the system.

Enterprise systems are still just software systems, and suffer from many of the same problems as other types of software systems. Many of the design patterns and principles presented in these notes are applicable to systems other than enterprise systems; however, these patterns and principles are presented because they are regularly applied in enterprise systems.

Some of the concepts in these notes can be considered architectural design patterns, but many are also design principles that apply at any level.

Architectural design vs. detailed design

An architecture is a blue print for the system. It is primarily concerned with what the high-level components are, the behaviour of each of these components, and how the components relate to one another. This is one step above the detailed design, which is concerned with class structure, data types, and algorithms. The line between software architecture and detailed design is blurred, and there is no clear place in which we stop architectural design and start detailed design: both are iterative activities.

One rule of thumb to use to determine what is architecture and what is detailed design is to look at the highest level components in the system, and think about what changes need to be made when one of these components change. If a change in one component requires a change in another, it is probably an architectural design decision.

For example, if a change in a database schema at the back-end of a system changes, we probably need to change the corresponding object/class structure that the data is transformed into. As such, how this data is transformed becomes an architectural design decision. This issue is dealt with in Chapter 5 *Object-to-relational structural design*.

Presenting an architectural design

There are a number of ways that architectural designs can be presented. As noted about, the three different views — structural, behavioural, and interaction — offer a good starting point. However, we exactly does this mean from a presentation viewpoint? One way to interpret this is:

1. *Structural*: What are the key high-level components? What are the high-level sub-components of these components that affect how they interact with other components?

   For this, we would need to present what the components are, and what the sub-components inside these components are — that is, those sub-components that influence other high-level components.

2. *Behavioural*: What is the behaviour of the high-level components and their sub-components? For this, we would need to present the *interfaces* to each of the high-level components, and the relevant aspects of the sub-components interfaces that affect the behaviour of the components; that is, those related to interaction.

An interface should describe the services that a (sub-)component provides to other components. Typically, this consists of a list of operations/methods that the components provides, the inputs and outputs of each operation, and a description of what the operation does for the calling component.

To describe operation behaviour, we have to specify the state space for the system (that is, what data does it encapsulate), and for each operation, how that data is changed for a set of inputs, and what the resulting output is. State diagrams and activity diagrams are one possible way to specify this, but informal descriptions suffice for many projects.

3. *Interaction*: How do the components and sub-components interact?

   Interaction protocols are typically used to specify the interactions. These provide readers, including developers, with information about how the components should interact.

   Typically, interaction protocols are presented relative to a set of use cases for the system. That is, for each use case, specify the component interactions that must occur to complete that use case. In many cases, doing this for only the key use cases is sufficient to establish interaction, and the remainder are either not specified, or only sketched.

The need for design

Hopefully you will already have an appreciation for why design is necessary in software systems. When engineering a large-scale software system, it is simply not possible to jump straight to coding and hope that everything comes out fine. The larger the project and the more people working on it, the more communication is required. Designing a system and recording that design are both important.

A design model or document becomes an important communication tool that demonstrates a *shared* understanding of the design of the system. It is tool for communication between developers: from those that design the system to the programmers that will code it; from those that designed the system to the test engineers that will verify it; and from those that design the system to those who will maintain it, long after the original designers, programmers, and testers are gone. That is, it is used as a tool throughout the development and maintenance processes.

A design model or document captures the early design decisions of a project — and mainly those that are difficult and expensive to correct once development has begun. Of course, this implies that it is valuable getting these decisions right. Being able to communicate the design with other stakeholders, thus giving them a chance to point out problems, is highly valuable, and can save much effort and cost.

As such a crucial communication tool, it is important that the design model/document accurately records the architecture of a system, and that this record is kept up to date. At a minimum, it should record the structural, behaviour, and interaction views of the system, plus others, such as the control flow, if the system is complex enough.

If you do not agree that recording designs is important, I have an exercise for you. Go to the following website, which allows you to browse the source code for the Eclipse Git repository: `http://git.eclipse.org/`. Now, find out how the Eclipse developers de-couple their user interfaces from the underlying Eclipse functionality and data. Note: I do not know the answer to this, but I am willing to bet that you cannot find this yourself from the source code.

If you attempted the above exercise, did you find it difficult? I am going to assume that you said "Yes". Now, think about this: Eclipse is a relatively simple application. It operates as a stand-alone application running Java, so has only one user at a time, and does not have many concerns regarding the hardware or operating system on which it runs. It does not maintain terabytes of data. It does not interact with other systems that are running in separate processes or on separate machines. Most enterprise applications must consider all of these aspects.

Principles

It is usually understood by software developers that coupling, cohesion, and encapsulation are important aspects of Object-Oriented Design (OOD). However, many of them find it is difficult to achieve the benefits of low coupling, high cohesion and strong encapsulation in practice. This issue is related to what is called ”dependency management” of OOD, which can -if performed poorly- lead to have a poor design that pervade the overall architecture of the software, and it will exhibit the following symptoms [6]:

- *Rigidity*: The design is difficult to change.
- *Fragility*: The design is easy to break.
- *Immobility*: The design is difficult to reuse.
- *Viscosity*: It is difficult to do the right thing.
- *Needless complexity*: The design contains elements that aren't currently useful.
- *Needless repetition*: Redundant modules/code in the system makes it is difficult to maintain it.
- *Opacity*: The design has modules that are difficult to understand.

To help developers eliminating the symptoms of poor software design, Robert Martin [6] defined a set of guidelines that helps to achieve a good software design, which is known as *Principles of Object Oriented Class Design*. The acronym SOLID was coined by Michael Feathers to help remembering them.

These five SOLID principles are:

1. **S**ingle Responsibility Principle

   A class should have one, and only one, responsibility. This principle states that if we have two reasons to change for a class this means the responsibilities become coupled, we have to split the functionality in two classes. Because when we need to make a change in a class having more responsibilities, the change might affect the other functionality of the classes.

2. **O**pen Closed Principle

   You should be able to extend a classes behaviour, without modifying it. It means simply this: We should write our modules so that they can be extended, without requiring them to be modified. Indeed, abstraction is the key here.

3. **L**iskov Substitution Principle

   Derived classes must be substitutable for their base classes. That is, a user of a base class should continue to function properly if a derivative of that base class is passed to it.

4. **I**nterface Segregation Principle

   Make fine grained interfaces that are client specific. For example, if you have a class that has several clients, rather than loading the class with all the methods that the clients need, create specific interfaces for each client and multiply inherit them into the class.

5. **D**ependency Inversion Principle

   Depend on abstractions, not on concretions. That is, depend upon interfaces or abstract functions and classes, rather than upon concrete functions and classes.

   By applying these principles during to design object oriented systems, it can be noticed that some structures appears repeatedly over and over again which are known as design patterns [7].

Patterns

Like "architecture", defining "pattern" is difficult. Fowler [3] does the wise thing and takes someone else's:

> "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." — Christopher Alexander, building architect.

The above quote is a great definition of a software design pattern, even if it was intended to be a definition of a building design pattern.

Design patterns and principles are not science: they are engineering. They do not result from theory or scientific thinking, but from practice and experience. Patterns come about when people identify that they are solving the same problem many times. This is why many people say that they have "discovered" a pattern, rather than "invented" one.

Like design models and documents, patterns are a communication device. The value of patterns come from the fact that they communicate knowledge. Their value is exploited as both a design tool, and a tool for understanding. On the one hand, design patterns communicate to developers a set of ideas that can be applied to their own project. This is valuable.

On the other hand, perhaps a more powerful aspect of a pattern is what it communicates about a design itself. That is, when a designer uses a pattern, someone who is familiar with that pattern can pick up the design and understand at least part of it straight away.

For example, consider the *Observer* pattern [4]. If a designer uses the same concept, they could describe their design as follows:

> "Some objects will want to know that the state of object A changes. So, object A maintains a list of objects that have registered with it. Those objects that want to know when the state of A changes can register with A. When A recognises its state has changed, it can inform all registered objects of this fact."

This is fine, and it communicates the idea of an observer. However, a much better description of the design is: "The Observer pattern is used here".

It is worth noting that patterns can never be applied straight out of the box. They must always be fitted to the particular problem at hand, and this requires creative design. First, one must figure out that the pattern is applicable. Second, one must figure out how to apply is to the given situation.

1.3 Layering

Layering is one of the design principles used in software engineering to break a system into smaller parts that fit together. Layers are used from low-level code, such as network stacks and operating systems, to high-level systems, such as enterprise systems and desktop applications.

1.3.1 Layering in software systems

Figure 1.1 presents a visualisation of the layering principle. Each layer depends on the layers below it, so Layer 1 uses Layer 2 to fulfil its responsibilities, but no layer should know about any layers above it. Further, it is also advised that a layer knows only about the layer directly below, meaning that each layer hides the layers below it.

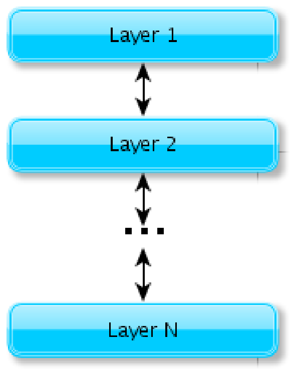


Figure 1.1: An abstraction of a layered architecture.

Pros

Layering, if done properly, minimises dependencies between layers using encapsulation, which has many benefits:

1. *Understandability*: We can understand a single layer independently by understanding it and the layer directly below it.
2. *Substitution*: We can substitute a layer with a new layer that shares the same interface with little disruption.
3. *Re-usability*: We can re-use a layer, or a group of layers, from one system in another without having to consider the layers higher up.

Cons

There are some downsides to layering, although for any non-trivial system, these are generally far outweighed by the pros:

1. *Change*: Layering does not necessarily isolate any layers from change. That is, if a change in requirements is made to the system, it will often result in a change to every layer, especially if that change adds a new type of data to the system.

2. *Performance*: Layering often has negative performance implications. At each layer, the inputs/outputs to the layer may have to be transformed from the format used in the neighbour layers.

1.3.2 Layering in enterprise systems

In this subject, we will concern ourselves only with a particular layering system, consisting of three layers. Such an approach is taken in most enterprise systems, and is a logical way to structure an enterprise system. These layers may consist of sub-layers, but from an architectural point of view, there are only three top-level layers.

The first 9 chapters of these notes are about how to break an enterprise system into layers, how to design these layers, and how to effectively interface between the layers. Most enterprise systems are layered using the scheme outlined in Figure 1.2, consisting of three layers: the *presentation* layer, the *domain* layer, and the *data source* layer. The responsibilities of each layer are outlined in Table 1.1, taken from Fowler [3].

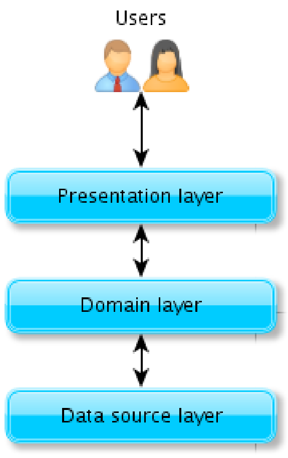


Figure 1.2: The typical three-layer architecture used in enterprise systems.

The *presentation* layer is responsible for handling interactions between the user and the system, by taking user input and calling the correct part of the system, and interpreting and displaying information sent back from the system. This is generally in the form of a user interface, such as a rich graphical interface, a web browser, a command line, or an interface to an external system (yes, users can be non-human).

The *domain* layer is responsible for implementing the logic of the system by interpreting commands received from the presentation layer. That is, doing calculations, determining what information to change and what information to display, and determining what to store/remove, etc.

The *data source* layer is responsible for the handling of communication with the systems that contain the data, such as databases, file systems, messaging systems, and external data sources.

Together, these three layers handle the data in the system, the logic, and the view, which is similar to the familiar *model-view-controller* pattern.

Table 1.1: The three principle layers.

Layer	**Responsibilities**
Presentation	Provision of services Display of information
Domain	Implementing the business logic of the system
Data source	Communication with data stores (e.g. databases, file systems) Messaging systems Transaction managers Interfacing to other packages

Where to run the layers

In an enterprise system, one question that must be answered is where to run the different layers: on a client, or on a server.

There are some simple trade-offs that must be made:

- *Presentation layer:* If the presentation layer is implemented on the client side, it provides faster response time, allows for disconnected operations, and for more powerful interfaces, because calculations can be done on the client side. This is known as a *rich client*.

  The opposite of a rich client is a *dumb client*. Typically, these are HTML interfaces. The advantage of this is that everything is done on the server, so a change in the system does not require an update on all of the clients (so there are no version incompatibility issues). Using a HTML interface means almost universal availability; all a user needs is a browser and a network connection.

- *Domain layer:* This has similar trade offs to the presentation layer in efficiency and maintenance issues. One can also split the domain logic layer between clients and servers, but this is quite messy, and so should be reserved for when is absolutely necessary.

- *Data layer:* The data layer is almost always run on servers. In an enterprise system, typically the data must be accessible enterprise wide, so there is little benefit in having it on the client. The only exception is session data, which is discussed further in Chapter 7.

These principles are rules of thumb, and ultimately the decisions must be made on a per-system basis, and will depend on many factors, such as the proposed system behaviour, the resources available, the preferences of the clients and other stakeholders, etc.

1.4 Transactions

As a final note in this chapter, we will discuss *transactions*. An understanding of transactions is key to understanding software architecture in enterprise systems, because so many enterprise systems are built around transactions. Transactions are the primary tool used in enterprise systems to prevent conflicts in concurrent access of data – a requirement in almost any enterprise system.

Definition 1.3 (Transaction)
A *transaction* is an atomic, bounded sequence of work with well-defined start and end points that maintains the integrity of the associated resources. □

To further understand this definition, we can break it down. First, a transaction is atomic, meaning that it must either succeed or fail. That is, there is no partial completion of a transaction; transactions must complete on an "all or nothing" basis. If a transaction commences but cannot complete, it must roll back to the starting state. Second, a transaction must ensure that all resources are consistent before and after execution.

In computing, transactions are often described using the *ACID* properties:

- *Atomicity*: The transaction must complete in its entirety, or roll back to the start state; the "all or nothing" property.
- *Consistency*: All associated resources must be in a consistent, non-corrupt state at the start and end of the transaction.
- *Isolation*: Any effect that a transaction has on resources must only be visible to other transactions after completion; that is, changes must not be visible during execution.
- *Durability*: The results of the transaction must be permanent; that is, it must survive system shutdowns and crashes.

Database management systems utilise transactions to preserve the integrity of their data in the face of concurrent access, and the use of database management systems in many enterprise systems has meant that transactions are also used in many enterprise systems.

Business transactions vs. system transactions

The term "transaction" may conjure visions of a transaction in commerce; e.g. paying for something in a shop. This is because the notion of a computing transaction came from the idea of a commercial transaction. When you buy something, the properties described above hold. If you hand over your money but then the seller realises they have run out of the goods you request, they will give you your money back, rolling back to the starting state. The transaction resources must be consistent, so the books must balance at the end of the transaction, etc. This is an example of a *business transaction*.

In enterprise systems, we must deal with both *business transactions* and *system transactions*. A business transaction is one that is related to the domain of the system, rather than the implementation itself. For example, a financial transaction on an online book seller is a business transaction. A system transaction is related to the *implementation* of business rules. For example, committing the order and payment details to the database. System transactions are hidden from the user, while the business transactions form how the user interacts with the system. A business transaction is related to the requirements, while a system transaction is related to the design and implementation.

The distinction between business and system transactions is important because there is often a nontrivial relationship between business transactions and related system transactions. For example, the implementation of a single business transaction may consist of multiple system transactions. The business transaction may have to obey the ACID properties, and the system transactions that implement it may have to do this as well. The example of the online book seller is one such example, in which the entire transaction of buying a book may consist of multiple system transactions: entering payment details, shipping details, confirmation, etc. These so called *long transactions* are not straightforward to implement, and can cause problems in enterprise systems. They are discussed further in the chapter on concurrency (Chapter 8).

1.5 Further reading

- To read further about software architecture and its importance in the software engineering process, read Linda Northop's chapter "What is software architecture?" in Bass et al. [1].
- To read further about Object-Oriented Design Principles, read the Chapters 7-12 in Martin et.al. [6].
- To learn more about layering in enterprise systems, read chapter 1 of the subject text (Fowler [3]), available as an e-book through the University of Melbourne library.

Chapter 2

Organising Domain Logic – Principle and Patterns

In the previous chapter, we discussed the concepts of *layers* in an enterprise system. Typically, an enterprise system is divided into the presentation, domain logic, and data source layers. These layers are coupled, and it is the job of the designers to keep this coupling as loose as possible.

In this chapter, we will look at three architectural design patterns and principles for use in the *domain logic layer* in enterprise systems. These patterns provide designers with ways to structure the domain layer of an enterprise system to help with *maintainability* and *understandability* (these two quality attributes are closely related).

The three patterns are: (1) *transaction script*; (2) *domain model*; and (3) *table module*. These three patterns are alternatives to each other, however, they can be used in tandem as well; two parts of the layer implemented using two different patterns[1]. We will discuss the pros and cons of each pattern, and in which circumstances they are considered better choices than the others.

We will also discuss a fourth pattern, *service layer*, which can be combined with any three of the above to separate the domain logic layer into its domain and application components.

Learning outcomes A person familiar with the material in this chapter should be able to:

1. describe the four design patterns related to domain logic, and what problem they aim to solve;
2. apply the design patterns in enterprise applications;
3. critique the choice of one of the four design patterns on a particular application; and
4. provide a rationale for the choice of one the presented patterns in an enterprise application.

2.1 The problem

The problem: How do we organise the domain logic of an enterprise system to make it understandable and maintainable?

[1]Although we will not cover such combination in this subject.

Table 2.1: Revenue booking for three products.

Product	**Today**	**30 days**	**60 days**	**90 days**
A	100%	-	-	-
B	$33.3^{*}\%$	-	$33.3^{*}\%$	$33.3^{*}\%$
C	$33.3^{*}\%$	$33.3^{*}\%$	$33.3^{*}\%$	-

The domain layer is where most of the business rules in a system are implemented, and so is where much of the complexity lays. The domain layer design should be structured make it clear how to identify the transactions that it supports, and the effects these transactions have on the domain. Further, the design should also be extensible so that new types of transactions and new business rules can be added while minimising the change impact on the rest of the layer, and the system in general.

The three design patterns that we will discuss each have strengths and weaknesses with respect to these goals.

2.2 Motivating example

As a motivating example, we will take the *revenue recognition problem* described by Fowler [3]. This is a common problem in business systems.

The goal of revenue recognition is to maintain, on your accounting books, the revenue that your business has received. The revenue recognition problem results because there are many different ways for revenue to be charged. For example, if I get paid a fee for 12 months service, I may not be able to book the entire fee straight away because I have not completed the service. I may book $\frac{1}{365}th$ of it each day, or $\frac{1}{12}th$ of it each month, so if you pull out after three months, I have only booked three months work.

In most domains, there are many such rules, and many of the rules have no relationship to each other. As an example, consider a company that sells three products: A, B, and C, along with some support for using these products. If a client buys product A, the company receives all of the revenue upfront. If the client buys product B, the company receives one third of the revenue now, one third in 60 days, and one third in 90 days. If the client buys product C, the company receives one third of the revenue now, one third in 30 days, and one third in 60 days. Table 2.1 outlines these figures.

Given this set of business rules, one problem is how to organise our domain logic such that we can implement these rules, while supporting maintainability, extensibility, and understandability; as well as the other "-ilities" that are important from both the customer and developer perspectives.

2.3 The patterns

2.3.1 Transaction Script

Pattern name	Transaction script
Description	Organises business logic by procedures where each procedure handles a single requirement from the presentation.

Many enterprise systems work around *transactions*[2]. Each transaction will read some information from the system, modify some information in the system, calculate some new information, or any combination of the three. Thus, a transaction captures some business logic that is used to support an enterprise.

The *transaction script* pattern is a simple pattern that organises the domain layer such that each transaction is organised into a single procedure (or *script*), with common behaviour factored into shared sub-procedures. In general, these scripts should not interact, and there should be no dependencies between them.

Implementation

There are a few approaches for implementing a transaction script. The common theme is that there is a single method/procedure for each transcript. While this method/procedure can use other methods/procedures to execute the transaction, each transaction is served by exactly one script, and each script serves exactly one transaction.

There are two common ways that these are implemented:

1. Group methods into classes, where a class groups together similar methods. This is best in most cases.
2. Have only one method per class, using the *command* pattern [4]. Figure 2.1 shows an example of this for the revenue recognition problem. This allows run-time management of transaction scripts, but comes with an associated design cost.

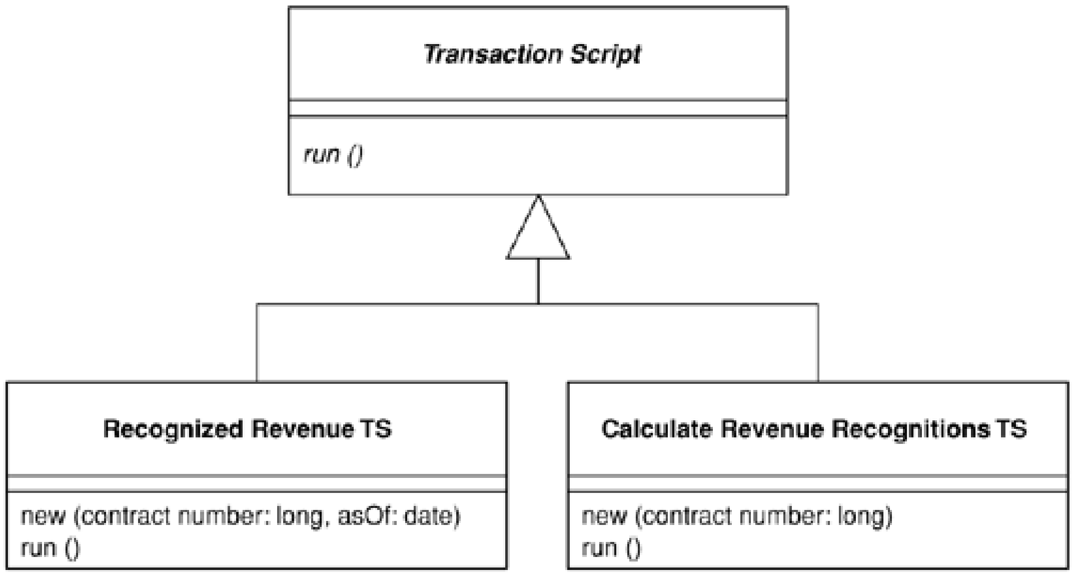

Transaction Script
run ()
Recognized Revenue TS
new (contract number: long, asOf: date)
run ()
Calculate Revenue Recognitions TS
new (contract number: long)
run ()

Figure 2.1: Using the *command* pattern to manage transaction scripts.

Pros

1. *Simplicity*: The major attraction of the transaction script pattern is its simplicity. It is simple

[2]See Definition 1.3 for a definition of "transaction" in the context of enterprise systems.

to model and understand. If one knows the transactions of the system, it is straightforward to find the corresponding behaviour. In addition, maintenance is straightforward because one needs only to learn about a single transaction to modify it, and a new transaction script can be added to correspond to new business rules.

2. *Transaction boundaries*: In addition, it is straightforward to identify the transaction boundaries; that is, where a transaction starts and ends. Each transaction is opened at the start of its corresponding transaction script, and closed at the end of its transaction script.

Cons

1. *Scalability*: The downside this pattern is that it does not scale well. As the business logic increases in complexity, the complexity of the domain layer increases exponentially.

2. *Duplication*: A second side effect is that it tends to result in duplicate code. Because a transaction script can be understood in isolation, and indeed implemented by different develops, there tends to be no over-arching view of the different transactions. Sub-routines can be used to factor out code duplication, but the common behaviour tends to be difficult to spot because: (a) the common behaviour will be implemented in different ways by different developers; and (b) generally no single person understands how all of the transactions are implemented, and the structure of the patterns encourages this.

Example

As an example, we consider the following transaction script for calculating the revenue of a contract, which specifies the conditions of buying a product of type A, B, or C. This example is taken directly from Fowler [3].

Figure 2.2 outlines a high-level model of the main concepts in this transaction script.

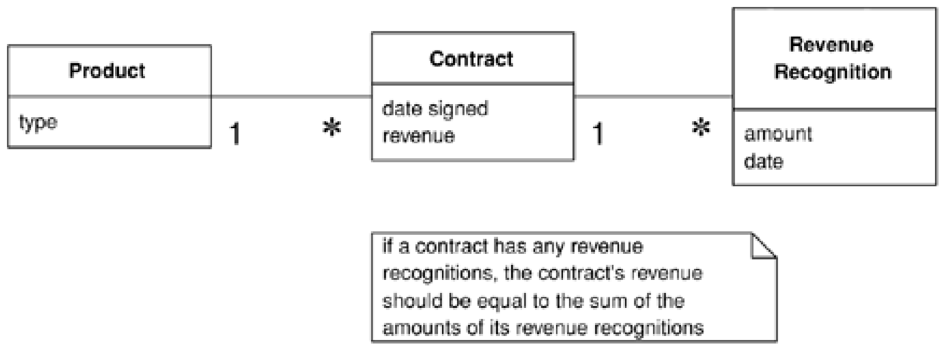

Product
type
1
*
Contract
date signed
revenue
1
*
Revenue Recognition
amount
date
if a contract has any revenue recognitions, the contract's revenue should be equal to the sum of the amounts of its revenue recognitions

Figure 2.2: A conceptual class model for the transaction script design of the revenue recognition problem.

The code itself is implemented as a service:

```
class RecognitionService {
  ...

  public void calculateRevenueRecognitions(long contractNumber) {
    try {
      Contract contract = db.findContract(contractNumber);
      Money totalRevenue = Money.dollars(contract.getBigDecimal("revenue"));
      MfDate recognitionDate = new MfDate(contract.getDate("dateSigned"));
      String type = contract.getString("type");

      if (type.equals("A")){
        db.insertRecognition(contractNumber, totalRevenue, recognitionDate);
      }
      else if (type.equals("B")){
        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("C")) {
        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);
    }
  }
```

The calls at the top of the method are not so important to understand in this context, although Fowler does go into significantly more detail on this in his book [3]. The important thing to take away from this example is the way that the contract recognition is done using a transaction script. Explicit branches are used to distinguish between contract/product types (e.g. `type.equals("A")`), and the corresponding behaviour is implemented in the true branch of the block.

The interaction diagram for this example is shown in Figure 2.3. From this, we extract the important call information that shows that when a call comes into the `calculateRevenueRecognititions` script, the script pulls the relevant contract information from the data base[3], resulting in a set of contracts, and then finally calculates the total revenue and inserts it into the database.

2.3.2 Domain model

Pattern name	Domain model
Description	An object model of the domain that incorporates both behaviour and data.

The *domain model* pattern involves constructing a model of the business domain of the system, usually

[3]This uses a *data gateway*, which is just a thin layer to a database, discussed in Chapter 3.

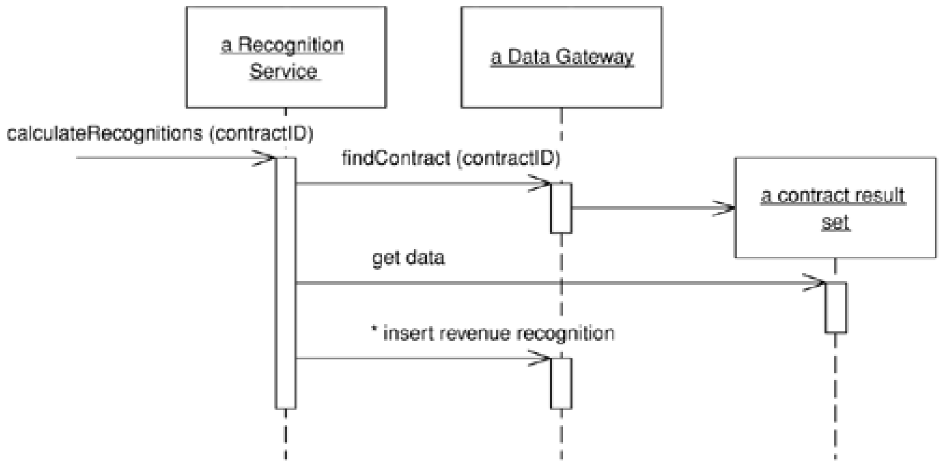

a Recognition Service
a Data Gateway
calculateRecognitions (contractID)
findContract (contractID)
a contract result set
get data
* insert revenue recognition

Figure 2.3: An interaction diagram for the transaction script design of the revenue recognition problem.

an object-oriented model, and then coupling the behaviour to the objects in the domain using methods. This is the very essence of object-orientation – each object takes the part of the logic that is related to it. In an enterprise system, each object created in the domain logic layer corresponds to an object in the business, and the methods on that object correspond to the business rules about that object.

Implementation

There are typically two broad approaches to implementing the domain model pattern. In both approaches, the domain model itself is the same, however, the way that the objects are created differs.

For small applications, one may simply load the object graph from a file and place it into memory. This is fine for a small application with a small number of objects, generally running on a single machine. However, for larger applications, this takes up too much memory.

An alternative approach is instead to store the information required in a database, and then only create the objects as required. For example, in the revenue recognition problem, only create objects that are related to a particular contract ID, execute the domain logic, and then destroy the objects. If the object graph is required over multiple transaction calls, it can either be created each time, or stored as part of the session state (covered later in Chapter 7).

Pros

1. *Extensibility*: A major attraction of the domain model pattern is that to add more behaviour, one only need to add the classes/objects to the domain model, and how to create these. Therefore, if a system has changing business rules or complex calculations/validations, the domain model pattern is highly suitable.

2. *Managing domain complexity*: A second attraction, which is related to the first, is that the domain logic itself reflects the business domain, so the layer itself is a more natural fit to the business logic than a transaction script approach. This aids in managing the complexity within the domain.

3. *Re-usability*: While the logic attached to the domain objects may be application specific, often parts of the domain model can be re-used in other applications. The high-level model itself is

a primary candidate for this. A transaction script, on the other hand, is unlikely to be re-usable unless the same transaction occurs in a different system.

Cons

1. *Complexity*: A major con is the complexity of the approach itself. For a system with a handful of simple transactions, the added effort of creating a domain model, implementing it, and mapping between the database and domain model is a high overhead that is probably not worth paying.
2. *Experience*: Experience seems to suggest that a development team unfamiliar with using the domain model pattern will require significant effort to make the paradigm shift. Transaction scripts, on the other hand, do not seem to suffer from this problem to the same extent.

Example

We use the revenue recognition example again to illustrate the concept, taking the example from Fowler [3]. First, consider the domain model in Figure 2.4. Note that, as opposed to Figure 2.2, that the type of revenue recognition is now modelled as classes, rather than an explicit field in the `Product` class. Further, note that the model contains both data *and* behaviour.

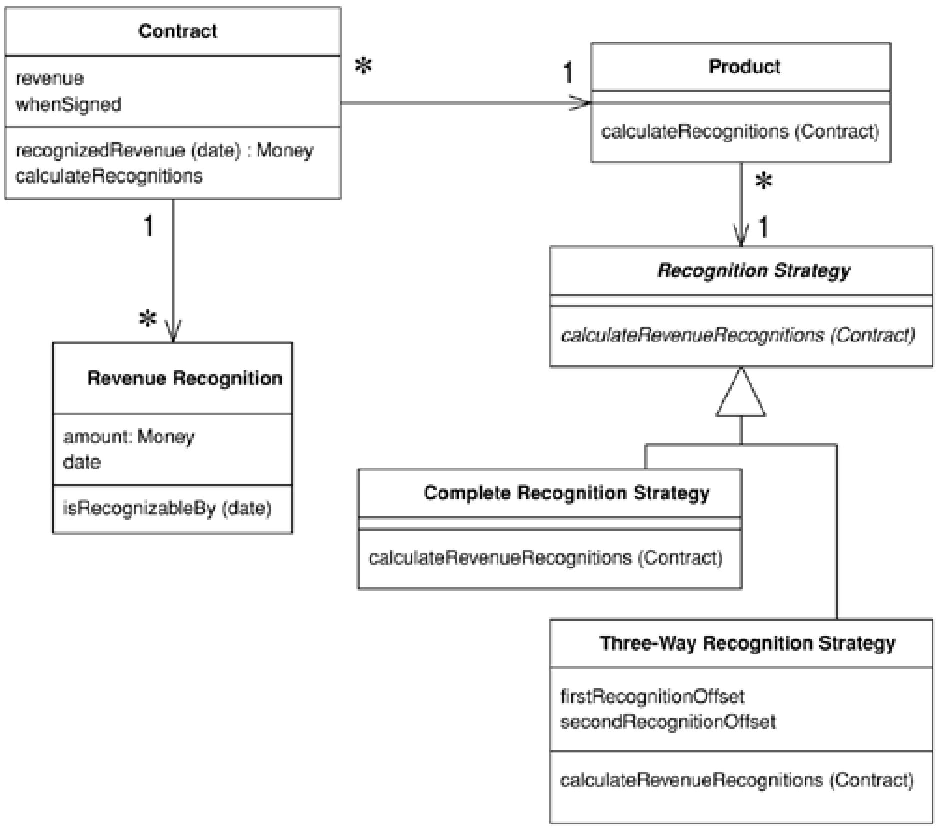

Contract
revenue
whenSigned
recognizedRevenue (date) : Money
calculateRecognitions
*
1
Product
calculateRecognitions (Contract)
*
1
1
*
Recognition Strategy
calculateRevenueRecognitions (Contract)
Revenue Recognition
amount: Money
date
isRecognizableBy (date)
Complete Recognition Strategy
calculateRevenueRecognitions (Contract)
Three-Way Recognition Strategy
firstRecognitionOffset
secondRecognitionOffset
calculateRevenueRecognitions (Contract)

Figure 2.4: A domain model for the domain logic design of the revenue recognition problem.

To use the domain model pattern, we again get the contract from the database, but now we create the domain objects, and call `calculateRecognitions` on the `Contract` class, which asks the product for its revenue. This is implemented as follows:

```java
class Contract {
  private Product product;
  ...

  public Money calculateRecognitions() {
    Money result = Money.dollars(0);
    Iterator<RevenueRecognition> it =
      product.getRevenueRecognitions().iterator();
    while (it.hasNext()) {
      RevenueRecognition r = it.next();
      result = result.add(r.getAmount());
    }
    return result;
  }
}

class Product {
  private RecognitionStrategy recognitionStrategy;
  ...

  public Money calculateRecognitions(Contract contract) {
    recognitionStrategy.calculateRevenueRecognitions(contract);
  }
}
```

From Figure 2.4, we can see that each `Product` has a reference to the type of revenue recognition associated with it. The call in the above code asks the recognition strategy subclass for its revenue. There are two such strategies: one that calculates that all revenue can be charged straight away (product type A), and one for “three-way” strategies, which are those with offsets (products B and C). These are implemented like so:

```java
class CompleteRecognitionStrategy extends RecognitionStrategy {

  void calculateRevenueRecognitions(Contract contract) {
    contract.addRevenueRecognition(
      new RevenueRecognition(
        contract.getRevenue(),
        contract.getWhenSigned()));
   }
}

class ThreeWayRecognitionStrategy extends RecognitionStrategy {

  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)));
  }
}
```

Again, the details of the implementation are not so important. Instead, this example illustrates how the behaviour related to the business logic is tied with the domain objects. The domain model captures that there are two types of charging: one that collects all revenue upfront, and one that has three periods.

The interaction diagram in Figure 2.5 shows the series of calls that are made. This illustrates how the behaviour related to revenue is passed on to the object to which it is related.

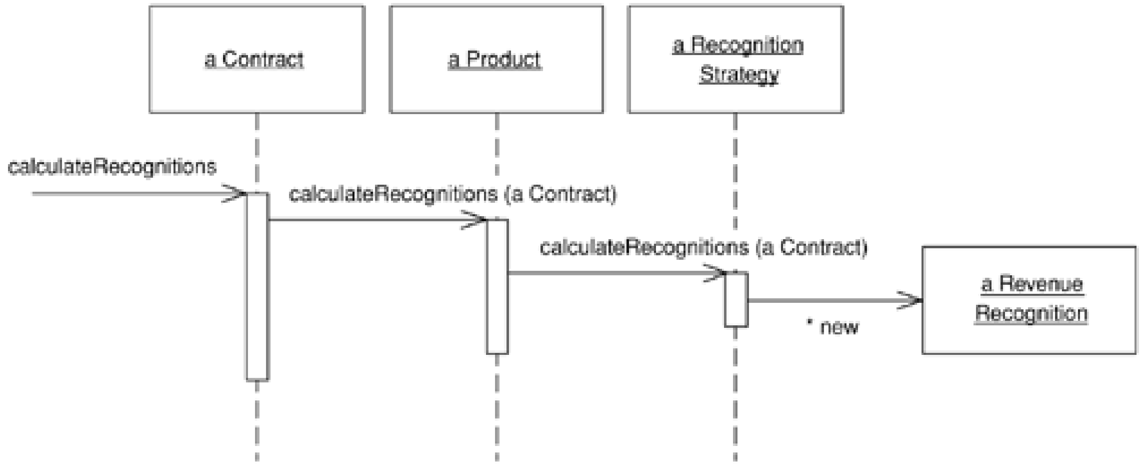


Figure 2.5: An interaction model for the domain logic design of the revenue recognition problem.

The strength of the domain model approach is that, if we wanted to add a new type of revenue recognition, such as paying weekly installments, we would just need a new subclass of `RecognitionStrategy`.

Comparison with the transaction script pattern

Pattern	When to use
Transaction scripts	– Simple transactions – Business rules not subject to change
Domain model	– Complicated business rules – Business rules that are subject to change – Team experienced with domain model pattern and domain models

2.3.3 Table Module

Pattern name	Table module
Description	A single instance that handles the business logic for all rows in a database table or view.

The *table module* pattern attempts to blend the transaction pattern and domain model patterns. It is a modification of the domain model pattern that ties the behaviour of the business rules to individual database tables. For example, in the revenue recognition problem, if there a database table called *Product* that records all products and their type, the implementation of the rules would be tied to this. Such an approach does not consist of scripts for transactions, but instead delegates the behaviour to the object most qualified to deal with it, like in the domain model pattern. However, when using a relational database, inheritance cannot be used, and therefore taking advantage of polymorphism, which is a key aspect of the domain model pattern, is not possible.

Example

As an example, consider the following implementation of the business rules for the revenue recognition problem:

```
class Product {
  ...

  public void calculateRecognitions (long productID) {
    DataRow productRow = table.getRow(productID);
    RevenueRecognition rr = new RevenueRecognition (productRow);
    Product prod = new Product(table);

    if (prod.getProductType(productID) == ProductType.A) {
      rr.insert(productID, amount, getWhenSigned(productID));
    }
    else if (prod.getProductType(productID) == ProductType.B) {
      Decimal [] alloc = allocate(amount, 3);
      rr.insert(productID, alloc[0], getWhenSigned(productID));
      rr.insert(productID, alloc[1], getWhenSigned(productID).addDays(60));
      rr.insert(productID, alloc[2], getWhenSigned(productID).addDays(90));
    }
    else if (prod.getProductType(productID) == ProductType.C) {
      Decimal [] alloc = allocate(amount, 3);
      rr.insert(productID, alloc[0], getWhenSigned(productID));
      rr.insert(productID, alloc[1], getWhenSigned(productID).addDays(30));
      rr.insert(productID, alloc[2], getWhenSigned(productID).addDays(60));
    }
    else throw new Exception("invalid product id");
  }
}
```

Again, we are not concerned so much about the details, but just how this compares to the other patterns. In this example, we can see that the behaviour is delegated to an object, in this case, an instance of `Product`, rather than implement this in the top-level script. However, the different types of contract are still handled in a single method.

