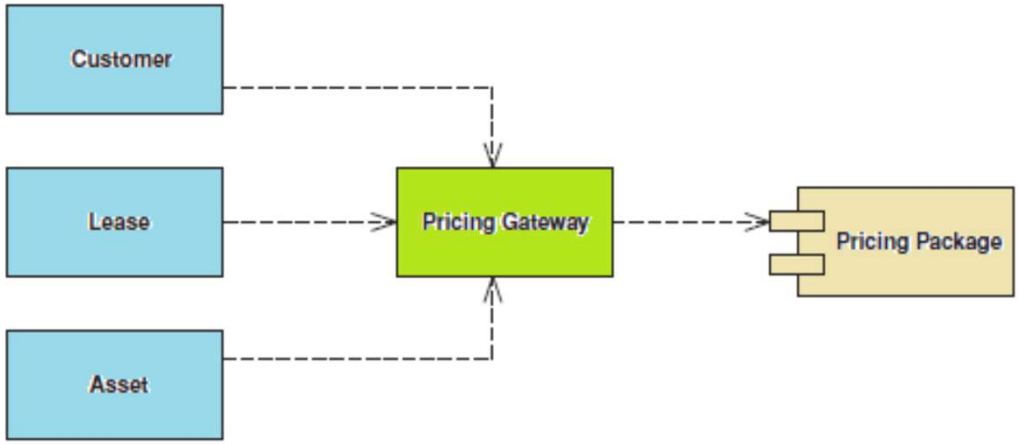
SOEN 387: Web-Based Enterprise Application Design

Chapter 18. Base Patterns

Gateway

An object that encapsulates access to an external system or resource.



When accessing external resources like this, you will usually get APIs for them.
However, these APIs are naturally going to be somewhat complicated because they take the nature of the resource into account.

Anyone who needs to understand a resource needs to understand its API—whether JDBC and SQL for relational databases or for XML.

Not only does this make the software harder to understand, it also makes it much harder to change.

The answer is to wrap all the special API code into a class whose interface looks like a regular object. Other objects access the resource through this *Gateway*, which translates the simple method calls into the appropriate specialized API.

The *Gateway* should be as minimal as possible and yet able to handle required tasks

You should consider *Gateway* whenever you have an awkward interface to something that feels external. Rather than let the awkwardness spread through the whole system, use a *Gateway* to contain it.

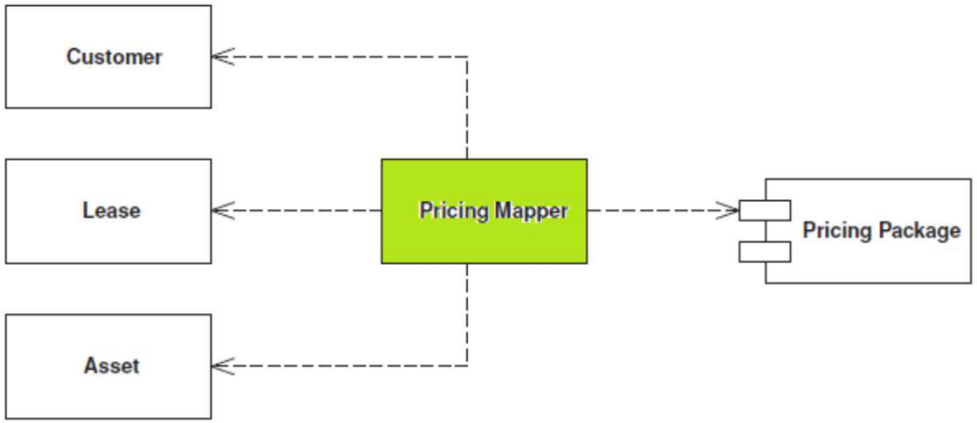
A clear benefit of *Gateway* is that it also makes it easier for you to swap out one kind of resource for another. Any change in resources means that you only have to alter the *Gateway* class—the change does not ripple through the rest of the system.

