VŠB TECHNICKÁ UNIVERZITA OSTRAVA

www.vsb.cz

VŠB TECHNICKÁ UNIVERZITA OSTRAVA | FAKULTA ELEKTROTECHNIKY A INFORMATIKY | KATEDRA INFORMATIKY

VEA – Vývoj Enterprise Aplikací

David Ježek
david.jezek@vsb.cz

Tel: 597 325 874
Místnost: EA406

Architecture?

- impressive-sounding words
- the highest-level breakdown of a system into its parts
- decisions that are hard to change
- architecture is a subjective thing, a shared understanding of a system's design by the expert developers on a project
- Commonly this shared understanding is in the form of the major components of the system and how they interact.
- decisions, in that it's the decisions that developers wish they could get right early on because they're perceived as hard to change
- if you find that something is easier to change than you once thought, then it's no longer architectural
- architecture boils down to the important stuff—whatever that is.

Architecture

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

"Architecture" is a term that lots of people try to define, with little agreement. There are two common elements: One is the highest-level breakdown of a system into its parts; the other, decisions that are hard to change. It's also increasingly realized that there isn't just one way to state a system's architecture; rather, there are multiple architectures in a system, and the view of what is architecturally significant is one that can change over a system's lifetime.

From time to time Ralph Johnson has a truly remarkable posting on a mailing list, and he did one on architecture just as I was finishing the draft of this book. In this posting he brought out the point that architecture is a subjective thing, a shared understanding of a system's design by the expert developers on a project. Commonly this shared understanding is in the form of the major components of the system and how they interact. It's also about decisions, in that it's the decisions that developers wish they could get right early on because they're perceived as hard to change. The subjectivity comes in here as well because, if you find that something is easier to change than you once thought, then it's no longer architectural. In the end architecture boils down to the important stuff—whatever that is.

In this book I present my perception of the major parts of an enterprise application and of the decisions I wish I could get right early on. The architectural pattern I like the most is that of layers, which I describe more in Chapter 1. This book is thus about how you decompose an enterprise application into layers and how these layers work together. Most nontrivial enterprise applications use a layered architecture of some form, but in some situations other approaches, such as pipes and filters, are valuable. I don't go into those situations, focusing instead on the context of a layered architecture because it's the most widely useful.

Some of the patterns in this book can reasonably be called architectural, in that they represent significant decisions about these parts; others are more about design and help you to realize that architecture. I don't make any strong attempt to separate the two, since what is architectural or not is so subjective.

Enterprise application characteristic

- Enterprise software, also known as enterprise application software (EAS), is purpose-designed computer software used to satisfy the needs of an organization rather than individual users. Such organizations can vary from businesses, schools, interest-based user groups[1] and clubs, retailers, or governments.[2] Enterprise software is an integral part of a (computer based) Information System, and as such includes web site software production.

https://en.wikipedia.org/wiki/Enterprise_software

Services provided by enterprise software are typically business-oriented tools such as online shopping and online payment processing, interactive product catalogue, automated billing systems, security, enterprise content management, IT service management, customer relationship management, enterprise resource planning, business intelligence, project management, collaboration, human resource management, manufacturing, enterprise application

enterprise software a software suite with common business applications, tools for modeling how the entire organization works, and development tools for building applications unique to the organizationintegration, and enterprise forms automation.

Enterprise application characteristic

- Persistent data
- A lot of data
- Access data concurrently
- A lot of user interface screens
- Integrate with other enterprise applications
- Conceptual dissonance
- complex business "illogic“

https://en.wikipedia.org/wiki/Enterprise_software

Services provided by enterprise software are typically business-oriented tools such as online shopping and online payment processing, interactive product catalogue, automated billing systems, security, enterprise content management, IT service management, customer relationship management, enterprise resource planning, business intelligence, project management, collaboration, human resource management, manufacturing, enterprise application

enterprise software a software suite with common business applications, tools for modeling how the entire organization works, and development tools for building applications unique to the organizationintegration, and enterprise forms automation.

Enterprise Applications

Lots of people write computer software, and we call all of it software development. However, there are distinct kinds of software out there, each of which has its own challenges and complexities. This comes out when I talk with some of my friends in the telecom field. In some ways enterprise applications are much easier than telecoms software—we don't have very hard multithreading problems, and we don't have hardware and software integration. But in other ways it's much tougher. Enterprise applications often have complex data—and lots of it—to work on, together with business rules that fail all tests of logical reasoning. Although some techniques and patterns are relevant for all kinds of software, many are relevant for only one particular branch.

In my career I've concentrated on enterprise applications, so my patterns here are all about that. (Other terms for enterprise applications include "information systems" or, for those with a long memory, "data processing.") But what do I mean by the term "enterprise application"? I can't give a precise definition, but I can give some indication of my meaning.

I'll start with examples. Enterprise applications include payroll, patient records, shipping tracking, cost analysis, credit scoring, insurance, supply chain, accounting, customer service, and foreign exchange trading. Enterprise applications don't include automobile fuel injection, word processors, elevator controllers, chemical plant controllers, telephone switches, operating systems, compilers, and games.

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

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

Usually many people access data concurrently. For many systems this may be less than a hundred people, but for Web-based systems that talk over the Internet this goes up by orders of magnitude. With so many people there are definite issues in ensuring that all of them can access the system properly. But even without that many people, there are still problems in making sure that two people don't access the same data at the same time in a way that causes errors.

Enterprise application characteristic

- An enterprise application usually means an application that:
  - Serves a large number of users at once
  - Works with large amounts of data in a database
  - Communicates with other systems
  - Is robust and secure

https://en.wikipedia.org/wiki/Enterprise_software

Services provided by enterprise software are typically business-oriented tools such as online shopping and online payment processing, interactive product catalogue, automated billing systems, security, enterprise content management, IT service management, customer relationship management, enterprise resource planning, business intelligence, project management, collaboration, human resource management, manufacturing, enterprise application integration, and enterprise forms automation.

Enterprise application characteristic

- **Enterprise applications include** payroll, patient records, shipping tracking, cost analysis, credit scoring, insurance, supply chain, accounting, customer service, and foreign exchange trading.
- **Enterprise applications don't include** automobile fuel injection, word processors, elevator controllers, chemical plant controllers, telephone switches, operating systems, compilers, and games.

https://en.wikipedia.org/wiki/Enterprise_software

Services provided by enterprise software are typically business-oriented tools such as online shopping and online payment processing, interactive product catalogue, automated billing systems, security, enterprise content management, IT service management, customer relationship management, enterprise resource planning, business intelligence, project management, collaboration, human resource management, manufacturing, enterprise application integration, and enterprise forms automation.

Kinds of Enterprise Application

- B2C (business to customer) online retailer
- system that automates the processing of leasing agreements
- simple expense-tracking system for a small company

- can't come up with a single architecture that will be right for all three
- Choosing an architecture means that you have to understand the particular problems of your system and choose an appropriate design based on that understanding
- don't give a single solution for your enterprise needs
- many of the patterns are about choices and alternatives
- You can't build enterprise software without thinking

Kinds of Enterprise Application

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

Consider a B2C (business to customer) online retailer: People browse and—with luck and a shopping cart—buy. For such a system we need to be able to handle a very high volume of users, so our solution needs to be not only reasonably efficient in terms of resources used but also scalable so that you can increase the load by adding more hardware. The domain logic for such an application can be pretty straightforward: order capturing, some relatively simple pricing and shipping calculations, and shipment notification. We want anyone to be able access the system easily, so that implies a pretty generic Web presentation that can be used with the widest possible range of browsers. Data source includes a database for holding orders and perhaps some communication with an inventory system to help with availability and delivery information.

Contrast this with a system that automates the processing of leasing agreements. In some ways this is a much simpler system than the B2C retailer's because there are many fewer users—no more than a hundred or so at one time. Where it's more complicated is in the business logic. Calculating monthly bills on a lease, handling events such as early returns and late payments, and validating data as a lease is booked are all complicated tasks, since much of the leasing industry's competition comes in the form of little variations over deals done in the past. A complex business domain such as this is challenging because the rules are so arbitrary.

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

A third example point is a simple expense-tracking system for a small company. Such a system has few users and simple logic and can easily be made accessible across the company with an HTML presentation. The only data source is a few tables in a database. As simple as it is, a system like this is not devoid of a challenge. You have to build it very quickly and you have to bear in mind

Thinking About Performance

- Response time
- Responsiveness
- Latency
- Throughput
- performance is either throughput or response time—whichever matters more to you
- Load
- Load sensitivity - degradation
- Efficiency
- The capacity of a system
- Scalability
  - Vertical scalability, or scaling up
  - Horizontal scalability, or scaling out

Thinking About Performance

Many architectural decisions are about performance. For most performance issues I prefer to get a system up and running, instrument it, and then use a disciplined optimization process based on measurement. However, some architectural decisions affect performance in a way that's difficult to fix with later optimization. And even when it is easy to fix, people involved in the project worry about these decisions early.

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

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

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

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

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

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

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

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

In this terminology performance is either throughput or response time—whichever matters more to you. It can sometimes be difficult to talk about performance when a technique improves throughput but decreases response time, so it's best to use the more precise term. From a user's perspective responsiveness may be more important than response time, so improving responsiveness at a cost of response time or throughput will increase performance.

Load is a statement of how much stress a system is under, which might be measured in how many users are currently connected to it. The load is usually a context for some other measurement, such as a response time. Thus, you may say that the response time for some request is 0.5 seconds with 10 users and 2 seconds with 20 users.

Load sensitivity is an expression of how the response time varies with the load. Let's say that system A has a response time of 0.5 seconds for 10 through 20 users and system B has a response time of 0.2 seconds for 10 users that rises to 2 seconds for 20 users. In this case system A has a lower load sensitivity than system B. We might also use the term degradation to say that system B degrades more than system A.

Efficiency is performance divided by resources. A system that gets 30 tps on two CPUs is more efficient than a system that gets 40 tps on four identical CPUs.

The capacity of a system is an indication of maximum effective throughput or load. This might be an absolute maximum or a point at which the performance dips below an acceptable threshold.