Pros

1. *Simplicity*: The implementation is simple, like the transaction script pattern, but is organised around tables instead of actions.
2. *Lack of duplication*: This pattern results in less duplicate code than the transaction script, because behaviour is tied to the concepts/classes.

Cons

1. *Scalability*: Like transaction script, this pattern does not scale well.
2. *Extensibility*: Also like transaction script, this pattern does not support extensibility particularly well. As can be seen by the revenue recognition example, adding a new type of revenue recognition requires us to modify the `calculateRecognitions` method, rather than adding a new class.

2.3.4 Choosing between the three patterns

Figure 2.6 is taken directly from Fowler [3]. As he notes, the axes are unlabelled and the plots are not based on any data, so this is not intended to be a scientific graph. Instead, it is a visualisation of how the patterns scale as the domain logic complexity increases. It intends to convey that for problems with relatively simple domain logic, the transaction script and table module patterns are more suitable. However, as the complexity increases, the effort required to add enhancements to the system increases quadratically for these two patterns, while for the domain logic, the effort increases linearly with the complexity of the domain logic.

Therefore, the conclusion to be drawn from this figure is that, if designing a system that you believe has simple domain logic, and is not subject to change, either the transaction script or table module patterns are fine. For a domain logic that is complex or subject to change, the domain model pattern would be more suitable.

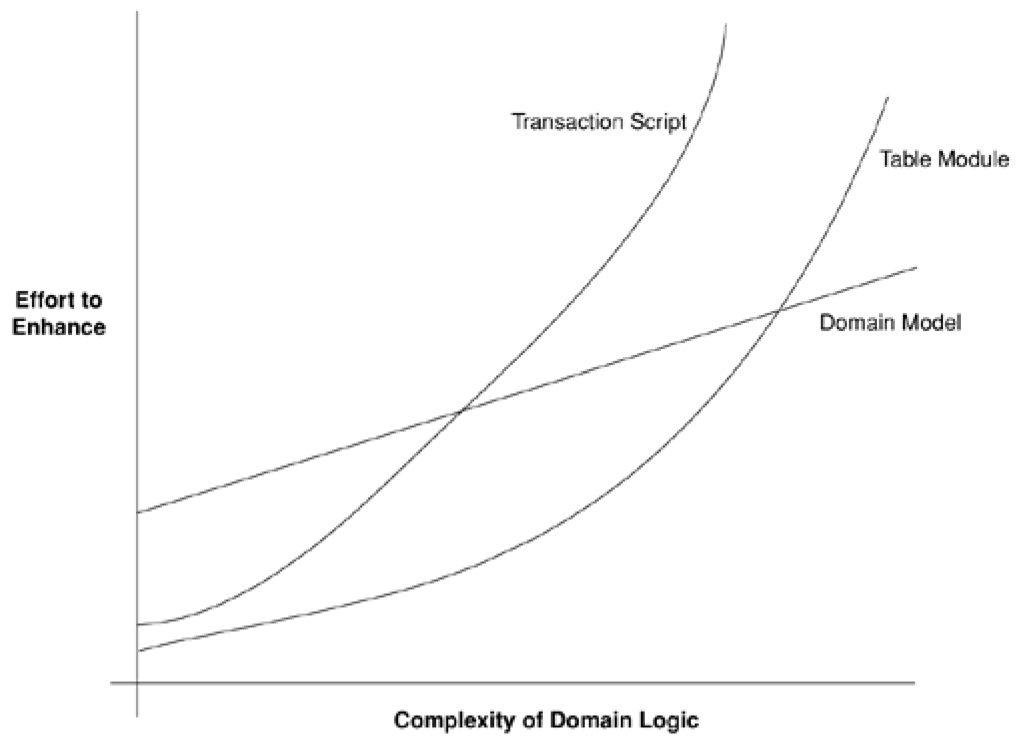


Figure 2.6: A visualisation of the relationship between complexity and maintainability for the three domain layer patterns.

2.3.5 Service Layer

Pattern name	Service layer
Description	Defines an application's boundary with a layer of services that establishes a set of available operations and coordinates the application's response in each operation.

The *service layer* pattern is one that is used in tandem with the previous three domain logic patterns that we have seen so far in this chapter. An enterprise system using one of these patterns may be required to publish an interface to more than one client; for example, a user interface and an external system. However, these clients may be exposed directly to the domain logic of the system. If the interactions involved in the system transactions involved multiple resources and multiple actions, then implementing a client may involve some logic to pull together the domain logic class etc. If we have multiple clients, this logic could be duplicated across clients, so it is worthwhile packaging these into shared methods if more than one interface will use them.

A service layer defines a set of operations that can be accessed directly by the interfacing clients, thus promoting re-use and reducing code redundancy. Figure 2.7 presents a sketch of this idea.

Implementation

What this pattern promotes is the division of the business logic into two layers: the service layer and the domain model. The decision that must be made from here is what goes into the service layer and what goes into the domain model. There are two basic variations:

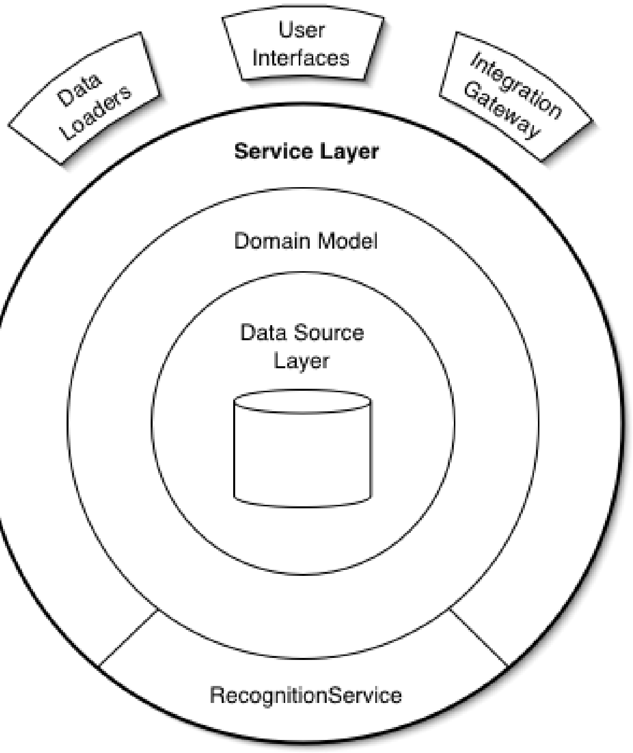


Figure 2.7: Service layer sketch.

1. *Domain facade*: the service layer is a simple facade that ties together the necessary operations in the domain layer, leaving all of the business logic itself in the domain layer. Thus, it is a true facade as defined by Gamma et al. [4].

2. *Operation script*: the domain logic is divided into *domain-specific logic* and *application-specific logic*. Domain-specific logic is that part of the domain logic about the domain itself that is independent of the system; for example, the types of revenue in the revenue recognition problem. Application-specific logic is that part of the domain logic specific to the application; for example, what information about revenue will be displayed to a particular type of user. This is sometimes called *workflow logic*.

   In the operation script approach, the service layer implements the application-specific logic, and delegates the domain-specific logic to the domain layer. The operations are implemented as scripts, generally packaged together in several classes, with each classes containing several related scripts.

The advantage of the domain facade approach is its relative simplicity. However, the operation script approach is better for large applications. In particular, it promotes re-use because the design and implementation fort the domain-specific logic can be re-use in other systems in the same domain that are required to exhibit different behaviour.

Pros

1. *Re-usability*: As discussed before, having a service layer, particularly one that separates domain-specific logic from application-specific logic, promotes re-use across applications.

2. *Multiple clients*: In addition, a service layer promotes re-use between clients. That is, if we have more than one client interface, collating shared code into service operations helps to reduce duplicate code.

3. *Maintainability*: Identifying operations for a service layer is typically straightforward. As discussed in Chapter 1, many enterprise applications consist of CRUD (create, read, updated, delete) use cases. Fowler [3] claims that there is almost always a one-to-one mapping between CRUD uses cases and service layer operations. This helps with maintainability, as the mapping from transactions to code is clearer, similar to taking a transaction script approach.

   In addition, using a service layer makes finding transactions much easier for clients.

Cons

1. *Complexity*: For a small application with a relatively straightforward domain layer, the overhead of a service layer is likely to cost more than it gains.

   Further, if a system has only one client interface (and this fact is unlikely to change), then the abstraction of scripts into a service layer will not bring as much benefit.

2. *Operation grouping*: While identifying operations is often straightforward, implementing these operations, and particularly grouping these operations into sets of related operations is far from trivial. Several heuristics exist, but this process typically requires one to consider different trade offs, which we will not discuss in this subject.

Thus, as a summary, a service layer is useful for an domain layer with multiple clients, and also promotes re-use of domain-specific logic.

Example

A short example suffices to illustrate the ideas of a service layer. In this example, we will assume that we have used the same domain logic implementation as in Section 2.3.2 (the domain model pattern). The implementation of revenue recognition in that particular instance models the domain-specific part of the application. However, as part of our application, we wish to send an email to the contract administrator notifying them that the revenue has been calculated:

```java
public class RecognitionService
  extends ApplicationService {
  ...

  public void calculateRevenueRecognitions(long contractNumber) {
    Contract contract = Contract.readForUpdate(contractNumber);
    contract.calculateRecognitions();
    getEmailGateway().sendEmailMessage(
      contract.getAdministratorEmailAddress(),
      "RE: Contract #" + contractNumber,
      contract + " has had revenue recognitions calculated.");
    getIntegrationGateway().publishRevenueRecognitionCalculation(contract);
  }
}
```

In this code excerpt, we have used domain-specific logic to calculate the revenue, but the service (note that this classed is called a `RecognitionService` — the same class name as in the transaction script approach) sends the email, which is application-specific. If we wish to build a different system in the same domain, we can re-use much of the domain logic layer as is, without bringing along the application-specific code.

2.4 Further reading

- Much of the material from this chapter is based on Chapters 2 and 9 of the subject text (Fowler [3]), available as an e-book through the University of Melbourne library.

Chapter 3

Architectural design for the data-source layer

So far, we have discussed using layers to design systems, and principles and patterns for the domain logic of the system. In the following three chapters, we will look at patterns and principles related to the *data-source layer*: both how to organise the data-source layer internally, as well as how to present the data-source layer to the domain layer.

The success of the *structured query language* (SQL) has led to the proliferation of *relational databases* as the underlying data storage system in most enterprise systems today. As a result, many of the design problems related to the data source layer are how the domain layer can query the database using SQL, while still preserving sound design principles, such as loose coupling and high cohesion. In this chapter, we will focus on patterns and principles for data-source layers using relational databases, as this is the most commonly used approach in enterprise systems.

The patterns and principles related to the data-source layer are divided into three categories:

1. *architectural*: how the domain layer and data-source layer interact with each other (Chapter 3);
2. *behavioural*: how data is loaded from and stored into the underlying database (Chapter 4); and
3. *structural*: how the domain model can be mapped to tables in the database (Chapter 5).

Learning outcomes A person familiar with the material in Chapters 3–5 should be able to:

1. describe the principles related to architectural, behavioural, and structural patterns, and what problem they aim to solve in the context of the data-source layer;
2. apply these patterns and principles in the design of a data-source layer for an enterprise systems;
3. provide a rationale for the choice of one or more of the presented patterns in an enterprise application; and
4. implement an enterprise system that used these patterns and principles.

In this chapter, we look at architectural patterns for the data-source layer.

3.1 The problem

The problem: How do we allow elegant querying of the data-source layer from the domain layer without exposing the implementation details of the data-source layer?

Accessing a database from the domain layer presents a series of problems. One such problem is how to query the database. Since SQL has established itself as a semi-standard of relational database querying, and is used in most enterprise systems today, using SQL is a given; the question is *how* to use it.

One solution is for a developer to encode the SQL query as a string, and send this string to the database, via some connection, directly from the domain layer. However, this approach breaks about every design principle ever proposed. Most critically, it breaks the design principle of maintaining loose coupling between the domain layer and the underlying database. If we want to change the underlying data store from a relational database using SQL to a semantic web approach using RDF and SPARQL, we will have to change the domain layer. Even if we want to change the underlying database schema, we would likely have to change the domain layer.

Employing SQL directly in the domain layer creates other problems as well:

- Developers working on the domain layer must be experts in SQL as well as the programming language used in the domain layer.
- If one of our queries is taking a particularly long time, and our database administrators want to performance tune it, they will have to dig around the domain layer to find it.
- The response to a query will come back as a string, which we will have to parse to get out the relevant information. Parsing such information is not part of the domain logic, so should not be done there.

For these reasons, a clear design principle that we should follow is to separate the domain logic from the data access, and access the data source layer via a well-defined interface. This chapter presents a collection of design patterns that recommend strategies for defining the interface, such as where querying methods go, how data is returned, and how these fit with domain layer patterns seen in Chapter 2.

3.2 The patterns

3.2.1 Table data gateway

Pattern name	Table data gateway
Description	An object that acts as a gateway to a database table. One instance handles all the rows in the table.

The *table data gateway* is a straightforward pattern for representing and accessing a relational database. The basic paradigm is that each table in the database is represented by a single object. The classes that define these objects encapsulate the underlying database queries (e.g. SQL queries), and each of these queries is represented by a method in the class.

The main decision to make when using the table data gateway pattern is how information is returned to the calling code, especially in the case where a set of database records is returned. One approach is to use a *result set* (or *record set*), which is a type of map that maps a key (the key of a row) to an object that contains the information relating to that key. This is the simplest approach, but is error prone because result sets do not (usually) contain type information, so no static type checking is available.

An alternative to the result set approach is to use a *data transfer object*, which is described in Chapter 9. A data transfer object is an object that carries specific information, such as the data in a row of a table, with the corresponding attributes, so that type correctness can be ensured at compile time.

Example

As an example, consider a database table called *Person*, which holds some basic information about people, including their first name, last name, and their number of dependents. The class `PersonGateway` is created, and an instance of this is used to access the information in this database.

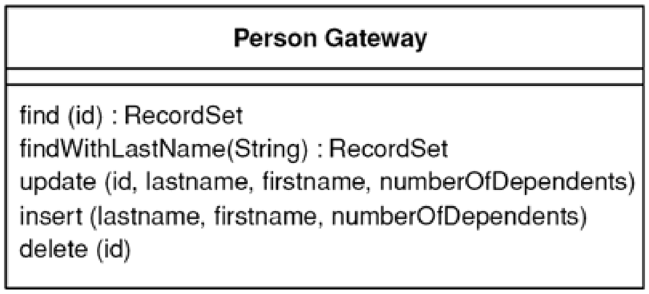

Person Gateway
find (id) : RecordSet
findWithLastName(String) : RecordSet
update (id, lastname, firstname, numberOfDependents)
insert (lastname, firstname, numberOfDependents)
delete (id)

Figure 3.1: The interface for the `PersonGateway` class, used as a gateway to the *Person* database table.

Figure 3.1 shows a possible interface for the `PersonGateway` class. Using an instance of this, we can search for a record of a person based on their system ID, which returns a single record, or on their last name, which returns a result set containing all people with the specified last name. Information can be added, modified, or deleted to/from the table using update, insert, and delete operations.

The following code snippet shows an implementation of the first three operations in the `PersonGateway` class:

```
class PersonGateway {

  public ResultSet find(long id) {
    String sql = "SELECT * FROM person WHERE id = {0}";
    String sqlPrepared = DB.prepare(sql, id);
    IDbCommand comm = new OleDbCommand(sqlPrepared, DB.Connection);
    ResultSet rs = comm.executeReader();
    return rs;
  }

  public ResultSet findWithLastName(String lastName) {
    String sql = "SELECT * FROM person WHERE lastname = {0}";
    String sqlPrepared = DB.prepare(sql, lastName);
    IDbCommand comm = new OleDbCommand(sqlPrepared, DB.Connection);
    ResultSet result = comm.executeReader();
    return result;
  }

  public void update
    (long key, String lastName, String firstName, int numberOfDependents) {

    String sql = "UPDATE person " +
                 "  SET lastname = {1}, firstname = {2}, " +
                 "      numberOfDependents = {3}, " +
                 "  WHERE id = {0}";
    String sqlPrepared =
      DB.prepare(sql, key, lastName, firstName, numberOfDependents);
    IDbCommand comm = new OleDbCommand(sqlPrepared, DB.Connection);
    comm.executeNonQuery();
  }

  ...
}
```

From this, we can see that the SQL is encapsulated in the gateway class and hidden from the calling code. In addition, the code used for handling connections and sending queries and updates to the database is also hidden. This encapsulation helps to separate the domain layer from the data-source layer, allowing us to, for example, go from SQL queries to stored procedures without disrupting the domain layer.

We can now use this gateway in our calling code to increase the number of dependents of a specified person:

```
  ...

  public void incrementNumberOfDependents(long id, int number) {
    PersonGateway gateway = new PersonGateway();
    Record record = gateway.find(id);
    Integer numberOfDependents =
      Integer.parseInt(record.get("number_of_dependents"));
    int updatedNumberOfDependents = numberOfDependents + number;
    gateway.update(id, record.get("last_name"), record.get("first_name"),
                   updatedNumberOfDependents);
  }
```

Pros

1. *Simplicity*: It is a simple pattern that works well for many applications.

2. *Precise mapping*: The mapping between the database and the gateways needed is straightforward — we require one table data gateway class per database table. This also makes it easy for users of the gateways to know which class a query resides in.

3. *Compatibility (table module)*: The table data gateway is highly compatible with the table module pattern described in Chapter 2, because it returns a result set for the table module classes to operate on.

Cons

1. *Incompatibility (domain model)*: It is not as compatible with the domain model pattern as other data-source patterns are.
2. *Scalability*: As a corollary of the previous item, it will not scale as well as the domain complexity increases.

3.2.2 Row data gateway

Pattern name	Row data gateway
Description	An object that acts as a gateway to a single record within a table. There is one instance per row in the table.

We saw in the previous section that one decision to be made with the table data gateway pattern was how results from the database are to be returned. The *row data gateway* pattern gets around this problem by using objects that mimic the records in the database table, and can also be used to update the table. So, unlike the *table* data gateway pattern, which creates one object per table, the *row* data gateway pattern creates one object per row/record; or more accurately, it creates an object for each row as it is accessed.

Example

The best way to explain the row data gateway pattern is via an example. Recall the earlier example from Section 3.2.1, in which the table data gateway was used to access the table. Compare the class design in that section (Figure 3.1) with the class design in Figure 3.2, which uses the row data gateway pattern.

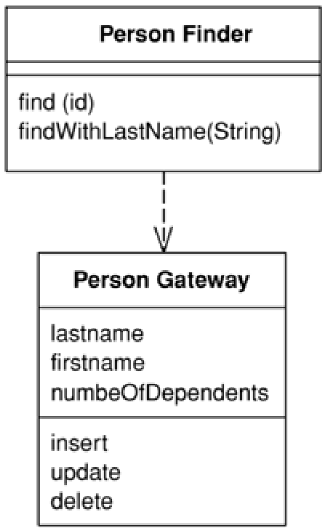


Figure 3.2: A class diagram using that row data gateway pattern for access to the *Person* database table.

In this new design, we can see that the `PersonGateway` class contains attributes corresponding to the columns in the database table: the last name, first name, and number of dependents. To modify a record, we use the operations associated with the row.

The only choice to be made here is where the "find" methods go. In Figure 3.2 they are placed in a new class *PersonFinder*, whose methods query the database and return *PersonGateway* objects, or collections of *PersonGateway* objects. Alternatively, one could use static find methods, but this removes the options of overriding.

Figure 3.3 demonstrates how the table is queried. In his case, the client class creates a *PersonFinder* instance, which queries the database for records, and then creates *PersonGateway* instances from these.

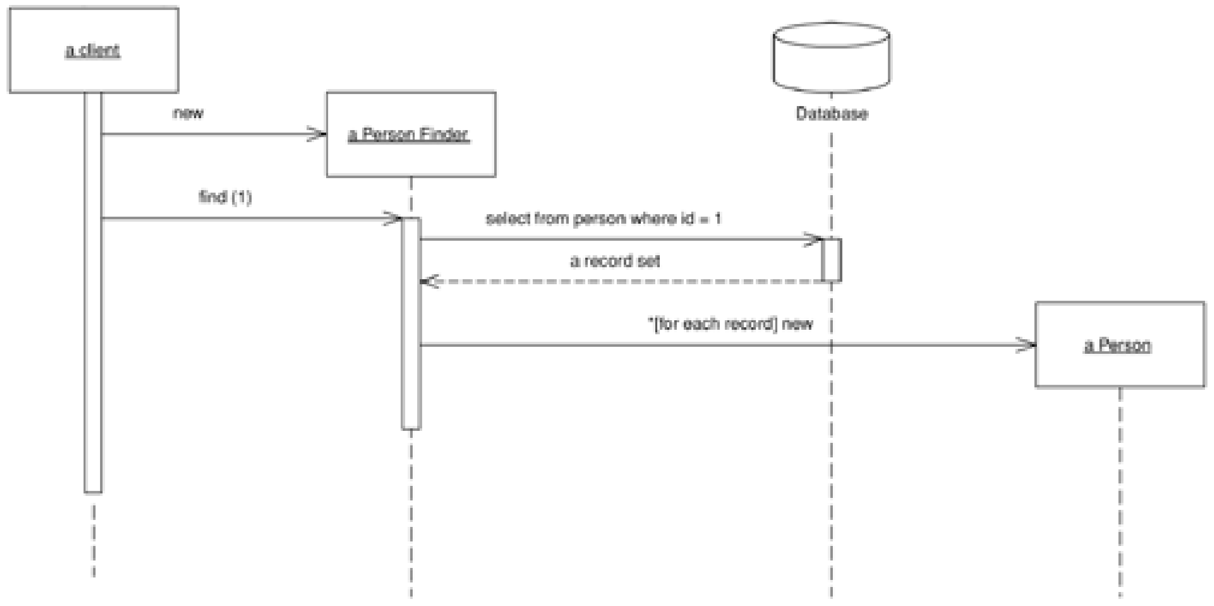


Figure 3.3: An interaction diagram demonstrating the use of the row data gateway pattern for querying a record from the `Person` database table.

The following code snippets demonstrate the implementation of this pattern, implementing the same functionality as in the table data gateway example. First, we implement the "finder" methods:

```
class PersonFinder {

  public PersonGateway find(long id) {
    String sql = "SELECT id, lastname, firstname, number_of_dependents " +
      "  from people " +
      "  WHERE id = {0}";
    String sqlPrepared = DB.prepare(sql, id);
    IDbCommand comm = new OleDbCommand(sqlPrepared, DB.Connection);
    ResultSet rs = comm.executeQuery();
    rs.next();
    PersonGateway result = PersonGateway.load(rs);
    return result;
  }

  public List<PersonGateway> findWithLastName(String lastName) {
    String sql = "SELECT id, lastname, firstname, number_of_dependents " +
      "  from people " +
      "  WHERE lastName = {0}";
    String sqlPrepared = DB.prepare(sql, lastName);
    IDbCommand comm = new OleDbCommand(sqlPrepared, DB.Connection);
    ResultSet rs = comm.executeQuery();
    List<PersonGateway> result = new ArrayList();
    while (rs.next()) {
      result.add(PersonGateway.load(rs));
    }
    return result;
  }
}
```

These finder methods return instances of the *PersonGateway* class, which is the class that mimics our database records. A partial implementation of this class is:

```
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;
  }

  public void update() {
    String sql =
      "UPDATE people " +
      "  set lastname = {0}, firstname = {1}, number_of_dependents = {2}" +
      "  where id = {3}";
    String sqlPrepared =
      DB.prepare(sql, this.lastName, this.firstName,
                      this.numberOfDependenents, getID());
    IDbCommand comm = new OleDbCommand(sqlPrepared, DB.Connection);
    comm.executeNonQuery();
  }
}
```

Note how we now have getter and setter methods for reader and updating the attributes of a person. A client using this pattern would now have access to instances of the `PersonGateway` class, and can use the regular mechanisms of the programming language to access the attributes. The details of the data source are still hidden. Note also that the `update` method contains no parameters, unlike in the table data gateway example, and instead uses the attributes as part of the update statement.

The re-implementation of our `incrementNumberOfDependents` is:

```
...

public void incrementNumberOfDependents(long id, int number) {
  PersonFinder personFinder = new PersonFinder();
  PersonGateway gateway = personFinder.find(id);
  int newNumberOfDependents = gateway.getNumberOfDependents() + number;
  gateway.setNumberOfDependents(newNumberOfDependents);
  gateway.update();
}
```

Now we are not dealing with records directly, so do not need to do potentially error-prone type conversions.

Pros

1. *Type safety*: The ability to use the programming language type system is a bonus, as we receive design-time type checking. Of course, there will still have to be a conversion between the database record and the object (this would be implemented in the `PersonGateway.load` method used in the code snippet above), but these conversions are now done in one place, rather than in every place the table is queried, and most importantly, they are encapsulated.

2. *Precise mapping*: Like the table data gateway pattern, the mapping between the database and gateways is straightforward – we require one gateway class per table.

3. *Compatibility (transaction script)*: The row data gateway pattern is compatible with the transaction script. One of the weaknesses of the transaction script is the duplication of code across transactions. If the row data gateway pattern is used, some of this duplicate code is instead re-factored into the gateway class, allowing it to be reused by different transaction scripts.

Cons

1. *Boiler plate code*: A downside is that we have to create loads of boiler plate code, including new classes, to act as the gateways. Much of this is simple getter and setter stuff that can be automated, but this still increases the maintenance headache.

2. *Database coupling*: This pattern tends to result in a close coupling with the underlying database because the gateway objects reflect the columns of tables.

3. *Incompatibility (domain model)*: This is not a downside, but more of an observation. When using the domain model pattern at the domain layer, using the row data gateway pattern results in three data representations: one at the domain layer, one for the gateway, and one in the database; only two of these are really necessary. For this reason, other architectural data-source patterns should be used with the domain model pattern, such as *active record* (Section 3.2.3).

3.2.3 Active record

Pattern name	Active record
Description	An object that wraps a row in a database table or view, encapsulates the database access, and adds domain logic on that data.

The *active record* pattern is an extension of the row data gateway pattern from Section 3.2.2 that contains domain logic in its objects. Like the row data gateway objects, active record objects classes mimic the record structure in the underlying database, and contain logic to read and modify the rows. In addition, an active record implements domain logic related to the data. It may implement all the domain logic, or leave some of its logic in the domain layer. Either way, the responsibilities of the domain layer are diminished, and consist of bringing together the various active records to implement the business logic of the application. It would seem that the soundest way to use this pattern is to include domain logic that touches the database, and move the rest into the domain layer.

A typical active record class will have static methods for creating instances from SQL results, creating new instances to be inserted into the table, and "finder" methods to wrap common database queries that return instances.

Example

Again we use the example of a database table that contains data about people. However, unlike the other patterns seen previously in this chapter, we will consider some simple domain logic, such as calculating tax exemptions based on the number of dependents. Figure 3.4 presents an example class interface for the `Person` active record, which contains the usual attributes, the usual methods for querying the database, and some additional domain logic.

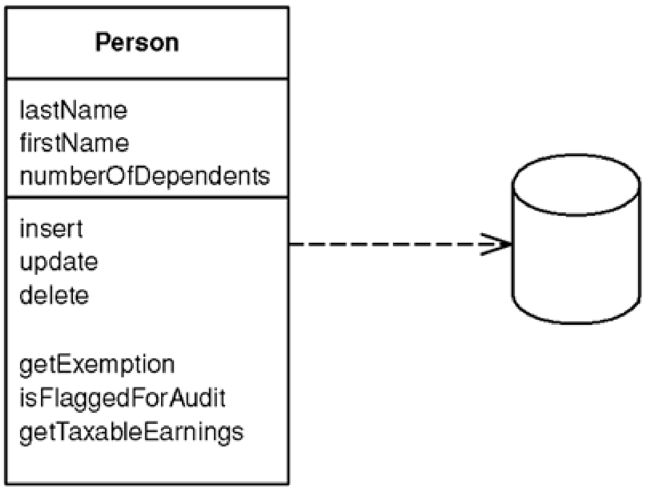


Figure 3.4: The `Person` class design using an active record that contains domain logic.

The implementation of the attributes and database access methods would remain much the same as in the row data gateway pattern. The additional domain logic methods are implemented in the class, such as the `getExemption` method, which calculates a tax exemption based on the number of dependents a person has:

```java
class Person {

  public double getExemption() {
    //$1500 is the base exemption
    double exemption = 1500D;

    for (int i = 0; i < this.getNumberOfDependents(); i++) {
      //each dependent provides a furhter $750 exemption
      exemption += 750D;
    }
    return exemption;
  }
  ...
}
```

Pros

1. *Type safety, precise mapping, and compatibility (transaction script)*: These three pros are inherited from the fact that the active record pattern is an extension of the row data gateway pattern.

2. *High cohesion*: This pattern promotes higher cohesion that the row data gateway pattern by placing logic related to the table rows in one class.

Cons

1. *Domain logic coupling*: Although this patterns promotes higher cohesion than the row data gateway pattern, it also promotes the coupling of data with domain logic, which reduces the potential for re-use. This can be somewhat mitigated by implementing active records in two classes: a base class containing the code related to database access (a row data gateway class), and a subclass of this that contains the business logic, and does not access the database except via methods in its superclass.

2. *Database coupling*: Like row data gateway, this pattern also encourages a close coupling with the database.

3. *Scalability*: Active records do not scale well as the complexity of the domain logic increases, and in particularly, the close coupling with the database schema make it difficult to use object-oriented constructs such as inheritance.

3.2.4 Data mapper

Pattern name	Data mapper
Description	A layer of mappers that moves data between objects and a database while keeping them independent of each other and the mapper itself.

A *mapper* in this context is similar to a *mediator* pattern described by the Gang of Four [4]. The *data mapper pattern* specifies a mediator/mapper that separates the in-memory domain objects from the underlying database. The intermediate mapping class takes result sets from database queries and creates domain objects representing that data, as well as doing the reverse. The result is a domain layer that does not need to know the underlying database schema (unlike with the previous patterns in this chapter).

The advantage of taking this approach is that we can use a rich domain layer that takes advantage of object oriented constructs, without this layer having to deal with the database layer itself. Therefore, the data mapper pattern is highly suited to the domain model pattern from Chapter 2.

A key decision that needs to be made is how many mapper classes are required. The default is to use one mapper class for the entire system, however, this clearly becomes huge and difficult to maintain as the size of the domain increases. For a non-trivial domain, it is neater to use one mapper per domain class, or one mapper class for every class that is the root of a domain hierarchy.

Example

Figure 3.5 shows a class diagram of how one could structure the classes related to the data mapper pattern. In this example, the `Person` class is a domain object, and the implementation of this will know nothing about the database or its schema. It is the responsibility of the `PersonMapper` class to query the database and create instances of the `Person` object, and to query instances of the `Person` class and store it in the database. Figure 3.6 shows an interaction diagram outlining how this would happen to an update.

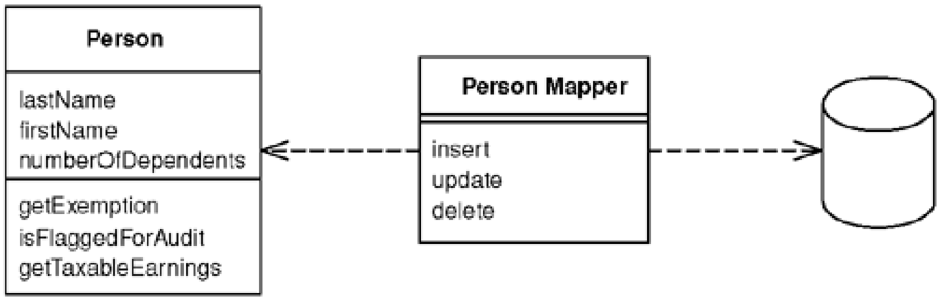


Figure 3.5: A class design that uses the data mapper pattern for the *Person* table.

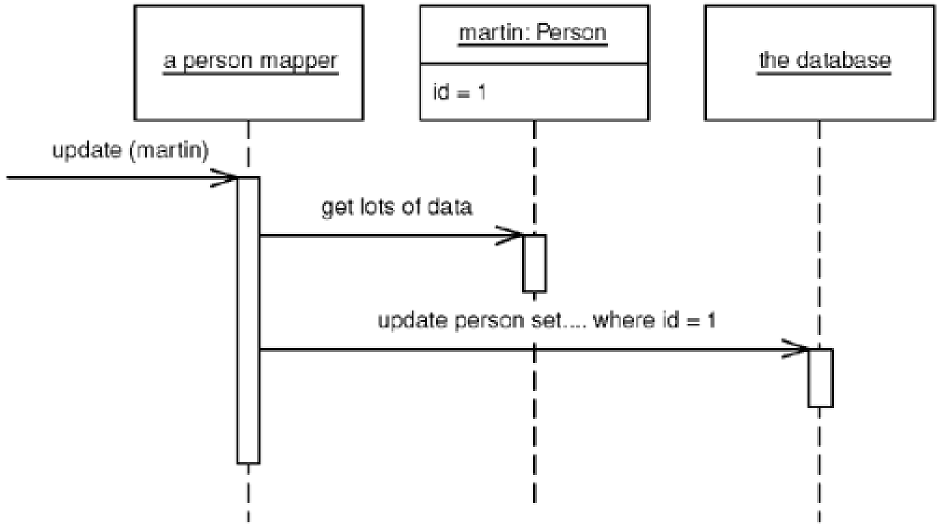


Figure 3.6: An interaction diagram that uses the data mapper pattern for the *Person* table.

The following code snippet show the implementation of static "find" and update methods. The find method queries the database and returns a list of `Person` instances corresponding to people with a specified last name, and the update method takes a `Person` instances and updates the database with the information contained in that object:

```
class PersonMapper {

  public static List<Person> findWithLastName(String lastName) {
    String sql = "SELECT id, lastname, firstname, number_of_dependents " +
      "  from people " +
      "  WHERE lastName = {0}";
    String sqlPrepared = DB.prepare(sql, lastName);
    IDbCommand comm = new OleDbCommand(sqlPrepared, DB.Connection);
    ResultSet rs = comm.executeQuery();
    List<Person> result = new ArrayList();
    while (rs.next()) {
      long rsId = rs.getLong(1);
      String rsLastName = rs.getString(2);
      String rsFirstName = rs.getString(3);
      int rsNumberOfDependents = rs.getInt(4);
      Person person =
        new Person(rsId, rsLastName, rsFirstName, rsNumberOfDependents);
      result.add(person);
    }
    return result;
  }

  public static void update(Person person) {
    String sql =
      "UPDATE people " +
      "  set lastname = {0}, firstname = {1}, number_of_dependents = {2}" +
      "  where id = {3}";
    String sqlPrepared =
      DB.prepare(sql, person.getLastName(), person.getFirstName(),
                    person.getNumberOfDependenents(), person.getID());
    IDbCommand comm = new OleDbCommand(sqlPrepared, DB.Connection);
    comm.executeNonQuery();
  }
  ...
}
```

Note the difference between this and the row data gateway. In the row data gateway, the `PersonGateway` object held the domain object attributes, and was also responsible for converting query results into *PersonGateway* instances. In the data mapper, the `Person` object, which is the domain object, knows nothing about the database.

The `update` method is almost identical to the one in the row data gateway example, except it takes a `Person` instance and reads the data from this.

Pros

1. *Loose coupling*: This pattern de-couples the database access from the domain objects that repre-

sent the information in the database. This further allows the shape of the domain model to differ from the underlying database, as well as allowing the design of the domain layer to proceed independently of the design of the database layer.

2. *Compatibility (domain model)*: This pattern is highly compatible with the domain model pattern from Chapter 2. The data mapper objects create domain model objects from the database. This allows high re-use of both the underlying database and the domain layer.

Cons

1. *Complexity*: An extra layer must be added to act as the mapper. This increases complexity in itself, so is only a worthwhile trade-off if the domain logic is particularly complex, or there is real need to have low coupling between the domain and data layers, such as cases in which there is a high probability that one layer will change in the future.

   Further to this, if the shape of the domain objects and the database differ significantly, the code that performs the mapping (converting query results into domain objects and back again) can be highly complex.

3.3 Choosing between the patterns

- *Table data gateway*: This should be used when there the domain is simple, and fits well with the table module pattern from Chapter 2.
- *Row data gateway*: This should be used when there the domain is simple, but also when design-time type safety is desired, and fits well the transcript script pattern from Chapter 2. It should not be used if there is a high likelihood of the database schema changing.
- *Active record*: This should be used when the domain logic is somewhat complicated, but not complicated enough to warrant the use of the domain model pattern from Chapter 2. It should not be used if there is a high likelihood of the database schema changing.
- *Data mapper*: This should be used when the domain logic is particularly complex, and should be used in conjunction with the domain model pattern from Chapter 2. It should also be used if there is a good chance that either the data source layer is likely to change, or if the domain logic is likely to change without a change to the underlying data. This includes cases where the "change" is re-using one of these layers in another application.

3.4 Further reading

- Much of the material from this chapter is based on Chapters 3 and 10 of the subject text (Fowler [3]), available as an e-book through the University of Melbourne library.

Chapter 4

Object-to-relational behavioural design

In this chapter, we look at principles and patterns for handling the process of translating objects to relational schemas. The relationship between the three patterns discussed in this chapter differ from previous chapters in that they are not alternatives. Each of them solves a different problem to the other, and can be used together if desired.

4.1 The problem

The problem: How do we get objects to efficiently load from and save to a data source while maintaining data integrity?

When people talk about object-to-relational mapping, they tend to think about how to map the domain objects to a database schema. While this is an important part of a design, there are also some behavioural problems that will be encountered in an enterprise system. That is, how to we get the data from the data source and load it into some domain objects, and then do the reverse once we have made some changes. Further, how to we make sure that the resulting data is consistent?

This problem sounds easier than it is. Surely we can just read records from a database, and parse them into domain objects, using active records or a data mapper (see Chapter 3), and then write them all back to the database when we are done?

That works if you load only a handful of records and modify them. However, this has some significant drawbacks:

1. What happens when a large collection of records of loaded? If we modify only a small number of them, we then have to write all of them back to the database, when it would likely be more efficient to only write back those records that have changed. To do this, we need a way of keeping track of which objects have changed.

2. If we read some objects, we must be sure that no other process is going to change those objects while we are working on them. We discuss that more in Chapter 8 – Concurrency, but some of the patterns in this chapter are relevant.

4.2 The patterns

4.2.1 Unit of work

Pattern name	Unit of work
Description	Maintains a list of objects affected by a business transaction and coordinates the writing out of changes and the resolution of concurrency problems.

The *unit of work* pattern describes a way to keep track of which domain objects have changed (or new objects created), so that only those objects that have changed need to be updated in the database.