However, even if you do not think the resource is ever going to change, you can benefit from the simplicity and testability *Gateway* gives you. When you have a couple of subsystems like this, another choice for decoupling them is a *Mapper*. However, *Mapper* is more complicated than *Gateway*.

While *Facade* simplifies a more complex API, it is usually done by the writer of the service for general use. A *Gateway* is written by the client for its particular use.

Mapper

An object that sets up a communication between two independent objects.



Sometimes you need to set up communications between two subsystems that still need to stay ignorant of each other. This may be because you can not modify them or you can but you do not want to create dependencies between the two.

A mapper is an insulating layer between subsystems. It controls the details of the communication between them without either subsystem being aware of it.

Essentially a *Mapper* decouples different parts of a system. When you want to do this you have a choice between *Mapper* and *Gateway*.

Gateway is by far the most common choice because it's much simpler to use a *Gateway* than a *Mapper* both in writing the code and in using it later.

As a result you should only use a *Mapper* when you need to ensure that neither subsystem has a dependency on this interaction.

The only time this is really important is when the interaction between the subsystems is particularly complicated and somewhat independent to the main purpose of both subsystems. Thus, in enterprise applications we mostly find *Mapper* used for interactions with a database, as in *Data Mapper.*

Layer Supertype

A type that acts as the supertype for all types in its layer.

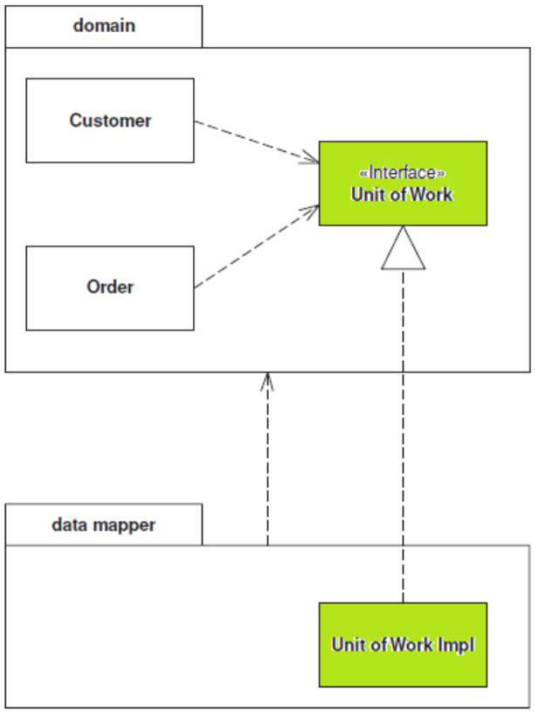
It is not uncommon for all the objects in a layer to have methods you do not want to have duplicated throughout the system. You can move all of this behavior into a common *Layer Supertype*.

Layer Supertype is a simple idea that leads to a very short pattern. All you need is a superclass for all the objects in a layer.
For example, a Domain Object superclass for all the domain objects in a *Domain Model* . Common features, such as the storage and handling of *Identity Fields* can go there.