Scalability is a measure of how adding resources (usually hardware) affects performance. A scalable system is one that allows you to add hardware and get a commensurate performance improvement, such as doubling how many servers you have to double your throughput. Vertical scalability, or scaling up, means adding more power to a single server, such as more memory. Horizontal scalability, or scaling out, means adding more servers.

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

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

Layering

- **benefits**
  - You can understand a single layer as a coherent whole without knowing much about the other layers.
  - You can substitute layers with alternative implementations of the same basic services
  - You minimize dependencies between layers.
  - Layers make good places for standardization.
  - Once you have a layer built, you can use it for many higher-level services.
- **downsides**
  - Layers encapsulate some, but not all, things well. As a result you sometimes get cascading changes.
  - Extra layers can harm performance.
- But the hardest part of a layered architecture is deciding what layers to have and what the responsibility of each layer should be.

Chapter 1. Layering

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

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

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

You can understand a single layer as a coherent whole without knowing much about the other layers. You can understand how to build an FTP service on top of TCP without knowing the details of how ethernet works.

You can substitute layers with alternative implementations of the same basic services. An FTP service can run without change over ethernet, PPP, or whatever a cable company uses.

You minimize dependencies between layers. If the cable company changes its physical transmission system, providing they make IP work, we don't have to alter our FTP service.

Layers make good places for standardization. TCP and IP are standards because they define how their layers should operate.

Once you have a layer built, you can use it for many higher-level services. Thus, TCP/IP is used by FTP, telnet, SSH, and HTTP. Otherwise, all of these higher-level protocols would have to write their own lower-level protocols.

Layering is an important technique, but there are downsides.

Layers encapsulate some, but not all, things well. As a result you sometimes get cascading changes. The classic example of this in a layered enterprise application is adding a field that needs to display on the UI, must be in the database, and thus must be added to every layer in between.

Extra layers can harm performance. At every layer things typically need to be transformed from one representation to another. However, the encapsulation of an underlying function often gives you efficiency gains that more than compensate. A layer that controls transactions can be optimized and will then make everything faster.

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

Layer vs. Tier

- **tier** as implying a physical separation
- **layer** to stress that you don't have to run the layers on different machines

When people discuss layering, there's often some confusion over the terms layer and tier. Often the two are used as synonyms, but most people see tier as implying a physical separation. Client–server systems are often described as two-tier systems, and the separation is physical: The client is a desktop and the server is a server. I use layer to stress that you don't have to run the layers on different machines. A distinct layer of domain logic often runs on either a desktop or the database server. In this situation you have two nodes but three distinct layers. With a local database I can run all three layers on a single laptop, but there will still be three distinct layers.

Layers

- **Presentation**
  - Provision of services, display of information (e.g., in Windows or HTML, handling of user request (mouse clicks, keyboard hits), HTTP requests, command-line invocations, batch API)
- **Domain**
  - Logic that is the real point of the system
- **Data Source**
  - Communication with databases, messaging systems, transaction managers, other packages

[Team LiB]

The Three Principal Layers

For this book I'm centering my discussion around an architecture of three primary layers: presentation, domain, and data source. (I'm following the names used in [Brown et al.]). Table 1.1 summarizes these layers.

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

Table 1.1. Three Principal Layers

Layer
Responsibilities
Presentation
Provision of services, display of information (e.g., in Windows or HTML, handling of user request (mouse clicks, keyboard hits), HTTP requests, command-line invocations, batch API)
Domain
Logic that is the real point of the system
Data Source
Communication with databases, messaging systems, transaction managers, other packages

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

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

Sometimes the layers are arranged so that the domain layer completely hides the data source from the presentation. More often, however, the presentation accesses the data store directly. While this is less pure, it tends to work better in practice. The presentation may interpret a command from the user, use the data source to pull the relevant data out of the database, and then let the domain logic manipulate that data before presenting it on the glass.

A single application can often have multiple packages of each of these three subject areas. An application designed to be manipulated not only by end users through a rich-client interface but also through a command line would have two presentations: one for the rich-client interface and one for the command line. Multiple data source components may be present for different databases, but would be particularly for communication with existing packages. Even the domain may be broken into distinct areas relatively separate from each other. Certain data source packages may only be used by certain domain packages.

So far I've talked about a user. This naturally raises the question of what happens when there is no a human being driving the software. This could be something new and fashionable like a Web service or something mundane and useful like a batch process. In the latter case the user is the client program. At this point it becomes apparent that there is a lot of similarity between the presentation and data source layers in that they both are about connection to the outside world. This is the logic behind Alistair Cockburn's Hexagonal Architecture pattern [wiki], which visualizes any system as a core surrounded by interfaces to external systems. In Hexagonal Architecture everything external is fundamentally an outside interface, and thus it's a symmetrical view rather than my asymmetric layering scheme.

I find this asymmetry useful, however, because I think there is a good distinction to be made between an interface that you provide as a service to others and your use of someone else's service. Driving down to the core, this is the real distinction I make between presentation and data source. Presentation is an external interface for a service your system offers to someone else, whether it be a complex human or a simple remote program. Data source is the interface to things that are providing a service to you. I find it beneficial to think about these differently because the difference in clients alters the way you think about the service.

Although we can identify the three common responsibility layers of presentation, domain, and data source for every enterprise application, how you separate them depends on how complex the application is. A simple script to pull data from a database and display it in a Web page may all be one procedure. I would still endeavor to separate the three layers, but in that case I might do it only by placing the behavior of each layer in separate subroutines. As the system gets more complex, I would break the three layers into separate classes. As complexity increased I would divide the classes into separate packages. My general advice is to choose the most appropriate form of separation for your problem but make sure you do some kind of separation—at least at the subroutine level.

Together with the separation, there's also a steady rule about dependencies: The domain and data source should never be dependent on the presentation. That is, there should be no subroutine call from the domain or data source code into the presentation code. This rule makes it easier to substitute different presentations on the same foundation and makes it easier to modify the presentation without serious ramifications deeper down. The relationship between the domain and the data source is more complex and depends upon the architectural patterns used for the data source.

One of the hardest parts of working with domain logic seems to be that people often find it difficult to recognize what is domain logic and what is other forms of logic. An informal test I like is to imagine adding a radically different layer to an application, such as a command-line interface to a Web application. If there's any functionality you have to duplicate in order to do this, that's a sign of where domain logic has leaked into the presentation. Similarly, do you have to duplicate logic to replace a relational database with an XML file?

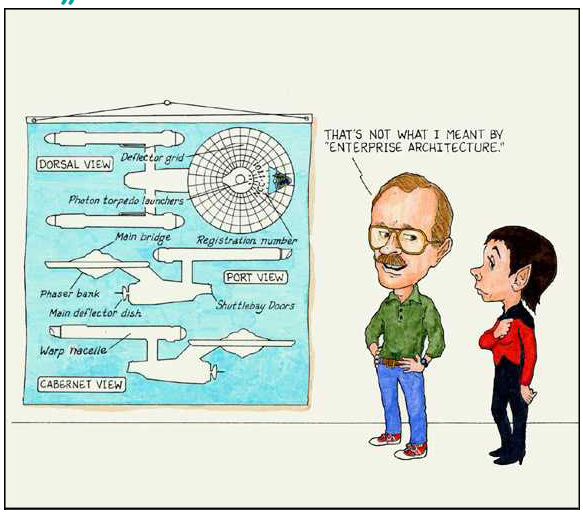
A good example of this is a system I was told about that contained a list of products in which all the products that sold over 10 percent more than they did the previous month were colored in red. To do this the developers placed logic in the presentation layer that compared this month's sales to last month's sales and if the difference was more than 10 percent, they set the color to red.

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

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

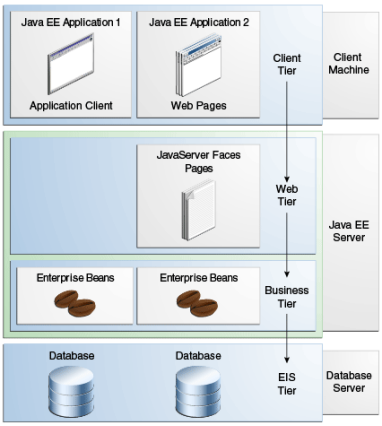
[Team LiB]

Example of „Good“ Architecture



Java EE Platform

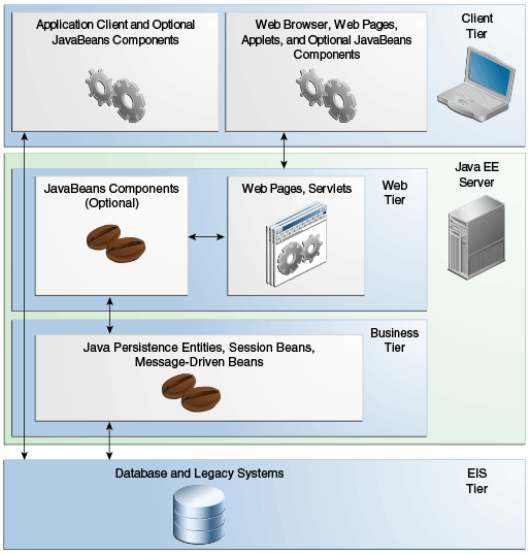
- Multi-tier architecture
- Java EE API
  - Servlets
  - Java Server Faces
  - Java Server Pages
  - Persistence API
  - …
- Runtime environment



The Java EE platform is built on top of the Java SE platform. The Java EE platform provides:

- Huge API with support for internet communication, HTTP connectivity, Web development, data persistency, …
- Runtime environment often called JavaEE container that provide server side runtime environment often included as part of web server.

We discuss this platform later in that course.