There are two naive solutions to this problem:

1. The database can be updated with every change to the objects. The problem with this is it can be highly inefficient. First, multiple connections are made for each change, which can be expensive itself. Second, changes in the object may not need to be committed because the object may change again before the transaction completes.

2. Each object can be "tagged" (using a boolean instance variable) indicating that it has been changed. This is better, but it requires us to search all objects to find those that have been changed.

The unit of work is a more elegant solution that keeps four lists of objects:

1. *new*: a list of new objects that have been created and must be inserted into the database;

2. *dirty*: a list of existing objects whose attributes have changed values since they were read from the database;

3. *clean*: a list of existing objects that have not been changed since they were last read from database; and

4. *deleted*: a list of existing objects that need to be removed from the database.

Each object that is created from a query must be in *exactly* one of these lists. In practice, the *clean* list need not be created, as the set of clean objects is just the set of all objects not in the new, dirty, or deleted lists. A typical interface for a unit of work class is shown in

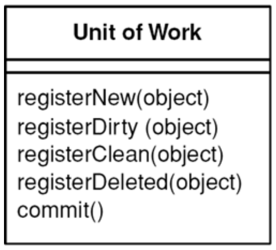


Figure 4.1: An interface for a typical unit of work class.

When the calling code decides to commit the objects back to the database by calling the `commit` operation, the unit of work commits only those objects in the new, dirty, or deleted lists.

Of course, a unit of work is not magic. It must be told which objects to keep track of in its lists. How to do this is a decision that must be made. There are two different ways to do it:

1. *Caller registration*: When some calling code changes the attributes of an object, the calling code is responsible for registering the object with the unit of work.

2. *Object registration*: When some calling code changes the attributes of an object, the object is responsible for registering itself with the unit of work. That is, setter methods register the object as dirty, constructors register it as new, etc.

Both of these suffer from the *forgetfulness problem*: where a developer forgets to write the code to register the object. However, it is clear that object registration will suffer from this far less, and is significantly more straightforward to verify using reviews.

The downside of object registration is that every object has to know how to get the unit of work object. Rather than pass the object around, it is much cleaner to use a *registry*, which is a class containing a static method that returns the unit of work based on its session information, or something similar. The concern is to make sure that only one process/thread can access its unit of work object at a time. The upside of object registration is that it opens up the possibility of automatically generating registration code, but this is not always straightforward.

Example

The following is a complete unit of work implementation:

```java
class UnitOfWork {
  private static ThreadLocal current = new ThreadLocal();

  private List<DomainObject> newObjects = new ArrayList<DomainObject>();
  private List<DomainObject> dirtyObjects = new ArrayList<DomainObject>();
  private List<DomainObject> deletedObjects = new ArrayList<DomainObject>();

  public static void newCurrent() {
    setCurrent(new UnitOfWork());
  }

  public static void setCurrent(UnitOfWork uow) {
    current.set(uow);
  }

  public static UnitOfWork getCurrent() {
    return (UnitOfWork) current.get();
  }

  public void registerNew(DomainObject obj) {
    Assert.notNull(obj.getId(), "id is null", );
    Assert.isTrue(!dirtyObjects.contains(obj), "object is dirty");
    Assert.isTrue(!deletedObjects.contains(obj), "object is deleted");
    Assert.isTrue(!newObjects.contains(obj), "object is new");
    newObjects.add(obj);
  }

  public void registerDirty(DomainObject obj) {
    Assert.notNull(obj.getId(), "id is null");
    Assert.isTrue(!deletedObjects.contains(obj), "object is deleted");
    if (!dirtyObjects.contains(obj) && !newObjects.contains(obj)) {
      dirtyObjects.add(obj);
    }
  }

  public void registerDeleted(DomainObject obj) {
    Assert.notNull(obj.getId(), "id is null");
    if (newObjects.remove(obj)) return;
    dirtyObjects.remove(obj);
    if (!deletedObjects.contains(obj)) {
      deletedObjects.add(obj);
    }
  }

  public void registerClean(DomainObject obj) {
    Assert.notNull(obj.getId(), "id is null");
  }

  public void commit() {
    for (DomainObject obj : newObjects) {
      DataMapper.getMapper(obj.getClass()).insert(obj);
    }
    for (DomainObject obj : dirtyObjects) {
      DataMapper.getMapper(obj.getClass()).update(obj);
    }
    for (DomainObject obj : deletedObjects) {
      DataMapper.getMapper(obj.getClass()).delete(obj);
    }
  }
```

The `java.lang.ThreadLocal` class is used so that each thread in the system has its own `UnitOfWork` instance, to avoid multiple threads accessing the same instance. The first three methods in the class – all static methods — use this to keep track of and return the `UnitOfWork` instance of each thread.

The remainder keeps track of the new, dirty, and deleted objects, and commits them. Note that the `commit` method assumes the use of a data mapper (see Chapter 3) that maintains a list of mapper instances for each object in the domain hierarchy.

Now, we create a class that uses the unit of work. This class maintains properties of music albums, such as title, artist, genre, tracks, etc. If we are using object registration, the code that handles the object registration is in the methods:

```java
class Album {
  private String title;
  private String artist;
  ...

  public Album(String title, String artist) {
    this.title = title;
    this.artist = artist;
    UnitOfWork.getCurrent().registerNew(this);
  }

  public void setTitle(String title) {
    this.title = title;
    UnitOfWork.getCurrent().registerDirty(this);
  }

  public String getTitle() {
    return this.title;
  }
  ...
}
```

The constructor registers the object as new, and `setTitle` registers it as dirty, noting that it has changed. The `getTitle` method does not register it because nothing has changed.

To use this, we show a snippet of a script that edits the title of an album:

```java
class EditAlbumScript {

  public static void updateTitle(Long albumId, String title) {
    UnitOfWork.newCurrent();
    DataMapper mapper = DataMapper.getMapper(Album.class);
    Album album = mapper.find(albumId);
    album.setTitle(title);
    UnitOfWork.getCurrent().commit();
  }
  ...
}
```

Note that, if we had used caller registration instead of object registration, we would have to have registered the `album` object as "dirty" in this script.

Pros

1. *Simplicity*: The unit of work pattern is an elegant and simple way to keep track of objects. An entire implementation of a unit of work class is shown above in under one page. Even for the most trivial applications, using a unit of work is likely to be worthwhile.

   An alternative approach, which we do not examine in detail in these notes, is called a *unit of work controller*. Using a unit of work controller, the controller registers all objects as clean after a database read and takes a copy of these objects. Just before a commit, it compares the two sets of objects an marks those that have changed. This would be more complicated, but has some efficiency advantages because it allows selective update of only those fields that have changed.

2. *Efficiency*: This is quite an efficient approach. Given a large set of objects with only minimal changes, committing to the database will be much faster, while the registration of objects would not take a considerable amount of resources.

3. *High cohesion*: All information regarding what has been changed, added, or deleted is contained in a single place for each thread.

Cons

1. *Forgetfulness*: Whether using caller or object registration, if a developer forgets to register an object, the resulting fault may prove difficult to trace.

4.2.2 Identity map

Pattern name	Identity map
Description	Ensures that each object gets loaded only once by keeping every loaded object in a map. Looks up objects using the map when referring to them.

The *identity map* pattern is a simple pattern that helps to maintain data integrity. An identity map keeps a record of all objects that have been read from the database in *a single business transaction*. Put simply, this pattern aims to prevent the same data from being loaded into more than one object. If this is not prevented, two or more of these objects may be changed in different ways, and then during the next update, there will be a clash of values, and we will not be able to determine the correct values to update the database with.

To solve this, the identity map simply maintains a mapping from the unique ID assigned by the database, to a reference/pointer to the instance of that object. Before an object is read from the database, a lookup is performed on the identity map to see if an object with that ID already exists. If it does, the map returns that instance. Otherwise, the database is queried and a new instance is created.

In addition, if we load the same data more than once, we are incurring unnecessary cost, so an identity map can give increased performance. Note though that this is not the primary purpose of an identity map. If the cost of looking up the objects is *more* than querying the database, we should still use an identity map.

One decision that needs to be made is how many maps are required: one map per class per session, in which each map holds all instances of a single class, or just one map per session, in which one maps holds all instances. The latter only works if the keys in the database are unique database-wide. The former can be difficult when the domain class hierarchy contains inheritance, which implies that a map should keep all instances for a single inheritance hierarchy.

Example

The following is a complete implementation of an identity map:

```java
import java.util.Map;
import java.util.HashMap;

class IdentityMap<E> {

  private Map<Long, E> map = new HashMap<Long, E>();

  private static Map<Class, IdentityMap> singletons =
    new HashMap<Class, IdentityMap>();

  public static <E> IdentityMap<E> getInstance(E e)
  {
    IdentityMap<E> result = singletons.get(e.getClass());
    if (result == null) {
      result = new IdentityMap<E>();
      singletons.put(e.getClass(), result);
    }
    return result;
  }

  public void put(long id, E obj) {
    map.put(id, obj);
  }

  public E get(long id) {
    return map.get(id);
  }
}
```

This class uses a singleton-like pattern [4] to prevent only one instance of each identity map (per class) from being created. We may also implement thread-specific instances of the identity map as we did for the unit of work pattern, but this has been omitted here.

A use of this is then a simple lookup of objects from the map:

```java
Person person = new Person();
IdentityMap<Person> map = IdentityMap.getInstance(person);
person = map.get(id);

if (person == null) {
  //read record from the database
  Person person = new Person();
  Record record = gateway.find(this.id);
  person.setLastName(record.get("last_name"));
  person.setFirstName(record.get("first_name"));
  person.setNumberOfDependents(
    Integer.parseInt(record.get("number_of_dependents"));
  map.put(id, person);
}
```

The creation of the variable `person` on the first line is so we can inform the `getInstance` method which identity map we want. We could just pass the class type using `Person.class`, however, we would not get the type safety for the identity map; that is, the `IdentityMap` class could not use generic types, and we would have to typecast down from `Object` to `Person`.

Pros

1. *Simplicity*: The identity map pattern is an elegant and simple solution to the problem of loading duplicate objects. An entire implementation is shown above in under half a page.

2. *Efficiency*: As well as preventing duplicate objects, the identity map has a side effect of not loading objects more than once, which in many cases increases efficiency.

3. *Prevents the == problem*: If we have one instance per piece of data in our system, then we know that value-equivalent objects must share the same identity. As a result, we can use the == operator to compare objects for value equality, instead of the `equals()` method.

When to use it

There are not really any downsides of using the identity map. It is simple, elegant, and solves the problem it is supposed to.

However, there are cases where the identity map may not be necessary. The most obvious case is when the object being read are immutable. If the value cannot change, then duplicate objects cannot create any data integrity issues. The identity map may still offer efficiency issues, and also allows us to use the == operator to compare for value equality.

A second case where an identity map is not required is if we are using the *dependent mapping* pattern, which we will discuss in Chapter 5. When using the dependent mapping pattern, child objects have their persistence controlled by their parents, so a map is not required.

4.2.3 Lazy load

Pattern name	Lazy load
Description	An object that does not contain all of the data that you need, but knows how to get it.

The unit of work pattern discussed earlier in this chapter aims to reduce the amount of data that is written back to the database by only writing back what has changed. The *lazy load* pattern is the inverse of this: it aims to reduce the amount of data read from database by only reading what it needs.

Essentially, a lazy loader loads some type of dummy objects (there are different ways to structure these) that contain no data, and only loads the corresponding data when we try to access that data. If an object is never accessed, the corresponding data is never read from the database, improving performance significantly when only a fraction of the objects are really needed. This process should be completely transparent to the calling code (except of course the measurable performance difference), and therefore must be encapsulated in the objects themselves.

Implementation

There are four main ways that the lazy load pattern can be implemented:

1. *Lazy initialisation*: This is the most straightforward implementation. Each field in a class is initialised to "null", and each field must contain a getter method that first checks if the field is null, and if so, loads the object. If the field is non-null, it returns the previously-initialised value. A constraint is that all other code, including methods within the class, must use the getter method to access the field.

   One possible problem is if null is a valid value of the field. In this case, another way is required to check if the object has been loaded, such as creating a boolean variable for each field indicating whether it has been loaded, or using a special value other than null.

Due to the tight coupling between the objects and the database, lazy initialisation works best with the table data gateway, row data gateway, or active record patterns.

2. *Virtual proxy*: This is the same pattern as the virtual proxy described by the Gang of Four [4]. A virtual proxy is a dummy object that wraps around the real object. The proxy has the same interface as the real object, so the calling code does not notice it is using a proxy. The corresponding data is only loaded when the real object is needed — that is, when one of the methods is called. At this point, the entire object is read from the database.

   A potential problem with this is that the proxy object and the real object represent the same instance, but do not have the same object identity. As such, we can run into trouble if identities are used.

3. *Value holder*: The value holder approach is the same as the virtual proxy, except that the value holder does not have the same interface as the virtual proxy. That is, there is one value holder class for all domain objects.

   There are two downsides with this. The first is that the using code has to know about value holders. The second is that typecasting may be required to get the values in some languages. In Java 5.0 and above, generics can be used to mitigate this, but many enterprise systems in use to day are older than Java 5.0.

4. *Ghost*: This approach is the same as a lazy initialisation implementation, except that all fields (or multiple fields) are initialised the first time any of the fields are accessed. The advantage of this over the lazy initialisation is that many of the fields will be kept in the same row of the database table, so multiple accesses to that row are not required. The most efficient approach is to follow the structure of the database table and initialise as many fields as possible with a single query; e.g. query a row that contains the required field, and populate all fields corresponding of other columns in that row.

   The downside of this approach is the additional overhead of linking related fields.

Example — lazy initialisation

Lazy initialisation is the most simple of all approaches. If we consider the `Person` class that we have been using in the last few chapters, we can perform lazy initialisation as follows:

```
String getLastName() {
  if (this.lastName == null) {
    //load the name from the database
    Record record = gateway.find(this.id);
    this.lastName = record.get("last_name")
  }

  return this.lastName;
}

String getFirstName() {
  if (this.firstName == null) {
    //load the name from the database
    Record record = gateway.find(this.id);
    this.firstName = record.get("first_name")
  }

  return this.firstName;
}
```

If we were using the ghost implementation, we could also create the first name and number of dependents here, because we have already loaded them from the database and can access them from the *record* object.

Example — ghost

The ghost implementation is similar to the lazy initialisation, except that the entire object is initialised at the same time:

```
  String getLastName() {
    if (this.lastName == null) {
      load();
    }
    return this.lastName;
  }

  String getFirstName() {
    if (this.firstName == null) {
      load();
    }
    return this.firstName;
  }

  void load() {
    Record record = gateway.find(this.id);
    if (this.lastName == null) {
      this.lastName = record.get("last_name");
    }
    if (this.firstName == null) {
      this.firstName = record.get("first_name");
    }
    if (this.numberOfDependents == -1) {
      this.numberOfDependents =
        Integer.parseInt(record.get("number_of_dependents");
    }
  }
```

Note that in the `load()` method we still need to check that an object is non-null. Even though load should only be called once, it may be that a field is set by some calling code before the object is read from the database. That is, the object is created (no fields initialised), one field is set with a value, and then the remainder of the object is initialised. In this case, we do not want to write over the value of the field that has been set, so we do not initialise it.

Example – virtual proxy

A virtual proxy implementation is far different, as there is a second object involve (the proxy). First, there needs to be an interface that both the proxy and the real object can implement:

```java
interface PersonI {
  long getID();
  void setLastName(String lastName);
  String getLastName();
  void setFirstName(String firstName);
  String getFirstName();
  void setNumberOfParticipants(int numberOfParticipants);
  int getNumberOfParticipants();
}
```

Then, a proxy object wraps around a real object, only creating the object is called upon, and delegating all behaviour after that point.

```java
class PersonProxy implements PersonI {
  //the real object
  private PersonI source;

  private PersonI getSource() {
    if (source == null) {
      load();
    }
  }

  public void setLastName(String lastName) {
    getSource().setLastName(lastName);
  }

  public String getLastName() {
    return getSource().getLastName();
  }

  ...
}
```

The implementation of the underlying domain object can be just a standard implementation. To use the proxy, the calling code has to know to create proxy objects (this could be placed in a *factory* [4] to hide it from the business logic, or in the class that communicates with the database), but can use the proxy object as if it is the real object:

```java
  ...

  public void incrementNumberOfDependents(long id, int number) {
    PersonFinder personFinder = new PersonFinder();
    PersonI person = personFinder.find(id);
    int newNumberOfDependents = person.getNumberOfDependents() + number;
    person.setNumberOfDependents(newNumberOfDependents);
  }
```

The type of the `person` object is `PersonI`, we the business logic code does not care that it is using a proxy instead of a `Person` object.

Pros

1. *Efficiency*: The clear advantage of using lazy loading is the efficiency gains it can potentially deliver. This is unsurprising, considering that this is the problem it is designed to solve.

Cons

1. *Complexity*: Clearly, implementing a system using lazy load is going to increase complexity over not using lazy load, so it should only be used in cases in which it is truly needed.
2. *Inheritance*: If the domain hierarchy contains uses inheritance, it can cause confusion over what type of object to create. For example, the fields contained in the database may affect that type of object to create. If we do not load these, then we cannot know what type of object needs to be created if we are using lazy initialisation or ghost (or using generics in the value holder case).
3. *Ripple loading*: A potential problem is with ripple loading. This is where many more database accesses are performed than is required. For example, our calling code may iterate over a collection of objects, which are each created when they are accessed. It may be much faster to query all of these at one time, and create the entire collection. Design trade-offs need to be made to decide whether to load the entire collection at once, or only load an object in the collection if it is accessed.

4.3 Further reading

- Much of the material from this chapter is based on Chapters 3 and 11 of the subject text (Fowler [3]), available as an e-book through the University of Melbourne library.

Chapter 5

Object-to-relational structural design

5.1 The problem

The problem: How do we structurally map our domain objects into a relational database?

At first glance, such a problem sounds easy — just map each class to a database table! Indeed, such an idea is the basis of the principles and patterns discussed in this chapter. However, there are a few fundamental problems that mean this is not as straightforward as it may seem:

1. *References*: Objects typically link together via memory management, using references or pointers. In contrast, relational databases use keys, and referencing a value from one table to another requires the use of a foreign key.

2. *Collections*: Programming languages use collections, such as lists and arrays, to store multi-value fields. For example, a single field in an object can be of type "array", which can hold references to multiple objects. Relational databases do not have this luxury, and force all relations to be single valued.

3. *Inheritance*: Relational databases do not support inheritance, so recording an inheritance hierarchy in a relational database is non-trivial.

The problem of mapping between the domain model and its relational database is a problem that will be encountered on any enterprise system project that uses an object-oriented programming language, and a relationship database. This probably accounts for most enterprise systems being developed today. However, even non-object-oriented languages will still create similar problems in the use of collections to store multi-value fields and pointers to link structures and collections. As such, sound solutions to these problems are important.

5.2 The patterns

5.2.1 Identity field

Pattern name	Identity field
Description	Saves a database ID field in an object to maintain object identity between an in-memory object and a database row.

This is a straightforward yet important pattern. In a relational database, rows are distinguished from each other by their *key*. In programming languages, objects or data structures are distinguished using memory addresses or object identities. When in-memory objects correspond to a row in a database, we need a method for ensuring that the correct objects are written back to to the correct rows.

The *identity field* pattern specifies a method such that, for any object that corresponds to a row, the object explicitly stores the primary key to the corresponding row.

This approach sounds straightforward and obvious. That's because it is. However, the sticky problems in this pattern are how we choose key representations, and how we generate new keys.

To choose keys, we have a number of options, each of which are applied in the database community already. If we are using an existing database, we will be constrained to use their keys. However, if we are designing the database as part of our system, we have to make the choice. There are several choices that must be made:

1. *Meaningful vs. meaningless keys*: Meaningful keys are those that come from the domain, while meaningless keys have no meaning in the context of the domain. For example, a meaningful key in a database may be your email address, while a meaningless key is a number that is never intended for human use. Meaningful keys tend to create problems, such as duplication; e.g. if my wife and I both want accounts with a vendor system, but share an email address. As such meaningful keys should in general be avoided.

2. *Simple keys vs. compound keys*: A simple key corresponds to one database field, while a compound key uses multiple. An example of a compound key is for a line item in an order. If an order contains a unique key, and a list of line items identified as a sequence of numbers, then the key for the line items is the order number and the sequence. Simple keys are often preferred because compound keys tend to be meaningful, such as in the line item example, where the meaning is that the line item is part of a specific order.

3. *Table-unique key vs. database-unique keys*: Keys must be unique in a table, but can also be unique database wide. The advantage of database-wide keys is evident if we consider the identity map pattern in Chapter 4. If we want to use a single identity map for all objects, then we must use a database-unique key. However, table-unique keys are fine and work in most cases.

To generate a new key, there are several different options, and trade-offs must be made depending on the application being built:

1. *Auto-generate*: One option is to use the database management system to automatically generate a key when a new record is inserted. This is efficient for creating keys, however, if the calling code needs to know what the value of the key is, it will have to run a new query. In some cases, you

may need this key *before* you even insert the row. For example, take the idea of an order number with line items from above. To insert a new row in the order table, we need the compound key for each line item to use as the foreign key in the order table. However, this foreign key is based on the order item key, which is the new row to be inserted. As a result, we need a new key before insertion.

   Some database management systems, such as Oracle products, support auto-generating outside of inserts, called a *database counter*. A query can be sent to the database asking for the next *X* number of unique keys.

2. *Globally Unique IDentified (GUID)*: To do this, a single machine is used to generate unique keys, and all code must query this machine to get a new key. Algorithms for this generally use information such as the caller, and the current date & time, to generate a key that will be unique across all time, not just across a single database. The downside is that the keys are generally long and make little sense, which causes problems if a human user needs to type it in.

3. *Table scan*: This is a method in which the key is generated by the calling code, not the database. The SQL function *max* can be used to get the largest key in the table. If we know this, we can increment this value by 1 to get a unique key.

   A downside of this is that the entire database must be read-locked during the call to avoid more than one thread running this at the same time.

4. *Key table*: Another method in which the key is generated by the calling code is the key table method. In this case, a single table in the database records the maximum key distributed so far. For a database-unique key, the key table consists of just a single row with the key. For a table-unique key, the key table must have one row per table. This requires only the key table to be read-locked.

Example

This example uses a key table to keep track of table-unique keys, which are implemented as simple integers:

```java
import java.util.*;

class KeyTable {

  public int getKey(String tableName) {
    //get the next key from the table
    String query = "SELECT nextID FROM keys WHERE name = {0} FOR UPDATE";
    String queryPrepared = DB.prepare(query, tableName);
    IDbCommand comm = new OleDbCommand(queryPrepared, DB.Connection);
    ResultSet rs = comm.executeReader();
    Record record = rs.get(0);
    int result = record.getLong(1);

    //update the table with the next key
    int nextKey = result + 1;
    String update = "UPDATE keys SET nextID = {0} WHERE name = {1}";
    String updatePrepared = DB.prepare(update, nextKey, tableName);
    comm = new OleDbCommand(queryPrepared, DB.Connection);
    comm.executeNonQuery();

    return result;
  }
}
```

Each time a new object is created that requires a new key, the creating code can call `getKey` to get a unique key.

In the above implementation, only one key can be created at a time, however, it would be straightforward to update this implementation to generate a sequence of keys.

Pros

1. *Simplicity*: The general idea is extremely simple, even if the implementation requires some thought.

Cons

1. *Keys*: As we saw already, creating keys for new data can be a non-trivial problem.
2. *Coupling*: Linking a domain object to its table key implies a higher coupling between the domain layer and the database. However, this is necessary in many systems.

Usage

The identify field pattern is not required when using the table module, transaction script, or table data gateway patterns from earlier chapters. This is because generic records are used, and the keys will be part of these records. Identity fields are only required when data is held in domain objects.

5.2.2 Foreign key mapping

Pattern name	Foreign key mapping
Description	Maps an association between objects to a foreign key reference between tables.

The identity field pattern describes how to link domain objects to their respective rows in database tables. However, objects contain references to other objects in the domain layer, and often it is necessary for these references to persist in the database. Simply saving the object identities is not sufficient, because these identities are lost between sessions. A further complication is that objects can refer to *collections* of other objects, while databases can only reference to single values.

The *foreign key mapping* pattern aims to solve this problem by using the *identity field* pattern described in Section 5.2.1 to map objects to rows, and to extend this concepts to *foreign keys* as well. If an object, o_1 refers to another object o_2, and this association should persist in the database, then in the database schema, map the association using a foreign key. In other words, the row corresponding to object o_1 should have a foreign key field that refers to the corresponding row for object o_2.

A more complicated case is when the object refers to a collection of objects. In this case, we cannot refer to a list of foreign keys, so instead, we reverse the link of the association in the database. That is, if object o refers to a collection of objects $o_1, ..., o_n$, then the rows corresponding to objects $o_1, ..., o_n$ should contain a foreign key field that refers to the row corresponding to object o.

Example

We use an example of a small system in which we are keeping track of albums, the artist that produced them, and the tracks on them, etc. We have three classes in our domain model: `Album`, `Artist`, and `Track`. We store the data about these objects in tables with the same names (*Album*, *Artist*, and *Track*) and maintain the links using identity field and foreign key mapping.

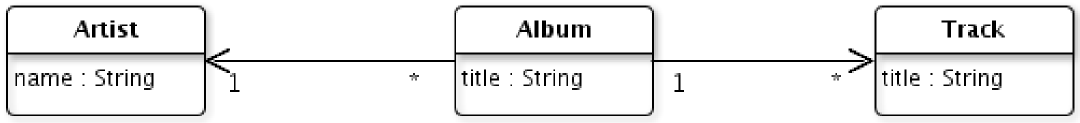


Figure 5.1: An excerpt from a domain model.

Figure 5.1 shows an excerpt of domain model for this system. The association from album to artist is implemented as single-valued reference, while the association from album to tracks is implemented as multi-valued reference. The implementation of this is below.

```
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;
  private Track [] tracks;

  public Album(long ID, String title, Artist artist, Track [] trackss) {
    super(ID);
    this.title = title;
    this.artist = artist;
    this.tracks = tracks;
  }
  public String getTitle() {
    return title;
  }
  public void setTitle(String title) {
    this.title = title;
  }
  //  ...
}
```

```
class Track {
  private String title;

  public Track(long ID, String title) {
    super(ID);
    this.title = title;
  }
  public String getTitle() {
    return title;
  }
  public void setTitle(String title) {
    this.title = title;
  }
}
```

The mapping to the table uses the identity field and foreign key mapping patterns. As a result, the *Albums* table has a foreign key referencing the artist ID in the *Artists* table, while reverse is true for the relationship between *Albums* and *Tracks*.

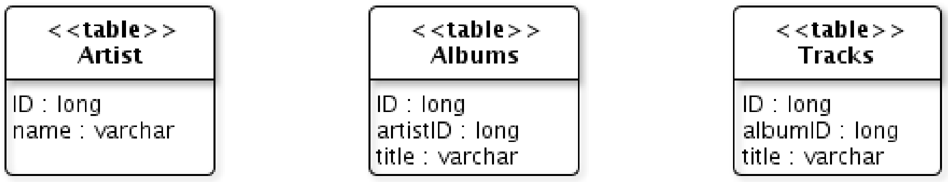


Figure 5.2: The table definitions corresponding to the domain model from Figure 5.1.

The `AlbumMapper` class, implemented using the data mapper pattern (Chapter 3), has a `find` method, which queries the *Album* table to get the album name and the artist ID (the foreign key value), and then uses the `ArtistMapper` class to get the `Artist` object. Finally, it uses the `TrackMapper` class to get the `Track` objects:

```
class AlbumMapper {

  public Album find(long id) {
    String sql = "SELECT ID, artistID, title " +
      "  from albums " +
      "  WHERE ID = {0}";
    String sqlPrepared = DB.prepare(sql, id);
    IDbCommand comm = new OleDbCommand(sqlPrepared, DB.Connection);
    ResultSet rs = comm.executeQuery();
    rs.next();

    //get the artist information
    long artistID = rs.getLong(1);
    ArtistMapper artistMapper = new ArtistMapper();
    Artist artist = artistMapper.find(artistID);

    //get the track information
    TrackMapper trackMapper = new TrackMapper();
    Track [] tracks = trackMapper.findForAlbum(id);
    Album result = new Album(id, title, artist, tracks);
    return result;
  }
}
```

The `TrackMapper` class is implemented as:

```
class TrackMapper {
  public Track [] findForAlbum(long albumId) {
    String sql = "SELECT ID, title " +
      "  from tracks " +
      "  WHERE albumID = {0}";
    String sqlPrepared = DB.prepare(sql, albumID);
    IDbCommand comm = new OleDbCommand(sqlPrepared, DB.Connection);
    ResultSet rs = comm.executeQuery();

    Track [] result = new Track[rs.size()];
    for (int i = 0; i < rs.size(); i++) {
      rs.next();
      result[i] = new Track(rs.getLong(0), rs.getString(1));
    }
    return result;
  }
}
```

Note the difference in `AlbumMapper` between getting the artist information and the track information. In the former case, the artist ID is read from the *Album* table, and then looked up, where as in the latter case, we do not have the track ID, so need to look it up based on the album ID using `findForAlbum(long albumID)`. This is because the track information is a collection, so the foreign key is kept in the *Track* table instead of the *Album* table.

While querying each table in isolation is a conceptually clean idea, issuing three calls to the database is inefficient compared to the case where we issue one query that obtains all information. First, because we have to issue three remote calls instead of one, and second, because database management systems are implemented to make such querying highly efficient, whereas our code does it the "grind" way.

As a result, we can issue fewer queries. For example, we could join the first two queries (to the *Album* and *Artist* tables) into one:

```
class AlbumMapper {

  public Album find(long id) {
    String sql = "SELECT a.ID, a.artistID, a.title, r.name " +
      "  from albums a, artists r " +
      "  WHERE ID = {0} and a.artistID = r.ID";
    String sqlPrepared = DB.prepare(sql, id);
    IDbCommand comm = new OleDbCommand(sqlPrepared, DB.Connection);
    ResultSet rs = comm.executeQuery();
    rs.next();

    //get the artist information
    long artistID = rs.getLong(1);
    String artistName = rs.getString(3);
    Artist artist = new Artist(artistID, artistName);

    //get the track information
    ...
  }
}
```

Joins can themselves be inefficient, especially if not implemented carefully. As a result, the choice of using a join or issuing multiple queries will depend on the properties of the system, and early profiling may help to decide which one to use.

We could extend the query above so that it also read the track information from the *Track* table where the album IDs were equal. This would require some careful implementation to keep track of album IDs to ensure that the correct tracks were added to the correct albums.

Pros

1. *Simplicity*: The mapping between the domain relationships and database tables can be done mechanically.

Cons

1. *Coupling*: As with identity field, using the foreign key mapping pattern couples the domain logic layer with the data-source laying due to their sharing of keys.

2. *Many-to-many associations*: This pattern does not support cases in which a collection of objects refers to a collection of objects; or *many-to-many* associations in domain modelling. Foreign keys must be single values, and databases in first normal form do not support multiple values in a single field.

3. *Updates*: A complication occurs when a collection is updated; e.g. a track is added to an album. We have to keep track of which tracks need to be written back to the database. Using a unit of work (Chapter 4) can help here, but other solutions may be just to remove all tracks from the database and re-insert them again, or doing of “diff” on before and after states.

5.2.3 Association table mapping

Pattern name	Association table mapping
Description	Saves an association as a table with foreign keys to the tables that are linked by the association.

One of the drawbacks with the foreign key pattern is that it cannot handle many-to-many associations. For a one-to-many mapping, we can use a foreign key for the single-valued end of the association, however, for a many-to-many relationship there is no single-valued end, so this will not suffice.

The *association table mapping* pattern uses a strategy from relational database design: create a new table to represent the mapping. The table contains pairs of foreign keys, in which a pair represents the individual relationships that must persist. The table has no corresponding in-memory object — the relationship in memory is recorded using references or pointers. Like the foreign key mapping pattern, the association table mapping pattern relies on the identity field pattern.

Example

Consider the domain model in Figure 5.3, which extends the domain model in Figure 5.1 by adding a many-to-many relationship between an artist and the instruments that the artist plays.

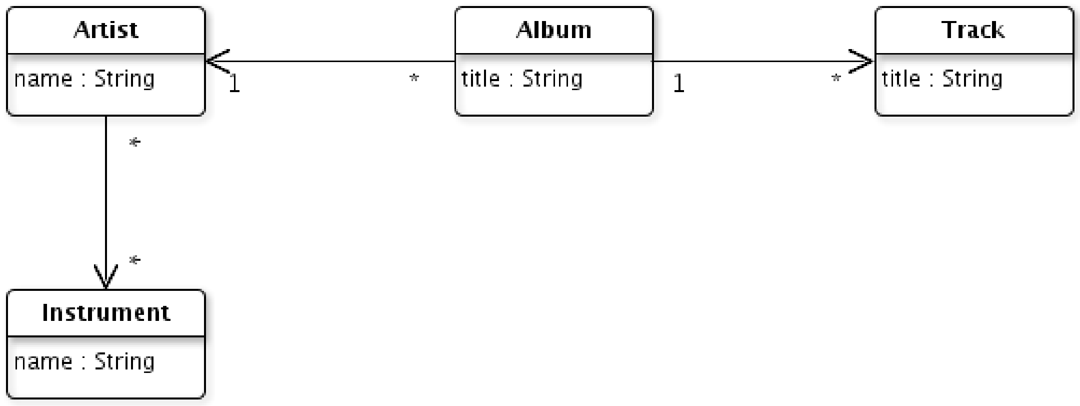


Figure 5.3: An extension of the domain model from Figure 5.1.

Mapping this using the association table mapping pattern will result in the table design shown in Figure 5.4. A new table *Artist-Instruments* records the relationship, and consists only of pairs of foreign keys.

To load the collection of instruments that an artist is able to play, the `ArtistMapper` class is extended to issue a query to the *Artist-Instrument* table:

```
class ArtistMapper {

  public Artist find(long id) {
    String sql = "SELECT ID, name " +
      "  from artists " +
      "  WHERE ID = {0}";
```

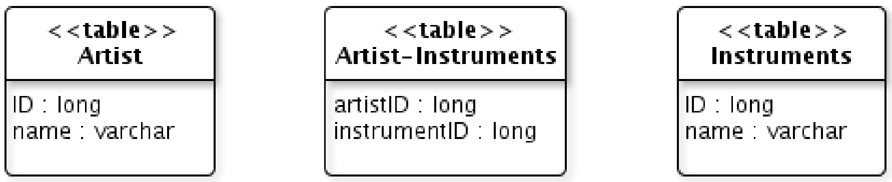


Figure 5.4: The table definitions corresponding to the domain model from Figure 5.3, with the *Album* and *Track* tables omitted.

```
    String sqlPrepared = DB.prepare(sql, id);
    IDbCommand comm = new OleDbCommand(sqlPrepared, DB.Connection);
    ResultSet rs = comm.executeQuery();
    rs.next();

    //get the name
    String name = rs.getString(1);

    //get the instruments information
    Instrument [] instruments = loadInstruments(id);

    Artist result = new Artist(id, name, instruments);
    return result;
  }

  public Instrument [] loadInstruments(long artistID) {
    String sql = "SELECT artistID, instrumentID " +
      "  from artist-instruments " +
      "  WHERE artistID = {0}";
    String sqlPrepared = DB.prepare(sql, id);
    IDbCommand comm = new OleDbCommand(sqlPrepared, DB.Connection);
    ResultSet rs = comm.executeQuery();

    //load the instrument details using an InstrumentMapper
    Instrument [] result = new Instrument[rs.size()];
    InstrumentMapper instrumentMapper = new InstrumentMapper();
    for (int i = 0; i < rs.size(); i++) {
      rs.next();
      result[i] = instrumentMapper.find(rs.getLong(2));
    }
    return result;
  }
}
```

Again, we note that this could be performed in a single query to improve efficiency.

Pros

1. *Simplicity*: Despite some sticky issues with implementing queries, the mapping from the domain relationships and the database tables can be done mechanically.

2. *Genericity*: The association table mapping pattern can be used to record one-to-many relationships, so can be used instead of the foreign key mapping.

Cons

1. *Complexity*: Yes, we've categorised this pattern as both simple and complex! It is more complex than the identity field and foreign key mapping patterns. This is noted because, while we note above that it is more generic that the foreign key mapping, the mapping is slightly more complicated, and adds an unnecessary table.

5.2.4 Embedded value (dependent mapping)

Pattern name	Embedded value
Description	Maps an object into several fields of another object's table.

In many domain models, it makes sense to have small classes/objects for recording information that do not make sense to represent in a database. For example, we may have a class called `Money` that records the cost of a purchase in a particular currency. However, it does not make sense to have a corresponding table for this class just to record purchase amounts.

An *embedded value* maps the values of an object into the fields of its owner. When the owning object is loaded/saved, the corresponding embedded values are loaded/saved as well. A pattern that is closely related to this is the *dependent mapping* pattern, discussed by Fowler [3], but which we do not discuss further.

Implementation

The concept of embedded values is simple, but deciding which objects to implement as embedded values can be less straightforward. In the example of the purchase amount above, it is clear that a *Money* table is unnecessary, but other cases such as a purchase order and its shipping details may not be so clear.

Usually the deciding factor is the relationship between the objects for loading and saving. If both objects are always loaded and saved at the same time, then it makes sense to save them in one table. On the other hand, if the shipping details are queried independently in some cases, then using separate tables with foreign keys is likely to be a better solution.

Generally, embedded values should only be used when the relationship is a one-to-one mapping.

Example

Consider the domain model implementation shown in Figure 5.5, which records information about employees, when their employment started and ended, and what their current salary is. A naive mapping of this, which uses the foreign key mapping pattern, is shown in Figure 5.6. This table design records information on date ranges and salary information.

However, such a mapping results in the two tables *DateRanges* and *Salary*, which record information that typically only belongs to one person, and are highly unlikely to be queried on their own. For example,

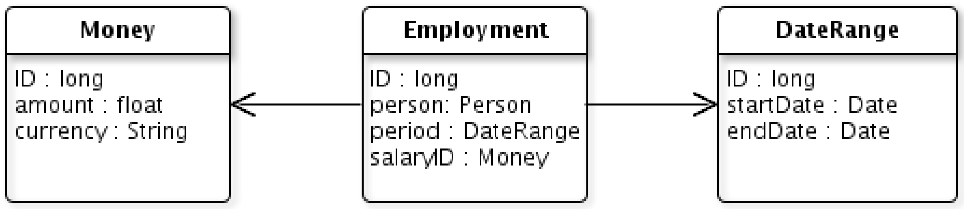


Figure 5.5: A domain model implementation that records employee information.

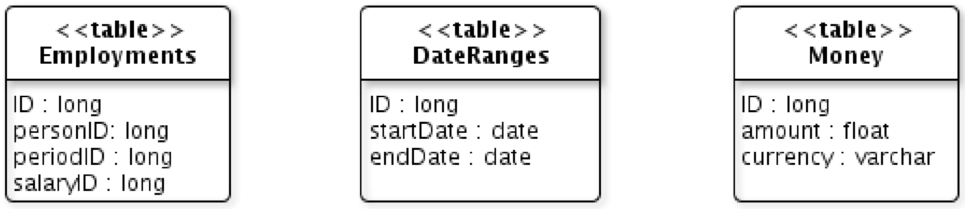


Figure 5.6: A naive mapping between the domain model and database.

below is a small instantiation of such a table. Note that the *Employments* table consists of rows that are entirely populated by keys.