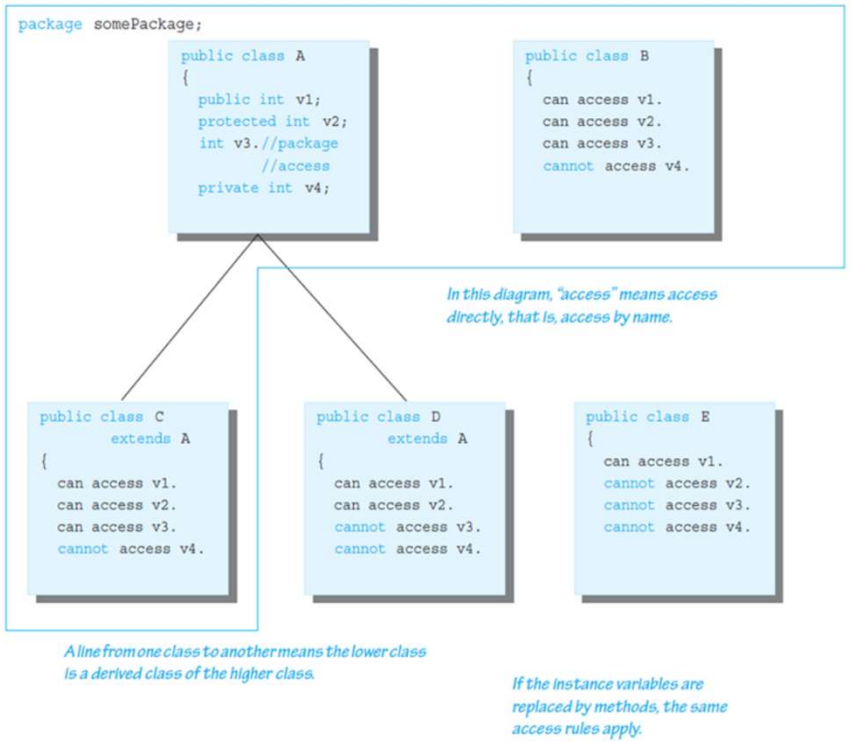
Use *Layer Supertype* when you have common features from all objects in a layer.

Separated Interface

Defines an interface in a separate package from its implementation.



use *Separated Interface* to define an interface in one package but implement it in another. This way a client that needs the dependency to the interface can be completely unaware of the implementation.

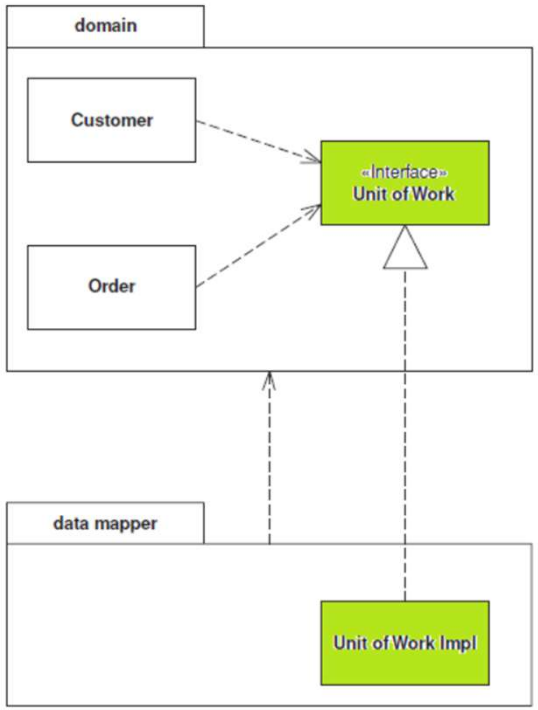


As you develop a system, you can improve the quality of its design by reducing the coupling between the system's parts. A good way to do this is to group the classes into packages and control the dependencies between them. You can then follow rules about how classes in one package can call classes in another—for example, one that says that classes in the domain layer may not call classes in the presentation package.

This pattern is very simple to employ. Essentially it takes advantage of the fact that an implementation has a dependency to its interface but not vice versa.

This means you can put the interface and the implementation in separate packages and the implementation package has a dependency to the interface package.

Other packages can depend on the interface package without depending on the implementation package.



Registry

A well-known object that other objects can use to find common objects and services.

When you want to find an object you usually start with another object that has an association to it, and use the association to navigate to it.

Thus, if you want to find all the orders for a customer, you start with the customer object and use a method on it to get the orders.

However, in some cases you won't have an appropriate object to start with. You may know the customer's ID number but not have a reference. In this case you need some kind of lookup method—a finder—but the question remains: How do you get to the finder?

A *Registry* is essentially a global object, or at least it looks like one—even if it is not as global as it may appear.

The *Registry* class contains a single static field that holds a *Registry* instance.