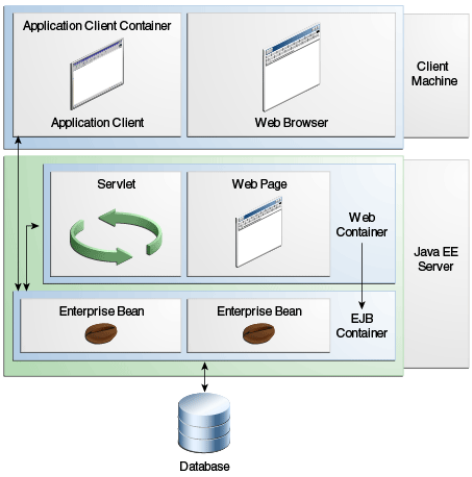


The Java EE platform is built on top of the Java SE platform. The Java EE platform provides:

- Huge API with support for internet communication, HTTP connectivity, Web development, data persistency, …
- Runtime environment often called JavaEE container that provide server side runtime environment often included as part of web server.

We discuss this platform later in that course.

Java EE Platform

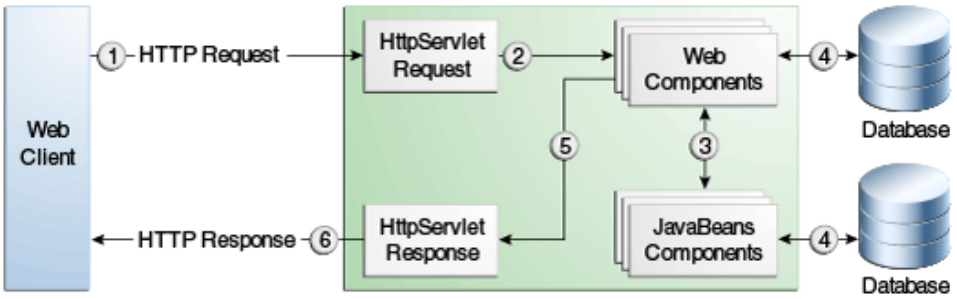


The Java EE platform is built on top of the Java SE platform. The Java EE platform provides:

- Huge API with support for internet communication, HTTP connectivity, Web development, data persistency, …
- Runtime environment often called JavaEE container that provide server side runtime environment often included as part of web server.

We discuss this platform later in that course.

Java EE – Web application



Web Applications

In the Java 2 platform, *web components provide the dynamic extension capabilities for a web* server. Web components are either Java servlets, JSP pages, or web service endpoints. The interaction between a web client and a web application is illustrated in Figure 3–1. The client sends an HTTP request to the web server. A web server that implements Java Servlet and JavaServer Pages technology converts the request into an HTTPServletRequest object. This object is delivered to a web component, which can interact with JavaBeans components or a database to generate dynamic content. The web component can then generate an HTTPServletResponse or it can pass the request to another web component. Eventually a web component generates a HTTPServletResponse object. The web server converts this object to an HTTP response and returns it to the client.

Servlets are Java programming language classes that dynamically process requests and construct responses. *JSP pages are text-based documents that execute as servlets but allow a more natural* approach to creating static content. Although servlets and JSP pages can be used interchangeably, each has its own strengths. Servlets are best suited for service-oriented applications (web service endpoints are implemented as servlets) and the control functions of a presentation-oriented application, such as dispatching requests and handling nontextual data.

JSP pages are more appropriate for generating text-based markup such as HTML, Scalable Vector Graphics (SVG), WirelessMarkup Language (WML), and XML.

Since the introduction of Java Servlet and JSP technology, additional Java technologies and frameworks for building interactive web applications have been developed. Figure 3–2 illustrates these technologies and their relationships.

Java EE – Technology for web services

Notice that Java Servlet technology is the foundation of all the web application technologies, so
you should familiarize yourself with the material in Chapter 4, “Java Servlet Technology,” even
if you do not intend to write servlets. Each technology adds a level of abstraction that makes
web application prototyping and development faster and the web applications themselves more
maintainable, scalable, and robust.
Web components are supported by the services of a runtime platform called a *web container. A*
web container provides services such as request dispatching, security, concurrency, and
life-cycle management. It also gives web components access to APIs such as naming,
transactions, and email.
Certain aspects of web application behavior can be configured when the application is installed,
or *deployed, to the web container. The configuration information is maintained in a text file in*
XML format called a *web application deployment descriptor (DD). ADDmust conform to the*
schema described in the Java Servlet Specification.
This chapter gives a brief overview of the activities involved in developing web applications.
First it summarizes the web application life cycle. Then it describes how to package and deploy
very simple web applications on the Application Server. It moves on to configuring web
applications and discusses how to specify the most commonly used configuration parameters. It
then introduces an example, Duke’s Bookstore, which illustrates all the Java EE web-tier
technologies, and describes how to set up the shared components of this example. Finally it
discusses how to access databases from web applications and set up the database resources
needed to run Duke’s Bookstore.

Java Servlet

Servlet

is a Java class that is used to extend the capabilities of a server that hosts applications accessed using the request-response model.

- Servlet technology is capable of handling any request, not just HTTP
- HTTP requests are the most common

As soon as the web began to be used for delivering services, service providers recognized the need for dynamic content. Applets, one of the earliest attempts toward this goal, focused on using the client platform to deliver dynamic user experiences. At the same time, developers also investigated using the server platform for this purpose. Initially, Common Gateway Interface (CGI) scripts were the main technology used to generate dynamic content. Although widely used, CGI scripting technology has a number of shortcomings, including platform dependence and lack of scalability. To address these limitations, Java Servlet technology was created as a portable way to provide dynamic, user-oriented content.

What Is a Servlet?
A *servlet is a Java programming language class that is used to extend the capabilities of servers* that host applications accessed by means of a request-response programming model. Although servlets can respond to any type of request, they are commonly used to extend the applications hosted by web servers. For such applications, Java Servlet technology defines HTTP-specific servlet classes.
The javax.servlet and javax.servlet.http packages provide interfaces and classes for writing servlets. All servlets must implement the Servlet interface, which defines life-cycle methods.
When implementing a generic service, you can use or extend the GenericServlet class provided with the Java Servlet API. TheHttpServlet class provides methods, such as doGet and doPost, for handling HTTP-specific services.
This chapter focuses on writing servlets that generate responses to HTTP requests.

HTTPServlet - example

```
@WebServlet(description = "desc", urlPatterns = {"/MyServlet"})
public class MyFirstServlet extends HttpServlet {
    public MyFirstServlet() {
        super();
    }
    public String getServletInfo() {
   return "My first servlet";
   }
   @Override
   protected void doGet(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
   doPost(request, response);
   }
   @Override
   protected void doPost(HttpServletRequest request, HttpServletResponse
response) throws ServletException, IOException {
   PrintWriter pw = response.getWriter();
   pw.println("<html><body>Hello world!</body></html>");
   pw.close();
   }
}
```


This example of source code for servlet generate web page with simple text “Hello world!”.

Servlet responses only on HTTP requests with HTTP methods GET and POST, because only methods **doPost()** and **doGet()** are overridden. Because we don’t need different response on method POST and GET the method **doGet()** simply call method **doPost()**.
Method **doPost()** just generates HTML code with simple text “Hello world!”

Patterns

- Name
- The intent and the sketch
- Describes a motivating problem
- How It Works
- When to Use
- Further Reading
- Examples

The Structure of the Patterns

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

The first item is the name of the pattern. Pattern names are crucial, because part of the purpose of patterns is to create a vocabulary that allows designers to communicate more effectively. Thus, if I tell you my Web server is built around a Front Controller (344) and a Transform View (361) and you know these patterns, you have a very clear idea of my web server's architecture.

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

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

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

When to Use It describes when the pattern should be used. Here I talk about the trade-offs that make you select this solution compared to others. Many of the patterns in this book are alternatives; such Page Controller (333) and Front Controller (344). Few patterns are always the right choice, so whenever I find a pattern I always ask myself, "When would I not use this?" That question often leads me to alternative patterns.

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

I like to add one or more examples. Each one is a simple example of the pattern in use, illustrated with some code in Java or C#. I chose those languages because they seem to be languages that the largest number of professional programmers can read. It's absolutely essential to understand that the example is not the pattern. When you use the pattern, it won't look exactly like this example so don't treat it as some kind of glorified macro. I've deliberately kept the example as simple as possible so you can see the pattern in as clear a form as I can imagine. All sorts of issues are ignored that will become important when you use it, but these will be particular to your own environment. This is why you always have to tweak the pattern.

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

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

The code in the examples is written with a focus on making the ideas understandable. As a result several things fall aside—in particular, error handling, which I don't pay much attention to since I haven't developed any patterns in this area yet. They are there purely to illustrate the pattern. They are not intended to show how to model any particular business problem.

For these reasons the code isn't downloadable from my Web site. Each code example in this book is surrounded with too much scaffolding to simplify the basic ideas so they're worth anything in a production setting.

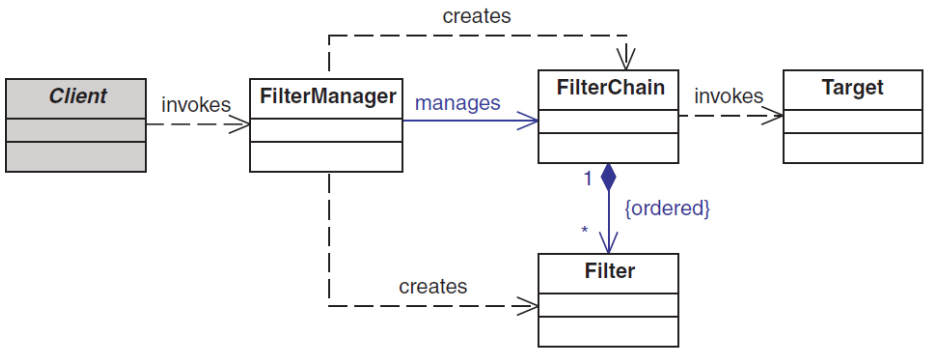
Not all the sections appear in all the patterns. If I couldn't think of a good example or motivation text, I left it out.

Patterns

- Why patterns
- Patterns and libraries
- Patterns and experts
- Overuse

http://martinfowler.com/ieeeSoftware/patterns.pdf

Intercepting Filter



Context

The presentation-tier request handling mechanism receives many different types of requests, which require varied types of processing. Some requests are simply forwarded to the appropriate handler component, while other requests must be modified, audited, or uncompressed before being further processed.

Problem

Preprocessing and post-processing of a client Web request and response are required.