Employments			
ID	**Person**	**Period**	**Salary**
123	1	1	1
456	2	2	2
789	3	3	3
	...		

DateRanges		
ID	**Start**	**End**
1	31/08/2000	31/12/2009
2	14/01/2008	-
3	01/01/2008	-
	...	

Money		
ID	**Start**	**End**
1	120,000	AUD
2	102,000	AUD
3	60,000	EUR
	...	

Using the embedded value pattern will group all of the three above objects into a single table, as shown in Figure 5.7. Here, the start and end date, and the amount and currency type are all recorded in a single table.

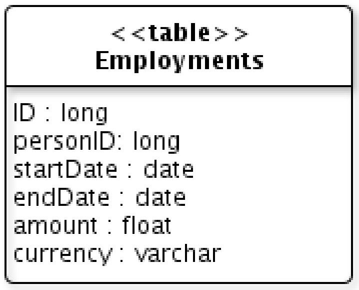


Figure 5.7: An embedded value mapping between the domain model and database.

The code snippet below shows how the objects are constructed from a single record:

```
class EmploymentMapping {

  public Employment find(long id) {
    String sql = "SELECT * from Employments WHERE id = {0}";
    String sqlPrepared = DB.prepare(sql, id);
    IDbCommand comm = new OleDbCommand(sqlPrepared, DB.Connection);
    ResultSet rs = comm.executeQuery();
    Record record = rs.get(0);

    //lookup the information from the Person table
    long personID = rs.getFloat(2);
    Person person = personMapper.find(personID);

    //create the data range and money objects
    Date startDate = rs.getDate(3);
    Date endDate = rs.getDate(4);
    DateRange dateRange = new DateRange(startDate, endState);

    float amount = rs.getFloat(5);
    String currency = rs.getString(6);
    Money money = new Money(amount, currency);

    //create the Employment instance
    Employment result = new Employment(id, person, dateRange, money);
    return result;
  }

  ...
}
```

Pros

1. *Neatness*: Using the embedded value pattern results in a much neater relational database schema, which is closer to the way it would be designed independent of the object-oriented domain model.

Cons

1. *Applicability*: Deciding when this should be applied is not always straightforward.
2. *Complexity*: An embedded value is only straightforward to apply for simple dependencies in a domain model. It is unsuitable for more complex relationships.

5.2.5 Single table inheritance

Pattern name	Single table inheritance
Description	Represents an inheritance hierarchy of classes as a single table that has columns for all fields of the various classes.

The next three patterns all deal with the structural mapping of inheritance hierarchies to relational models. Relational databases do not support inheritance, yet we need to determine how to structure the relational database such that our inheritance hierarchy is preserved when objects are re-loaded.

The *single table inheritance* pattern maps all fields/attributes of all classes in the hierarchy into a single table, recording the type of each object, and leaving null/empty all the fields in row that are not in the corresponding object. When loading a row into an in-memory object, the type field in the database is read first to determine which type of object to create, and the relevant fields are extracted from the row to create the object. All instances are saved in the one table.

Example

Figure 5.8 shows the single table mapping from an inheritance hierarchy that describes players in sports[1]. The abstract class `Player` records the name, while the lower-level classes record attributes that are specific to certain sports.

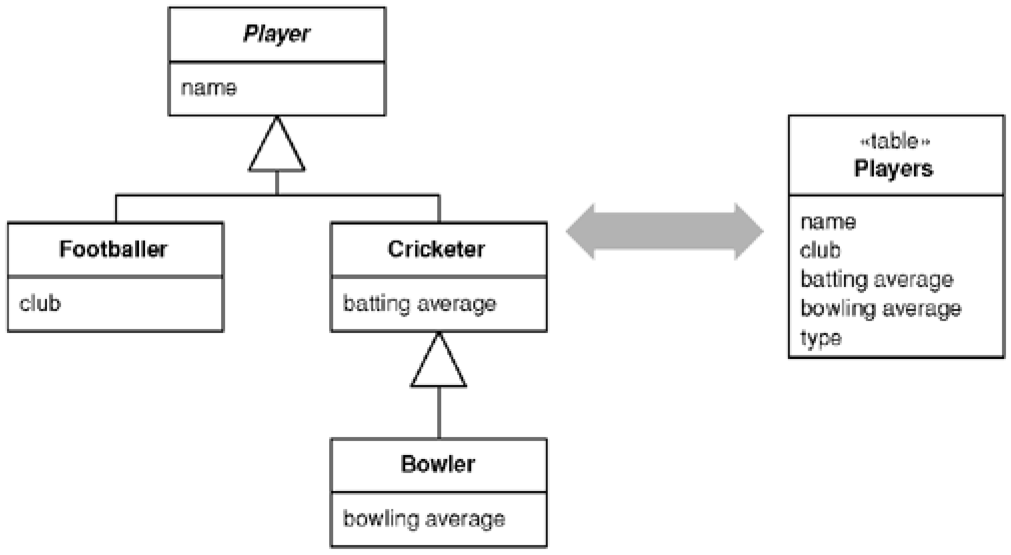


Figure 5.8: An single table inheritance mapping between the domain model and database.

The corresponding table records all fields from all player types, which implies that the rows will have fields for information that is not relevant. For example, the field for "batting average" will be null for a footballer.

Figure 5.9 shows how to structure the mapper layers for the player example. This design uses a pattern called *inheritance mappers* [3], which we will not present any further in these notes. This is an elegant and extensible design for all three inheritance-related patterns that we will discuss.

The `Mapper` class implements some generic behaviour to find rows and convert them into objects:

```
class Mapper {

  protected DomainObject abstractFind(long id) {
```

[1]In cricket, all players must bat, but only some players will bowl. Therefore, all players have a batting average, but not all have a bowling average.

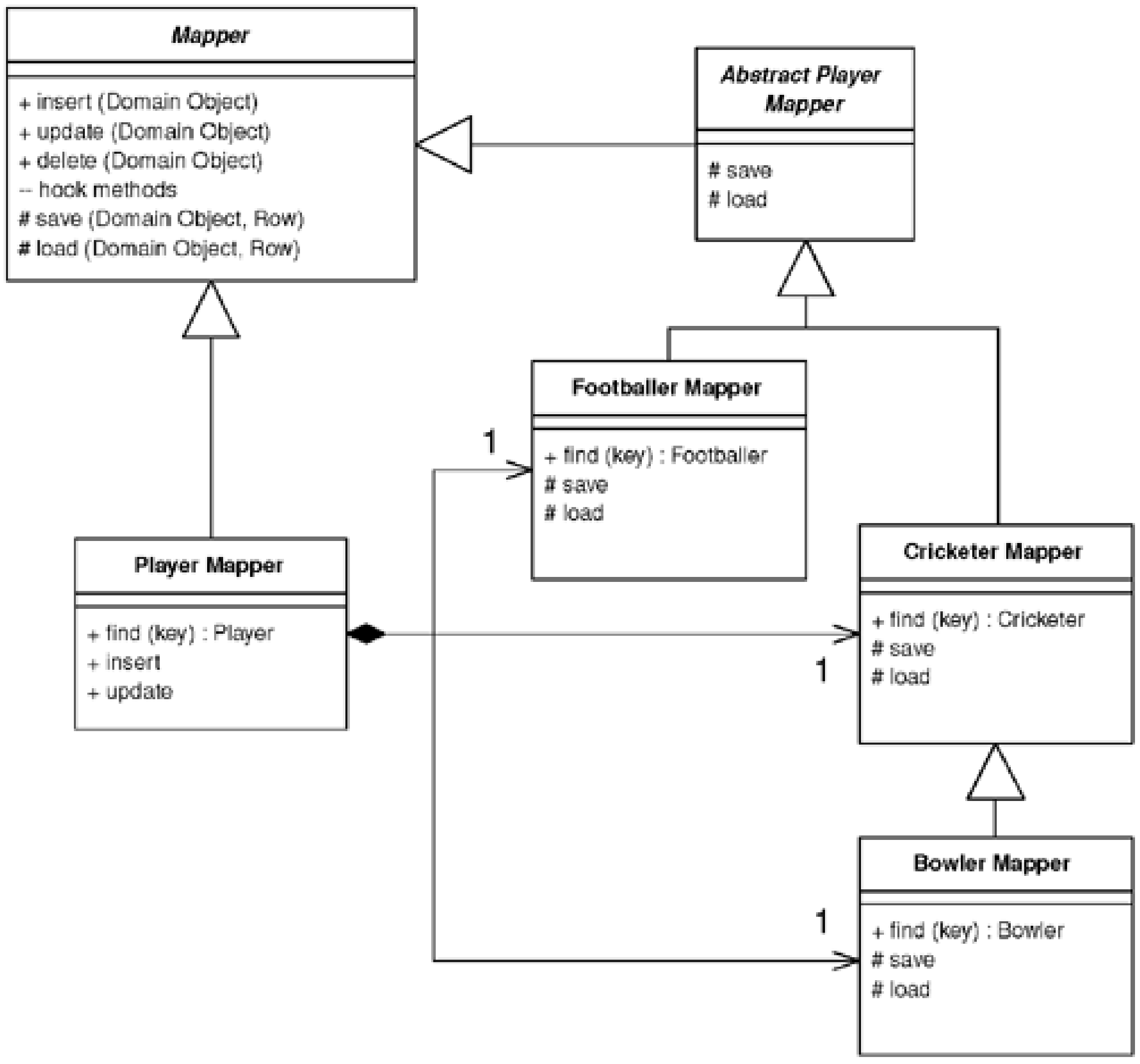


Figure 5.9: A class diagram for mappers using table inheritance, using the *inheritance mappers* pattern from Fowler [3].

```
    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;
  }

```

```java
  protected void load(DomainObject obj, DataRow row) {
    obj.setId("id", row.getInt("id"));
  }

  abstract protected DomainObject createDomainObject();
}
```

The `AbstractPlayerMapper` class implements loading related to the fields in the `Player` class, and the subclasses implement the behaviour related to them specifically. For example, the `CricketMapper` class implements the loading of the batting average, and delegates the more-common behaviour to its superclass (`AbstractPlayerMapper`):

```java
class AbstractPlayerMapper extends Mapper {
  @Override
  protected void load(DomainObject obj, DataRow row) {
    super.load(obj, row);
    Player player = (Player) obj;
    player.name = row.getString("name");
  }
  ...
}

class CricketerMapper {
  @Override
  protected void load(DomainObject obj, DataRow row) {
    super.load(obj, row);
    Cricketer cricketer = (Cricketer) obj;
    cricketer.setBattingAverage(row.getLong("battingAverage"));
  }
  ...
}
```

Finally, the `PlayerMapper` class, which is the interface used to query the database itself, must check the type of the object and delegate to the correct mapper:

```java
class PlayerMapper extends Mapper {

  //the sub-class mappers
  private BowlerMapper bmapper;
  private CricketerMapper cmapper;
  private FootballerMapper fmapper;

  public PlayerMapper (Gateway gateway) {
    bmapper = new BowlerMapper(gateway);
    cmapper = new CricketerMapper(gateway);
    fmapper = new FootballerMapper(gateway);
  }

  public Player find (long key) {
    DataRow row = findRow(key);
    if (row == null) return null;
    else {
      String typecode = row.getString("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");
      }
    }
  }

  ...
}
```

Inserting and updating a row is similar: each class implements its own `save` method that reads the attributes of the object that are relevant to it, and delegates the remainder to its superclass/subclasses.

Pros

1. *Simplicity*: There is only one table to hold the entire inheritance hierarchy.
2. *Stability*: Refactoring the design to move fields around the hierarchy does not require a change in the table, only the corresponding code.
3. *No joins*: Neither table joins nor multiple database queries are not required to retrieve an instance.

Cons

1. *Table design*: Some fields in the table will be left null because they do not correspond to a field of an in-memory object (e.g. the column "batting average" for a footballer). This can result in confusion for those maintaining the database.
2. *Space complexity*: The additional fields may also take up space in memory if the database is not designed carefully (e.g. pushing all optional columns to the right of tables), or if the database management system does not deal with compression well.
3. *Locking*: For a large inheritance hierarchy, pushing all information into one table may result in frequent locking of the table, therefore affecting performance.

5.2.6 Class table inheritance

Pattern name	Class table inheritance
Description	Represents an inheritance hierarchy of classes with one table for each class.

The *class table inheritance* pattern divides up field information over several tables, with each class mapping directly to a single table in the database. Fields from each class are stored in the respective table, which implies that instances of objects are recorded over multiple tables, rather than in a single table.

Implementation

One implementation concern is how to record that rows from different tables relate to the same instance. The most straightforward way to record the sharing of rows is to use unique primary keys across all tables. A second way is to use foreign keys and place the information into the superclass table.

A second concern is how to query and update the table. Using joins can be inefficient for more than a handful of tables, especially when we only want to retrieve a single row. Updating rows is even more of a problem, because if we want to update a set of instances of different types, we will not even know which tables to include. When some tables have no data, an outer join will be required, but this is inefficient and also non-standard. The alternative is to issue multiple queries – also inefficient. The choice depends on the properties of the system itself.

Example

Figure 5.10 shows the class table mapping from the `Player` inheritance hierarchy. Note in this mapping, the fields for classes (e.g. `Bowler`) are spread over multiple tables.

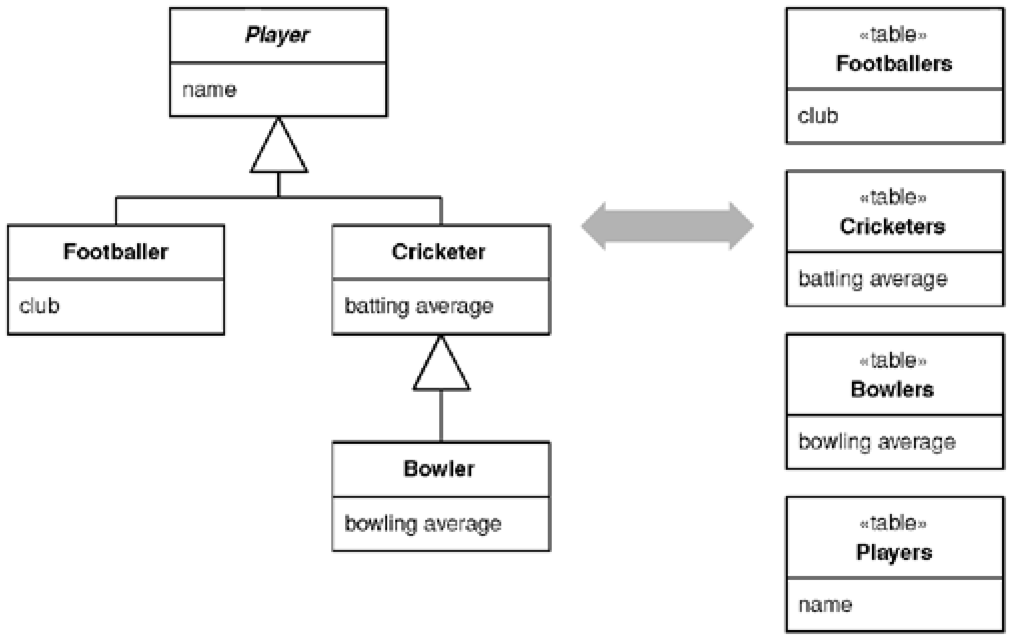


Figure 5.10: An class table inheritance mapping between the domain model and database.

One implementation of this is to use the inheritance mapper pattern again, as shown in Figure 5.9, and to issue queries for each of the classes, rather than using a table join query in SQL. In this case, the design is the same as in Figure 5.9, however, the load, save, update, and delete methods change. Instead of receiving the row as an argument to the method, they instead use the object ID to query the the corresponding table:

```java
class AbstractPlayerMapper extends Mapper {
  @Override
  protected void load(DomainObject obj) {
    DataRow row = findRow(obj.id, "Players")
    Player player = (Player) obj;
    player.name = row.getString("name");
  }
  ...
}

class CricketerMapper {
  @Override
  protected void load(DomainObject obj) {
    super.load(obj);
    DataRow row = findRow(obj.id, "Crickets");
    Cricketer cricketer = (Cricketer) obj;
    cricketer.setBattingAverage(row.getLong("battingAverage"));
  }
  ...
}
```

Note that the `findRow` method now also requires us to pass the name of the table that we are querying. The `Mapper` and `PlayerMapper` classes remain the same as for the single table inheritance pattern, except that they are also required to specify which table is queried.

Pros

1. *Table design*: All columns in all tables are relevant, so there are no issues with confusion or with wasted space.
2. *Simplicity*: The relationship between the domain model and the tables is straightforward: one table per class in the hierarchy.

Cons

1. *Joins*: To load an object into memory, either table joins or multiple queries are required, which can be inefficient in terms of both memory and processor time.
2. *Refactoring*: Moving a field from one class to another in the inheritance hierarchy requires a change to the database as well as the code.
3. *Superclass table bottlenecks*: The superclass tables may become a bottle neck because they are used in each query.

5.2.7 Concrete table inheritance

Pattern name	Single table inheritance
Description	Represents an inheritance hierarchy of classes with one table per concrete class in the hierarchy.

The final inheritance-related pattern is the *concrete table inheritance* pattern, which maps only *concrete* classes to tables in the database. Each table contains columns for all corresponding fields from the mapped classes, including their superclasses.

Note that the mapping is between tables and *concrete* classes — not tables and *leaf node* classes. Classes that are not leaf nodes but are still concrete (such as the `Cricketer` class) still require a table.

Implementation

The handling of keys is crucial to this pattern, because retrieving an instance based on its ID, with no type information, requires us to be able to uniquely identify which row it is in over multiple tables. The primary keys of each table must therefore be unique over the entire hierarchy, so using the functionality provided by th database management system to automatically assign keys is not an option.

Example

Figure 5.11 shows the concrete table mapping from the `Player` inheritance hierarchy. Note that, in this mapping, the tables hold all field relevant to their instances.

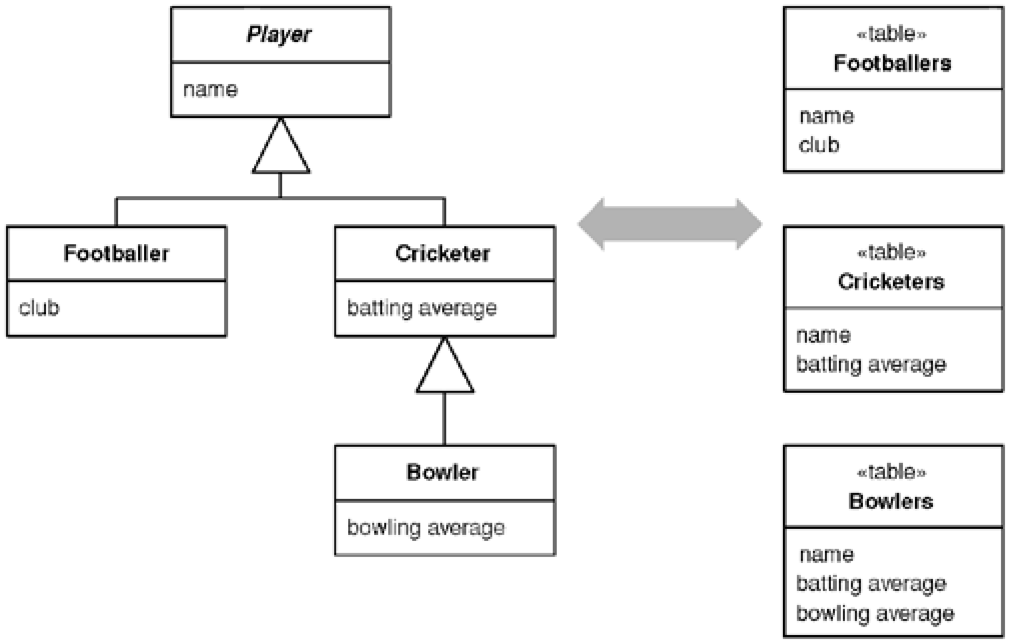


Figure 5.11: An concrete table inheritance mapping between the domain model and database.

Implementing this pattern is much the same as the previous two. We can again the inheritance mapper pattern (Figure 5.9) delegating the behaviour to the class in which the respective fields are defined. The main difference between the implementation for the concrete table inheritance pattern and the class table inheritance pattern is in the `PlayerMapper.find` method. This can be implemented as follows:

```
class PlayerMapper extends Mapper {
  ...

  public Player find (long key) {
    Player result = null;
    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 result;
  }

  ...
}
```

This is highly inefficient if the objects are not already stored in memory and cached by the mapper classes, because in the worst case, we have to issue three queries to the database. In the case that most objects are not already stored in memory, it may be more efficient to instead to a join of all concrete tables and search the joined table. This can be inefficient itself, so the choice is specific to the system, and doing some benchmarking with large data sets is important.

Pros

1. *Simplicity*: The mapping from the domain model to the tables is straightforward: one table per concrete class.
2. *Table design*: All columns in all tables are relevant, so there are no issues with confusion or with wasted space.
3. *No joins*: Each instance is recorded in a single table, so there is no need for joins or multiple queries just to load/save an object (although this may be required to load/save a *collection* of objects of different types, or find an instance if we do not know the type).
4. *Load spread*: There is no single bottleneck for the database, as only a single table is accessed to load/save an object.

Cons

1. *Keys*: Primary keys must be unique across all tables in the hierarchy, which can be difficult to handle.
2. *Refactoring*: Moving a field from one class to another in the inheritance hierarchy requires a change to the database as well as the code.
3. *Maintenance*: If a field from one of the superclasses changes (e.g. the name or type), the corresponding change must be made to *all* subclasses of that class.
4. *Global finds*: To query an instance whose type is not known, all tables must be checked (at least, until the relevant instance is found), requiring a large join or multiple queries.

5.2.8 Discussion about inheritance

As with many of the other patterns, choosing which of the inheritance patterns to use requires trade-offs to be made. Assessing the pros and cons of each with respect to the system being designed will be required, and running some early benchmarking tests will also help.

However, it is important to note that the three options are not mutually exclusive for a system – only for a single inheritance hierarchy. For example, if we have multiple hierarchies in our domain model, we can use single class inheritance for one, and concrete class inheritance for others.

The patterns discussed all assume single inheritance. Most designers tend to avoid multiple inheritance these days, so it is generally not such an issue. However, the issue cannot always be ignored. Interfaces are used in some languages to mimic something that is close to multiple inheritance. Further, a domain model with multiple inheritance may be re-used from an existing system. In these situations, variations of the three patterns presented in this section are required.

For the single table version, all classes and interfaces are assembled into a single table. For the class table version, separate tables are made for each interface and superclass. For the concrete table version, all implemented interfaces and all superclasses are included in each concrete table.

5.3 Further reading

- Much of the material from this chapter is based on Chapters 3 and 12 of the subject text (Fowler [3]), available as an e-book through the University of Melbourne library.

Chapter 6

Presentation layer

As discussed in the introductory chapter, the high level logical architecture of an enterprise system can be viewed as a layered style architecture, consisting three layers: presentation, domain logic, and data source layers.

In this chapter, we will look at the design patterns and principles for the design of the *presentation layer*. These patterns provide designers with ways to structure the presentation layer of an enterprise system in order to meet design goals.

The presentation layer of the application deals with the presentation and handling of the user input from the user interface (normally a web-based graphical user interface in the context of enterprise applications), and displaying the results to the user. The most widely used design pattern for the presentation layer of user interface based applications is the *Model View Controller (MVC)* or its more recent adaptation the *Model View Presenter (MVP)* — both these patterns are considered prerequisite knowledge for this subject, but will be briefly reviewed at the beginning of this chapter.

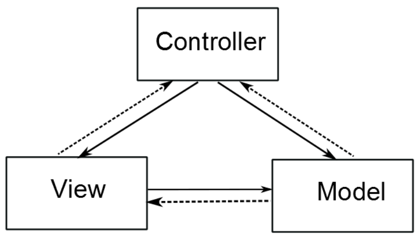
The controller in the MVC architecture is referred to as the *input controller*, to avoid confusion with the different types of controllers used in enterprise applications. We will study two patterns that are specific to the design of the input controller in the MVC architecture: (1) *page controller*; and (2) *front controller*.

We will also look at three patterns related to view handling in the MVC architecture: (1) *template view*; (2) *transform view*; and (3) *two step view*.

A common design issue related to complex enterprise applications is handling the overall flow of the application while separating the presentation objects from the domain objects. We will study the *application controller* design pattern which serves this purpose.

Learning outcomes A person familiar with the material in this chapter should be able to:

1. describe the six design patterns related to the presentation layer, and what problem they aim to solve;
2. apply these design patterns in enterprise applications;
3. critique the choice of one of the different design patterns on a particular application; and
4. provide a rationale for the choice of one the presented patterns in an enterprise application.



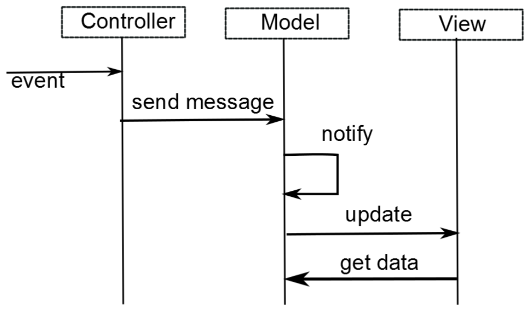


Figure 6.1: Structure and Behaviour of the MVC Pattern

6.1 The problem

The problem: How do we organise the presentation logic of an enterprise system to make it understandable and maintainable?

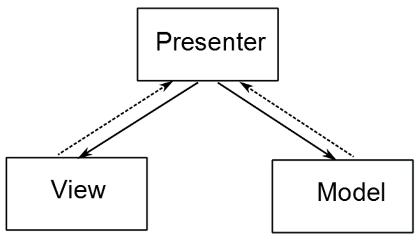
The presentation layer deals with the user interactions of the system. Often user interactions are facilitated through a graphical user interface. In the case of web-based enterprise applications the user interface is often a web browser, however this need not be the case. The goal of the patterns related to the presentation layer is to decouple the user interface layer from the lower layers that deal with domain logic and database access. Good abstraction of the presentation layer will allow the user interface of the system to be changed without any changes to the lower layers, and will make it possible to support multiple UIs using the same underlying layers.

6.2 Overview of Patterns

The most widely used web presentation patterns are the Model View Controller (MVC) and Model View Presenter (MVP). These patterns are well documented in the literature; they will be reviewed in this section as revision. Both these patterns abstract the logic related to presentation to three modules: (1) Model; (2) View; and (3) Presenter/Controller.

1. **Model:** Model represents the information about the domain.
2. **View:** View deals with the display of information to the client through a user interface.
3. **Controller/Presenter:** Handles the interactions between the view and the model.

Figures 6.1 and 6.2 show the structure and behaviour of the MVC and MVP design patterns respectively. The main difference between the two patterns lie in the interactions of the view and the model. In the MVC pattern the view directly queries the model while in the MVP pattern the view is managed by the presenter.



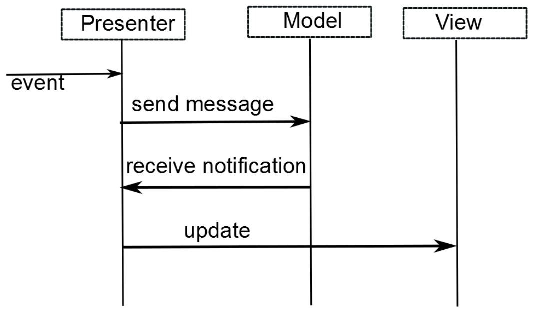


Figure 6.2: Structure and Behaviour of the MVP Pattern

In the context of the layered architecture, the view and the presenter/controller are on the layer above the model layer. Both these patterns provide a clear separation between the model layer and the upper layer (view-presenter/controller) layer, however, less separation is provided between the view and the controller/presenter.

6.3 Patterns for Input Controller

The input controller acts as the controller in the MVC architecture. *Page Controller* and the *Front Controller* are two widely used patterns for the design of the controller; these patterns will be discussed in the next sections.

6.3.1 Page Controller

Pattern name	Page controller
Description	An object that handles a request for a specific page or action on a Web site.

When the page controller design pattern is used, each web page or each action of the web page (for example each submit button) will have its own controller. The controller is responsible for handling the logic related to user actions, as well as displaying the view. The basic structure of the pattern is shown in Figure 6.3.

The responsibilities of a Page Controller are:

- Extract user data either from a post method via a form or as parameters in the URL.
- Invoke the appropriate methods in the domain logic layer or external services to perform the appropriate actions based on the user action.
- Determine which view to display and forward the model information to it.

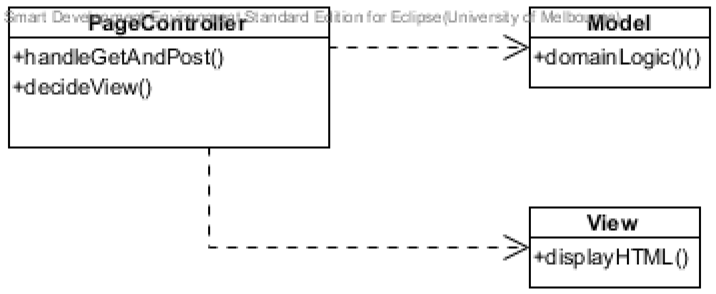


Figure 6.3: Structure of the *Page Controller* design pattern.

Implementation

For simple applications where the action to be performed based on the user request is straightforward, the controller is quite simple and using a script such as a servlet or a CGI script is an acceptable solution. However, in cases where more complex logic is needed for handling the user requests, for example choosing between different views to display based on user actions, the script code can become complicated and messy. In such cases helper objects can be used to encapsulate the logic.

In some cases, especially when the logic that needs to be performed is relatively simple, server pages can be used as the controller.

Pros

1. *Ease of understanding:* Due to the natural structuring mechanism of particular actions being handled by particular server pages code is easy to understand and intuitive.
2. *Simplicity:* Compared to the alternative design pattern, the front controller, developing presentation layer using the page controller pattern is much simpler.

Cons

1. *Duplication of logic:* If there are generic actions that need to be performed across pages, there could be code duplication. This can be minimised by having parent classes that handle generic logic.
2. *Complex logic in the controller:* Since a single controller is responsible for handling all the logic related to the page, this could result in complex logic in the controller.
3. *Extensibility:* Adding new pages cannot be done dynamically because controllers are static objects.

Example

To better understand the class structure and behaviour of the page controller design pattern we will look at an example of a simple web page that displays information regarding a recording artist given the artist name, implemented using J2EE technologies (a servlet at the controller and a JSP as the view), Figure 6.4 (reproduced from [3]) shows the design class diagram for this case. This design is very similar to the simple MVC Login application we developed as a workshop exercise.

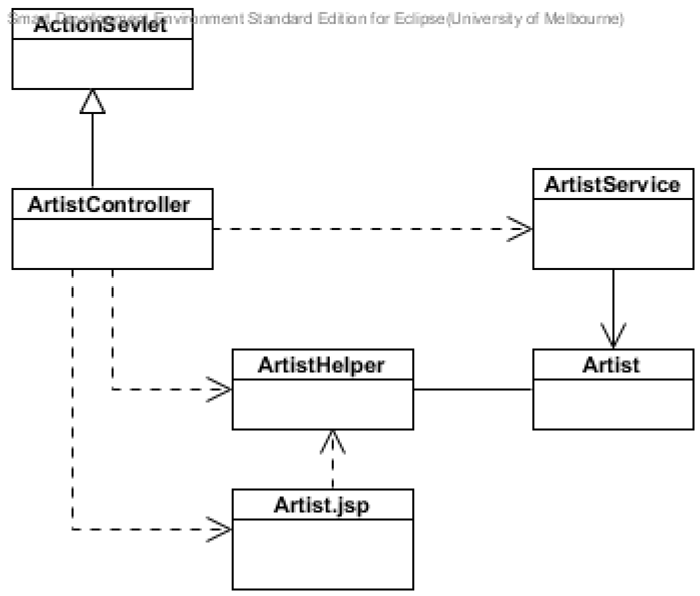

Smart Development Environment Standard Edition for Eclipse(University of Melbourne)
ActionSevlet
ArtistController
ArtistService
ArtistHelper
Artist
Artist.jsp

Figure 6.4: Structure of the *Page Controller* design pattern.

Listings 6.1 and 6.2 show the Java code for the controller classes.

Listing 6.1: **ArtistController.java class**

```java
package com.swen90007.mvc.controller;

import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import com.swen90007.mvc.service.*;
import com.swen90007.mvc.model.*;
/**
 * Servlet implementation class ArtistController
 */
@WebServlet("/ArtistController")
public class ArtistController extends ActionServlet {
  private static final long serialVersionUID = 1L;

  @Override
  public void init(){
    ArtistService a = new ArtistService();
  }

  /**
   * @see HttpServlet#doGet(HttpServletRequest request, HttpServletResponse
       response)
   */
  protected void doGet(HttpServletRequest request, HttpServletResponse
      response) throws ServletException, IOException {

    System.out.println(request.getParameter("name"));
    Artist artist = ArtistService.findNamed(request.getParameter("name"));

    if (artist == null)
      forward("/MissingArtistError.jsp", request, response);
    else {
      request.setAttribute("helper", new ArtistHelper(artist));
      forward("/artist.jsp", request, response);
    }

  }

}
```

Listing 6.2: **ActionServlet.java class**

```java
package com.swen90007.mvc.controller;

import java.io.IOException;

import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

/**
 * Servlet implementation class ActionServlet
 */
@WebServlet("/ActionServlet")
public class ActionServlet extends HttpServlet {
  private static final long serialVersionUID = 1L;

    protected void forward(String target, HttpServletRequest request,
       HttpServletResponse response) throws IOException, ServletException{

      RequestDispatcher dispatcher = getServletContext().
         getRequestDispatcher(target);
      dispatcher.forward(request, response);

    }
}
```

Listings 6.3 - 6.5 show the corresponding model and service classes.

Listing 6.3: **Artist.java class**

```java
package com.swen90007.mvc.model;

public class Artist {
  private String name;
  private String album;


  public Artist(String name, String album) {
    super();
    this.name = name;
    this.album = album;
  }

  public String getName() {
    return name;
  }
  public void setName(String name) {
    this.name = name;
  }
  public String getAlbum() {
    return album;
  }
  public void setAlbum(String album) {
    this.album = album;
  }


}
```

Listing 6.4: **ArtistHelper.java class**

```
package com.swen90007.mvc.model;


public class ArtistHelper {
  Artist artist;


  public ArtistHelper(Artist artist){
    this.artist = artist;
  }

  public String getName(){
    return artist.getName();
  }

  public String getAlbum(){
    return artist.getAlbum();
  }

}
```

Listing 6.5: **ArtistService.java class**

```
package com.swen90007.mvc.service;
```

```java

import java.util.HashMap;

import com.swen90007.mvc.model.Artist;

public class ArtistService {

  public static HashMap artists = new HashMap();

  public ArtistService(){
    artists.put("Joe_Do",  new Artist("Joe Do", "XXXX"));
    artists.put("John_Brown", new Artist("John Brown", "YYYY"));
  }

  public static Artist findNamed(String name){
    return (Artist)artists.get(name);
  }

}
```

An alternative to the above implementation is using JSPs as the controller with helper classes for managing the logic.

Listings 6.6-6.8 show the classes that perform the controller actions — the login associated with the controller are performed in the helper classes. The model and service classes are the same as the previous example and, therefore, the code included here.

Listing 6.6: **artistController.jsp**

```jsp
<%@ page language="java" contentType="text/html; charset=ISO-8859-1"
    pageEncoding="ISO-8859-1" import="com.swen90007.mvc.controller.*"%>
<!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://www.
   w3.org/TR/html4/loose.dtd">
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=ISO-8859-1">
<title>Insert title here</title>
</head>
<body>
<jsp:useBean id="helper" class="com.swen90007.mvc.controller.
   ArtistControllerHelper" scope="request"></jsp:useBean>

<%helper.init(request, response); %>
</body>
</html>
```

Listing 6.7: **ArtistControllerHelper.java class**

```java
package com.swen90007.mvc.controller;

import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import com.swen90007.mvc.model.*;
import com.swen90007.mvc.service.ArtistService;
/**
 * Servlet implementation class ArtistController
 */


public class ArtistControllerHelper  extends HelperController {


  public void init(HttpServletRequest request, HttpServletResponse response
     )  {

    super.init(request, response);
    ArtistService a = new ArtistService();

    Artist artist = ArtistService.findNamed(request.getParameter("name"));

    if (artist == null)
      forward("/MissingArtistError.jsp", request, response);
    else {
      request.setAttribute("helper", new ArtistHelper(artist));
      forward("/artist.jsp", request, response);
    }

  }
}
```

Listing 6.8: **HelperController.java class**

```java
package com.swen90007.mvc.controller;

import java.io.IOException;

import javax.servlet.RequestDispatcher;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public class HelperController {

  private HttpServletRequest request;
  private HttpServletResponse response;

  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.getServletContext().
          getRequestDispatcher(target);
      dispatcher.forward(request, response);
    }
    catch(IOException e){
      //throw new Exception(e);
    }
    catch(ServletException e){
      //throw new Exception(e);
    }

  }
}
```

6.3.2 Front Controller

Pattern name	Front controller
Description	A controller that handles all the requests for a Web site.

In the case of complex enterprise applications, generic operations such as security checking and internationalisation have to be performed when handling any user action regardless of the view that is displayed to the user. In such a scenario using the page controller design pattern could result in dispersed logic; hence duplication. The front controller design pattern solves this problem by channelling all requests through a single controller.

The basic structure and the behaviour of the pattern is shown in Figures 6.5 and 6.6 respectively.

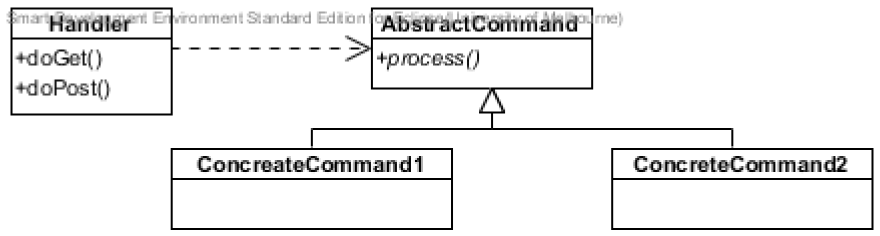


Figure 6.5: Structure of the *Front Controller* design pattern.

Implementation

The front controller design pattern can be viewed as consisting of two parts: (1) a handler; and (2) a command hierarchy.