Singletons are widely used in single-threaded applications, but can be a problem for multi-threaded applications.

This is because it's too easy for multiple threads to manipulate the same object in unpredictable ways.

You may be able to solve this with synchronization which has its own challenges.

Using a singleton for mutable data in a multi-threaded environment is not recommended. It does work well for immutable data, since anything that can not change is not going to run into thread clash problems.

Value Object

A small simple object, like money or a date range, whose equality isn't based on identity.

Difference between reference objects and *Value Objects:*
A *Value Object* is usually the smaller; it is similar to the primitive types present in many languages that are not purely object-oriented.

In a broad sense we like to think that *Value Objects* are small objects, such as a date object, while reference objects are large, such as an order or a customer.

The key difference between reference and value objects lies in how they deal with equality. A reference object uses identity as the basis for equality—maybe some kind of ID number, such as the primary key in a relational database.

A *Value Object* bases its notion of equality on field values within the class. Thus, two date objects may be the same if their day, month, and year values are the same.

Since *Value Objects* are small and easily created, they're often passed around by value instead of by reference.

For value objects to work properly in these cases it's a very good idea to make them immutable, that is, once created none of their fields change.

The reason for this is to avoid aliasing bugs. An aliasing bug occurs when two objects share the same value object and one of the owners changes the values in it.

Thus, if Martin has a hire date of March 18 and we know that Cindy was hired on the same day, we may set Cindy's hire date to be the same as Martin's. If Martin then changes the month in his hire date to May, Cindy's hire date changes too.

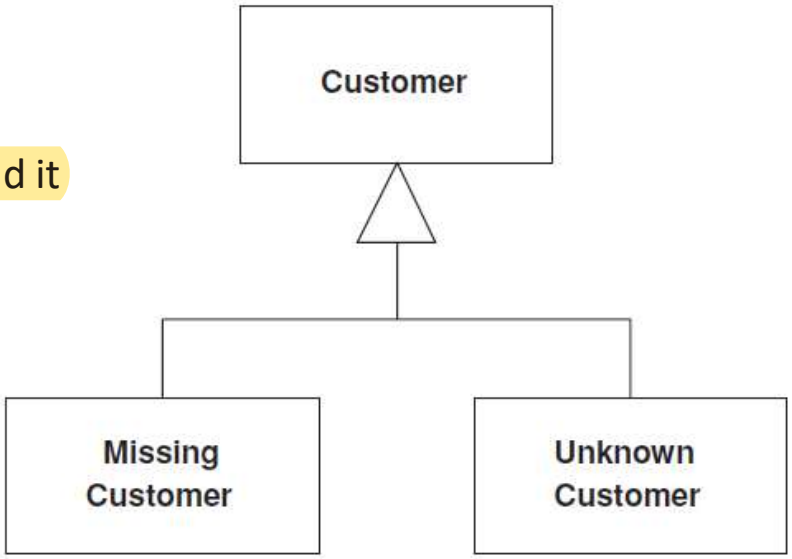
Special case

A subclass that provides special behavior for particular cases.

If it's possible for a variable to be null, you have to remember to surround it with null test code so you'll do the right thing if a null is present.

Often the right thing is the same in many contexts, so you end up writing similar code in lots of places

In number systems you have to deal with infinity, which has special rules for things like addition that break the usual invariants of real numbers.

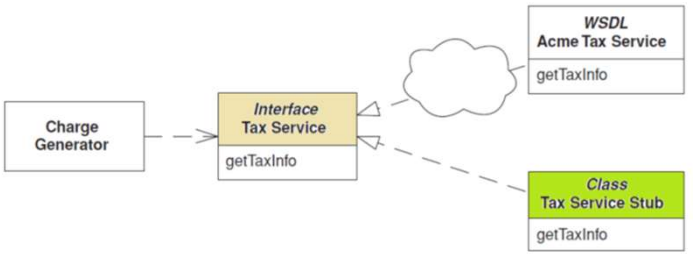


null customer may mean no customer or it may mean that there's a customer but we don't know who it is. Rather than just using a null customer, consider having separate *Special Cases* for missing customer and unknown customer.