When a request enters a Web application, it often must pass several entrance tests prior to the main processing stage. For example,

- Has the client been authenticated?
- Does the client have a valid session?
- Is the client's IP address from a trusted network?
- Does the request path violate any constraints?
- What encoding does the client use to send the data?
- Do we support the browser type of the client?

Some of these checks are tests, resulting in a yes or no answer that determines whether processing will continue. Other checks manipulate the incoming data stream into a form suitable for processing.

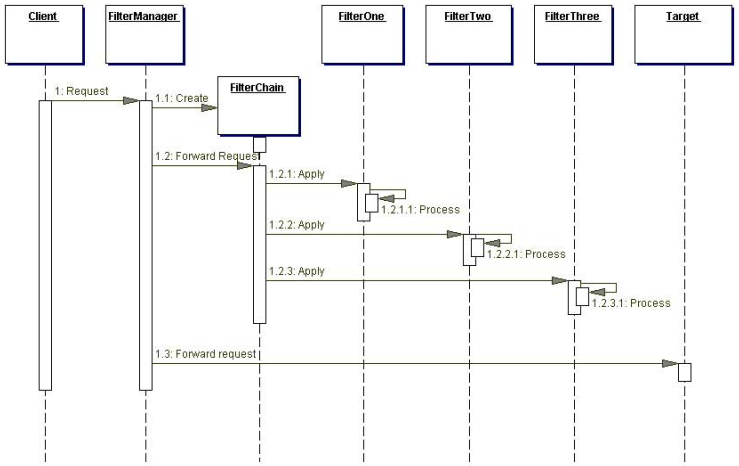
The classic solution consists of a series of conditional checks, with any failed check aborting the request. Nested if/else statements are a standard strategy, but this solution leads to code fragility and a copy-and-paste style of programming, because the flow of the filtering and the action of the filters is compiled into the application.

The key to solving this problem in a flexible and unobtrusive manner is to have a simple mechanism for adding and removing processing components, in which each component completes a specific filtering action.

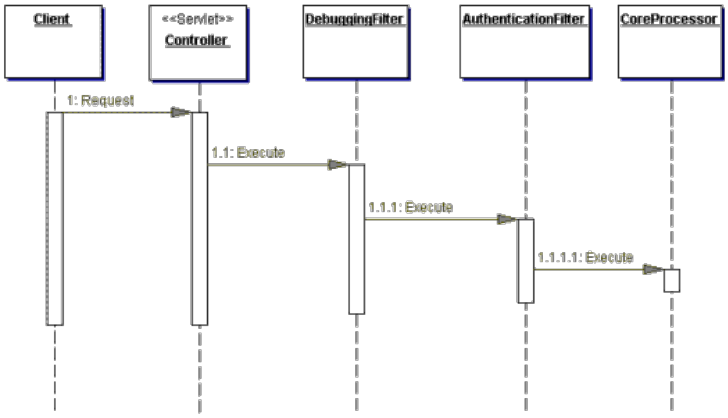
Forces

- Common processing, such as checking the data-encoding scheme or logging information about each request, completes per request.
- Centralization of common logic is desired.
- Services should be easy to add or remove unobtrusively without affecting existing components, so

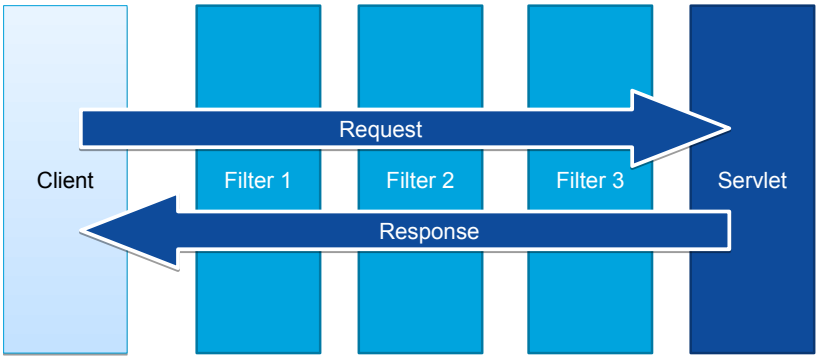
Intercepting Filter



Intercepting Filter



Servlet - Filters



Request filtering is another useful mechanism in web development. Java EE provides possibility of filter definition and mapping to URL pattern.

When client send request, web container build filter chain (ordered set of filters) according to requested URL. Request have to pass through all filters in the filter chain then is processed by servlet and have to go back through filter chain in reverse order.

Servlet - Filters

- Filter can change request before and response after servlet processing.
- Each filter have to implements interface **javax.servlet.Filter**
  - Filter method **doFilter()** is most important, because this method performs filtering.
- Interface **javax.servlet.FilterChain** is a parameter of method **Filter.doFilter()** and each filter should call method **FilterChain.doFilter()** to pass control to next filter in chain.

Servlet – Filter Example

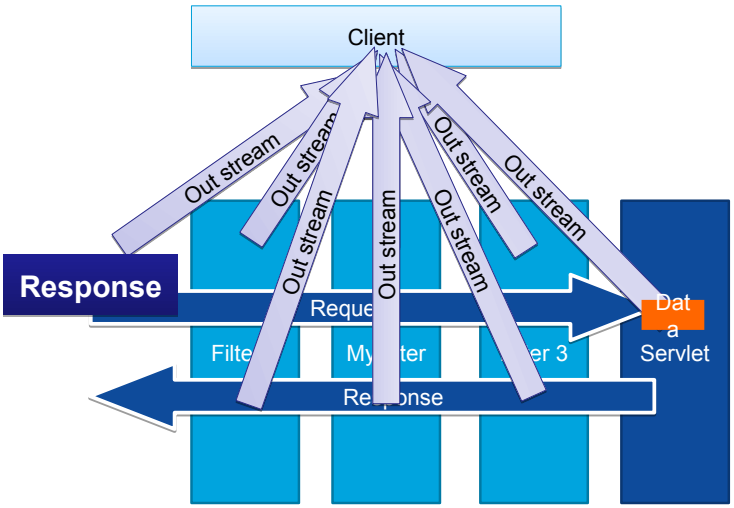
```
@WebFilter(filterName="/MyFilter", urlPatterns={"",""})
public class MyFilter implements Filter {
   public MyFilter() {}
   public void doFilter(ServletRequest request,
         ServletResponse response, FilterChain chain)
            throws IOException, ServletException {

      MyWrappedHttpResponse wrapper = new
            MyWrappedHttpResponse( (HttpServletResponse)
            response);
         chain.doFilter(request, wrapper);

      response.getWriter().write(wrapper.toString()
         .toUpperCase());
   }}
```


Source code implements simple filter example. Shown filter just convert all text from response to upper case.

Implemented class contains annotation that can substitute configuration from “web.xml” file.

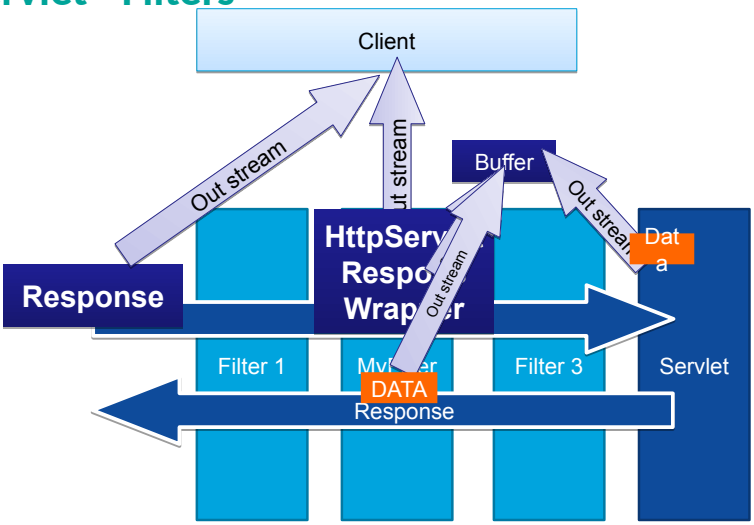
Method doFilter() just create response wrapper, call method FilterChain.doFilter() to pass control to next filter in the chain. When control is returned from method FilterChain.doFilter(), all other filters and servlet already process request and our filter can change text to upper case.

Servlet - Filters



If filter want process data for client from servlet or other filter it need response wrapper.
The animation describes filtering process if filter doesn't create response wrapper.
When request is passed to filtering process a response object is already created and contains output stream. Response output stream is connected directly to client and data passed to the output stream are immediately sent to client (web browser).
Servlet generate response data and pass the data to output stream.
Our filter "MyFilter" cannot convert already sent data to upper case.

Servlet - Filters



This animation describes filtering process if our filter create a response wrapper.

A response wrapper implements interface HttpServletResponse and the default implementation of wrapper (class HttpServletResponseWrapper) just forward all methods call to the original response object.

Implementation of the response wrapper in our example just creates a data buffer and redirect output stream to the data buffer. Servlet generates data and pass them to the output stream. The output stream sent data to the buffer and our filter “MyFilter” can read data from the buffer and change all character to upper case.

Servlet – Filter – ResponseWrapper

```
public class MyWrappedHttpResponse extends
             HttpServletResponseWrapper {
     private CharArrayWriter buffer;
     public MyWrappedHttpResponse(
          HttpServletResponse response) {
          super(response);
          buffer = new CharArrayWriter();
     }
     public String toString() {
          return buffer.toString();
     }
     public PrintWriter getWriter() {
          return new PrintWriter(buffer);
     }
}
```


Implementation of response wrapper from our example inherits from default response wrapper HttpServletResponseWrapper.

Our class add private field “buffer” of type CharArrayWriter, initialize the filed in constructor and override two methods getWriter() and toString().

Method getWriter() return output stream connected to buffer.

Method toString() return content of buffer as string.