The handler is responsible for receiving the request and identifying the appropriate command class to execute; it is usually implemented as a servlet as opposed to a server page. Identification of the command in the handler can be done either statically or dynamically:

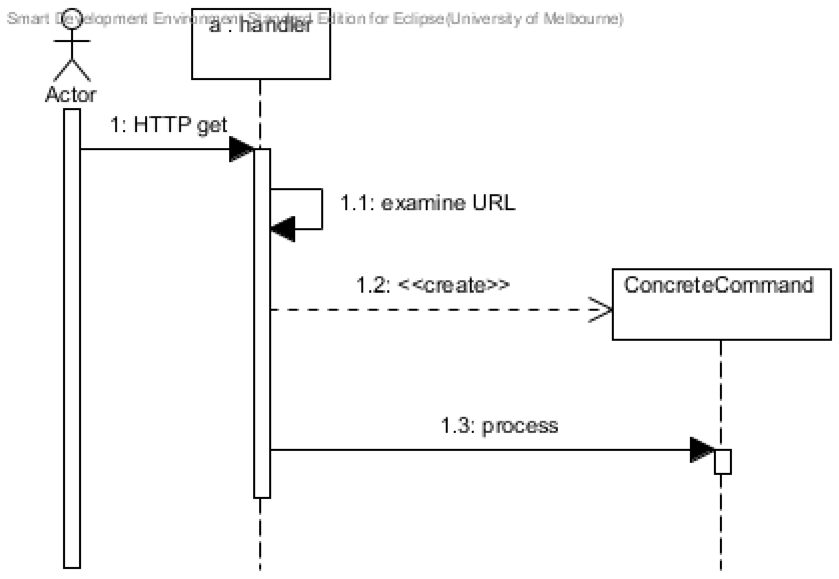


Figure 6.6: Behaviour of the *Front Controller* design pattern.

1. *Static identification:* The URL is parsed to identify the command, and an appropriate command object is created based on conditional logic.
2. *Dynamic identification:* Based on the command associated with the URL, dynamic object instantiation is used to create a command class.

Pros

1. *Single entry point:* Having a single controller makes the web server configuration much easier compared to the page controller.
2. *Ease of extension:* Runtime dynamic command instantiation gives the option of adding new commands at runtime.

Cons

1. *Design complexity:* More complicated than the page controller design pattern.

Example

We will now look at an example of implementing the front controller design pattern in a J2EE application. The specific example we will look at is an application which supports two commands: one that retrieves information regarding an artist given the name of the artist, and the second that retrieves information regarding an album given the album title. The command are accessed as: `http://localhost:8080/isa/music?name=barelyWorks&command=Artist` or `http://localhost:8080/isa/music?tilte=xxxYYY&command=Album`.

Figures 6.7 shows the controller classes in the application.

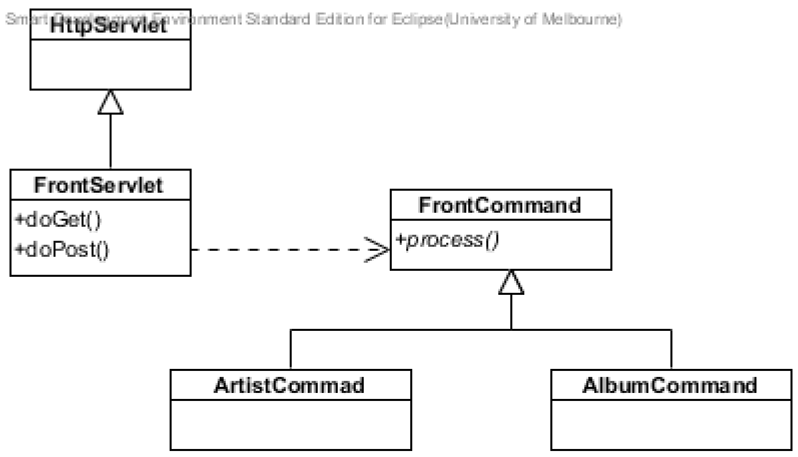


Figure 6.7: Class Diagrams of the controller classes.

Listings 6.9 - 6.12 show the Java code for the implementation classes.

Listing 6.9: **FrontServlet.java class**

```
package com.swen90007.mvc.controller;

import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

import org.omg.CORBA.portable.ApplicationException;

/**
 * Servlet implementation class FrontServlet
 */
@WebServlet("/FrontServlet")
public class FrontServlet extends HttpServlet {
  private static final long serialVersionUID = 1L;

    /**
     * @see HttpServlet#HttpServlet()
     */
    public FrontServlet() {
        super();
        // TODO Auto-generated constructor stub
    }

   /**
    * @see HttpServlet#doGet(HttpServletRequest request, HttpServletResponse
        response)
    */
  protected void doGet(HttpServletRequest request, HttpServletResponse
     response) throws ServletException, IOException {
```

```
    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){
      return null;
    }

  }

  private Class getCommandClass(HttpServletRequest request){
    Class result;
    final String commandClassName = "com.swen90007.mvc.controller." + (
       String) request.getParameter("command") + "Command";

    System.out.println(commandClassName);
    try{
      result = Class.forName(commandClassName);
      System.out.println(result);
    } catch(ClassNotFoundException e){
      result = UnknownCommand.class;
    }
    System.out.println(result);
    return result;
  }
}
```

Listing 6.10: **FrontCommand.java class**

```
package com.swen90007.mvc.controller;

import java.io.IOException;

import javax.servlet.RequestDispatcher;
import javax.servlet.ServletContext;
import javax.servlet.ServletException;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

public abstract 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;

  }

  abstract public void process() throws ServletException, IOException;

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

Listing 6.11: **ArtistCommand.java class**

```
package com.swen90007.mvc.controller;

import java.io.IOException;

import javax.servlet.ServletException;

import com.swen90007.mvc.model.Artist;
import com.swen90007.mvc.model.ArtistHelper;
import com.swen90007.mvc.service.ArtistService;

public class ArtistCommand extends FrontCommand {

  public void process() throws ServletException, IOException{

    Artist artist = ArtistService.findNamed(request.getParameter("name"));
    if (artist == null)
      forward("/MissingArtistError.jsp");
    else {
      request.setAttribute("helper", new ArtistHelper(artist));
      forward("/artist.jsp");
    }
  }
}
```

Listing 6.12: **AlbumCommand.java class**

```
package com.swen90007.mvc.controller;

import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.ServletException;

import com.swen90007.mvc.model.Album;
```

```java
import com.swen90007.mvc.model.Artist;
import com.swen90007.mvc.model.AlbumHelper;
import com.swen90007.mvc.service.AlbumService;
import com.swen90007.mvc.service.ArtistService;

public class AlbumCommand extends FrontCommand {

  public void process() throws ServletException, IOException{
    AlbumService a = new AlbumService();

    Album album = AlbumService.findNamed(request.getParameter("title"));

    if (album == null)
      forward("/MissingAlbum.jsp");
    else {
      request.setAttribute("helper", new AlbumHelper(album));
      forward("/album.jsp");
    }

  }
}
```

6.4 Patterns for View

We will look at three patterns for handling the view in the MVC architecture: *Template View*; *Transform View*; and *Two Step View*.

6.4.1 Template View

Pattern name	Template view
Description	Renders information into HTML by embedding markers in an HTML page.

One obvious approach to generating web pages containing dynamic information (information generated at runtime based on user queries) is to generate HTML programmatically in the servlet. However, this is not very straightforward because manipulating layout of pages through programming logic is not very intuitive; WYSWIG editors are much better at generating view pages.

With the template view design pattern, the view is generated using static HTML with dynamic content created through programming logic embedded in markers. The static part of the page acts as a template for the view; hence the term *Template View.*

Implementation

Server page technologies, such as JSP, ASP and PHP, are examples where the template view pattern is used for generating views. These technologies allow *scriptlets*, containing programming logic to be embedded in the pages.

Pros

1. *Simplicity:* The pages are easy to design and the programming task is simplified.

Cons

1. *Reduced maintainability:* Makes it hard for non-programmers (who usually do the web page design), to design/maintain views.
2. *Promotes bad design practises:* Embedding code within pages results in designs that do not follow good design principles. Code is not well encapsulated and does not follow the modularity and layering principles, which further discourages reuse.
3. *Challenging to test:* It is difficult to isolate the code for unit testing, and is therefore generally tested within a web server.

Example

We will look at the same example we used for the page controller, where the ArtistController.java class (Listing 6.13) used a JSP (artist.jsp), to display information related to the artist.

Listing 6.13: **ArtistController.java class**

```
protected void doGet(HttpServletRequest request, HttpServletResponse
    response) throws ServletException, IOException {

    System.out.println(request.getParameter("name"));
    Artist artist = ArtistService.findNamed(request.getParameter("name"));

    if (artist == null)
      forward("/MissingArtistError.jsp", request, response);
    else {
      request.setAttribute("helper", new ArtistHelper(artist));
      forward("/artist.jsp", request, response);
    }

  }
```

The JSP can then access the helper using the following code in Listing 6.14.

Listing 6.14: **Code to access the helper class**

```
 <jsp:useBean id="helper" type="actionController.ArtistHelper" scope="
    request"/>
```

Then the information can be retrieved from the helper class using a Java expression (Listing 6.15) or a property (Listing 6.16).

Listing 6.15: **Accessing the helper as a java expression**

```
<B> <%=helper.getName()%></B>
```

Listing 6.16: **Accessing the helper as a property**

```
<B><jsp:getProperty name="helper" property="name"/></B>
```

Using a helper, the developer can often avoid complicated scriptlet code by putting in the logic in the helper.

The example in Listing 6.17 show the scriptlet code for displaying a list of albums:

Listing 6.17: **Scriptlet code to display a list of albums**

```
 <UL>
<%
   for (Iterator it = helper.getAlbums().iterator(); it.hasNext();) {
      Album album = (Album) it.next();%>
   <LI><%=album.getTitle()%></LI>
<%   } %>
</UL>
```

This messy code could be avoided by implementing a `getAlbumList()` method in the helper class, that returns HTML with the list of albums. The HTML is generated in Java.

6.4.2 Transform View

Pattern name	Transform view
Description	A view that processes domain data element by element and transforms it into HTML.

As opposed to the template view which focuses on the structure of the display as a template, the transform view focuses on the structure of the domain data; this data is then transformed to different display formats; hence the name *Transform View*. While the template view is designed based on the output, the transform view is based on the input — the domain object.

Implementation

Two widely-used transform languages are XSLT, which is a functional programming language, and Javascript, which is a multi-paradigm language.

XSLT performs the rendering transformations based on the elements in the domain data expressed in XML.

Javascript is typically used to render transformations based on domain data expessed in formats such as XML or JSON.

Pros

1. *Portability:* Portability across different web platforms due to language independence. Because the views are only tied to an XML output of the domain object, it is more standardised.
2. *Encapsulation:* The logic in the view is limited to displaying, which is a good design practice.
3. *Improved Testability:* The views can be tested independent from the web server.

Cons

1. *Programming Complexity:* Implementation is much harder than template view; expertise in XSLT is required.

Example

Listing 6.18 shows the AlbumCommand class in the previous example modified to use a transform view.

Listing 6.18: **AlbumComamnd.java class**

```java
package com.swen90007.mvc.controller;

import java.io.IOException;
import java.io.PrintWriter;

import javax.servlet.ServletException;


import com.swen90007.mvc.model.Album;
import com.swen90007.mvc.service.AlbumService;

public class AlbumCommand extends FrontCommand {

  public void process() throws ServletException, IOException{
    AlbumService a = new AlbumService();

    Album album = AlbumService.findNamed(request.getParameter("title"));
    Assert.notNull(album);

    PrintWriter out = response.getWriter();

    XsltProcessor processor = new SingleStepXsltProcessor("album.xls");

    out.print(processor.getTransformation(album.toXmlDocument()));

  }

}
```

The corresponding XML and XSL are shown in Listings 6.19 and Listing 6.20 respectively.

Listing 6.19: **XML for Album**

```
<?xml version="1.0" encoding="UTF-8"?>
<?xml-stylesheet href="album.xsl" type="text/xsl" ?>
<album>
  <title>Stormcock</title>
  <artist>Roy Harper</artist>
  <trackList>
    <track><title>Hors</title><time>8:37</time></track>
    <track><title>The Same Old Rock</title><time>12.34</time></track>
      <track><title>One Man Rock and Roll Band</title><time>7.23</time></
         track>
  </trackList>
</album>
```

Listing 6.20: **XSL for Album**

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet version="1.0" xmlns:xsl="http://www.w3.org/1999/XSL/
   Transform">

  <xsl:template match="album">
    <HTML>
    <BODY>
    <h1><xsl:value-of select="title"/></h1>
    <h1><xsl:value-of select="artist"/></h1>
    <table border="1">
    <tr>
        <th>Track</th>
        <th>Time</th>
    </tr>

    <xsl:for-each select="trackList/track">
    <tr>
            <td><xsl:value-of select="title"/></td>
            <td><xsl:value-of select="time"/></td>
    </tr>
    </xsl:for-each>


    </table>
    </BODY>
    </HTML>
  </xsl:template>
</xsl:stylesheet>
```

6.4.3 Two Step View

Pattern name	Two-step view
Description	Turns domain data into HTML in two steps: first by forming some kind of logical page, then rendering the logical page into HTML.

The transformation of output data into HTML is performed in two stages: first the model data is converted to a logical presentation format without formatting; and the second stage converts the logical presentation to HTML as shown in Figure 6.8.

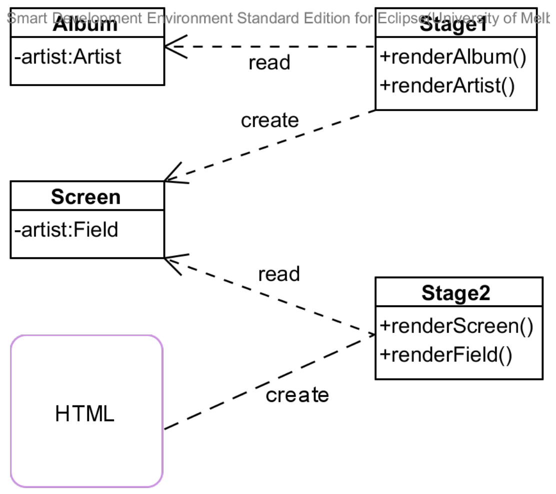


Figure 6.8: Structure of the *Two Step View* design pattern.

Implementation

The most common implementation of the two step view pattern is using two-step XSLT.

1. The first stage: transform domain-oriented XML to presentation-oriented XML.
2. The second stage: render the presentation-oriented XML in HTML.

An alternative is to have two different classes: *domain-oriented classes*, which model the domain information independent of the presentation; and *presentation-oriented classes*, which model the presentation elements. Domain-oriented objects are translated into presentation-oriented objects, which are presented as the view. Different translators can be implemented to give different presentations.

Pros

1. *Improved maintainability:* Making global changes to the view format is much easier due to good encapsulation.

Cons

1. *Design Complexity:* Developing a good presentation logic structure requires expertise.
2. *Implementation Complexity:* Programming knowledge is required to build the views — a non-programmer cannot be used for developing the view layer. The programming model is harder to learn.

Example

Listings 6.21 and 6.22 show the domain-oriented and presentation-oriented XML for our previous example.

Listing 6.21: **Domain-Oriented XML**

```xml
<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>
```

Listing 6.22: **Presentation-Oriented XML**

```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>
```

6.5 Application Controller

Pattern name	Application controller
Description	A centralised point for handling screen navigation and the flow of an application.

In the patterns we have looked at so far, the input controller is responsible for the logic related to which screen is displayed next based on the user input. This works well in the case of simple page transition logic. However, in applications which require more complex screen navigation, this could result in dispersed and duplicated logic that is difficult to manage and maintain. The application controller design pattern addresses this problem through an Application Controller that handles the flow of logic. The input controller then calls the application controller to get the domain object and the view for the particular command.

Implementation

The application controller has two main responsibilities (Figure 6.9):

1. deciding which domain logic object to run; and
2. deciding the view for displaying the response.

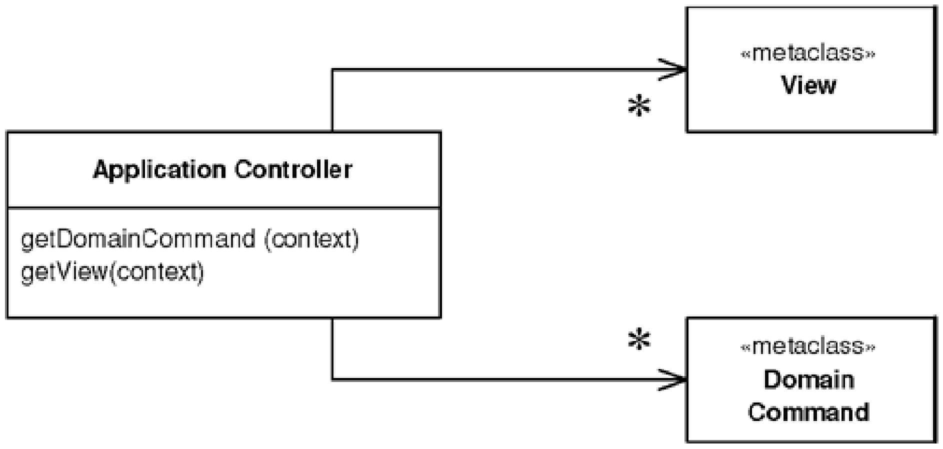


Figure 6.9: Structure of the *Application Controller* design pattern.

Pros

1. *Simplified input controller:* Enables handing complex screen navigation logic; avoid this being done in the input controller.
2. *Less duplicated code:* Code is less likely to be dispersed around the application due to the central design, resulting is less duplication.

Cons

1. *Complexity:* Over-complicates the design in cases where screen transition logic is relatively simple.

Example

In the case of using an Application Controller, the Front Controller design pattern can be used. The FrontController invokes methods in the ApplicationController to find the appropriate Command class and the View (Listing 6.23). The class diagram in Figure 6.10 shows the class diagram for the implementation.

Listing 6.23: **FrontServlet.java**

```
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);
   }

}
```

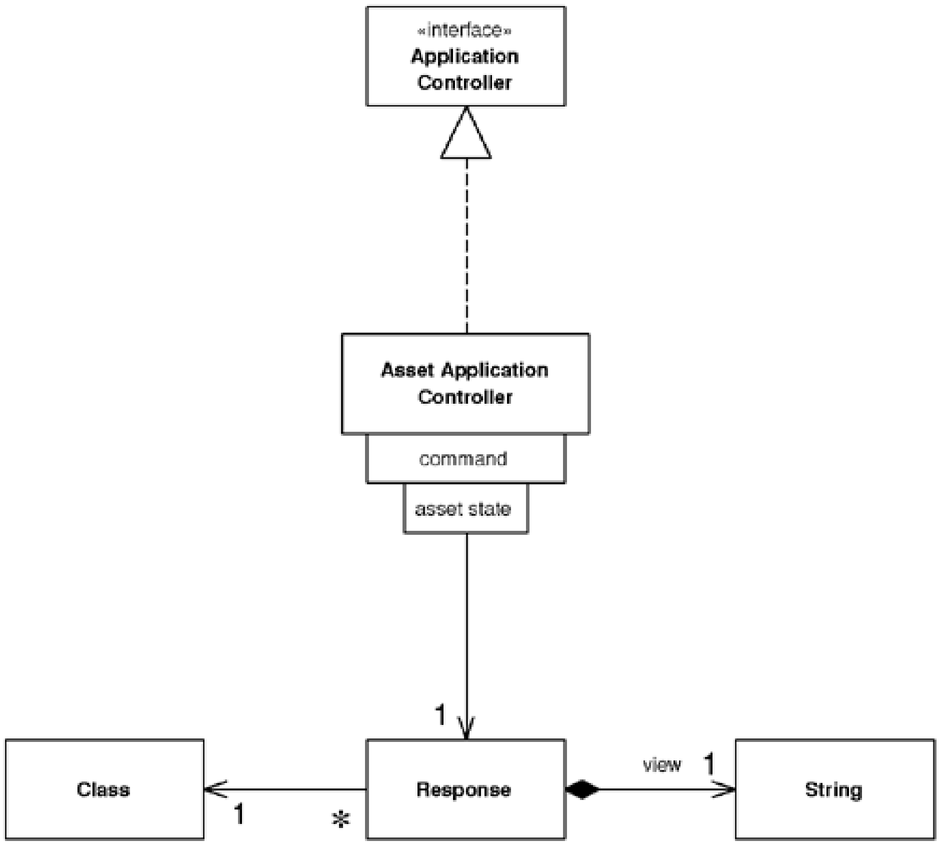


Figure 6.10: Class diagram of the *Application Controller* design pattern implementation.

Chapter 7

Sessions

Web servers are normally stateless. This implies that state/information is not persisted across requests; each new request is treated independently. However, this does not suit most client applications, because client interactions are normally stateful. For example, consider a simple shopping cart application, where the user chooses items to purchase that he/she places on a shopping cart. The contents of the shopping cart must be persisted across requests and therefore the interaction is stateful; in particular the state must to be persisted for the session, referred to as *session state*.

In this chapter, we will look at how the session state can be persisted in the web server, which is otherwise stateless. We will particularly look at three different design solutions to this problem; *(1) Client Session Sate; (2) Server Session State*; and (3) *Database Session State.*

Learning outcomes A person familiar with the material in this chapter should be able to:

1. describe three different ways of persisting session state;
2. critique the choice of one of the different approaches for persisting session state on a particular application; and
3. provide a rationale for the choice of the session state solution in an enterprise application.

7.1 The problem

The problem: How to we support states using servers that are stateless?

Although web servers are stateless, client applications need state persisted across sessions. The obvious approach to this is to maintain an object per session. However, given the large number of clients simultaneously connecting to the servers, and the inactive time between requests, the server will have to maintain a large number of objects in memory.

In this chapter, we will look at different ways of maintaining session state, and the trade-off between the approaches.

7.2 The patterns

7.2.1 Client Session State

Pattern name	Client Session State
Description	Stores session state on the client.

Using this pattern, the session state is maintained in the client (as opposed to the server). Most designs require at least the session ID to be persisted as Client Session State. In some applications all session state will be maintained in the client whereas in other applications it could be in the server or a combination of the two.

Implementation

With an HTML interface as the client there are three different ways to maintain client session state:

1. *URL parameters:* In this approach a URL parameter is used to maintain session state — this works well for small amounts of data (for example maintaining session ID) but does not scale well.
2. *Hidden fields:* The data is sent to the browser as a hidden field that is not displayed using `<INPUT type="hidden">` tag. The data is sent in serialised form.
3. *Cookies:* These are sent back and forth automatically — session state can be serialised as a cookie.

Pros

1. *Resilience*: Resilient to server failures.
2. *Clustering*: Works well with server clusters.

Cons

1. *Scalability*: Does not scale well for large amounts of data, because all relevant data must be sent with each request, which may entail sending megabytes of data.
2. *Security*: Data can be looked at and tampered with as it sent across a network.

7.2.2 Server Session State

Pattern name	Server Session State
Description	Keeps the session state on a server system in a serialised form.

Session state is maintained in the server memory.

Implementation

The simplest implementation is an object maintained in the server for each object. The server can maintain a map of objects indexed by the session ID. The assumption is that the server has enough memory to maintain this for all active sessions. In the case of memory issues on the server of maintaining session state, the state can be persisted in a serialised form.

This solution assumes that there is only one server as opposed to a cluster of servers. In the case of clusters a more complex solution is needed.

Pros

1. *Simplicity:* The solution is intuitive and is easy to implement for the simple case of a single server.

Cons

1. *Clustering*: The simple form of solution does not work with failover and clustering — solving this problem will result in a complicated solution.

7.2.3 Database Session State

Pattern name	Database Session State
Description	Stores session data as committed data in the database.

Session state is maintained in the database itself based on the session ID.

Implementation

The import issue to consider here is how the session data (for which the business transaction is most probably not completed) is separated from the data that is already committed to the database.

There are several alternative solutions that are possible:

1. *Additional field:* Adding either a pending flag or a session ID to each record — the downside of this solution is that queries to non-session data also gets impacted because a "not NULL" clause needs to be specified for queries. This can be solved thorough defining a database view, but this will have additional costs.

2. *Additional table:* For each of the tables an additional table to maintain pending data can be added. This will solve the previous problem related to queries but add the complexity of maintaining two sets of tables.

In both the approaches, mechanisms must be in place to clean the data for cancelled or abandoned sessions. In the case of cancelled sessions, data related to the session ID can be removed. Timeout mechanisms are required for abandoned sessions.

Pros

1. *Clustering a failover*: Can easily handle clustering and failover.

Cons

1. *Performance*: May impact performance due to having to pull data from the database with each request. Caching can help solve this to some extent.
2. *Complexity*: More difficult to implement compared to server session state.

Chapter 8

Concurrency

In this chapter, we discuss the issue of concurrency. Enterprise systems almost always need to support concurrent access to the data in the data-source layer. What this means is that multiple processes within the system will be trying to access the same data at the same time. If all of the processes are trying to read the data only, then this is not such an issue, but if some are trying to write, then problems can occur, such as interference or inconsistent reads.

While database management systems offer support for dealing with concurrency, they do not solve all of the problems. First, some of the data may reside outside of the database. Second, while database management systems use transactions to mitigate problems with concurrency, they do not know about the business or system transactions that are being implemented. Typically, business transactions may consist of many database transactions. As such, we cannot rely solely on the database transaction manager to maintain data integrity.

The general strategy for handling these multi-span business transactions is *offline concurrency*. Offline concurrency simply means that we are supporting the ACID properties (see Chapter 1) between database transactions. This chapter looks at offline concurrency on the server side of an enterprise system. That is, handing multiple requests from many clients that want to access the same data on the data layer.

We will study three patterns for concurrency in enterprise systems: (1) *pessimistic offline lock*; (2) *optimistic offline lock*; and (3) *implicit lock*.

Learning outcomes A person familiar with the material in this chapter should be able to:

1. describe the problems that concurrent access can cause in an enterprise system;
2. describe the different types of transactions, and the effects they have on the system;
3. describe the difference between offline pessimistic locking and offline optimistic locking, the problems with each, and when to apply them;
4. describe the three design patterns related to concurrency, and what problem they aim to solve;
5. apply the design patterns in enterprise applications;
6. critique the choice of one of the three design patterns on a particular application; and
7. provide a rationale for the choice of one the presented patterns in an enterprise application.

8.1 The problem

The problem: How do we support multiple processes accessing the same data in an enterprise system while preserving the integrity of the data, and providing efficient access to that data?

The concurrency-related problems encountered in enterprise systems are the same as other systems, and the solutions are much the same as well. If all processes just want to read this data, there is typically no problem. Problems occur when at least one process wants to add, delete, or update the data, including:

1. *Interference (or lost updates)*: This occurs if two or more processes write data to the same location in quick succession. For example, process P_1 reads a row from a table, then process P_2 reads that same row. Both processes then make different changes to their local copy of the row. Process P_1 then writes its change to the table, followed by process P_2. The updates made by P_1 will be lost.

2. *Inconsistent read*: This occurs if a process reads data from multiple places, such as multiple tables or rows, while another process is updating it. For example, consider a process, P_1, that wants to find the number of new sales that have come in so far on a particular day, and the number of returns that have come in that same day, in order to calculate the net number of sales. If process P_1 issues two queries for this (one for sales and one for returns), and another process, P_2, is writing to this data at the same time, the result may be inconsistent. For example, if there are 1000 new sales and 300 returns, the net sales is 700. Process P_1 first reads the number of new sales, retrieving 1000. Process P_2 then updates the number of sales and returns to 1100 and 500 respectively (adding 100 and 200 to each respectively). The net sales is now 600. After this, process P_1 issues its query for the number of returns, retrieving 500. P_1 then calculates the net sales at $1000 - 500 = 500$, which was never the correct answer. Either 700 or 600 would be consistent, but 500 is inconsistent.

There are other problems that are critical but not so difficult to solve in enterprise systems due to the underlying technology that is used.

This chapter presents control mechanism patterns and principle for dealing with the issues interference and inconsistent reads. The solutions discussed in this chapter bring their own problems, however, these problems are *generally* considered less serious than interference and consistent reads.

Execution Contexts

Requests and sessions are two important contexts from the perspective of an enterprise application interacting with the outside world.

Enterprise system are generally designed to handle *requests*. A request is a single call generated from outside the system, which often requires a response. Typically, systems are implemented such that the client issues a request, and waits for the response.

A *session* is a long-running interaction between a client and a server. Often, sessions consists of a series of requests that are related to each other in a logical sequence. For example, buying an item from a website often requires multiple requests, such as choosing the item, selecting the properties (such as size or colour), checking out, submitting payment options, submitting delivery details, and finally confirming. Each of these is issued as a request from the server, in which the behaviour of each request is dependent on the requests before it.

Transactions are another important context when interacting with databases. Transactions pull together several requests that the client wants treated as if they were a single request. They can occur from the application to the database (a system transaction) or from the user to an application (a business transaction). There terms will be discussed in more detail later on.

Transactional resources

In enterprise systems, most system transactions relate to the underlying database. Data sources other than databases may also be used, however, a database is most common. A *transactional resource* is a resource that uses transactions to mitigate concurrency problems. Database management systems use transactions to handle concurrency, as do other data sources. In this chapter, we will assume that the underlying data sources are transactional resources.

We can categorise system transactions based on their relationship to the business transactions/rules that they implement:

- *Request transaction*: a transaction that spans exactly one request to the system.
- *Long transaction*: a transaction that spans two or more requests to the system.
- *Late transaction*: a transaction that does as much reading as possible, calculates what changes are required, and then does an update as late as possible. Using late transactions leaves the system open to inconsistent reads.

When locking a database table, we must be careful not to lock things we do not touch. If we are writing to a particular set of rows, most database management systems will lock only those rows. However, if the database is asked to lock more rows than it can handle, it will often lock an entire table, meaning that other processes cannot lock the rows in that table, even though they are not being used by another process. This is called *lock escalation*, and can cause serious performance issues.

The nomenclature can be confusing here. We are discussing two different levels of transactions:

1. *System transaction*: transactions supported by transactional resources. E.g., a database transaction (a group of SQL commands delimited by instructions to begin and end it).
2. *Business transaction*: those at the enterprise level that may bring together several system transactions.

One design option is to limit all system transactions to single transactions on the underlying transactional resource. This allows us to employ the transactional resource's capability to handle concurrency, roll backs, etc. However, this may complicate the design, and may also not be possible.

8.2 The patterns

8.2.1 Optimistic offline lock

Pattern name	Optimistic offline lock
Description	Prevents conflicts between concurrent business transactions by detecting a conflict and rolling back the transaction.

An *optimistic lock* is a late transaction pattern that validates the changes about to be committed in one session do not conflict with changes in another. If they do, the changes cannot be committed, and the business transaction fails. It does this by confirming that, since a record has been loaded, no other session has changed that record.

Implementation

The most common and perhaps most straightforward method involves associating a version number with each record in a database. Then, the following steps are used to implement the optimistic lock:

1. When a record is read, note the version number and store this; e.g. in the session state.
2. Make changes to that record.
3. Before writing the record, read the version number of that record in the table.
4. If the two version numbers are equivalent, the record has not changed, so the data can be safely written back.
5. Obtain the lock on the rows, and update the data, incrementing the version number of that record at the same time.
6. If the two version numbers are different, then another process has modified the record. Fail the business transaction and do not write to the database.

Example

The example below extends the example of the data mapper from Section 3.2.4. In the example below, the find and update methods consider the modification of the rows. The database table containing the information is extended with three additional columns: the version of the row, the last modification time, and the session that modified it.

When a list of person objects are searched, the version number, modified time, and the session that modified the row are stored in the *Person* instances that are returned. When an update is performed, a new query is issued to find the latest version of a row, and if this version is inconsistent with the version in the object being updated, an exception is thrown. In addition, if no such row exists any longer, then the row has been deleted, and an exception is thrown.

Listing 8.1: **PersonMapper.java**

```java
class PersonMapper {

  public static List<Person> findWithLastName(String lastName) {
    String sql = "SELECT id, lastname, firstname, number_of_dependents " +
      "  version, modified_by, modified_time " +
      "  FROM people " +
      "  WHERE lastName = {0}";
    String sqlPrepared = DB.prepare(sql, lastName);
    IDbCommand comm = new OleDbCommand(sqlPrepared, DB.Connection);
    ResultSet rs = comm.executeQuery();
    List<Person> result = new ArrayList();
    while (rs.next()) {
      long rsId = rs.getLong(1);
      String rsLastName = rs.getString(2);
      String rsFirstName = rs.getString(3);
      int rsNumberOfDependents = rs.getInt(4);
      int rsModifiedBy = rs.getString(5);
      int rsVersion = rs.getInt(6);
      int rsModifiedTime = rs.getInt(7);
      Person person =
        new Person(rsId, rsLastName, rsFirstName, rsNumberOfDependents,
       rsModifiedBy, rsVersion, rsModifiedTime);
      result.add(person);
    }
    return result;
  }

  public static void update(Person person) {
    String sql =
      "SELECT version, modified_by, modified_time FROM people " +
      "  WHERE id = {0}" +
      "  LOCK IN SHARE MODE;"  //write lock the relevant rows
    String sqlPrepared = DB.prepare(sql, person.getID());
    IDbCommand comm = new OleDbCommand(sqlPrepared, DB.Connection);

    //Start a transaction before locking the rows
    SqlTransaction trans = DB.beginTransaction("updatePerson");
    ResultSet rs = comm.executeQuery();

    if (rs.next()) {
      int rsVersion = rs.getInt(1);
      //if the versions are inconsistent, throw an exception
      if (rsVersion != person.getVersion()) {
        String modifiedBy = rs.getString(2);
        String modifiedTime = rs.getString(3);
        trans.commit();  //"commit" to release the DB lock
        throw new ConcurrencyException("Row " + person.getID() +
               " in table person was modified by " +
               modifiedBy + " at " + modifiedTime);
      }
      //if the version are consistent, update the row
      else {
        sql = "UPDATE people " +
          "SET lastname = {0}, firstname = {1}, " +
          "    number_of_dependents = {2}, version = {3}, " +
          "    modified_by = {4} modified_time = {5} " +
          "WHERE id = {6}";
        Date currentTime = new Date();
        sqlPrepared =
          DB.prepare(sql, person.getLastName(), person.getFirstName(),
               person.getNumberOfDependenents(),
         person.getVersion() + 1, Session.getID(),
         currentTime, person.getID());
        comm.executeNonQuery();
        trans.commit(); //commit the transaction, releasing the lock
      }
    }
    //if this person does not exist, the record has been deleted
    else {
      throw new ConcurrencyException("Row " + person.getID() +
             " in table person has been deleted");
    }
  }
  ...
}
```

Note that the code that checks the version numbering is not specific to `Person` objects. A better design of this is to encapsulate the information about the latest modification in a superclass (one for all domain objects), and to implement a single method that checks version numbering on any domain object. This can be called from the respective update methods.

Figure 8.1 shows an interaction diagram for an instance of the optimistic lock pattern. In this figure, two sessions obtain data about a customer, and both edit it. The one that updates the two second receives a failure.

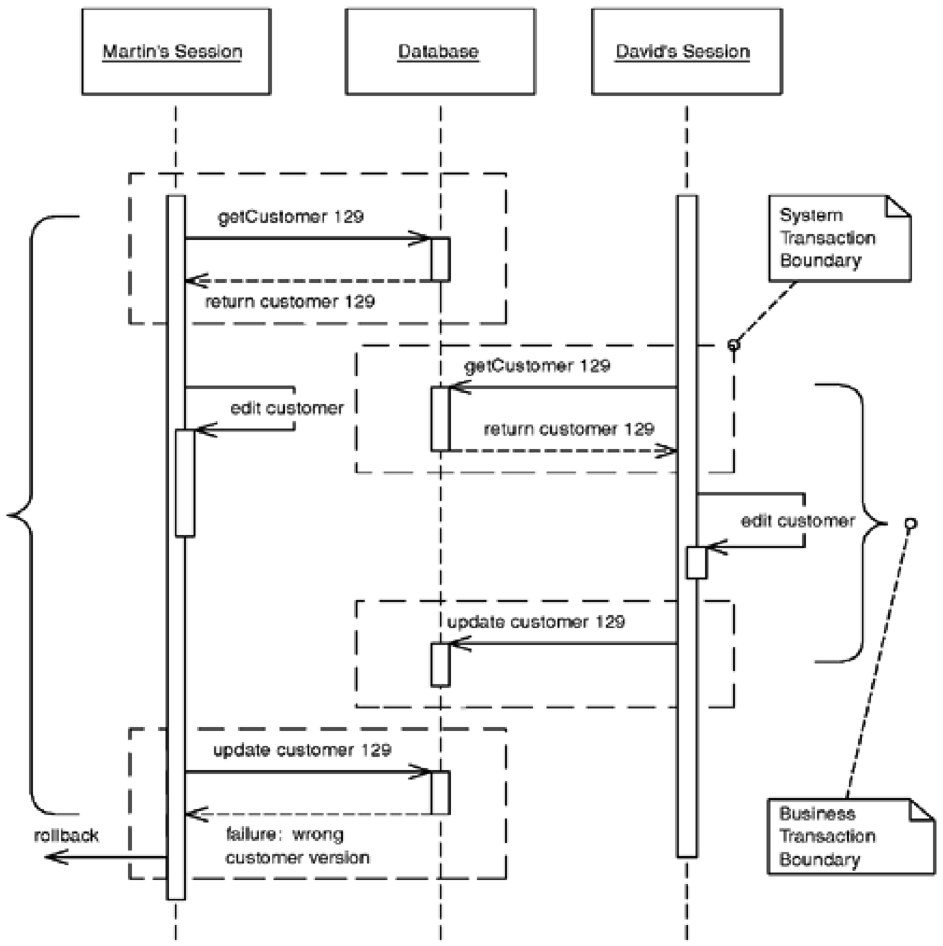


Figure 8.1: An interaction diagram for an instance of the optimistic lock pattern.

Pros

1. *Good liveness*: It allows concurrent access from other processes, except for the short period that the lock is obtained.
2. *Simplicity*: An extra column in the database and a simple check of version numbers will implement this pattern.

Cons

1. *Lost work/data*: If the entire business transaction is executed, and only then it is discovered that there is an inconsistency, it is possible that some or all of the work/data up until that point will be lost. This could be incredibly frustrating for a user.

8.2.2 Pessimistic offline lock

Pattern name	Pessimistic offline lock
Description	Prevents conflicts between concurrent business transactions by allowing only one business transaction at a time to access data.

Losing work/data using an optimistic lock may not be an option in some systems, due to the criticality of the data itself.

A *pessimistic lock* is a long transaction pattern that aims to prevent this by locking the required data over the entire business transaction. Thus, it prevents the "end of transaction" failures by avoiding conflicts all together.

Implementation

There are three steps required in a pessimistic lock:

1. *Determine the types of locks.*

   There are three types of locks that can be used:

   (a) *Exclusive write lock*: A process obtains an exclusive write lock, which allows other processes to read while the process has the lock. It only requires processes to acquire the lock if they want to write data. This type of lock does not prevent inconsistent reads.

   (b) *Exclusive read lock*: A process obtains an exclusive read lock. This means that an exclusive lock must be obtained to either read or write. This prevents inconsistent reads by requiring processes to acquire the lock if they want to read data. This results in less concurrent access than the exclusive write lock, but prevents inconsistent reads.

   (c) *Read/write lock*: This requires processes to acquire locks for either reading or writing data, however, multiple processes can acquire a read lock at the same time. A write lock can only be obtained if no process is reading, and a read lock can only be obtained if no process is writing. Only one process can obtain the write lock at one time.

   Determining which lock to use is a trade off of liveness vs. data integrity.