Instead of returning null, or some odd value, return a *Special Case* that has the same interface as what the caller expects.

The basic idea is to create a subclass to handle the *Special Case*. Thus, if you have a customer object and you want to avoid null checks, you make a null customer object. Take all of the methods for customer and override them in the *Special Case* to provide some harmless behavior. Then, whenever you have a null, put in an instance of null customer instead.

Service Stub



Removes dependence upon problematic services during testing.

Enterprise systems often depend on access to third-party services such as credit scoring, tax rate lookups, and pricing engines.
Any developer who has built such a system can speak to the frustration of being dependent on resources completely out of his control.
Feature delivery is unpredictable, and as these services are often remote reliability and performance can suffer as well.

At the very least these problems slow the development process. Developers sit around waiting for the service to come back on line or maybe put some hacks into the code to compensate for yet to be delivered features.

Replacing the service during testing with a *Service Stub* that runs locally, fast, and in memory improves your development experience.

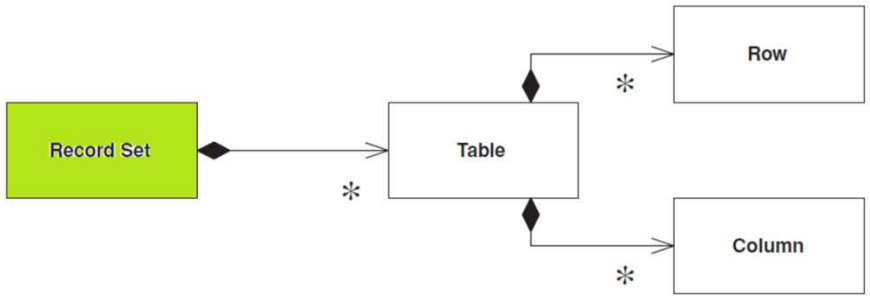
The key to writing a *Service Stub* is that you keep it as simple as possible

Let's walk through the process of stubbing a sales tax service that provides state sales tax amount and rate, given an address, product type, and sales amount. The simplest way to provide a *Service Stub* is to write two or three lines of code that use a flat tax rate to satisfy all requests.

Tax laws are not that simple, of course. Certain products are exempt from taxation in certain states, so we rely on our real tax service to know which product and state combinations are tax exempt. However, a lot of our application functionality depends on whether taxes are charged, so we need to accommodate tax exemption in our *Service Stub*. The simplest means of adding this behavior to the stub is via a conditional statement that exempt a specific combination of address and product and then uses that same data in any relevant test cases.

Use *Service Stub* whenever you find that dependence on a particular service is hindering your development and testing. Many practitioners of Extreme Programming use the term **Mock Object**, for a *Service Stub*.

Record Set



An in-memory representation of tabular data.

The idea of the *Record Set* is to provide you with an in-memory structure that looks exactly like the result of an SQL query but can be generated and manipulated by other parts of the system.

A *Record Set* is usually something that you won't build yourself, provided by the vendor of the software platform you're working with. Examples include the data set of ADO.NET and the row set of JDBC 2.0.

The ability to disconnect the *Record Set* from its link to the data source is very valuable. This allows you to pass the *Record Set* around a network without having to worry about database connections. Furthermore, if you can then easily serialize the *Record Set* it can also act as a *Data Transfer Object* for an application.

Disconnection raises the question of what happens when you update the *Record Set*. Increasingly platforms are allowing the *Record Set* to be a form of *Unit of Work*, so you can modify it and then return it to the data source to be committed.

The value of *Record Set* comes from having an environment that relies on it as a common way of manipulating data.

Record Set so valuable appeared because of the ever-presence of relational databases and SQL and the absence of any real alternative structure and query language.