Servlet – Filter Example

```
@WebFilter(filterName="/MyFilter", urlPatterns={"",""})
public class MyFilter implements Filter {
   public MyFilter() {}
   public void doFilter(ServletRequest request,
         ServletResponse response, FilterChain chain)
            throws IOException, ServletException {

      MyWrappedHttpResponse wrapper = new
            MyWrappedHttpResponse( (HttpServletResponse)
            response);
      chain.doFilter(request, wrapper);

      response.getWriter().write(wrapper.toString()
         .toUpperCase());
   }
}
```

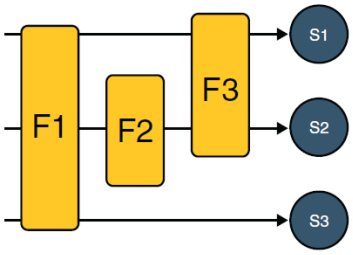

Source code implements simple filter example. Shown filter just convert all text from response to upper case.

Implemented class contains annotation that can substitute configuration from “web.xml” file.

Method doFilter() just create response wrapper, call method FilterChain.doFilter() to pass control to next filter in the chain. When control is returned from method FilterChain.doFilter(), all other filters and servlet already process request and our filter can change text to upper case.

Servlet – Filter Mapping

```
<filter>
   <display-name>MyFilter</display-name>
   <filter-name>MyFilter</filter-name>
   <filter-class>MyFilter</filter-class>
 </filter>
 <filter-mapping>
   <filter-name>MyFilter</filter-name>
   <url-pattern>/*</url-pattern>
   <dispatcher>REQUEST</dispatcher>
   <dispatcher>FORWARD</dispatcher>
   <dispatcher>INCLUDE</dispatcher>
   <dispatcher>ERROR</dispatcher>
</filter-mapping>
```



Filters are connected to the filter chain based on filter mapping. The filter mapping is defined in configuration file “web.xml” or can be specified by annotations in filter class.

The filter mapping contains URL pattern. If the URL pattern match with requested URL, the filter is added to the filter chain.

All URL pattern strings have to match excatly with requested URL except these:

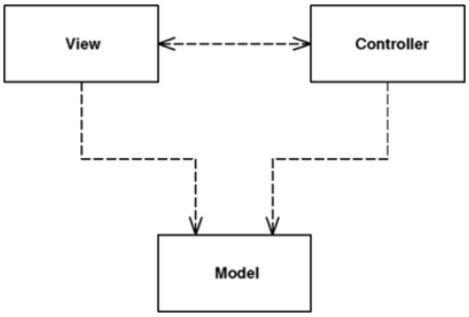
Pattern contains characters “/*” at end of the pattern string. Requested URL match even if contains suffix string.

Pattern contains characters “*.” at the beginning of the pattern string. Requested URL match if ends with specified extension.

Web presentation layer

Model View Controller

- separation of presentation from model
- the presentation depends on the model but the model doesn't depend on the presentation
- separation of view and controller, is less important
- The model and the view are obvious, but where's the controller? The common idea is that it sits between the model and the view



[Team LiB]

Model View Controller

Splits user interface interaction into three distinct roles

Model View Controller (MVC) is one of the most quoted (and most misquoted) patterns around. It started as a framework developed by Trygve Reenskaug for the Smalltalk platform in the late 1970s. Since then it has played an influential role in most UI frameworks and in the thinking about UI design.

How It Works

MVC considers three roles. The model is an object that represents some information about the domain. It's a nonvisual object containing all the data and behavior other than that used for the UI. In its most pure OO form the model is an object within a Domain Model (116). You might also think of a Transaction Script (110) as the model providing that it contains no UI machinery. Such a definition stretches the notion of model, but fits the role breakdown of MVC.

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

As I think about MVC I see two principal separations: separating the presentation from the model and separating the controller from the view.

Of these the separation of presentation from model is one of the most fundamental heuristics of good software design. This separation is important for several reasons.

Fundamentally presentation and view are about different concerns. When you're developing a view you're thinking about the mechanisms of UI and how to lay out a good user interface. When you're working with a model you are thinking about business policies, perhaps database interactions. Certainly you will use different very different libraries when working with one or the other. Often people prefer one area to another and they people specialize in one side of the line.

Depending on context, users want to see the same basic model information in different ways. Separating presentation and view allows you to develop multiple presentations—indeed, entirely different interfaces—and yet use the same model code. Most noticeably this could be providing the same model with a rich client, a Web browser, a remote API, and a command-line interface. Even within a single Web interface you might have different customer pages at different points in an application.

Nonvisual objects are usually easier to test than visual ones. Separating presentation and model allows you to test all the domain logic easily without resorting to things like awkward GUI scripting tools.

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

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

The second division, the separation of view and controller, is less important. Indeed, the irony is that almost every version of Smalltalk didn't actually make a view/controller separation. The classic example of why you'd want to separate them is to support editable and noneditable behavior, which you can do with one view and two controllers for the two cases, where the controllers are strategies [Gang of Four] for the view. In practice most systems have only one controller per view, however, so this separation is usually not done. It has come back into vogue with Web interfaces where it becomes useful for separating the controller and view again.

The fact that most GUI frameworks combine view and controller has led to many misquotations of MVC. The model and the view are obvious, but where's the controller? The common idea is that it sits between the model and the view, as in the Application Controller (379)—it doesn't help that the word "controller" is used in both contexts. Whatever the merits of a Application Controller (379), it's a very different beast from an MVC controller.

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

When to Use It

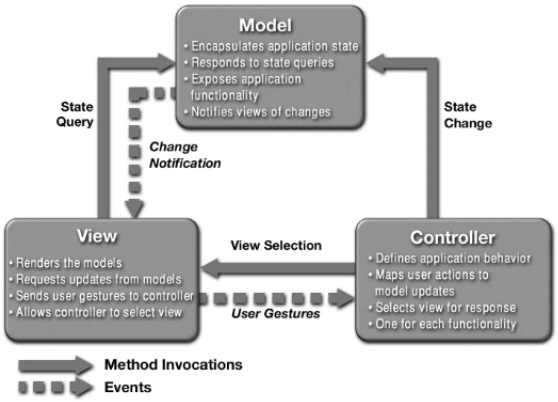
As I said, the value of MVC lies in its two separations. Of these the separation of presentation and model is one of the most important design principles in software, and the only time you shouldn't follow it is in very simple systems where the model has no real behavior in it anyway. As soon as you get some nonvisual logic you should apply the separation. Unfortunately, a lot of UI frameworks make it difficult, and those that don't are often taught without a separation.

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

[Team LiB]

Model View Controller

- Desktop application
- Java AWT GUI, Java SWING GUI



[Team LiB]

Model View Controller

Splits user interface interaction into three distinct roles

Model View Controller (MVC) is one of the most quoted (and most misquoted) patterns around. It started as a framework developed by Trygve Reenskaug for the Smalltalk platform in the late 1970s. Since then it has played an influential role in most UI frameworks and in the thinking about UI design.

How It Works

MVC considers three roles. The model is an object that represents some information about the domain. It's a nonvisual object containing all the data and behavior other than that used for the UI. In its most pure OO form the model is an object within a Domain Model (116). You might also think of a Transaction Script (110) as the model providing that it contains no UI machinery. Such a definition stretches the notion of model, but fits the role breakdown of MVC.

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

As I think about MVC I see two principal separations: separating the presentation from the model and separating the controller from the view.

Of these the separation of presentation from model is one of the most fundamental heuristics of good software design. This separation is important for several reasons.

Fundamentally presentation and view are about different concerns. When you're developing a view you're thinking about the mechanisms of UI and how to lay out a good user interface. When you're working with a model you are thinking about business policies, perhaps database interactions. Certainly you will use different very different libraries when working with one or the other. Often people prefer one area to another and they people specialize in one side of the line.

Depending on context, users want to see the same basic model information in different ways. Separating presentation and view allows you to develop multiple presentations—indeed, entirely different interfaces—and yet use the same model code. Most noticeably this could be providing the same model with a rich client, a Web browser, a remote API, and a command-line interface. Even within a single Web interface you might have different customer pages at different points in an application.

Nonvisual objects are usually easier to test than visual ones. Separating presentation and model allows you to test all the domain logic easily without resorting to things like awkward GUI scripting tools.

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

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

The second division, the separation of view and controller, is less important. Indeed, the irony is that almost every version of Smalltalk didn't actually make a view/controller separation. The classic example of why you'd want to separate them is to support editable and noneditable behavior, which you can do with one view and two controllers for the two cases, where the controllers are strategies [Gang of Four] for the view. In practice most systems have only one controller per view, however, so this separation is usually not done. It has come back into vogue with Web interfaces where it becomes useful for separating the controller and view again.

The fact that most GUI frameworks combine view and controller has led to many misquotations of MVC. The model and the view are obvious, but where's the controller? The common idea is that it sits between the model and the view, as in the Application Controller (379)—it doesn't help that the word "controller" is used in both contexts. Whatever the merits of a Application Controller (379), it's a very different beast from an MVC controller.

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

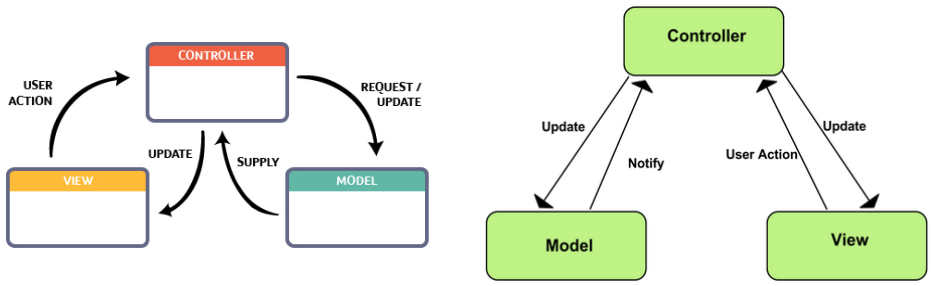
When to Use It

As I said, the value of MVC lies in its two separations. Of these the separation of presentation and model is one of the most important design principles in software, and the only time you shouldn't follow it is in very simple systems where the model has no real behavior in it anyway. As soon as you get some nonvisual logic you should apply the separation. Unfortunately, a lot of UI frameworks make it difficult, and those that don't are often taught without a separation.