2. *Define a lock manager.*

   A lock manager is a component that keeps track of which processes are locking which data items. Typically, it is implemented as a simple map from locks to owners, where owners are represented via their session ID. There must be exactly one lock manager in a data-source layer, and code implementing business transactions should interaction only with lock managers, and never with locks themselves.

3. *Define the lock protocol.*

   The lock protocol determines what to lock, when to lock it, when to release it, and what to do when a lock cannot be acquired.

   A general policy is:

   (a) Lock at the start of the business transaction.
   (b) Lock based on the ID (or primary key) of the object/record.
   (c) Release the lock at the end of the business transaction.
   (d) In the case that a lock cannot be acquired, either abort early, or wait until it can be acquired.

Example

The first step is to implement a lock manager. Below is an excerpt from a simple write lock manager. This is implemented using a database table to maintain the locks and owner of the locks. This particular implementation assumes globally-unique primary keys over the entire database. If we assume keys are only unique to individual tables, the lock table must also keep track of the table on which a lock as been acquired.

In this particular lock manager, a table called *lock* maintains rows of pairs: the ID of the row to be locked, and the owner. The ID column is declared as a *lockableid* primary key, which means that, if we try to insert a new row to the *lock* table with an ID that already exists in the table, the insertion will fail. Therefore, to implement a lock manager, all we require is that the `acquireLock` method tries to insert the row `(id, owner)`, which will fail if `id` is already in the table.

Listing 8.2: **ExclusiveWriteManager.java**

```java
class ExclusiveWriteLockManager implements LockManager {

  public static ExclusiveWriteLockManager getInstance() {...}

  public boolean acquireLock(long lockable, String owner)
    throws ConcurrencyException {

    boolean result = true;

    //if the owner already has the lock, this should succeed
    if (!hasLock(lockable, owner)) {
      try {
        String sql = "INSERT INTO lock ({0}, {1})";
        String sqlPrepared = DB.prepare(sql, lockable, owner);
        IDbCommand comm = new OleDbCommand(sqlPrepared, DB.Connection);
        comm.executeQuery();
      }
      //an exception will be thrown if the ID is already in the table
      catch (SQLException e) {
        result = false;
      }
    }

    return result;
  }

  public void releaseLock(long lockable, String owner) {
    try {
      String sql = "DELETE FROM lock" +
        "WHERE lockableid = {0} and owner = {1}";
      String sqlPrepared = DB.prepare(sql, lockable, owner);
      IDbCommand comm = new OleDbCommand(sqlPrepared, DB.Connection);
      comm.executeQuery();
    }
    catch (SQLException e) {
      throw new SystemException("Unexpected error releasing lock on " +
        lockableid);
    }
  }

  public void releaseAllLocks(String owner) {...}
}
```

In the above listing, the `hasLock` method is a straightforward query to the lock table to see if the pair `(lockableid, owner)` is already in the table.

Next, we look at how to use the lock manager. In the following example, the front controller pattern, presented in Section 6.3.2, is used to handle requests. The abstract class `BusinessTransactionCommand` provides methods to signal whether a request is the start of a new transaction, or the continuation of an existing one. If it is a new transaction, all locks owned by the session are removed:

Listing 8.3: **BusinessTransactionCommand.java**

```
abstract class BusinessTransactionCommand implements Command {

  public void init(HttpServletRequest req, HttpServletResponse rsp) {
    this.req = req;
    this.rsp = rsp;
  }

  protected void startNewBusinessTransaction() {
    HttpSession httpSession = getReq().getSession(true);
    AppSession appSession = httpSession.getAttribute(APP_SESSION);
    if (appSession != null) {
      ExclusiveReadLockManager.getInstance().relaseAllLocks(appSession.
          getId());
    }
    appSession = new AppSession(getReq().getRemoteUser(),
        httpSession.getId(), new IdentityMap());
```

```java
    AppSessionManager.setSession(appSession);
    httpSession.setAttribute(APP_SESSION, appSession);
    httpSession.setAttribute(LOCK_REMOVER,
           new LockRemover(appSession.getId()));
  }

  protected void continueBusinessTransaction() {
    HttpSession httpSession = getReq().getSession();
    AppSession appSession = httpSession.getAttribute(APP_SESSION);
    AppSessionManager.setSession(appSession);
  }

  protected HttpServletRequest getReq() {
    return req;
  }

  protected HttpServletResponse getRsp() {
    return rsp;
  }
}
```

Finally, commands must be implemented to execute the domain logic, and to acquire and release the locks when required. Below are two related commands: one which reads customer information from a database and presents this to the user to allow editing; and one which saves the edits back to the database. In this example, the edit command starts the transaction, acquiring a lock on the row, while the save command writes the information back and releases the lock:

```java
class EditCustomerCommand extends BusinessTransactionCommand {

  public void process()  {
    startNewBusinessTransaction();
    Long customerId = new Long(getReq().getParameter("customer_id"));
    LockManager lm = ExclusiveReadLockManager.getInstance();
    if (lm.acquireLock(customerId, SessionManager.getSession().getId())) {
      Mapper customerMapper =
        MapperRegistry.getInstance().getMapper(Customer.class);
      Customer customer = (Customer) customerMapper.find(customerId);
      getReq().getSession().setAttribute("customer", customer);
      forward("/editCustomer.jsp");
    }
    else {
      //report that the row has been locked
      ...
    }
  }
}

class SaveCustomerCommand extends BusinessTransactionCommand {

  public void process() throws Exception {
    continueBusinessTransaction();
    Customer customer =
      (Customer) getReq().getSession().getAttribute("customer");
    String name = getReq().getParameter("customerName");
    customer.setName(name);
    Mapper customerMapper =
      MapperRegistry.getInstance().getMapper(Customer.class);
    customerMapper.update(customer);
    LockManager lm = ExclusiveReadLockManager.getInstance();
    lm.releaseLock(customer.getId(), SessionManager.getSession().getId());
    forward("/customerSaved.jsp");
  }
}
```

Figure 8.2 shows an interaction diagram for an instance of the pessimistic lock pattern. As with the optimistic lock example in Figure 8.1, two sessions try to load the same data, however, the pessimistic lock will report a failure when the second session attempts to read the data, rather than write it.

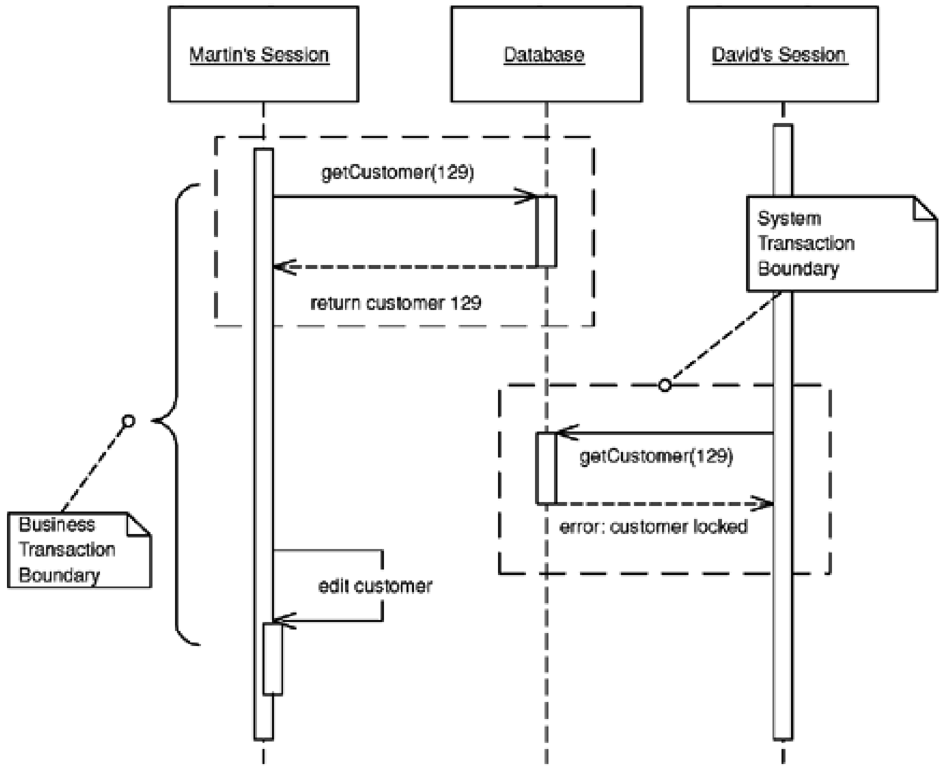


Figure 8.2: An interaction diagram for an instance of the pessimistic lock pattern.

Pros

1. *No work/data lost*: Failing up front means that no work/data is thrown away, unlike with the optimistic lock.

Cons

1. *Low liveness*: When a lock cannot be acquired, no processing can be done, forcing the user to wait or the transaction to fail.

Deadlocks

The dreaded deadlock occurs when processes are mutually waiting on data that each other has locked. For example, two process, P_1 and P_2 both want to write to tables T_1 and T_2, so they need to lock the tables to prevent others from writing to them. Now, if process P_1 locks table T_1 at the same time that process P_2 locks table T_2, both will then be unable to lock the other table. Both will wait for the other table to become unlocked, but neither will relinquish the lock on their table. A deadlock results where neither process will do anything. If these processes are performing tasks on behalf of a user, they will never return a result.

8.2.3 Comparing optimistic and pessimistic locking

When designing an enterprise system, the decision to make is whether to use optimistic or pessimistic lock. The essential difference between the two is that optimistic lock detects conflict, while pessimistic prevents it. The answer of which to use depends on the type of application. The following policy below provides some help in making this decision.

When to use	
Optimistic lock	**Pessimistic lock**
The chance of conflict is low	The chance of conflict is high
The cost of conflict is low	The cost of conflict is high
A business transaction fits into one request	

Of course, determining the chance of conflict for types of data may be non-trivial and highly subjective.

In general, optimistic locks are more straightforward to implement, but come at a cost in terms of losing data. Pessimistic locks are more complicated to implement, but preserve data. However, pessimistic locks lose some concurrency/liveness, which can be frustrating to the user.

8.2.4 Implicit lock

Pattern name	Implicit lock
Description	Allows a framework or layer supertype code to acquire offline locks.

A problem with the example shown in the pessimistic lock section is that the domain logic code is responsible for acquiring and releasing locks. Implementations of optimistic locks may also have the same problem. Generally speaking, if an item may be locked *anywhere*, then it must be locked *everywhere*. Forgetting to acquire a lock could result in loss of data integrity, while forgetting to release a lock could result in a lack of liveness.

The *implicit lock* pattern aims to prevent this by placing the code to acquire and release locks in a mapper layer, which sits between the domain logic code and the data-source code for accessing the database. Developers implementing the domain logic are not responsible for acquiring and releasing their own locks, therefore minimising the probability of forgotten locks.

Example

As an example, consider using a generic data mapper (Section 3.2.4) to retrieve rows from tables. The interface for a mapper is:

Listing 8.4: **Mapper.java**

```java
interface Mapper {
  public DomainObject find(Long id);
  public void insert(DomainObject obj);
  public void update(DomainObject obj);
  public void delete(DomainObject obj);
}
```

Mappers at the data-source layer implement this interface. However, instead of having the domain logic layer interact directly with mappers, we wrap the mappers in `LockingMapper` instances, which take care of the acquiring and releasing of locks:

Listing 8.5: **LockingMapper.java**

```java
class LockingMapper implements Mapper {

  private Mapper impl;
  private LockingManager lm;
  private String sessionId;

  public LockingMapper(Mapper impl) {
    this.impl = impl;
    this.lm = ExclusiveWriteLockManager.getInstance();
    this.sessionId = SessionManager.getSession().getId();
  }

  public DomainObject find(long id) {
    if (lm.acquireLock(id, sessionId)) {
      return impl.find(id);
    }
    else {
      //handle the error
      ...
    }
  }

  public void insert(DomainObject obj) {
    impl.insert(obj);
  }

  public void update(DomainObject obj) {
    if (lm.hasLock(obj.getID(), sessionId)) {
      impl.update(obj);
    }
    else {
      //handle the error
      ...
  }

  public void delete(DomainObject obj) {
    if (lm.hasLock(obj.getID(), sessionId)) {
      impl.delete(obj);
    else {
      //handle the error
      ...
    }
  }
}
```

Note that the lock is acquired for a lookup, and we must check that the current session holds the lock for an update and read. For an insertion, no lock was required (the row is new). The problem with this design is that the lock is not released automatically. There needs to be a signal that the business transaction is complete so the lock can be released.

Pros

1. *Low coupling*: The locking mapper de-couples the domain logic from the locking mechanism that is used.
2. *High cohesion*: Much of the implementation of the locking mechanism is contained in a single place, rather than spread out over the domain layer.

Cons

1. *Complacency*: While using an implicit lock allows developers to ignore the locking mechanics, they must still consider the effects of the locking. That is, they must consider that in when they issue a command that acquires a lock, it is not released until a corresponding update/delete method releases the lock. This prevents other processes from accessing this data in the meantime, so the developer should acquire the lock as late as possible, and release it as early as possible.

8.3 Further reading

- Much of the material from this chapter is based on Chapters 5 and 16 of the subject text (Fowler [3]), available as an e-book through the University of Melbourne library.

Chapter 9

Distribution

Many enterprise systems are *distributed*. By this, we mean that parts of the system run in different processes: on the same machine, on multiple machines, or (mostly) both. In this chapter, we will look at design principles and patterns for distributed architectures in enterprise systems. These principles and patterns provided designers with a way to meet a common design goal (and type of non-functional requirement): high performance.

The two patterns that will be discussed are: (1) *data transfer object*, which provides an object that can be transferred outside of local method calls; and (2) *remote facade*, which provides a coarse-grained interface for an object that can be accessed by other processes.

Learning outcomes A person familiar with the material in this chapter should be able to:

1. describe the problems relating to data communication between distributed process in enterprise systems;
2. critique the *first law of distributed object design*;
3. describe the two design patterns related to distribution;
4. apply these design patterns in enterprise applications; and
5. provide a rationale for the choice the presented patterns in an enterprise system.

9.1 The problem

The problem: How do we *efficiently* pass objects and their data between processes in a distributed enterprise system?

In a non-distributed system, passing objects and data is trivial: object references are shared as variables, which can be passed to other objects using method parameters and return values.

However, in a distributed system, an object reference on one machine, or even in another process on the same machine, does not refer to the same memory location. As a result, passing object references does not result in meaningful data being shared.

The obvious solution is to write code so that objects pass and return formatted data, rather than object references. This is a first step, however it has some pitfalls:

1. *Multiple calls*: In object-oriented design, it is considered a good principle to make objects small and fine-grained. That is, objects should hold only a small amount of related data types (high cohesion), and their interfaces should allow fine-grained access to this data. However, such an approach results in many small calls.

   Querying an object over a network (remote method calling) is extremely costly in compared to local method calling. As such, we want to cut down on the number of calls that are made over a network.

2. *Local and remote method calls*: Passing object references and using local method calls are much more efficient than making remote method calls and passing formatted copies of data. As a result, reference passing and local method calls should still be used within a single process. However, some objects will need to communicate with objects in other processes as well as within their own process. Preferably, the distinction should be transparent to calling code. Such objects need to support both local and remote method calls.

This section first discusses some principles regarding distributed architectures, and then presents two patterns that aim to mitigate the above problems.

9.2 Distributed architecture

In many enterprise systems, distribution is a necessary part of the architectural solution. The most obvious example of this is client-server architectures that require machine separation of clients and their servers. Other examples include using different processes or machines for the application and database servers, and different machines for a web server and its application server.

9.2.1 Distribution strategies

Often, distributed architectures are pursued within applications. Fowler [3] discusses his experience with software designers architecting systems with distributed objects in order to increase performance. In his experience, many such solutions hinder performance more than they help.

Fowler presents the example shown in Figure 9.1 of a typical distribution strategy that distributes a system by placing different components on different machines. In Fowler's experience, such a design will cripple performance, rather than improve it. The reason for this is that remote calls made between processes are an order of magnitude slower than local method calls. Remote calls made to another machine are an order of magnitude slower than remote calls on the same machine. The architecture in Figure 9.1 is likely to increase the number of inter-machine remote method calls, because the components are tightly coupled.

From the empirical evidence regarding this comes the *first law of distributed object design*: Do not distribute objects.

Instead of distributing objects, a solution that is generally more efficient is *clustering*. With a clustering solution, all classes are designed to work in a single process environment, and multiple copies of the

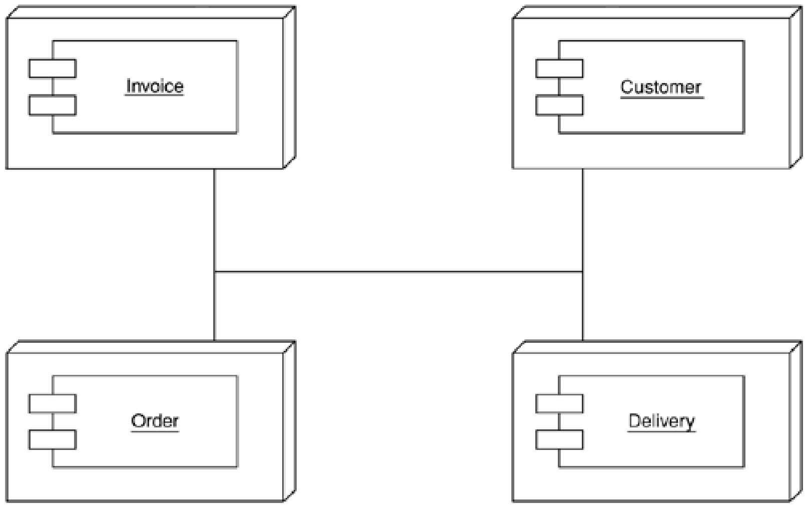


Figure 9.1: Distributing a system by its components.

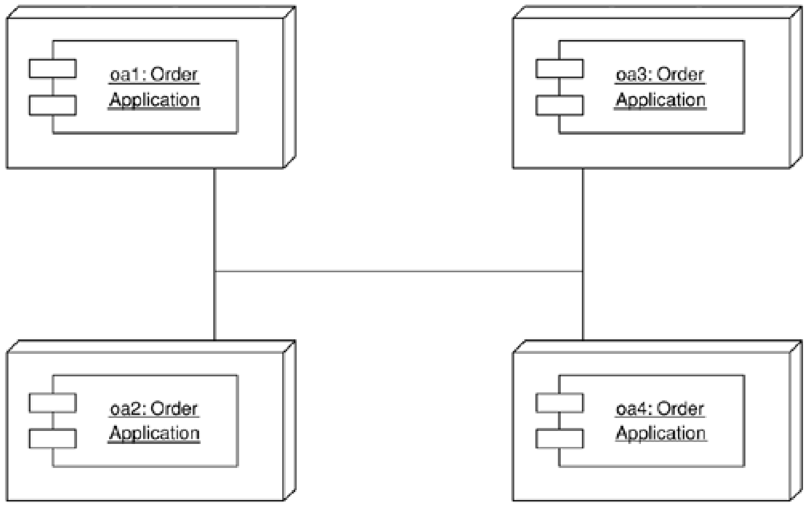


Figure 9.2: Distributing a system by clustering.

process are run on different machines. Figure 9.2 demonstrates how a clustering approach may look. The processes on each machine using local method calls for communication, and only require remote method calls when contacting another process.

Clustering is not always an option for a distributed system. For example, it does nothing for distributing our clients and servers, because they run different types of process, not copies of the same process. Clustering is generally only an option when our system needs to solve multiple instances of the same type of problem. Even then, it may require shared data sources (e.g. a shared database), which means the entire system is not a cluster.

9.2.2 Remote and local interfaces

As a result of the difference in efficiency between local and remote calls, class interfaces that will be accessed remotely should be designed differently to interfaces that will be accessed locally.

In object-oriented design, a good design principle is to make objects small and *fine-grained*. This means that a class is made up of a collection of related instance variables (high cohesion), and each instance variable has its own setter and getter methods. This allows small classes to be pieced together to form larger classes, and permits overriding of specific behaviour.

However, using fine-grained objects is not effective for remote method calls. For example, if we would like to obtain the values of five instance variables, we would need to issue five remote calls, which is costly. Instead, more *coarse-grained* interfaces are recommended. A coarse-grained interface will provide one call to get values for all five instance variables, which is more efficient for remote method calls.

A good strategy for dealing with this problem is to have fine-grained interfaces for local calls, and use coarse-grained interfaces to remote calls. This requires you to *know you distribution boundary*. That is, you have to know which objects are at the boundary of the distribution, and use coarse-grained interfaces there.

This does not seem like such a problem, however, many interfaces will be used both remotely and locally. Either you provide a fine-grained interface, a coarse-grained interface, or an interface with both. The problem with the latter is that you have more than one way to do the same thing, which is not good design.

In the following section, we look at some patterns for mitigating these problems.

9.3 The patterns

9.3.1 Data transfer object

Pattern name	Data transfer object
Description	An object that carries data between processes in order to reduce the number of method calls.

Most object-oriented languages are designed for fine-grained objects, and as such, return only single values. If a remote method is designed to be coarse-grained, it may have to return multiple values, which creates an issue. Simply placing them in a list or array eliminates any chance of design-time type checking.

A *data transfer object* assembles together multiple attributes from possibly multiple objects into a single object, which can then be serialised and sent over a network. The object contains only enough methods to get & set each attributed, and to serialise/deserialise the object.

Data transfer objects must be understood by both sides of the remote method call; e.g both the client and the server.

Implementation

Some important implementation issues must be considered for data transfer objects.

First, data transfer objects tend to be sent over a network, and must therefore be serialisable. As a result, the fields of a data transfer object should consist of only simple primitive types, such as integers, simple objects, such as strings and dates, or other data transfer objects.

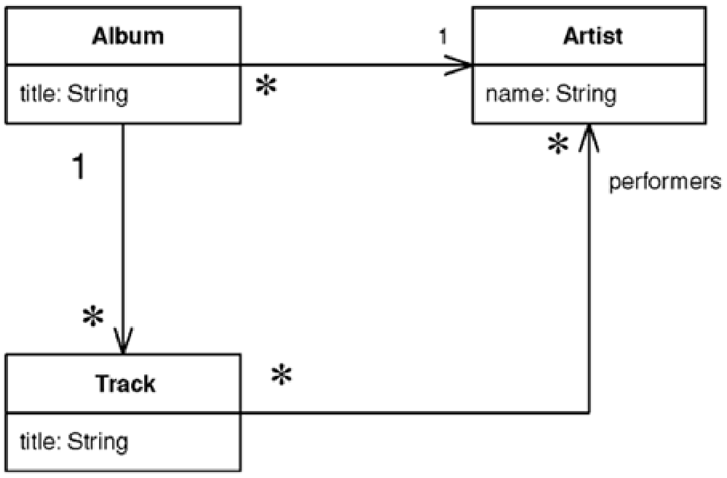


Figure 9.3: An excerpt of a class diagram.

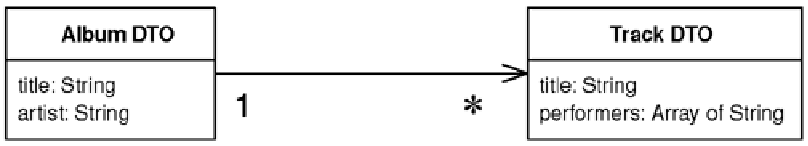


Figure 9.4: A class diagram for a data transfer object.

Second, they should be independent of the domain objects. That is, if a client makes a remote method call to a server, it should receive a data transfer object, and should not have to know the details of the domain logic classes to understand the result. Thus, the data transfer objects need to be simplified compared to the domain objects.

Finally, a decision must be made as the type of data transfer objects used. A simple solution is to create one type of data transfer object, such as a map, in which the key is a meaningful string (e.g. “city”). However, this loses the advantage of strong typing, and results in a lot of typecasting. A neater solution is to create a data transfer object per request or use case. This results in many types of data transfer object, however, these allow strong typing, and further, can be generated automatically.

Example

In this section, we use an example directly from Fowler [3], which extends the example of albums, tracks, and artists from Chapter 5. Figure 9.3 shows a class diagram in which albums have one artist name, a collection of tracks, and each track has a collection of performers.

Figure 9.4 shows a class diagram for a possible data transfer object for the classes in Figure 9.3. This design is much simplified. The album and artist name are in the same object, and the performers on an album are represented using an array. Note that this design implies that the corresponding use case is asking about specific albums, the artists, tracks and performers on these albums. If the use case was to request information about performers and which albums they had played on, an alternate design would be more suitable; for example, a performer data transfer class would be at the top of the object hierarchy, rather than the album data transfer class.

The first step is to implement the assembly of the data transfer object. Fowler [3] uses separate *assem-*

blers for each type of data transfer object. These are called by whatever object is handling the remote call, such as remote facade (Section 9.3.2). An implementation for an assembler of the album data transfer object is below:

Listing 9.1: **AlbumAssembler.java**

```java
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 albumDTO, Album subject) {
    List<Track> tracks = new ArrayList<Track>();
    Iterator<Track> it = subject.getTracks().iterator();
    while (it.hasNext()) {
      TrackDTO newDTO = new TrackDTO();
      Track thisTrack = it.next();
      newDTO.setTitle(thisTrack.getTitle());
      writePerformers(newDTO, thisTrack);
      tracks.add(newDTO);
    }
    albumDTO.setTracks(tracks);
  }

  private void writePerformers(TrackDTO trackDTO, Track subject) {
    List<Artist> performers = new ArrayList<Artist>();
    Iterator<Artist> it = subject.getPerformers().iterator();
    while (it.hasNext()) {
      Artist each = it.next();
      performers.add(each.getName());
    }
    trackDTO.setPerformers(performers);
  }
}
```

Once the `AlbumDTO` is assembled, it is likely that it has to travel across a network. For this, serialisation and deserialisation are required. The example below shows how to Java's XML API to implement serialise and deserialise methods for the `AlbumDTO` class:

Listing 9.2: **AlbumDTO.java**

```java
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;
  }

  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();
    }
  }

  static AlbumDTO readXml(Element source) {
    AlbumDTO result = new AlbumDTO();
    result.setTitle(source.getAttributeValue("title"));
    result.setArtist(source.getAttributeValue("artist"));
    List<Track> trackList = new ArrayList<Track>();
    Iterator<Element> it = source.getChildren("track").iterator();
    while (it.hasNext()) {
      trackList.add(TrackDTO.readXml(it.next()));
    }
    result.setTracks(trackList);
    return result;
  }

  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();
    }
  }
}
```

The `toXmlElement` method converts the object into an `javax.xml.bind.Element` object, which is part of the Java API. The `toXmlString` gets the `Element` instance using `toXmlElement`, and then serialised this into a string. The remaining two methods do the inverse: take a string and convert it into an `AlbumDTO` instance. This demonstrates the nice part of using an API to handle serialisation: the parsing from strings into `Element` instances is handled by a single line: `Element root = doc.getRootElement()`.

Libraries such as the *Java Architecture for XML Binding* (JAXB) can actually generate serialisation and deserialistion methods automatically for known classes, eliminating the need to manually implement specific toXML/fromXML methods for each class.

Pros

1. *Simplicity*: A simple collection of relevant objects is a straightforward solution that solves its problem.
2. *Compatibility (remote facade (see Section 9.3.2))*: Data transfer objects are a good fit for remote facades. Remote facade interfaces are generally required to return data from multiple fine-grained method calls, and data transfer objects can be used to assemble these into a single value. In fact, the remote facade example in Section 9.3.2 uses data transfer objects.

Cons

1. *Low cohesion*: A data transfer class tends to violate the design principle of high cohesion, in that it couples together data that may be only loosely related. However, no domain logic should be implemented in a data transfer class, so this is less of an issue that in a domain objects, for example.

9.3.2 Remote facade

Pattern name	Remote facade
Description	Provides a coarse-grained facade on fine-grained objects to improve efficiency over a network.

Objects on the distribution boundary may be queried by both local and remote objects. As such, it is sensible to provide both fine-grained and coarse-grained access to these objects. However, merging these into a single interface violates the design principles of fine-grained objects, and may also result in inconsistent use; e.g. using the coarse-grained method calls in a local object. Further, such an approach cannot be taken if we want a coarse-grained interface that brings together *more than one* fine-grained interface.

A *remote facade* is a facade [4] that provides a coarse-grained interface to a fine-grained object. It contains no domain logic, and exists only to translate coarse-grained method calls into fine-grained method calls and collate the results. Instances of the facade sit in the same process as their fine-grained counterparts.

Figure 9.5 presents example class and interaction diagrams showing how a remote facade may work. In this example, an object holds information about an address, such as the city, state, and postcode. To obtain all address information, three method calls are required, which is expensive for a remote method call. Instead, the facade packages these into a single call. The interaction diagram in Figure 9.5 demonstrates how the remote facade object issues the calls. In this diagram, the call to `getAddressData` is a remote method call from another process, perhaps on another machine, and the calls to `getCity`, `getState`, and `getZip` are made locally between the remote facade and its local object.

In more complex cases, the remote facade may collate more than one object. For example, the remote call may allow the retrieval of some person's data of birth, as well as their address, which are held in different objects.

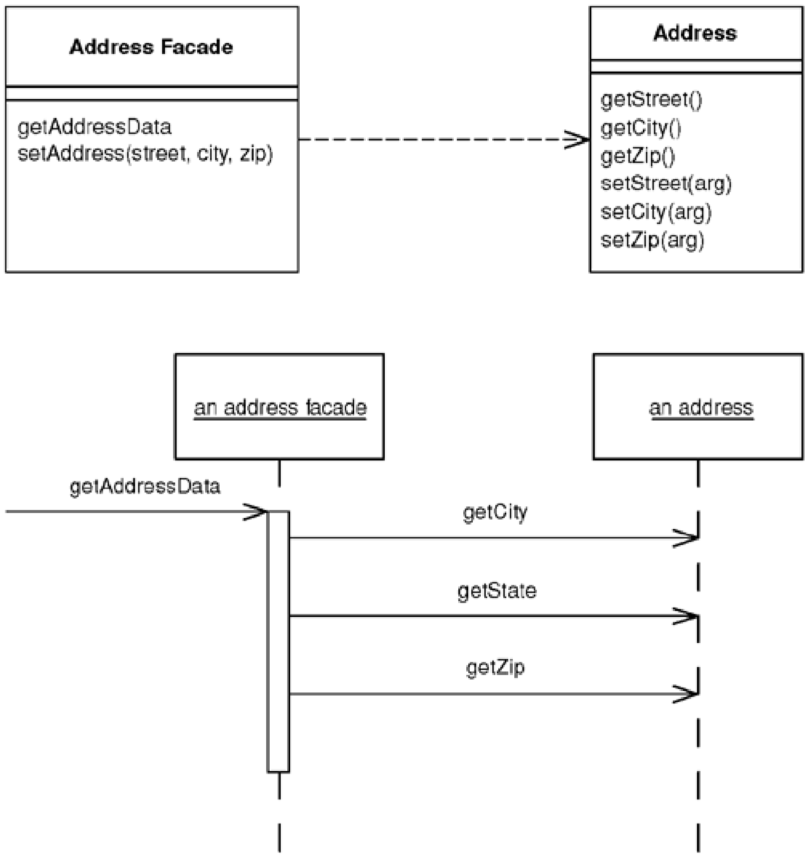


Figure 9.5: A class diagram and interaction diagram demonstrating how a remote facade call may work.

Implementation

An implementation decision that needs to be made is how coarse-grained the remote facade interface is. The most typical way to determine this is based on the use cases of the system: implement one remote facade per use case. Another common way is to have just one system-wide remote facade with multiple remote methods. This may be considered poor due to the low cohesion; however, each method is small and each does a similar task.

The most important thing to remember is not to place domain logic in the remote facade methods. Facades should be thin and provide only translation services. Any domain logic should be in the fine-grained objects, or placed into a transaction script (see Section 2.3.1).

Example

The following example uses a Java Enterprise Bean to construct an `AlbumDTO` instance. In this example, the album ID is used to retrieve the album via an album mapper (see Section 3.2.4), and the `AlbumAssembler` (Section 9.3.1) is used to assemble the instance into a single data transfer object:

Listing 9.3: **AlbumServiceBean.java**

```
class AlbumServiceBean {

  public AlbumDTO getAlbum(String id) throws RemoteException {
    return new AlbumAssembler().writeDTO(AlbumMapper.find(id));
  }
  public String getAlbumXml(String id) throws RemoteException {
    AlbumDTO dto = getAlbum(id);
    return dto.toXmlString();
  }
  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);
    updateAlbum(id, dto);
  }
}
```

Note that each method is small and simple: they simply delegate to the assembler object. One could implement the code for assembling objects in the remote facade, which would work for simple cases, however, separating out to an assembler object is good design for more complex cases. The above is an example of how a remote facade (or a facade in general) should be implemented: a list of short, simple methods with no domain logic — a thin layer.

Pros

1. *Lightweight*: Providing a thin, coarse-grained facade to a fine-grained object is an elegant and maintainable design, and is straightforward to understand once the granularity of the remote facade interface is defined.

2. *Compatibility (domain model)*: This pattern is particularly compatible with the domain model pattern (see Section 2.3.2). It is not typically used with the transaction script or table module patterns, which tend to imply coarse-grained data access at all levels.

Cons

1. *Determining granularity*: There is no easy solution to the problem of deciding granularity. While having one remote facade per use case provides a simple mapping, it may result in code duplication across different remote facades. Using assemblers helps to reduce this.

9.4 Further reading

- Much of the material from this chapter is based on Chapters 7 and 15 of the subject text (Fowler [3]), available as an e-book through the University of Melbourne library.

Chapter 10

Security

So far we have looked at architectural design for functional requirements in enterprise systems. That is, we have discussed patterns for helping us to assign functional requirements to different parts of the system. In this chapter, we will consider design for a type of *non-functional* requirement: security.

Just like functional requirements, non-functional requirements need to be *engineered* into the system from day one. It is not possible to simply measure non-functional qualities after a system has been built, and then retrospectively fit those qualities in should the system fail to meet them. We may get lucky once or twice, but over multiple projects, we will fail much of the time.

In this chapter, we look at design patterns for security. In particular, we will look at patterns for security at the presentation layer. We focus on the presentation layer in this subject because it is the most frequent point of attack for an adversary attempting to locate security weaknesses in an enterprise application.

In this chapter, we will look at four patterns related to security: 1) *authentication enforcer*; 2) *authorisation enforcer*; 3) *intercepting validator*; and 4) *secure pipe*.

A person familiar with the material in this chapter should be able to:

1. describe the issues related to presentation-layer security in enterprise applications;
2. describe the four design patterns related to presentation-layer security, and what problems they aim to solve;
3. apply the design patterns in enterprise applications;
4. critique the choice of one of the four design patterns on a particular application; and
5. provide a rationale for the choice of one the presented patterns in an enterprise application.

10.1 The problem

The problem: How do we organise security-related code in an enterprise system that supports correct access to that system or data within that system, and ensures data integrity and confidentiality?

Solving the problem is not simply about having a username & password login for every user. There are several problems that we need to solve. In this chapter, we will look at four of the most common:

1. *Authentication*: We need to verify that each request sent is from an authenticated entity; that is, a user is who they say they are.
2. *Authorisation*: We need to verify that each request is being performed on behalf of an authorised entity; that is, a user is permitted to do what they are trying to do.
3. *Validation*: We need to validate that each request is validated as well-formed and non-malicious content; for example, requests with injected SQL queries should be rejected.
4. *Integrity and confidentiality*: Some data sent over a network may contain sensitive or private information. We need to take measures to minimise the risk of that data being read or spoofed by external parties.

In this chapter, we look at four patterns: one to help solve each of the four problems above.

10.2 The patterns

10.2.1 Authentication enforcer

Pattern name	Authentication enforcer
Description	A centralised authentication enforcement that performs authentication of users and encapsulates the details of the authentication mechanism.

Authentication of users is one of the primary aspects of security. If a user presents themselves as a particular identity, we must take steps to verify that the person really is who they say that are. Note that a user may be an external (sub-)system — not just a human user.

Generally, authentication mechanisms consist of at least two pieces of information: one unique, public (or semi-public) piece that establishes a claim of the identity of the user, such as a username, email address, or account number; and one private piece that verifies that claim of identity, such as a password or one-time key. On top of this, a *mechanism* is required to authenticate this. For example, a simple database containing usernames and passwords can be looked up, or, in the case of verification based on email addresses, the mechanism may send an email to the email address requiring confirmation.

The choice of an authentication mechanism is mostly based on business requirements, and is subject to change. Further, authentication may be required in several parts of the system, and the authentication mechanism may be different in some of these different parts.

The *authentication enforcer* patterns specifies a centralised authentication mechanism that handles authentication of all users of all actions of the presentation layer, and encapsulate these details.

Implementation

There are four general strategies for implementing this pattern:

1. *Container authenticated strategy*: This is where the framework's underlying container takes responsibility for authentication, such as J2EE's specification that mandates support for HTTP authentication, form-based authentication, digest-based authentication, and client-certificate authentication. These do not define the mechanism itself, but they define how to retrieve credentials from

the user. The specific J2EE implementation of these specifications define how the authentication mechanism works.

2. *Third-party provider strategy*: This is where the authentication enforcer delegates responsibility to a third-party (external) component, and simply creates the mechanism to hold the authentication information.

3. *Use an external library*: For example, the *Java Authentication and Authorization Service* (JAZZ) is a Java-based implementation of the standard *Pluggable Authentication Module*.

4. *Use a proprietary approach*: Implement your own.

Using a proprietary approach is likely to increase the risk that a vulnerability can be used to gained access.

Example

Figure 10.1 shows a class diagram for an example of the authentication enforcer pattern, taken from Steel and Nagappan [9]. In this figure, a `RequestContext` object contains information about the request, including the user's credentials (e.g a username and password), and a `Subject`, which is a session object that is used to represent an authenticated user for subsequent queries. A `Subject` is not created/updated if the authentication fails. The `AuthenticationEnforcer` object is responsible for reading the credentials from the `RequestContext`, attempting to authenticate the user based on those credentials, and creating a `Subject` object if the authentication is successful.

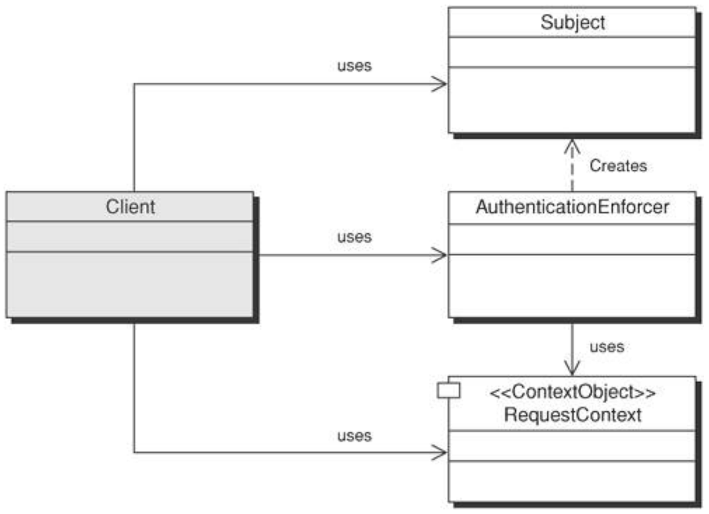


Figure 10.1: A class diagram for the authentication enforcer pattern.