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

[Team LiB]

Model View Controller - Architectonical style



[Team LiB]

Model View Controller

Splits user interface interaction into three distinct roles

Model View Controller (MVC) is one of the most quoted (and most misquoted) patterns around. It started as a framework developed by Trygve Reenskaug for the Smalltalk platform in the late 1970s. Since then it has played an influential role in most UI frameworks and in the thinking about UI design.

How It Works

MVC considers three roles. The model is an object that represents some information about the domain. It's a nonvisual object containing all the data and behavior other than that used for the UI. In its most pure OO form the model is an object within a Domain Model (116). You might also think of a Transaction Script (110) as the model providing that it contains no UI machinery. Such a definition stretches the notion of model, but fits the role breakdown of MVC.

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

As I think about MVC I see two principal separations: separating the presentation from the model and separating the controller from the view.

Of these the separation of presentation from model is one of the most fundamental heuristics of good software design. This separation is important for several reasons.

Fundamentally presentation and view are about different concerns. When you're developing a view you're thinking about the mechanisms of UI and how to lay out a good user interface. When you're working with a model you are thinking about business policies, perhaps database interactions. Certainly you will use different very different libraries when working with one or the other. Often people prefer one area to another and they people specialize in one side of the line.

Depending on context, users want to see the same basic model information in different ways. Separating presentation and view allows you to develop multiple presentations—indeed, entirely different interfaces—and yet use the same model code. Most noticeably this could be providing the same model with a rich client, a Web browser, a remote API, and a command-line interface. Even within a single Web interface you might have different customer pages at different points in an application.

Nonvisual objects are usually easier to test than visual ones. Separating presentation and model allows you to test all the domain logic easily without resorting to things like awkward GUI scripting tools.

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

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

The second division, the separation of view and controller, is less important. Indeed, the irony is that almost every version of Smalltalk didn't actually make a view/controller separation. The classic example of why you'd want to separate them is to support editable and noneditable behavior, which you can do with one view and two controllers for the two cases, where the controllers are strategies [Gang of Four] for the view. In practice most systems have only one controller per view, however, so this separation is usually not done. It has come back into vogue with Web interfaces where it becomes useful for separating the controller and view again.

The fact that most GUI frameworks combine view and controller has led to many misquotations of MVC. The model and the view are obvious, but where's the controller? The common idea is that it sits between the model and the view, as in the Application Controller (379)—it doesn't help that the word "controller" is used in both contexts. Whatever the merits of a Application Controller (379), it's a very different beast from an MVC controller.

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

When to Use It

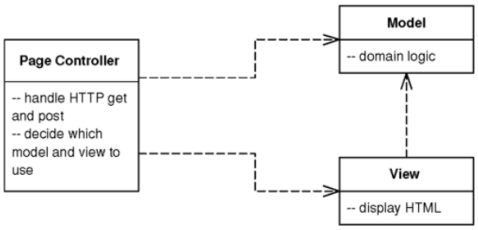
As I said, the value of MVC lies in its two separations. Of these the separation of presentation and model is one of the most important design principles in software, and the only time you shouldn't follow it is in very simple systems where the model has no real behavior in it anyway. As soon as you get some nonvisual logic you should apply the separation. Unfortunately, a lot of UI frameworks make it difficult, and those that don't are often taught without a separation.

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

[Team LiB]

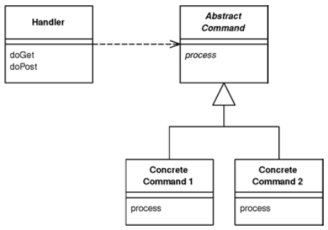
Page Controller

- An object that handles a request for a specific page or action on a Web site.
- As a result, Page Controller has one input controller for each logical page of the Web site. That controller may be the page itself, as it often is in server page environments, or it may be a separate object that corresponds to that page.



Front Controller

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



[Team LiB]

Front Controller

A controller that handles all requests for a Web site.

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

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

How It Works

A Front Controller handles all calls for a Web site, and is usually structured in two parts: a Web handler and a command hierarchy. The Web handler is the object that actually receives post or get requests from the Web server. It pulls just enough information from the URL and the request to decide what kind of action to initiate and then delegates to a command to carry out the action (see Figure 14.2).

Figure 14.2. How the Front Controller works.

The Web handler is almost always implemented as a class rather than as a server page, as it doesn't produce any response. The commands are also classes rather than server pages and in fact don't need any knowledge of the Web environment, although they're often passed the HTTP information. The Web handler itself is usually a fairly simple program that does nothing other than decide which command to run.

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

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

With dynamic invocation you can put the name of the command class into the URL or you can use a properties file that binds URLs to command class names. The properties file is another file to edit, but it does make it easier to change your class names without a lot of searching through your Web pages.

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

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

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

When to Use It

The Front Controller is a more complicated design than its obvious counterpart, Page Controller (333). It therefore needs a few advantages to be worth the effort.

Only one Front Controller has to be configured into the Web server; the Web handler does the rest of the dispatching. This simplifies the configuration of the Web server, which is an advantage if the Web server is awkward to configure. With dynamic commands you can add new commands without changing anything. They also ease porting since you only have to register the handler in a Web-server-specific way.

Because you create new command objects with each request, you don't have to worry about making the command classes thread-safe. In this way you avoid the headaches of multi-threaded programming; however, you do have to make sure that you don't share any other objects, such as the model objects.

A commonly stated advantage of a Front Controller is that it allows you to factor out code that's otherwise duplicated in Page Controller (333). To be fair, however, you can also do much of this with a superclass Page Controller (333).

There's just one controller, so you can easily enhance its behavior at runtime with decorators [Gang of Four]. You can have decorators for authentication, character encoding, internationalization, and so forth, and add them using a configuration file or even while the server is running. ([Alur et al.] describe this approach in detail under the name Intercepting Filter.)

Further Reading

[Alur et al.] give a detailed description of how to implement Front Controller in Java. They also describe Intercepting Filter, which goes very well with Front Controller.

A number of Java Web frameworks use this pattern. An excellent example appears in [Struts].

Example: Simple Display (Java)

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

Figure 14.3. The classes that implement Front Controller.

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

class FrontServlet... public void doGet(HttpServletRequest request, HttpServletResponse response) throws IOException, ServletException { FrontCommand command = getCommand(request); command.init(getServletContext(), request, response); command.process(); } private FrontCommand getCommand(HttpServletRequest request) { try { return (FrontCommand) getCommandClass(request).newInstance(); } catch (Exception e) { throw new ApplicationException(e); } } private Class getCommandClass(HttpServletRequest request) { Class result; final String commandClassName = "frontController." + (String) request.getParameter("command") + "Command"; try { result = Class.forName(commandClassName); } catch (ClassNotFoundException e) { result = UnknownCommand.class; } return result; } The logic is straightforward. The handler tries to instantiate a class named by concatenating the command name and "Command." Once it has the new command it initializes it with the necessary information from the HTTP server. I've passed in what I need for this simple example. You may well need more, such as the HTTP session. If you can't find a command, I've used the Special Case (496) pattern and returned an unknown command. As is often the case, Special Case (496) allows you to avoid a lot of extra error checking.

Commands share a fair bit of data and behavior. They all need to be initialized with information from the Web server.

class FrontCommand... protected ServletContext context; protected HttpServletRequest request; protected HttpServletResponse response; public void init(ServletContext context, HttpServletRequest request, HttpServletResponse response) { this.context = context; this.request = request; this.response = response; } They can also provide common behavior, such as a forward method, and define an abstract process command for the actual commands to override.

class FrontCommand... abstract public void process()throws ServletException, IOException ; protected void forward(String target) throws ServletException, IOException { RequestDispatcher dispatcher = context.getRequestDispatcher(target); dispatcher.forward(request, response); } The command object is very simple, at least in this case. It just implements the process method, which involves invoking the appropriate behavior on the model objects, putting the information needed for the view into the request, and forwarding to a Template View (350).

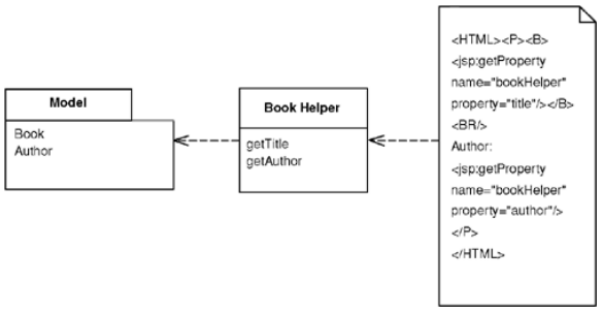
class ArtistCommand... public void process() throws ServletException, IOException { Artist artist = Artist.findNamed(request.getParameter("name")); request.setAttribute("helper", new ArtistHelper(artist)); forward("/artist.jsp"); } The unknown command just brings up a boring error page.

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

[Team LiB]

Template View

- Renders information into HTML by embedding markers in an HTML page.



[Inserts 1.00]

Template View
Renders information into HTML by embedding markers in an HTML page.
Writing a program that spits out HTML is often more difficult than you might imagine. Although programming languages are better at creating text than they used to be (some of us remember character handling in Fortran and standard Pascal), creating and concatenating string constructs is still painful. If there isn't much to do, it isn't too bad, but a whole HTML page is a lot of text manipulation.
With static HTML pages—those that don't change from request to request—you can use nice WYSIWG editors. Even those of us who like raw text editors find it easier to just type in the text and tags rather than fiddle with string concatenation in a programming language.
Of course the issue is with dynamic Web pages—those that take the results of something like database queries and embed them into the HTML. The page looks different with each result, and as a result regular HTML editors aren't up to the job.
The best way to work is to compose the dynamic Web page as you do a static page but put in markers that can be resolved into calls to gather dynamic information. Since the static part of the page acts as a template for the particular response, I call this a Template View.
How It Works
The basic idea of Template View is to embed markers into a static HTML page when it's written. When the page is used to service a request, the markers are replaced by the results of some computation, such as a database query. This way the page can be laid out in the usual manner, often with WYSIWYG editors, often by people who aren't programmers. The markers then communicate with real programs to put in the results.
A lot of tools use Template View. As a result this pattern isn't about how to build one yourself, but about how to use one effectively and what the alternative is.
Embedding the Markers
There are a number of ways markers can be placed in the HTML. One is to use HTML-like tags. This works well with WYSIWYG editors because they realize that anything between the angled brackets (<>) is special and so either ignore it or treat it differently. If the tags follow the rules for well-formed XML you can also use XML tools on the resulting document (providing your HTML is XHMTL, of course).
Another way to do this is to use special text markers in the body text. WYSIWYG editors treat that as regular text, still ignoring it but probably doing annoying things to it like spell checking. The advantage is that the syntax can be easier than the clunky syntax of HTML/XML.
Many environments provide the set of tags you use but more and more platforms now give you the ability to define your own tags and markers so you can design them to fit in with your particular needs.
One of the most popular forms of Template View is a server page such as ASP, JSP, or PHP. These actually go a step further than the basic form of a Template View in that they allow you to embed arbitrary programming logic, referred to as scriptlets, into the page. In my view, however, this feature is actually a big problem and you're better off limiting yourself to basic Template View behavior when you use server page technology.
The most obvious disadvantage of putting a lot of scriptlets into a page is that it eliminates the possibility of nonprogrammers editing the page. This is particularly important when you're using graphic designers for the page design. However, the biggest problems of embedding scriptlets into the page come from the fact that a page is poor module for a program. Even with an object-oriented language the page construct loses you most of the structural features that make it possible to do a modular design either in OO or in procedural style.
Even worse, putting a lot of scriptlets into the page makes it too easy to mingle the different layers of an enterprise application. When domain logic starts turning up on server pages it becomes far too difficult to structure it well and far too easy to duplicate it across different server pages. All in all, the worst code I've seen in the last few years has been server page code.
Helper Object
The key to avoiding scriptlets is to provide a regular object as a helper to each page. This helper has all the real programming logic. The page only has calls into it, which simplifies the page and makes it a more pure Template View. The resulting simplicity allows nonprogrammers to edit the page and programmers to concentrate on the helper. Depending on the actual tool you're using, you can often reduce all the templates in a page to HTML/XML tags, which keeps the page more consistent and more amenable to tool support.
This sounds like a simple and commendable principle, but as ever quite a few dirty issues make things more complicated. The simplest markers are those that get some information from the rest of the system and put in the correct place on the page. They are easily translated into calls to the helper that result in text, (or something that's trivially turned into text), and the engine places the text on the page.
Conditional Display
A more knotty issue is conditional page behavior. The simplest case is the situation where something is displayed only if a condition is true. That might be some kind of conditional tag along the lines of <IF condition = "$pricedrop > 0.1"> ...show some stuff </IF>, The trouble with this is that when you start having conditional tags like this, you start going down the path of turning the templates into a programming language in and of themselves. This leads you into all the same problems you face when you embed scriptlets in the page. If you need a full programming language, you might as well use scriptlets, but you know what I think of that idea!
As a result, I see purely conditional tags as a bad smell, something you should try to avoid. You can't always avoid them, but you should try to come up with something more focused than a general purpose <IF> tag.
If you're displaying some text conditionally, one option is to move the condition into the helper. The page will then always insert the result of the call into helper. It's just that if the condition isn't true the helper will send back an empty string, but this way the helper holds all the logic. This approach works best if there's no markup for the returned text or it's enough to return empty markup that gets ignored by the browser.
This doesn't work if, say, you want to highlight good-selling items in a list by putting their names in bold. In such a situation you always need the names displayed but sometimes you want the special markup. One way to get it is to have the helper generate the markup. This keeps all the logic out of the page, at the cost of moving the choice of highlighting mechanism away from the page designer and giving it to the programming code.
In order to keep the choice of HTML in the hands of the page design, you need some form of conditional tag. However it's important to look beyond a simple <IF>. A good route to go is a focused tag, so rather than a tag that looks like this:
<IF expression = "isHighSelling()"><B></IF> <property name = "price"/> <IF expression = "isHighSelling()"></B></IF> you have one like
<highlight condition = "isHighSelling" style = "bold"> <property name = "price"/> </highlight> In either case it's important that the condition be done based on a single Boolean property of the helper. Putting any more complex expression into the page is actually putting the logic into the page itself.
Another example would be putting information on a page that depends on the locale on which the system is running. Consider some text that should only be shown in the United States or Canada, which, rather than
<IF expression = "locale = 'US' || "CA"> ...special text </IF> Would be something like
<locale includes = "US, CA"> ...special text </locale> **Iteration**
Iterating over a collection presents similar issues. If you want a table where each line corresponds to a line item on an order, you need a construct that allows easy display of information for each line. Here it's hard to avoid a general iteration over a collection tag, but it usually works simply enough to fit in quite well.
Of course the kinds of tag you have to work with are often limited by the environment you're in. Some environments give you a fixed set of templates, in which case you may be more limited than you would like in following these kinds of guidelines. In other environments, however, you may have more choice in the tags to use; many of them even allow you to define your own tag libraries.
When to Process
The name Template View brings out the fact that the primary function of this pattern is to play the view in Model View Controller (330). For many systems the Template View should only play the view. In simpler systems it may be reasonable for it to play the controller, and possibly even the model, although I would strive to separate model processing as much as possible. Where Template View is taking on responsibilities beyond the view, it's important to ensure that these responsibilities are handled by the helper, not by the page. Controller and model responsibilities involve program logic, which program logic everywhere, should sit in the helper.
Any template system needs extra processing by the Web server. This can either be done by compiling the page after it's created, compiling the page or on its first request, or by interpreting the page on each request. Obviously the latter isn't a good idea if the interpretation takes a while to do.
One thing to watch with Template View is exceptions. If an exception works its way up to the Web container, you may find yourself with a half-handled page that provides some decidedly odd output to the calling browser instead of a redirect. You need to look into how your Web server handles exceptions; if it does something strange, catch all exceptions yourself in the helper class (yet another reason to disdain scriptlets.)
Using Scripts
Although server pages are one of the most common forms of Template View these days, you can write scripts in a Template View style. I've seen a fair bit of Perl done this way. Most noticeably demonstrated by perl's CGI.pm, the trick is to avoid concatenating strings by having function calls that output the appropriate tags to the response. This way you can write the script in your programming language and avoid the mess of interspersing print strings with programming logic.
When to Use It
For implementing the view in Model View Controller (330) the main choice is between Template View and Transform View (361). The strength of Template View is that it allows you to compose the content of the page by looking at the page structure. This seems to be easier for most people to do and to learn. In particular it nicely supports the idea of a graphic designer laying out a page with a programmer working on the helper.
Template View has two significant weaknesses. First, the common implementations make it too easy to put complicated logic in the page, thus making it hard to maintain, particularly by nonprogrammers. You need good discipline to keep the page simple and display oriented, putting logic in the helper. The second weakness is that Template View is harder to test than Transform View (361). Most implementations of Template View are designed to work within a Web server and are very difficult or impossible to test otherwise. Transform View (361) implementations are much easier to hook into a testing harness and test without a running Web server.
In thinking about a view you also need to consider Two Step View (365). Depending on your template scheme you may be able to implement this pattern using specialized tags. However, you may find it easier to implement it based on a Transform View (361). If you're going to need Two Step View (365) you need to take that into account in your choice.
Example: Using a JSP as a View with a Separate Controller (Java)
When using a JSP as a view only, it's always be invoked from a controller rather than directly from the servlet container. Thus, it's important to pass to the JSP any information it will need to figure out what to display. A good way to do this is to have the controller create a helper object and pass it to the JSP using the HTTP request. We'll show this with the simple display example from Page Controller (333). The Web-handling method for the servlet looks like this:
class ArtistController... public void doGet(HttpServletRequest request, HttpServletResponse response) throws IOException, ServletException { Artist artist = Artist.findNamed(request.getParameter("name")); if (artist == null) forward("/MissingArtistError.jsp", request, response); else { request.setAttribute("helper", new ArtistHelper(artist)); forward("/artist.jsp", request, response); } } As far as the Template View is concerned the important behavior is creating the helper and placing it in the request. The server page can now reach the helper with the useBean tag.
<jsp:useBean id="helper" type="actionController.ArtistHelper" scope="request"/> With the helper in place we can use it to access the information we need to display. The model information the helper needs was passed to it when it was created.
class ArtistHelper... private Artist artist; public ArtistHelper(Artist artist) { this.artist = artist; } We can use the helper to get appropriate information from the model. In the simplest case we provide a method to get some simple data, such as the artist's name.
class ArtistHelper... public String getName() { return artist.getName(); } Then we access this information by a Java expression.
<B> <%=helper.getName()%></B> or a property
<B><jsp:getProperty name="helper" property="name"/></B> The choice between properties or expressions depends on who is editing the JSP. Programmers find expressions easy to read and more compact, but HTML editors may not be able to handle them. Nonprogrammers will probably prefer tags, since they fits in the general form of HTML and leave less room for confusing errors.
Using a helper is one way to remove awkward scriptlet code. If you want to show a list of albums for an artist, you need to run a loop, which you can do with a scriptlet in the server page.
<UL> <% for (Iterator it = helper.getAlbums().iterator(); it.hasNext();) { Album album = (Album) it.next();%> <LI><%=album.getTitle()%></LI> <% } %> </UL> Frankly, this mix of Java and HTML is really horrible to read. An alternative is to move the for loop to the helper.
class ArtistHelper... public String getAlbumList() { StringBuffer result = new StringBuffer(); result.append("<UL>"); for (Iterator it = getAlbums().iterator(); it.hasNext();) { Album album = (Album) it.next(); result.append("<LI>"); result.append(album.getTitle()); result.append("</LI>"); } result.append("</UL>"); return result.toString(); } public List getAlbums() { return artist.getAlbums(); } I find this easier to follow because the amount of HTML is quite small. It also allows you to use a property to get the list. Many people don't like putting HTML code in helpers. While I prefer not to, given the choice between this and scriptlets I'd choose HTML in helpers any day or night.
The best route to go is a specialized tag for iteration.
<UL><tag:forEach host = "helper" collection = "albums" id = "each"> <LI><jsp:getProperty name="each" property="title"/></LI> </tag:forEach></UL> This is a much nicer alternative as it keeps scriptlets out of the JSP and HTML out of the helper.
Example: ASP.NET Server Page (C#)
This example continues the one I started in Page Controller (333) (page 340). To remind you, it shows the scores made by batsmen in a single innings of a cricket match. For those who think that cricket is a small noisy insect, I'll pass over the long rhapsodies about the world's most immortal sport and boil it all down to the fact that the page displays three essential pieces of information:
An ID number to reference the match
Which team's scores are shown and which innings the scores are for
A table showing each batsman's name, score, and run rate (the number of balls he faced divided by the runs he scored)
If you don't understand what these statistics mean, don't worry about it. Cricket is full of statistics—perhaps its greatest contribution to humanity is providing odd statistics for eccentric papers.
The Page Controller (333) discussion covered how a Web request is handled. To sum up, the object that acts as both the controller and the view is the aspx ASP.NET page. To keep the controller code out of a scriptlet, you define a separate code behind class.
<%@ Page language="c#" Codebehind="bat.aspx.cs" AutoEventWireup="false" Inherits="batsmen.BattingPage" %> The page can access the methods and properties of the code behind class directly. Furthermore, the code behind can define a Page_Load method to handle the request. In this case I've defined the Page_Load as a template method [Gang of Four] on a Layer Supertype (475).
class CricketPage... protected void Page_Load(object sender, System.EventArgs e) { db = new OleDbConnection(DB.ConnectionString); if (hasMissingParameters()) errorTransfer (missingParameterMessage); DataSet ds = getData(); if (hasNoData (ds)) errorTransfer ("No data matches your request"); applyDomainLogic (ds); DataBind(); prepareUI(ds); } For the purposes of Template View I can ignore all but the last couple of lines of the page load. The call to DataBind allows various page variables to be properly bound to their underlying data sources. That will do for the simpler cases, but for more complicated cases the last line calls a method in the particular page's code behind to prepare any objects for its use.
The match ID number, team, and innings are single values for the page, all of which came into the page as parameters in the HTTP request. I can provide these values by using properties on the code behind class.
class BattingPage... protected String team { get {return Request.Params["team"];} } protected String match { get {return Request.Params["match"];} } protected String innings { get {return Request.Params["innings"];} } protected String ordinalInnings{ get {return (innings == "1") ? "1st" : "2nd";} } With the properties defined, I can use them in the text of the page.
<P> Match id: <asp:label id="matchLabel" Text="<%# match %>" runat="server" font-bold="True"> </asp:label> </P> <P> <asp:label id=teamLabel Text="<%# team %>" runat="server" font-bold="True"> </asp:label> <asp:Label id=inningsLabel Text="<%# ordinalInnings %>" runat="server"> </asp:Label> innings</P> <P> The table is a little more complicated, but actually works easily in practice because of the graphical design facilities in Visual Studio. Visual Studio provides a data grid control that can be bound to a single table from a data set. I can do this binding in the prepareUI method that's called by the Page_Load method.
class BattingPage... override protected void prepareUI(DataSet ds) { DataGrid1.DataSource = ds; DataGrid1.DataBind(); } The batting class is a Table Module (125) that provides domain logic for the batting table in the database. Its data property is the data from that table enriched by domain logic from Table Module (125). Here the enrichment is the run rate, which is calculated rather than stored in the database.
With the ASP.NET data grid you can select which table columns you wish to display in the Web page, together with information about the table's appearance. In this case we can select name, runs, and rate columns.
<asp:DataGrid id="DataGrid1" runat="server" Width="480px" Height="171px" BorderColor="#336666" BorderStyle="Double" BorderWidth="3px" BackColor="White" CellPadding="4" GridLines="Horizontal" AutoGenerateColumns="False"> <SelectedItemStyle Font-Bold="True" ForeColor="White" BackColor="#339966"></ SelectedItemStyle> <ItemStyle ForeColor="#333333" BackColor="White"></ItemStyle> <HeaderStyle Font-Bold="True" ForeColor="White" BackColor="#336666"></HeaderStyle> <FooterStyle ForeColor="#333333" BackColor="White"></FooterStyle> <Columns> <asp:BoundColumn DataField="name" HeaderText="Batsman"> <HeaderStyle Width="70px"></HeaderStyle> </asp:BoundColumn> <asp:BoundColumn DataField="runs" HeaderText="Runs"> <HeaderStyle Width="30px"></HeaderStyle> </asp:BoundColumn> <asp:BoundColumn DataField="rateString" HeaderText="Rate"> <HeaderStyle Width="30px"></HeaderStyle> </asp:BoundColumn> </Columns> <PagerStyle HorizontalAlign="Center" ForeColor="White" BackColor="#336666" Mode="NumericPages"></PagerStyle> </asp:DataGrid></P> The HTML for this data grid looks intimidating, but in Visual Studio you don't manipulate it directly but through property sheets in the development environment, as you do for much of the rest of the page.
This ability to have Web form controls on the Web page that understand the ADO.NET abstractions of data sets and data tables is the strength, and limitation, of this scheme. The strength is that you transfer information through data sets, thanks to the kind of tools that Visual Studio provides. The limitation is that it only works seamlessly when you use patterns such as Table Module (125). If you have very complex domain logic, then a Domain Model (116) becomes helpful; to take advantage of the tools, the Domain Model (116) needs to create its own data set.