Figure 10.2 shows the corresponding interaction diagram for this pattern. In this example, the client, such as a front controller or application controller (see Chapter 6), creates the `RequestContext` instance, and simply requests authentication from the authentication controller. The authentication controller will lookup the known credentials from some data source (`UserStore`), and will create a `Subject` object if the credentials in the `RequestContext` match the known credentials. Subsequent requests use the `Subject` instance to check whether the user is already authenticated, rather than requiring the user to authenticate themselves each request.

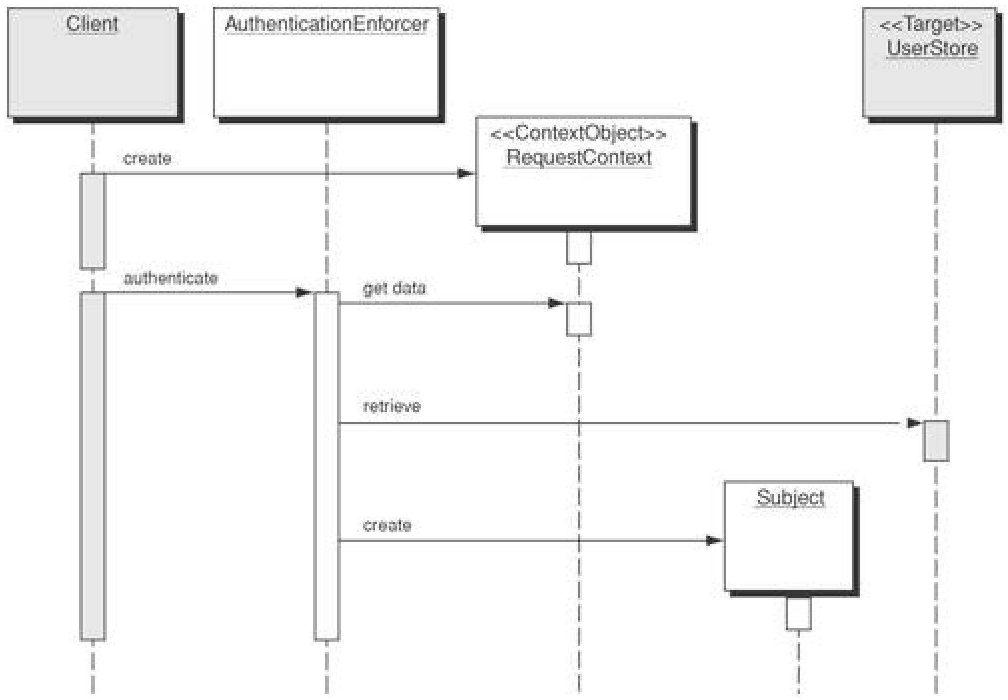


Figure 10.2: An interaction diagram for the authentication enforcer pattern.

Pros

1. *Improved security*: By centralising the authentication, this pattern reduces the number of places within the presentation layer that the mechanism is accessed, and therefore reduces the chance of security holes due to incorrect usage.
2. *Maintainability*: Encapsulating the security policy in a centralised module supports migration to different types of authentication; e.g. if business requirements change.
3. *Reuse*: Encapsulating the code consolidates authentication into a single place, which both reduces duplicate code within the system via reuse, and supports reuse in other systems.

Cons

1. *Making it usable*: Experience suggests that if the authentication mechanism is difficult to use or does not provide the functionality as expected, developers will add their own authentication mechanisms directly into their applications. This increases the risk of vulnerabilities that can be exploited to gain access.
2. *Confidentiality*: This is not so much a disadvantage, but just a reality check. During authentication, some sensitive information, such as a password, will be sent across a network, and is therefore at risk of being read by an attacking. As such, this pattern requires the use of some form of confidentiality pattern to keep these details secure (see the *secure pipe* pattern in Section 10.2.4).

10.2.2 Authorisation enforcer

Pattern name	Authorisation enforcer
Description	A centralised access controller that performs authorisation based on standard security API classes.

An authenticated user should not necessarily have access to all parts of an application. For example, an end user should not be able to access administrator functionality. Instead, the application must authorise that a user can perform a function.

Throughout an application, many classes may want to verify that a request has come from a user that is authorised to perform an action; e.g. to call a method that accesses certain data. Just like the authentication enforcer, spreading this out over the different parts of the application can lead to problems.

The *authorisation enforcer* patterns provides a generic, centralised encapsulation of authorisation mechanisms by defining a standard way for controlling access to particular functions or data within a system. It provides more fine-grained access than simply restricting URL access to users, supporting the ability to restrict aspects such as the links, buttons, or data displayed based on the user permissions.

To verify that a particular request has authorisation to perform a particular function on behalf of a user, the identity of that user may have to be established first. Therefore, any authorisation mechanism will generally be partnered with an authentication mechanism.

Example

Figure 10.3 shows a class diagram of a use of the authorisation enforcer pattern, taken from Steel and Nagappan [9]. There are several classes in this context. The `Subject` and `RequestContext` are as in the authentication enforcer pattern. The `SecurityBaseAction` is an abstract base class for all security actions (a pattern that we will not discuss further in these notes), while the `AuthorisationProvider` is a class that implements the authorisation logic. The `PermissionsCollection` is an application-specific class for storing permissions, with methods for each type of permission that is required in the system.

Figure 10.4 shows the corresponding interaction diagram for this pattern. This model assumes that `SecureBaseAction` object has already authenticated the user, and has placed that user's `Subject` object into the session state. From the perspective of the `SecurityBaseAction` object, it must get the `Subject` instance for a user, and then only makes two calls: one to authorise a user to perform a specific action; and one to check whether a user has authorisation to perform a specific action. The remaining logic is implemented behind the `AuthorisationEnforcer` interface.

When a request for authorisation comes in, the `AuthorisationEnforcer` object attempts to authorise this with the `AuthorisationProvider`. The `AuthorisationProvider` creates the necessary permissions collection by reading the user permissions from a data source, and updates the `Subject` object with these. Now, the `Subject` instance holds all information about what permissions a user has.

When the `SecurityBaseAction` object later queries the `AuthorisationEnforcer` object asking whether a user is authorised for a particular action, the `AuthorisationEnforcer` object gets the user's permissions, and queries the `PermissionsCollection` object to see if the retrieved permissions are sufficient to grant access to the user.

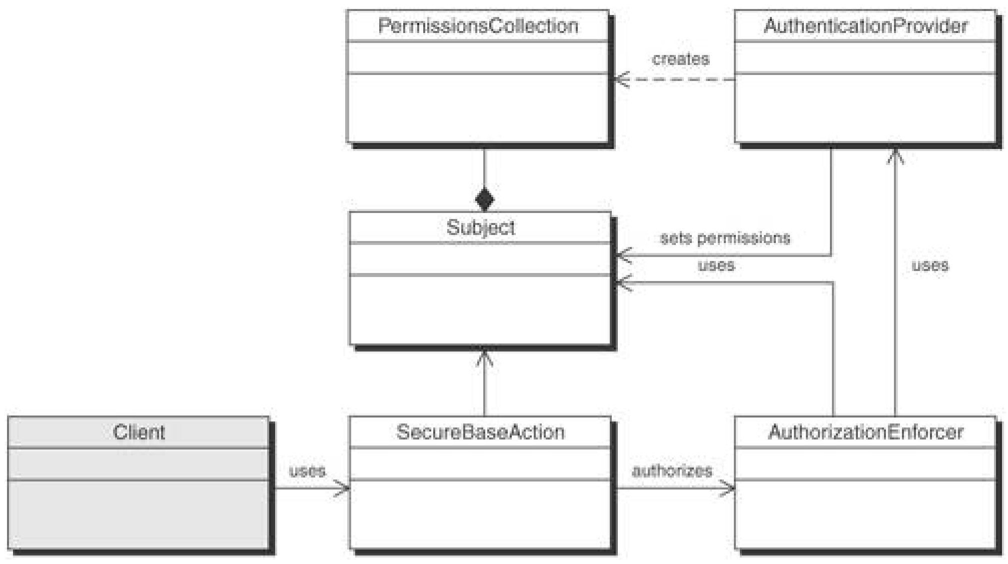


Figure 10.3: A class diagram for the authorisation enforcer pattern.

Pros

1. *Reuse*: Developers are encouraged to encapsulate the access control policy, eliminating the chance for repetitive code, and allowing greater reuse through encapsulation of disparate access-control mechanisms through common interfaces.
2. *Maintainability*: Encapsulating the security policy in a centralised module supports a change to new authorisation policies and makes it more straightforward to change permissions/roles.
3. *Promotes separation of responsibility*: Partitions authentication and authorisation responsibilities, insulating one from the other.

Cons

1. *Complexity*: Implementing anything but the most trivial policy requires a thorough understanding of either the security APIs in a language/framework, or a custom solution, which is complex on its own.

10.2.3 Intercepting validator

Pattern name	Intercepting validator
Description	A component that cleanses and validates data prior to use in a system using dynamically loadable validation logic.

Several prominent strategies for attacking enterprise systems involve the use of requests containing bogus data or malicious code, such as embedded SQL commands. For example, consider a request that allows a user to lookup their information based on their email address. The email address is entered via a text box on a web page, a request is submitted, and the query is presented to the database as:

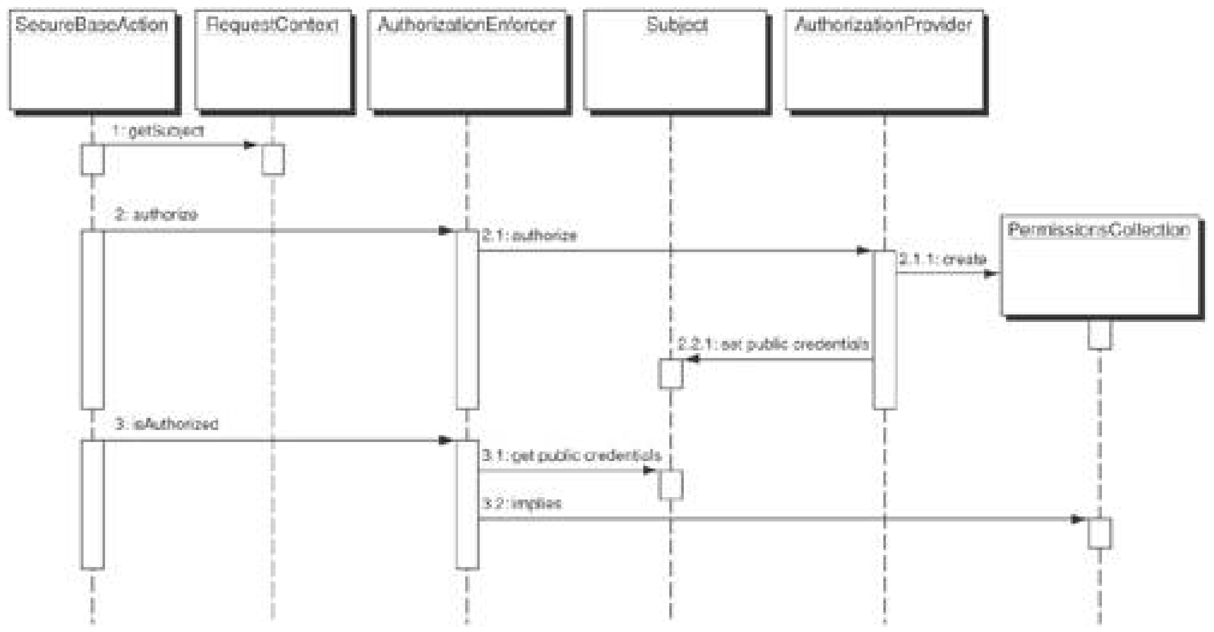


Figure 10.4: An interaction diagram for the authorisation enforcer pattern.

```
SELECT *
FROM really_personal_details
WHERE email = '$EMAIL_ADDRESS'
```

in which `$EMAIL_ADDRESS` is the text copied from the text box.

If the text `myemail@somedomain.com` is entered, then this works fine. However, if the user is aware of such a vulnerability, they may instead enter: `myemail@somedomain.com' OR '1=1`. If the developers have not been careful, this may result in the following query being issued:

```
SELECT *
FROM really_personal_details
WHERE email = 'myemail@somedomain.com' OR '1=1'
```

The `WHERE` clause evaluates to $true$, and therefore, the user will be able to gain unauthorised access to the data of other users.

One naïve approach around this problem is to tell our developers to check for them. For example, database management systems often have helper function to prepare SQL statements, which prevents these incidents from occurring; e.g. my stripping out all occurrences of `'`. However, we cannot leave this up to individual developers, because some may not be particularly security-aware, and will likely miss some validations. Further, if we want to add a new validation, we have to update the code in several places.

The *intercepting validator* pattern specifies that all input should be cleansed and validated before its use in a system. A series of pluggable *filters* are applied to the input, and are typically not tied to any particular business rules. An intercepting validator is performed on the server-side, which prevents any spoofing attacks; e.g. SQL injection after the query is sent from the client.

Example

Figures 10.5 and 10.6 show class and interaction diagrams for an instance of the intercepting validator pattern. Prior to any requests, the intercepting validator creates a collection of validation objects; in this example, objects for validating parameter correctness and cleansing SQL queries. When a request for a specific resource, which we will call the *target*, comes in, the `SecureBaseAction` object is called by a client, and it the responsibility of this action object to call the intercepting validator. The intercepting validator calls the necessary validator objects (e.g. commands implemented using the command pattern [4]), and hides the details of which checks it is performing from the `SecureBaseAction` object. The `SecureBaseAction` object receives the validated input, and sends this onto the target (the client-requested resource).

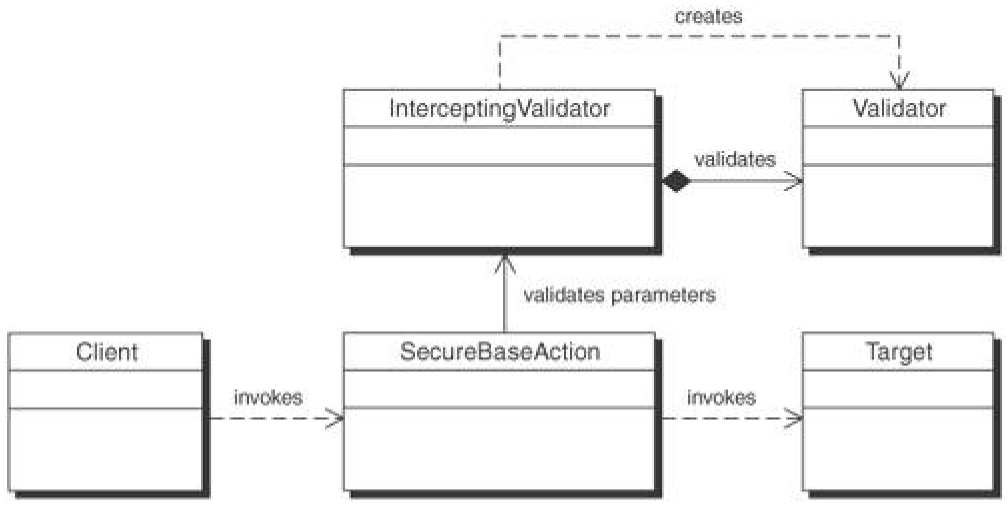


Figure 10.5: A class diagram for the intercepting validator pattern.

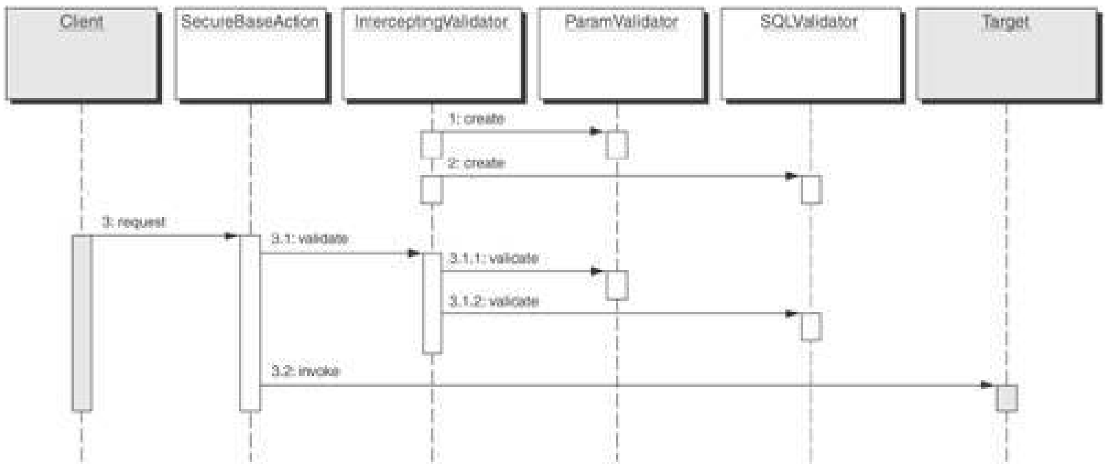


Figure 10.6: An interaction diagram for the intercepting validator pattern.

Pros

1. *Centralisation*: All security validations are centralised (although multiple instances can be used to ensure no bottleneck), supporting a standard way to address validations.

2. *Low coupling*: This decouples security validations from both the presentation and domain logic.

3. *Flexible*: This pattern provides a straightforward way to add validations. Further, if implemented correctly, these can be deployed dynamically, eliminating the need to a redeployment of the system.

Cons

1. *Complexity*: A series of validator rules can be highly complex, and may not be needed for applications that are less security-critical.

2. *Performance*: The intercepting validators may apply unnecessary filters to data that does not require them, affecting performance. Further, the validators may be required to read lockable session data, which could lead to concurrency issues such as long wait times or deadlocks.

10.2.4 Secure pipe

Pattern name	Secure pipe
Description	A generic and standardised way to ensure the integrity and privacy of data sent over a network.

Systems that send data over a network are open to security vulnerabilities, such as eavesdropping and spoofing. This is a problem that many Internet-based applications face. Applications that send sensitive data will need to address these vulnerabilities by sending the sensitive data over a secure network.

The *secure pipe* pattern specifies a generic and standardised way to send requests over a secure network using encryption/decryption. The pattern is not application specific, which reduces the complexity of the solution, and also allows instances of it to be reused over multiple applications. Further, a secure pipe can delegate encryption and decryption to hardware accelerators that are designed specifically to secure data.

Implementation

Several strategies can be used to implement the secure pipe pattern. The most common for web-based applications is to use web-server-based Transport Layer Security (TSL) or its predecessor, Secure Sockets Layer (SSL). These are specifications that are supported by almost all web browsers, and which use asymmetric cryptography to negotiate a symmetric key for data exchange.

Hardware-based accelerator cards are available to TSL and SSL cryptography. When a new session is requested, the server delegates the setting up of keys etc. to the accelerator card, as well as the encryption and decryption of data, which are expensive processes to execute using software.

Example

Figure 10.7 shows a class diagram for an example of the secure pipe pattern, taken from Steel and Nagappan [9]. This model shows the simple nature of the pattern. A client, instead of communicating with objects on the server directly, communicate via a secure pipe. In fact, the secure pipe is present on both the client side and server side of an application. The application on the server creates the pipe for the data to be sent, and then the sending side encrypts data and the receiving side decrypts it. These components are generally standard, such as TSL and SSL libraries.

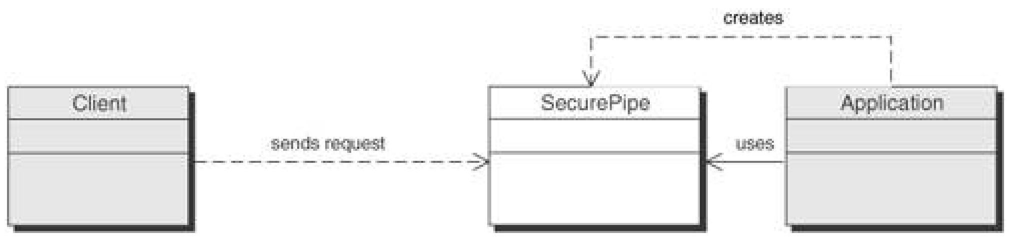


Figure 10.7: A class diagram for the secure pipe pattern.

Figure 10.8 shows the corresponding interaction diagram for this pattern. To interact with the server application, the client must first send a login request. However, before any login details are sent, the server creates a secure pipe, which then negotiates the security parameters with the client; e.g. SSL will be use for encryption and decryption. Any requests sent after this by the client will go to the client-side of the secure pipe, will be processed (encrypted) by the pipe, and then forwarded on to the server side of the secure pipe. The server side component decrypts the data and forwards this to the application. The secure pipe is destroyed when the client sends a logout request, or its session times out. Data sent from the application back to the client can also be sent over the secure pipe. The application knows nothing about the secure pipe, and receives requests and send responses as if no the pipe is not in place.

Pros

1. *Integrity and confidentiality*: A secure pipe will decrease the likelihood of a successful attack from an adversary.

2. *Promotes interoperability*: Using a standardised approach with standardised infrastructure components such as this achieves interoperability with other clients.

3. *Low coupling*: This patterns separates the cryptographic algorithms from the business logic.

Cons

1. *Performance*: Encrypting and decrypting data is a costly operation that can have performance effects. A secure pipe should only be used for requests in which integrity and/or confidentiality are necessary. Delegating cryptographic operations to hardware can mitigate this.

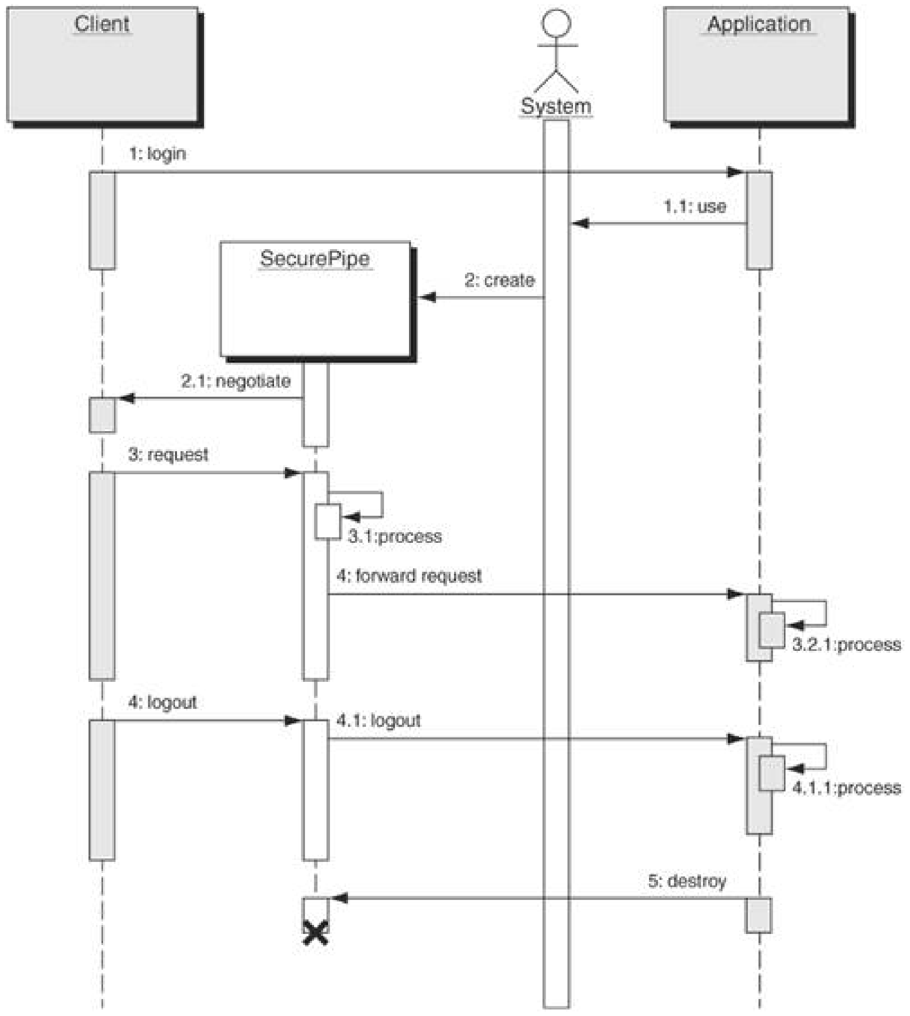


Figure 10.8: An interaction diagram for the secure pipe pattern.

10.3 Further reading

- Much of the material from this chapter is based on Chapter 9 of *Core Security Patterns: Best Practices and Strategies for J2EE, Web Services, and Identity Management* by Steel and Nagappan [9] (see `http://www.coresecuritypatterns.com/`).

Chapter 11

Performance

Minimal use of resources, such as CPU, memory, storage, and time is a key quality that encourages users to adopt and continue using systems. In this chapter, we look at three design principles for improving the use of resources: *Bell's principle*, *pipelining*, and *caching*. These three principles are techniques that can be used to improve performance of systems.

Learning outcomes A person familiar with the material in this chapter should be able to:

1. describe how the memory hierarchy of computer system relates to performance;
2. describe the the principles related to performance, and use them in enterprise application designs;
3. critique the choice of these principles for a particular application; and
4. provide a rationale for the choice of any of principles in an application.

11.1 The problem

The problem: How do we use resources such as memory and time more efficiently in software systems?

Many design principles are aimed at performance, and in fact, we have already seen several patterns that aim to improve performance, mostly time and memory; e.g., lazy load (Section 4.2.3), data-transfer object (Section 9.3.1), and offline locking solutions (Chapter 8). The principles in this chapter are more general design principles that are commonly used in enterprise systems.

11.1.1 Why care about performance?

The speed of a typical CPU has increased by more than a factor of a thousand in the last two decades; the speeds of the memories and disks have also improved significantly, although nowhere near as much as the speeds of CPUs. As a result of these advances, more and more programs are "fast enough" even if you do not try to improve their performance. However, the tasks people expect computers to solve have also increased greatly in their size and scope, so there are still many programs which will not run fast enough unless you invest some effort in performance engineering.

More efficient software also allows organisations to use their (leased/virtualised/private) hardware more effectively by allowing the same work to be done by fewer servers, reducing cost and energy consumption.

Software companies therefore compete on performance as well as functionality, so performance is important for just about all commercial software.

11.1.2 Performance engineering

When should you care about performance?:

1. *During requirements elicitation:* You cannot design a program to be “fast enough” or “small enough” if you do not know what “fast enough” or “small enough” mean.
2. *During design:* Some of the actions that a design calls for have minimum execution times. If performance is important, then it must be addressed at the design stage. If it isn’t addressed then, then achieving acceptable performance may require redoing large parts of the design, which in turn will require redoing large parts of the implementation.
3. *After debugging has been done:* Engineers should measure where the program is spending its time, and try to speed up any parts that consume nontrivial amounts of time.
4. *NOT during implementation:* Trying to speed up code that has not been debugged yet, or that consumes trivial amounts of time, is wasted effort, as the code will change in many places.

Premature optimisation is the root of all evil

The famous quote, attributed to Donald Knuth: “*Premature optimisation is the root of all evil*”, is an exaggerated aphorism that is intended to mean that any time used to optimise code before profiling is likely to be waste, because bottlenecks are too difficult to predict.

For example, reducing the time spent in a particular function from 10% of the runtime of the whole program to 6% is worthwhile, while reducing the time spent in a particular function from 0.01% of the runtime to 0.006% has an effect that is so small that it is usually not even measurable.

The problem with premature optimisation is that you do not know in advance which class a given function falls into.

If you guess that the fraction of the runtime spent inside a given function is closer to 10% than 0.01%, you are very likely to be wrong, simply because only a few functions can be close to 10%, while hundreds can be close to 0.01%.

11.2 The principles

The main part of design for performance is looking out for operations that are inherently slow (operations that you have no way to speed up), and either removing them from the design whenever possible, or replacing them with something faster (and probably simpler).

Inherently slow operations include:

- network message exchanges;
- spinups of sleeping disks;
- disk accesses;
- bulk memory copying;
- synchronisation;
- system calls; and
- process creation.

In this section, we will look at three simple principles for designing for performance: *Bell's principle*, *pipelining*, and *caching*.

11.2.1 Bell's principle

Principle name	Bell's principle
Description	Simple designs often show better performance than more complicated designs.

Gordon Bell's famous quote is:

> "*The cheapest, fastest and most reliable components of a system are the ones that aren't there.*

In other words, the best way to prevent a system component from slowing the whole system down is to eliminate that component.

Nonexistent components:

- take no time to execute;
- require no memory to store;
- cost nothing to write;
- take no time to debug;
- cannot fail; and
- do not consume any power.

Bell's principle is an argument in favour of the proposition that the best way to design for performance is to design for simplicity. A simpler design is more likely to be both correct and fast, compared to a more complex design.

11.2.2 Pipelining

Principle name	Pipelining
Description	Pipelining turns a problem whose performance is limited by *latency* into a problem whose performance is limited by *throughput*.

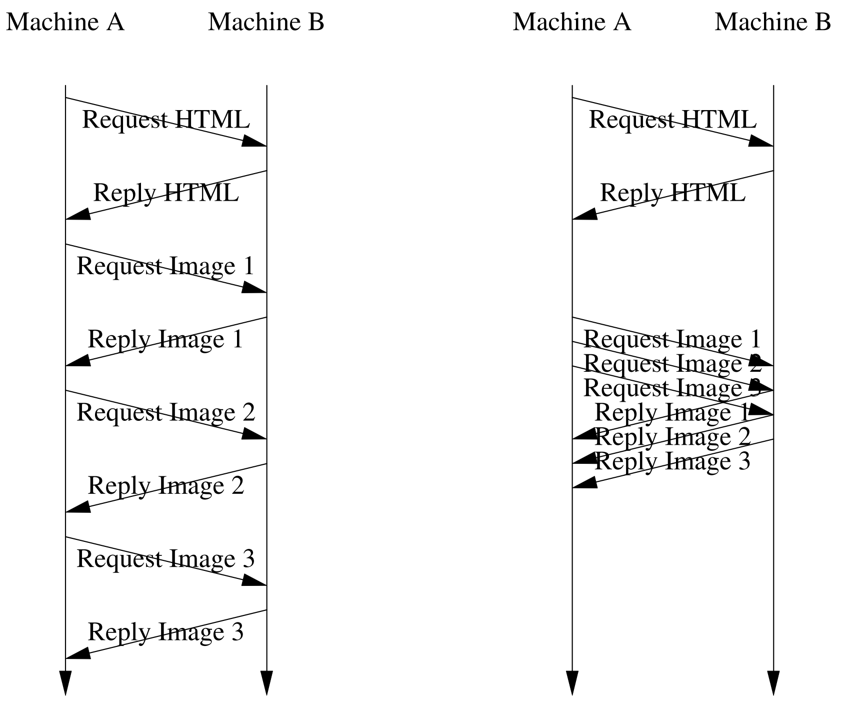
In some cases, you do not need the result of one task to decide what to do next.

For example, a browser sends a request for a HTML page and receives back the HTML content. In that content is a series of items, such as images, that the browser needs to request to present the complete page. Once the browser gets back the HTML file, it knows what items the page refers to and send requests for them. However, it does not need to receive image 1 before it requests image 2, because the context of image 1 is *independent* of image 2.

The technique of issuing each request *without* waiting for the result of the previous request is called *pipelining*.

Example

Figure 11.1 shows the example of receiving the HTML content and then requesting images one by one, or using pipelining. In this example, the application sends of multiple independent requests, resulting in a faster turnaround for loading the page.



(a) Issuing requests one at a time

(b) Issuing requests concurrently using pipelining

Figure 11.1: Two ways of retrieving images for a web page.

Why use pipelining?

Pipelining turns a problem whose performance is limited by *latency* into a problem whose performance is limited by *throughput*.

Latency: Latency is the length of time needed to complete a task.

Throughput: Throughput, also called *bandwidth*, is the number of tasks that can be completed in a given amount of time.

If each task in a sequence depends on the previous one, then improving (increasing) throughput requires improving (reducing) latency.

However, if tasks do *not* depend on each other, then one can use parallelism to improve throughput even if there is no way to improve latency. Pipelining is one form of this.

In networks, the speed of light imposes a hard floor on latency, but bandwidth has been improving at a phenomenal rate.

Pros

1. *Decreased time*: Using pipelining can decrease the throughput (e.g. over a network), thereby decreasing the time that users (including other systems) wait for access to resources.

Cons

1. *Dependencies*: The main limitation on pipelining is the availability of independent requests.

   If the time required for a request/reply cycle (the delay) is N seconds, and the bandwidth between the two machines is M bits/second, then keeping the pipeline full requires that at every point in time, the amount of data sent or received by independent requests exceeds the bandwidth delay product, M bits/second * N seconds, which is M*N bits.

   In most enterprise applications, we will run out of independent tasks long before this limit achieved. Designers should try to eliminate as many dependencies between tasks as possible during the design stage.

2. *Complexity*: The complexity of the resulting code is increased slightly (e.g. in some cases to keep track of the ordering of requests), although not to a point that pipelining should not be considered wherever possible.

11.2.3 Caching

Principle name	Caching
Description	Transparently store data in a faster-access location to decrease the time is takes to access the data.

Probably the most widely used speedup technique is *caching*.

The idea is that once you have read a piece of information from a slower medium into a faster medium, you keep it in the faster medium for a while. If you need that same piece of information again, you look for it in the faster medium first, and access the slower medium only if you have to.

The basic algorithm for caching is as follows:

Listing 11.1: **Pseudocode for caching**

```
if (data_is_in_faster_medium(key)) {
    value = get_value_from_faster_medium(key);
} else {
    if (there_is_no_room_in_faster_medium()) {
        make_room_in_faster_medium();
    }
    value = get_value_from_slower_medium(key);
    add_to_faster_medium(key, value);
}
```

The memory hierarchy

The hierarchy of memory for storing data is important in the concept of caching. Figure 11.2 presents a model of the hierarchy of computer memory.

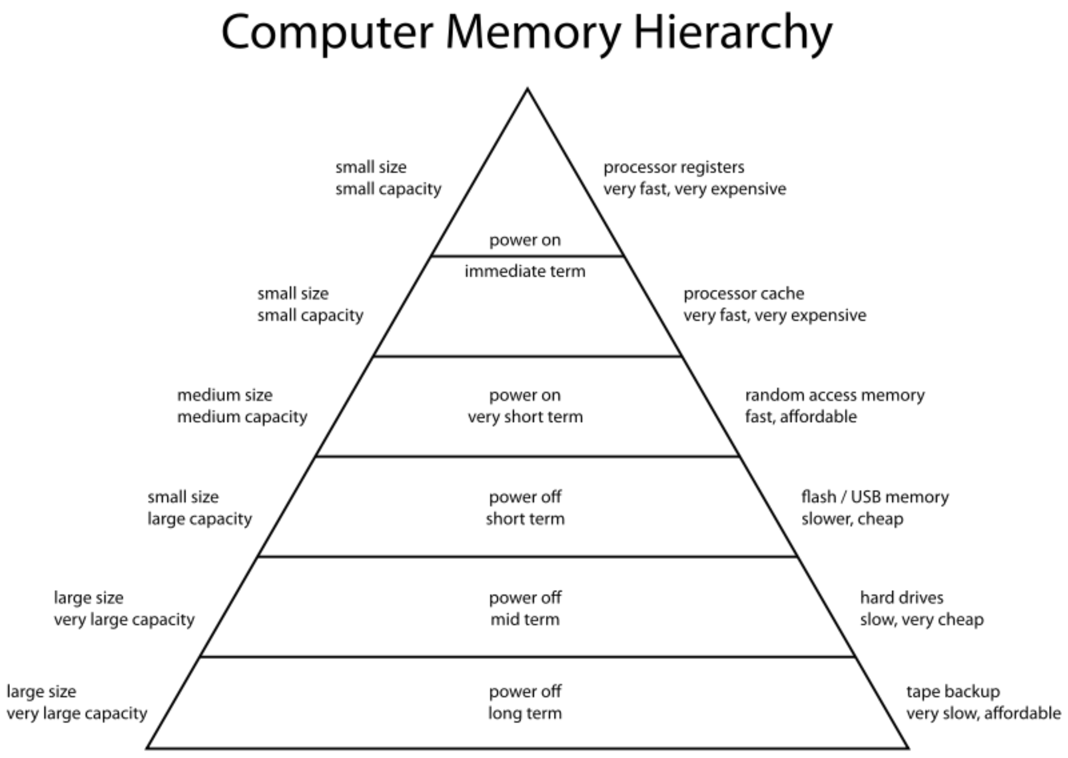


Figure 11.2: A model of the computer memory hierarchy (taken from Wikimedia Commons)

From this figure, we can see the standard progression from small, fast, expensive memory to large, slow, and inexpensive memory. One way to increase system performance is to figure out which data will be accessed the most, and store it as high up the hierarchy as possible.

However, what is missing from Figure 11.2 is *remote* storage. Remote storage typically falls into the slow, cheap, and large category, but is further slowed by the fact that data must be transmitted e.g. over a network. Thus, data stored in RAM a remote machine is significantly slower to access that data stored on disk on a local machine.

Expected access time

If the access time in the two mediums are T_{fast} and T_{slow}, it takes T_{check} time to check whether the faster medium has the info, and the probability that the check succeeds is hit_rate, then the *expected access time* of the whole system is:

$$\begin{aligned} T_{exp} &= T_{check} + (hit\_rate * T_{fast}) + ((1 - hit\_rate) * T_{slow}) \\ &= T_{check} + T_{fast} + ((1 - hit\_rate) * (T_{slow} - T_{fast})) \end{aligned}$$

Caching is worthwhile if $T_{exp} < T_{slow}$, and if the cost in space in the faster medium is acceptable.

You can increase the hit rate by using more space in the faster medium, but you will eventually either run out of space there, or take that space away from more worthwhile uses.

Why does caching work?

Caching works because most streams of accesses exhibit two properties: *temporal locality* and *spatial locality*:

1. *Temporal locality* says that if an item has been accessed lately, it is likely to be accessed again soon.

   This is why keeping a copy of a recently accessed item in the faster medium is likely (though not guaranteed) to pay off.

2. *Spatial locality* says that if an item has been accessed lately, then other items near it (related to it in some way) are also likely to be accessed again soon.

This is why it is often useful to bring into the cache not just the item being requested now, (e.g. the first disk block of a file, a web page, a single column from a record in a database), but also nearby related items (e.g. the 2nd, 3rd, 4th etc blocks of that file, the images referred to by the page, the entire record from a database).

Example — RAM access

Figure 11.3 shows a graph[1] of an application that demonstrates strong temporal and spatial locality properties of RAM memory. In this figure, X axis represents CPU cycles, and the Y axis represents memory locations.

[1]Taken from `http://www.eetimes.com/design/signal-processing-dsp/4017551/Optimizing-for-instruction-caches-part-1`

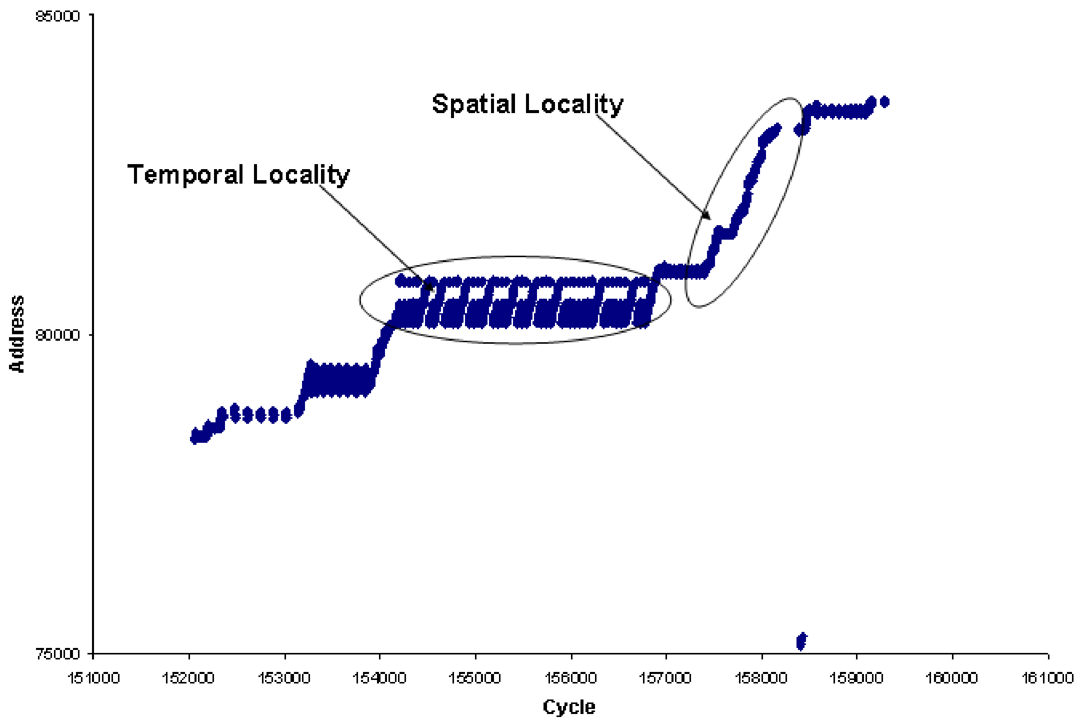


Figure 11.3: An example of strong temporal and spatial locality

Temporal locality is strong because, as one can see throughout, but especially in the middle, the same memory locations are being access several times in a short number of CPU cycles. Spatial locality is also strong throughout, because memory locations next to each other are accessed together in short succession.

One explanation of the noted temporal locality could be that the application must calculate the same thing a number of times from several variables, but only one or two of these variables changing each time. The spatial locality is likely to be iteration over an array. The section highlighted as temporal locality in Figure 11.3 demonstrates spatial locality as well, in that it is not just the same memory locations accessed several times, but accesses around the same time are also of memory locations nearby.

An application that has properties such as this one would benefit from caching certain data up the memory hierarchy. Compilers and interpreters perform loading in a certain fashion to allow exploitation of these.

Consider the example of adding up the elements in a two-dimensional array. One way is to do this row first, the other way is column first:

Listing 11.2: **Row-first matrix iteration**

```
for (i = 0; i < rows; i++) {
  for (j = 0; i < cols; j++) {
    sum += a[i][j];
  }
}
```

Listing 11.3: **Column-first matrix iteration**

```
for (i = 0; i < cols; i++) {
  for (j = 0; i < rows; j++) {
    sum += a[i][j];
  }
}
```

Using the C programming language, and many other modern programming languages, each row will be a contiguous block of memory. In the row-first solution, each time the outer loop is executed, the array `a[i]` will be loaded into cache when `a[i][0]` is accessed, and the remainder of the elements from `1` to `cols - 1` will be accessed from the cache. Whereas, in the column-first solution, the array will be loaded into the cache when `a[i][0]` is accessed, however, this may be discarded without ever being accessed from the cache because the next element of the matrix accessed is `a[i+1][0]`, meaning that the array `a[i+1]` is loaded into the cache.

Example – Content delivery networks

Some websites get many thousands of requests per second, and some of these are for the same content. Instead of delivering the same content, e.g. from Seattle to Sydney now and from Seattle to Melbourne in a few seconds, many website operators cache a copy somewhere in Australia for a while after the first access from Australia.

To avoid having servers in every country, many websites use a *content delivery network* (CDN). A CDN is a distributed network of servers that host data for other people to allow "local" caching, where local is in a geographical sense; e.g. hosting in Melbourne to avoid repeated connections overseas.

CDN service providers, such as Akamai or University of Melbourne-grown MetaCDN, allow companies to preposition their data near the ISP points-of-presence used by most of their customers. Such CDNs can be thought of as pre-populated caches, containing copies of anything from a few pages to an entire website.

Websites using CDNs exploit the temporal and spatial properties of website data. For example, a news website will infer that new articles are likely to get a lot of hits when they first appear, but will get less and less the older the news becomes. As such, the website operators may store copies of recent articles all around the world in a CDN, while keeping old articles only on their home site.

Similarly, website browsing demonstrates a strong presence of spatial locality. If a person clicks on a web page, they are likely to click on one of the links on that web page. As such, some content providers may choose to bring across more than one page and store it in the CDN when a user clicks on a home site.

Implementation

Different kinds of caches use different mechanisms to map keys to values. When you have to remove an item from the cache to make room, you want to remove the item that will not be needed for the longest time. Since we almost never know for sure what the sequence of accesses will be in the future, we have to estimate.

In many situations, the best estimate is that the near future will be similar to the near past. This suggests that the best item to remove is the item that was *least recently used* (LRU). This *LRU algorithm* requires marking each item when it is accessed with a time stamp or access sequence number.

There are several software packages (such as *memcache*) available that implement caching for you; some are available in several programming languages. These are suitable for some kinds of uses, but not all.

One obvious question is: how should the programmer decide whether some data item should be cached or not?

The answer depends on the relative costs and relative speeds of the faster and slower mediums, such as main memory, disks, and network latency. Since these change over time, the answers can change as well.

In 1985, a famous paper proposed that databases should keep items in memory if and only if they are used every five minutes, or more often. This five minute rule was still valid fifteen years later. Today, memories are big enough and cheap enough that you want to cache items that are used significantly less frequently than that.

Similar considerations apply to other caching situations, such as caching remote web pages on local disks, but the answers can be *very* different.

Pros

1. *Decreased time*: Using caching can decrease the amount of time that users wait to access resources.
2. *Relatively simple*: The idea of caching is relatively straightforward, even if the optimal policy for deciding what to cache is not.

Cons

1. *Caching limitation*: Increases in the size of a cache are subject to diminishing returns. If adding 1 megabyte to the size of a 1 megabyte cache improves execution time by a certain amount, then adding X megabytes to a 1 megabyte cache will NOT improve execution time by X.

   This is because a small cache should already store the most frequently reused items. As a cache grows, each increment of extra space stores only less and less frequently reused items.
2. *Lack of temporal locality*: If a stream of accesses does not exhibit temporal locality, it will not benefit from caching, because once a piece of data is stored in the faster medium, it will never be needed again. This is why bulk copying of memory is slow.
3. *Lack of temporal and spatial locality*: If a stream of accesses exhibits *neither* form of locality, then caching it can even lead to a *slowdown*. This is because bringing in nearby items is wasted work if the program will never request them, and will also hog valuable fast memory with data that is not used. Avoiding unproductive or counterproductive "optimisations" is one reason why designers should seek to understand the memory behaviour of their programs.

11.3 Profiling and performance tuning

Once the design has been implemented and the implementation has been debugged, it is time to pay attention to performance again.

Performance tuning is a loop:

1. profile the program, measuring where it is spending its time;
2. choose a bottleneck, a part of the program that seems to take more time than is necessary;
3. design and implement ways to speed up this bottleneck; and
4. return to 1.

Developers are often surprised by the location of bottlenecks, partly because they are often hidden behind abstraction boundaries.

You should back out any changes that turn out to cause slowdowns. You should also think twice about changes that significantly complicate the design, unless the speedup they yield is significant enough to be useful.

11.4 Further reading

- The performance chapters of *Code complete* by Steve McConnell [8].
- *Writing efficient programs* by Jon Bentley [2].

Chapter 12

Enterprise system integration

Enterprise software solutions often consists of many different systems that are designed and developed independently. These systems need to be seamlessly integrated to provide end-to-end business solutions. In this chapter we will look at architectural solutions and patterns for integrating enterprise applications. You will learn four basic patterns for enterprise integration, and will study the trade-offs between these patterns. The patterns are: (1) *file transfer*; (2) *shared database*; (3) *remote procedure invocation* and (4) *messaging* [5]. Due to many desirable properties messaging provides over the other three integration patterns, it is the widely accepted standard for enterprise integration and most enterprise standards such as J2EE support messaging. Chapter 13 will introduce a series of patterns specifically for messaging.

Learning outcomes A person familiar with the material in this chapter should be able to:

1. describe four design patterns related to enterprise integration;
2. apply these design patterns in enterprise applications;
3. critique the choice of one of the different design patterns on a particular application; and
4. provide a rationale for the choice of one the presented patterns in an enterprise application.

12.1 The problem

The problem: How do we efficient share information between different systems that form part of a larger enterprise system?

Although one could think of an organisation using a single, large, monolithic software system to support the business processes for the whole organisation, this is often not the case. Some of the reasons for this are:

- Organisations often purchase third party packages that only support certain functionalities of the business needs.
- Systems are built at different times, using the latest technology and platforms each time.

- Systems are developed by different groups, and solutions chosen depend very much on the expertise and preferences of the teams.
- Developing one large system would take a long time and it is often better to provide solutions to specific business needs in a timely manner.

Therefore, an enterprise solution often consists several different software systems, that are required to communicate with other to meet the business needs; in some cases to get information, and in other cases to access services provided by these systems.

For example, consider the University as a enterprise, which has a large number of different systems related to managing students, staff, and teaching/learning. Although you could think of a single system that provides all these functionalities, in which case integration is not required, in reality different functionalities are supported by different software systems (e.g. ISIS for student information management, LMS for teaching support, Themis, Travel Portal etc for staff support). These systems contain data that must be shared; for example the student enrolment data which is managed by the student system is required by the system that manages the teaching/learning, and therefore information needs to be shared between these systems. Following are some of the challenges related to enterprise integration.

- ***Unreliable networks:*** Integration solutions have to transfer data across systems, in most cases over a network (a local area network (LAN), or the Internet). The data has to be transferred over different media such as LAN segments, routers, switches, satellite links etc. This communication could result in interruptions and delays. Enterprise integration solutions need to deal with these issues.
- ***Network latency:*** Accessing information over a network is slow compared to accessing local data. Therefore integration can result in performance issues; seamless integration is a challenge for this reason.
- ***Heterogeneity:*** The applications that need to be integrated may not be compatible. They could be in different languages, on different platforms and operating systems, and have different data formats.
- ***Evolution:*** Software systems evolve over time. Integration solutions need to be designed such that such changes have minimal effect on integration; loose coupling between applications is desirable.

The remainder of this chapter describes four patterns related to enterprise integration.

12.2 The patterns

12.2.1 File Transfer

Pattern name	File transfer
Description	Uses files to transfer data/information between multiple applications.

The File Transfer solution, as the name implies, uses files as a means of exchanging data between systems.

Each application writes it output data (the information to be shared) to a file, at specific times (regular intervals or as and when data is available), and applications that consume the data simply read the data in the files. This is one of the simplest patterns for enterprise integration. One of the most important design decisions related to file transfer is the content and the format of the files; file transformation may be needed if the format of the output and the format expected by the consumer are different. This transformation would require processing.

Pros

1. *Simplicity:* File transfer is one of the simplest approaches to enterprise integration. Integrators do not require knowledge about the internals of the application. They only need to understand the format of the files and the transformation that is required.
2. *Tool independence:* Additional third party tools and software frameworks are not required.

Cons

1. *Timing*: Applications need to agree on when to write the files and when to consume the files. Writing files is a overhead to the producer. However the files need to be produced based on consumer needs.
2. *Synchronisation:* Because updates occur infrequently, results can get out of synchronisation. Therefore file transfer as an integration solution is only suitable for applications where the synchronisation requirements are less important; for example for a student management system it may be good enough that the subject enrolment information is only known by the learning management system the following day.
3. *Format Incompatibility:* Often the output format of the file by the producer does not match the needs of the consumer and therefore file transformation is required.
4. *Performance Overhead:* File reading/writing has associated overheads. However all other solutions that require input/output also have similar overheads. When applications are distributed across multiple locations a shared file that can be accessed by all locations thorough a wide area network is not a good solution in terms of latency. A distributed database may be more suitable in such cases.

12.2.2 Shared Database

Pattern name	Shared Database
Description	Uses a shared database to transfer data/information between multiple applications.

In this solution, a databased that is shared between applications is used for sharing information.

Producers write the data to a shared database and the consumers read the data from the database. The format of the database has to be agreed upon between the producers and the consumers or else transformation is required as before. One of the main advantages of the shared database over file transfer is the timeliness and synchronisation. When updates and reads are frequent, the shared database is a better solution than file transfer. However, the overhead is more than the file transfer solution.

Pros

1. *Timeliness*: One of the biggest advantages of the shared database approach is the data consistency. Because the reads and writes happen through a single server, the database server, it is much easier to manage synchronisation issues. Database servers take care of concurrency and synchronisation issues, and the integrators do not have to worry about it.

2. *Format Enforcement:* Because the data format has to agreed upon at the time the data base is designed, the solution enforces the data format more strictly, but this could be a disadvantage as well.

Cons

1. *Increased coupling via the database:* Any applications that are part of the information sharing must use the same database schema (or essentially same mechanism for accessing information from the database), meaning that the applications are coupled via the database.

2. *Design complexity*: Systems have to agree on the format of the shared database in advance, and all systems have to designed to use this data format. Coming up with a database schema design that all applications will agree on is challenging.

3. *Implementation complexity:* All systems have to be implemented such that they use the same database.

4. *Performance overhead:* A single central database can become the performance bottleneck, unless it is designed carefully.

12.2.3 Remote Procedure Invocation

Pattern name	Remote Procedure Invocation
Description	Uses remote procedure calls to transfer data/information between multiple applications.

Remote Procedure Invocation allows one application to invoke functions in another application. The data that is required is passed to the application during the invocation of the remote procedure call.

Many software vendors support middleware for remote procedure invocation (also referred to as *remote method invocation* in object oriented terminology). Some well known middleware solutions are CORBA, COM, .NET Remoting, and Java RMI. In some cases the middleware supports additional capabilities such as transactions, synchronisation, and security. One of the more recent developments related to remote procedure invocation is *Web Services*, which use standards such SOAP and XML.

Pros

1. *Functionality Sharing:* Enables an application to provide/use services to/of another application

2. *Better Encapsulation*: Because the complete data structure is not exposed as in the case of file transfer and shared databases, in principle this provides better encapsulation of the data; only the required data is transferred based on the functionality that is invoked in the procedure.

Cons

1. *Coupling*: Remote procedure invocation requires applications to adhere of the interface of the remote call in and therefore is tightly coupled. Application coupling is increased. Changes to the interface will trigger changes in all applications that use the interface. Systems cannot be changed independently.

2. *Synchronous:* The system in which the remote procedure/method is invoked must be up and running to receive the call; therefore this solution is essentially synchronous.

3. *Performance overhead:* Although remote calls can be designed to have the same syntax as local calls making it easy from a design point of view, there is a runtime overhead for remote calls compared to local calls, and therefore application will slow down during remote call invocation.

12.2.4 Messaging

Pattern name	Messaging
Description	Uses messages to transfer data/information between multiple applications.

File transfer and shared databases allow applications to share data but not their functionality. File transfer results in good decoupling between applications, but at the cost of timeliness. Shared database solves the problem of timeliness, but introduces coupling due to the shared database. Remote procedure invocation allows sharing functionality but increases the coupling between applications.

Messaging attempts to provide a solution that overcomes the tight coupling of remote procedure invocation through a more file transfer type communication mechanism, but allows functionality to be invoked similar to remote procedure invocation. Messaging allows data to be transferred frequently, immediately, reliably and asynchronously, using customisable formats.

Implementation

Messaging is provided by the third party messaging system. Figure 12.1 shows the schematic representation of a messaging system. Message producers generate messages (could be data or events/commands) that are placed on the message bus. These messages get routed to the appropriate message consumers via a message router. Message consumers consume the message by using the data in case of data messages, or taking appropriate actions in case of command messages.

The design of a messaging system must take into consideration what type of *Message* is be sent, and the type of *Message Channel* to be used, how it gets sent to the appropriate sender/s using appropriate *Message Routers* , the format of the data can be managed using *Message Translators* , and connection of the applications using *Message Endpoints.* The patterns related to each of these will be discussed in the next chapter.

Pros

1. *Remote Communication:* Messaging allows communication with remote applications, without the application developer needing to worry about data translation, networking etc.

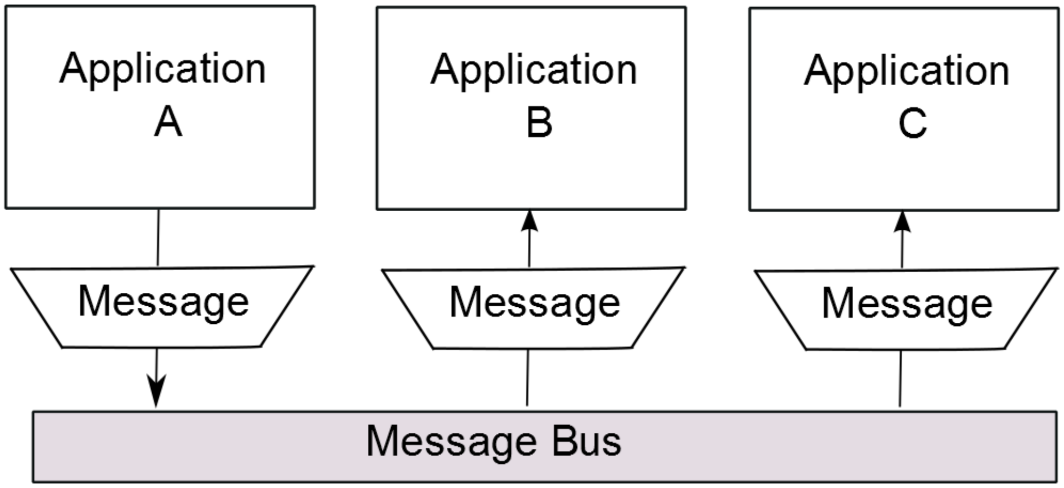


Figure 12.1: Schematic representation of a messaging solution.

2. *Platform/Language Integration:* Messaging solutions handle the heterogeneity issues across platforms. Applications developed in different languages, running on different platforms can communicate with each other.

3. *Asynchronous/Non-blocking Communication:* Messaging uses a *send-and-forget* (asynchronous) style of communication. The sender does not have to wait for the receivers to receive the message. Therefore, the communication is non-blocking.

4. *Throttling:* Applications are isolated from message overload conditions because queueing and throttling is handled by the messaging system.

5. *Reliable Communication:* Through building appropriate storing and retrying strategies, messaging can provide a reliable communication mechanism that is not easy to design with other forms of integration.

6. *Disconnected Operation:* Allows applications to run in a mode that is disconnected from the network and then receive messages stored in the messaging server when they eventually connect to the server.

Cons

1. *Complexity:* Using a messaging solution for integration increases the complexity of the programming model; since the communication is asynchronous, an *event-driven* programming model must be used.

2. *Sequencing Issues:* Messaging systems can have guaranteed message delivery, but they cannot guarantee when the messages will be delivered. Therefore, there can be sequencing issues. In cases where the message sequence matters, additional design considerations must be made to take care of this.

3. *Synchronous Scenarios:* Some applications require synchronous behaviour. For such applications messaging solutions must be designed such that such synchronous messaging can be supported.

4. *Performance:* Messaging systems impose a performance overhead on applications.

5. *Vendor lock-in:* Although there are enterprise specifications for messaging, messaging solutions across vendors are not fully compatible.

6. *Limited platform support:* Many messaging systems are not supported across all platforms.

Chapter 13

Enterprise Messaging

As discussed in Chapter 12, messaging provides a promising solution for enterprise integration because of the many benefits it offers. At the same time, messaging solutions also introduce many challenges, and careful design is required to overcome these challenges. A messaging system must deal with the creation, sending, delivering, receiving and processing of messages. This chapter will discuss some of the patterns related to the different aspects of messaging. Within the scope of this subject, we will only cover some of the basic and important patterns related to messaging. For a more complete reference on enterprise messaging related patterns is available in Hohpe and Woolf [5].

Learning outcomes A person familiar with the material in this chapter should be able to:

1. identify the main components of a messaging system;
2. describe some of basic design patterns related to message channel and message construction; and
3. understand the implementation and the application of these patterns in enterprise messaging.

13.1 The problem

The problem: The problem of messaging is the same as the previous chapter: how do we efficient share information between different systems that form part of a larger enterprise system?

Figure 13.1 shows an abstract view of a messaging system in terms of the components the system must provide in order to create, send, deliver, receive and process messages.

A messaging system and also an application that chooses messaging for integration must make design choices for each of these components. This chapter introduces the patterns for some of these components.

- *Message Channels:* A message channel is logical entity that connects a sender (producer, publisher) and a receiver (consumer, subscriber).
- *Messages:* A message is a packet of data, that is transmitted through the message channel. A message has one producer, but could have multiple receivers.

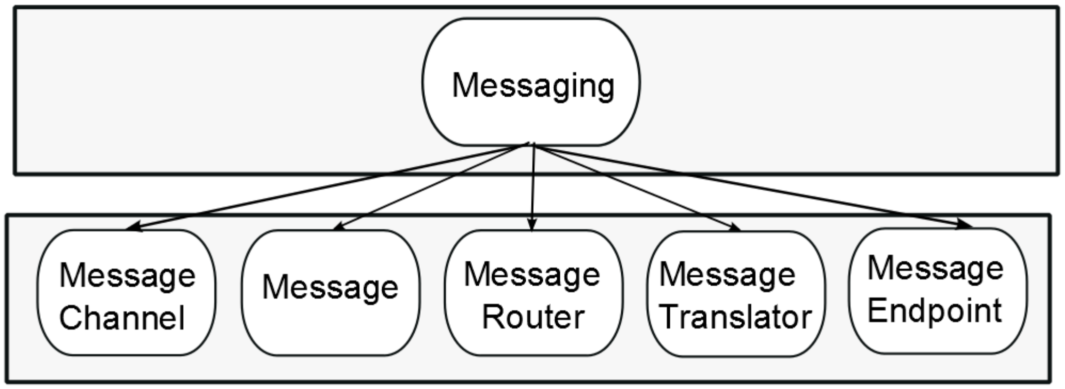


Figure 13.1: Abstract view of a messaging system.

- *Message Router:* Large enterprise systems have many different channels; a message may have to go through many different channels to reach the destination. In such cases, message routers are responsible for channelling a message through the appropriate channels to reach the final destination.
- *Message Translator:* In some cases the message that the producer sends may not be in the format the consumer is ready to accept. In such cases, a message translator in responsible for translating the message to the format the consumer can understand.
- *Message Endpoint:* Applications do not have the built-in functionality to interface with a messaging system. Such interfaces need to be built as a separate layer on top of the messaging. This is referred to as the message endpoint.

13.2 Message Channel Patterns

In a messaging system, data senders and receivers connect through message channels; a channel is a logical communication path between the sender and the receiver. Messaging systems support mechanisms to define message channels and the applications need to decide on the number and the types of channels that need to be defined for the particular system. Applications use this predefined set of channels — agreed upon at design time — for communications. Messaging channels are normally unidirectional; applications use separate channels for sending and receiving. There are several important decision related to the design of channels which result in several patterns related to message channel design:

- *One-to-one vs one-to many: Point-to-Point Channel* is used if the message must be received by only one consumer. *Publish-Subscribe Channel* is used if multiple receivers should receive the message.
- *Dealing with undeliverable messages:* Messaging is asynchronous. Therefore, once the sender places the message in the channel the sender does not deal with the message any more. There are several patterns that deal with how to handle undeliverable/invalid messages. *Invalid Message Channel* is a special channel to deal with message that the receiver cannot interpret due the format being invalid. Message systems provide a *Dead Letter Channel* to deal with messages that are received by the system but cannot be delivered. For applications that must ensure that messages received by the messaging system are delivered to the destination even if the system crashes, a *Guaranteed Delivery Channel* is the solution.
- *Integration of non-messaging clients:* In some scenarios client applications cannot connect directly connect to the messaging system. In such cases, a *Channel Adaptor* is used.
- *Messaging as a communication backbone:* As the number of applications that communicate through messages grows, the number of channels increase, so the messaging system becomes a generalised communication channel, referred to as a *Message Bus*.

13.2.1 Point-to-point Channel

Pattern name	Point-to-point Channel
Description	A channel in which a sender sends a message to exactly one receiver.

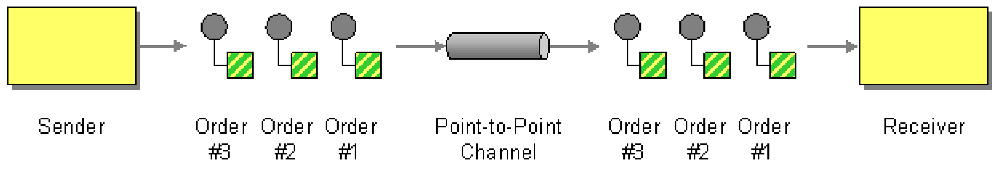


Figure 13.2: Point to point channel (from `http://fusesource.com/docs/router/2.8/eip/MsgCh-P2P.html`)

The point-to-point channel ensures that only one of the receivers out of all receivers listening on the channel will receive the message. If multiple receivers are listening on the channel the messages are distributed in a round robin fashion. Therefore, it can be ensured that only one receiver will receive and act on the message. An example implementation of this pattern in J2EE is the *JMS Queue*.

Applications

This pattern is useful for load balancing, where the work needs to be distributed to different workers.

13.2.2 Publish-Subscribe Channel

Pattern name	Publish-Subscriber Channel
Description	A channel in which a sender can send a broadcast to a set of interested parties.

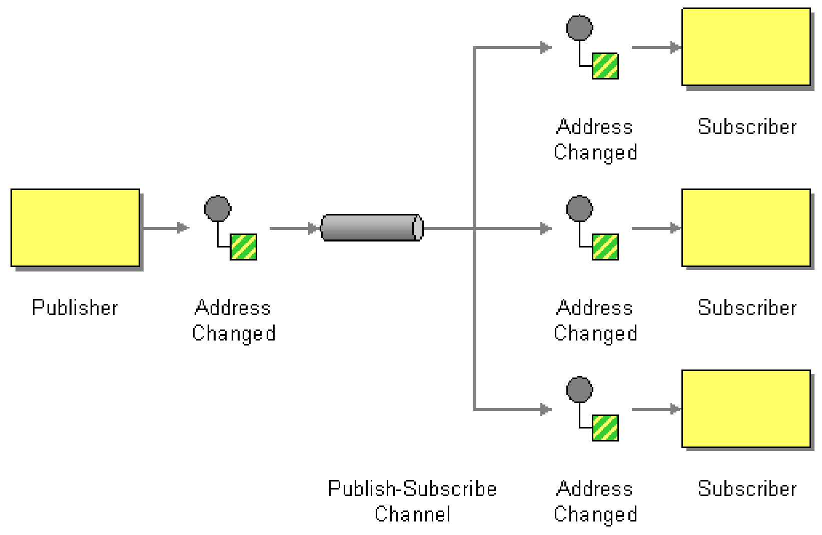


Figure 13.3: Publish-Subscribe Channel (from `http://fusesource.com/docs/router/2.8/eip/MsgCh-PubSub.html`)

Publish-subscribe channel is essentially the application of "Gang of Four" Observer Pattern to messaging. A message by a publisher is received by all subscribers listening for the type of message. The event is considered consumed when all subscribers receive the message. One input channel is split to multiple output channels. When an event is published to the channel, a copy of the message is delivered to all the output channels. An example implementation of this pattern in J2EE is the *JMS Topic*.

Applications

A publish-subscriber channel can be used in applications that have multiple receivers interested in the contents of a publisher. In some cases, this type of a channel is useful for debugging purposes — to ensure that a message is received. Design should take into consideration the possible implications of securing — eavesdropping is possible.

13.2.3 Datatype Channel

Pattern name	Datatype Channel
Description	A channel that only permits data of a specific type, ensuring that the receiver will know the type of data that it will receive.

A separate Datatype Channel is maintained for each datatype — all messages in the channel will be of the same type. The sender will have to select the appropriate channel based on the data type, the receiver will know the type of data based on from where it is received. The problem with the datatype channel approach is the number of channels growing with the number of data types. Several other related patterns, which we will not cover in these lectures. can be used for solving this issue, such as *Selective Consumer*, *Canonical Data Model*, and *Content-based Router* [5].

Applications

In applications that have a few different distinct types of messages, a Datatype Channel will be a good design choice. For example, in a stock trading application, different channels can be used for stock quote request and trade requests.

13.2.4 Invalid Message Channel

Pattern name	Invalid Message Channel
Description	A channel that allows a message receiver to gracefully handle a message that it cannot process.

Figure 13.4 shows an abstract view of the Dead Letter Channel pattern. This is essentially an error log for messaging. A messaging system can define one or more channels especially for placing invalid messages. When the receiver receives a message it cannot understand and interpret, rather than just discarding it, it places it in the invalid message channel. Normally the messages that are placed in the invalid message channel are those that have invalid format (syntax), rather than the messages that are semantically incorrect.

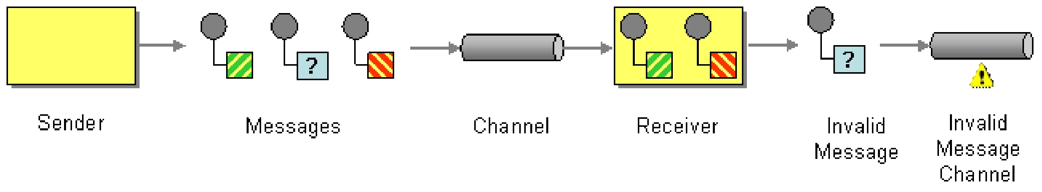


Figure 13.4: Invalid message channel (from `http://fusesource.com/docs/router/2.8/eip/MsgCh-Invalid.html`)

In the JMS specification, if the MessageListener gets a message it cannot process, the listener should divert it to a *unprocessable message destination* — this is the Invalid Message Queue.

Applications

This pattern can be used in any application that receives messages, and expects the message to adhere to a predefined format in order to deal with the message. If the format of the message is incorrect, and therefore the application cannot take the required action, the message is placed in the Invalid Message Channel.

13.2.5 Dead Letter Channel

Pattern name	Dead Letter Channel
Description	A channel that elegantly handles a message not being delivered.

Figure 13.5 shows an abstract view of the Dead Letter Channel pattern. A message that is placed in a channel may not be deliverable to the destination for different reasons, some of which are: incorrectly configured channel; channel being removed after the message being placed; message expiration before being delivered; and message timeout. Once a messaging system detects that it cannot be delivered

it is placed in a Dead Letter Channel. Unlike the Invalid Message Channel, which is designed by the developer, the Dead Message channel is normally supported by the messaging system.

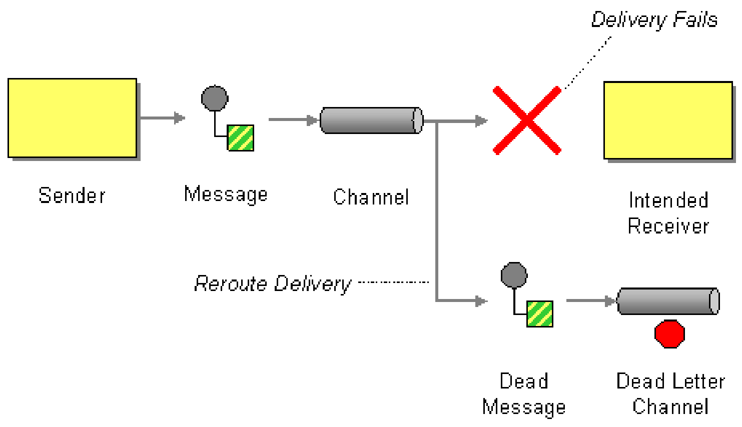


Figure 13.5: Dead Letter Channel (from `http://fusesource.com/docs/router/2.4/eip/MsgCh-DeadLetter.html`)

Applications

Messaging systems support a Dead Letter channel; any application messages that cannot be delivered to a receiver are placed in this channel.

13.2.6 Guaranteed Delivery Channel

Pattern name	Guaranteed Delivery
Description	A channel that ensures that a message will be delivered, even if the messaging system fails.

Figure 13.6 shows an abstract view of the Guaranteed Delivery pattern. Messaging systems handle messages from senders/publishers and delivers messages to receivers/subscribers. Until messages are delivered to subscribers they are normally held in memory. If the messaging system crashes, the messages are lost. If guaranteed delivery is required, all messages received are persisted in a database as soon as they are received, before the sender is acknowledged. Persistence increases reliability, but at the expense of performance. Guaranteed delivery must take care of many design/implementation issues such as, disk usage, clean up, retry strategies etc.

In JMS, persistence can be set on a per message basis, by setting the JMSDeliveryMode to PERSISTENT:

```
producer.setDeliverMode(javax.jms.DeliveryMode.PERSISTENT);
```

Applications

In applications where messages are critical, and loss cannot be afforded, messages have to follow the Guaranteed Delivery option.

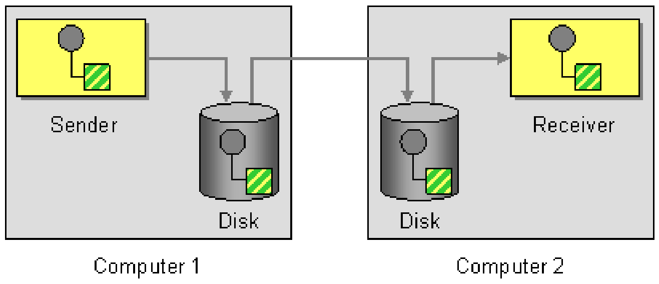


Figure 13.6: Guaranteed Delivery Channel (from `http://fusesource.com/docs/router/2.8/eip/MsgCh-Guaranteed.html`)

13.2.7 Message Bus

Pattern name	Message Bus
Description	A bus for sending messages that decouples integrated applications such that applications can be easily added/removed without affecting others.

Figure 13.7 shows an abstract view of the Message Bus. A Messages Bus integrates several applications by providing a unified interface across a number of applications. Applications that require services from other applications directly do so by connecting to the message bus. In some cases *channel adaptors* are needed to adapt to the interface provided by the message.

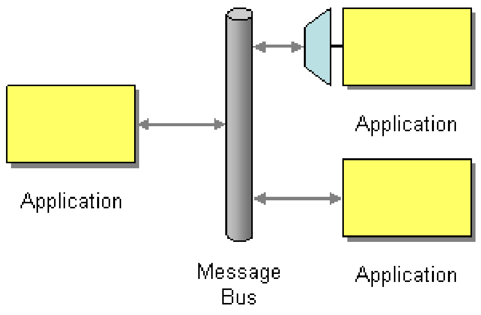


Figure 13.7: Message Bus (from `http://fusesource.com/docs/router/2.8/eip/MsgCh-Guaranteed.html`)

Applications

This pattern is used for integrating a large number of applications.

13.3 Message Patterns

In the previous section, we looked at patterns related to the design of the Message Channel. In this section, we will focus on the Message and its construction. Message-related patterns can be classified based on: the message intent; returning a response; managing large volumes of data; and strategy for discarding messages.

- *Message intent:* Messages can be classified into one of the following based on the message intent:

  1. *Command Message:* specifies the method or the function to the invoked when the message is received;
  2. *Document Message:* this is essentially a data structure sent to the receiver; and
  3. *Event Message*: informs the receiver of a change in the sender but not how to react to it.

- *Returning the response:* A *Request-Reply* scenario is when a sender expects response confirming that the message has been processed. Usually the request is a Command Message, and the response is a Document Message which returns the results of executing the command. For example the command could be a RPC to be executed and the receiver and the reply will be the results of the RPC. Another example is a request which is a query and the result which is are the query results. It is important that the requester sends the *Return Address*, which is the channel for the replier to place the results in. A *Correlation Identifier* to correlate the request to the reply is also important.

- *Managing large volumes of data:* In cases where the data structure to be transmitted is large, the message can be transmitted as a *Message Sequence* — a set of related messages.

- *Strategy for discarding messages:* Since messaging is asynchronous, the sender does not know how long it will take for the receiver to receive the message. However, the message contents may be time sensitive. In such cases *Message Expiration* strategies are important.

13.3.1 Command Message

Pattern name	Command Message
Description	A type of message used to invoke a procedure in another application.

Remote Procedure Invocation allows procedures/methods to be executed in remote applications in a synchronous fashion; the application needs to be up and running for the procedure to execute. Even in the case of an asynchronous RPC, the message will be lost if the application is not up and running. A command message is a message that is designed to carry information regarding a remote method that need to be executed in response to the message; the message must contain information regarding the command to be executed and its parameters.

Applications

Command messages can be used in cases where a command needs to be executed in a different application, and usually command messages are sent over point-to-point channels.

13.3.2 Document Message

Pattern name	Document Message
Description	A type of message used to transfer data between applications.

A Document Message is a message that carries plain data; no command or event. In JMS it can be an object message `ObjectMessage` or a `TextMessage` containing data in XML format.

Applications

A Document Message is used in the case where the data needs to be transmitted to message receiver(s), and the sender does not really care what the receiver does with the data. In the case of Document Messages, the timing of the delivery is not normally important except that it needs to be delivered in the specific time window. Message Expiration can be used in this context.

13.3.3 Event Message

Pattern name	Event Message
Description	A type of message used to transmit events from one application to another.

A Event Message is essentially a notification of a particular event that has taken place; this is normally used in the publish/subscribe communication pattern, where the Observer Pattern is used for event registration/notification. Event notifications are sent to Observers that have pre-registered for events. In the case of a push model, a combined document/event message would be used. In the case of a pull model an *update* will be sent as an Event Message that will be sent to all observers. In response, interested observers will send a *state request* as a Command Message to request for the state. The *state reply* will be a document message.

Applications

That pattern is used in applications that use the publish/subscribe design pattern for communication, to inform subscribers that some event has occurred.

Bibliography

[1] L. Bass, P. Clements, and R. Kazman. *Software architecture in practice*. Addison-Wesley Longman Publishing Co., Inc., 2003.

[2] J. Bentley. *Writing efficient programs*. Prentice-Hall, Inc., 1982.

[3] M. Fowler. *Patterns of enterprise application architecture*. Addison-Wesley Professional, 2003.

[4] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: elements of reusable object-oriented software*. Addison-Wesley Professional, 1995.

[5] G. Hohpe and B Woolf. *Enterprise Integration Patterns*. Addison-Wesley Professional, 2004.

[6] Micah Martin and Robert C Martin. *Agile principles, patterns, and practices in C#*. Pearson Education, 2006.

[7] Robert C Martin. Design principles and design patterns. *Object Mentor*, 2000.

[8] S. McConnell. *Code complete*. Microsoft press, 2004.

[9] C. Steel and R. Nagappan. *Core Security Patterns: Best Practices and Strategies for J2EE, Web Services, and Identity Management*. Prentice Hall, 2005.