[Inserts 1.00]

Transform View

- A view that processes domain data element by element and transforms it into HTML.
- The key difference between Transform View and Template View (350) is the way in which the view is organized. A Template View (350) is organized around the output. A Transform View is organized around separate transforms for each kind of input element.

[Team LiB]

Transform View

A view that processes domain data element by element and transforms it into HTML.

When you issue requests for data to the domain and data source layers, you get back all the data you need to satisfy them, but without the formatting you need to make a proper Web page. The role of the view in Model View Controller (330) is to render this data into a Web page. Using Transform View means thinking of this as a transformation where you have the model's data as input and its HTML as output.

How It Works

The basic notion of Transform View is writing a program that looks at domain-oriented data and converts it to HTML. The program walks the structure of the domain data and, as it recognizes each form of domain data, it writes out the particular piece of HTML for it. If you think about this in an imperative way, you might have a method called renderCustomer that takes a customer object and renders it into HTML. If the customer contains a lot of orders, this method loops over the orders calling renderOrder.

The key difference between Transform View and Template View (350) is the way in which the view is organized. A Template View (350) is organized around the output. A Transform View is organized around separate transforms for each kind of input element. The transform is controlled by something like a simple loop that looks at each input element, finds the appropriate transform for that element, and then calls the transform on it. A typical Transform View's rules can be arranged in any order without affecting the resulting output.

You can write a Transform View in any language; at the moment, however, the dominant choice is XSLT. The interesting thing about this is that XSLT is a functional programming language, similar to Lisp, Haskell, and other languages that never quite made it into the IS mainstream. As such it has a different kind of structure to it. For example, rather than explicitly calling routines, XSLT recognizes elements in the domain data and then invokes the appropriate rendering transformations.

To carry out an XSLT transform we need to begin with some XML data. The simplest way this can happen is if the natural return type of the domain logic is either XML or something automatically transformable to it—for example, a .NET. Failing that, we need to produce the XML ourselves, perhaps by populating a Data Transfer Object (401) that can serialize itself into XML. That way the data can be assembled using a convenient API. In simpler cases a Transaction Script (110) can return XML directly.

The XML that's fed into the transform don't have to be a string, unless a string form is needed to cross a communication line. It's usually quicker and easier to produce a DOM and hand that to the transform.

Once we have the XML we pass it to an XSLT engine, which is becoming increasingly available commercially. The logic for the transform is captured in an XSLT style sheet, which we also pass to the transformer. The transformer then applies the stylesheet to the input XML to yield the output HTML, which we can write directly to the HTTP response.

When to Use It

The choice between a Transform View and a Template View (350) mostly comes down to which environment the team working on the view software prefers. The presence of tools is a key factor here. There are more and more HTML editors that you can use to write Template Views (350). Tools for XSLT are, at least so far, much less sophisticated. Also, XSLT can be an awkward language to master because of its functional programming style coupled with its awkward XML syntax.

One of the strengths of XSLT is its portability to almost any Web platform. You can use the same XSLT to transform XML created from J2EE or .NET, which can help in putting a common HTML view on data from different sources.

XSLT is also often easier if you're building a view on an XML document. Other environments usually require you to transform such a document into an object or to indulge in walking the XML DOM, which can be complicated. XSLT fits naturally in an XML world.

Transform View avoids two of the biggest problems with Template View (350). It's easier to keep the transform focused only on rendering HTML, thus avoiding having too much other logic in the view. It's also easy to run the Transform View and capture the output for testing. This makes it easier to test the view and you don't need a Web server to run the tests.

Transform View transforms directly from domain-oriented XML into HTML. If you need to change the overall appearance of a Web site, this can force you to change multiple transform programs. Using common transforms, such as with XSLT includes, helps reduce this problem. Indeed it's much easier to call common transformations using Transform View than it is using Template View (350). If you need to make global changes easily or support multiple appearances for the same data, you might consider Two Step View (365), which uses a two-stage process.

Example: Simple Transform (Java)

Setting up a simple transform involves preparing Java code for invoking the right style sheet to form the response. It also involves preparing the style sheet to format the response. Most of the response to a page is pretty generic, so it makes sense to use Front Controller (344). I'll describe only the command here, and you should look at Front Controller (344) to see how the command object fits in with the rest of the request-response handling.

All the command object does is invoke the methods on the model to obtain an XML input document, and then pass that XML document through the XML processor.

class AlbumCommand... public void process() { try { Album album = Album.findNamed(request.getParameter("name")); Assert.notNull(album); PrintWriter out = response.getWriter(); XsltProcessor processor = new SingleStepXsltProcessor("album.xsl"); out.print(processor.getTransformation(album.toXmlDocument())); } catch (Exception e) { throw new ApplicationException(e); } } The XML document may look something like this:

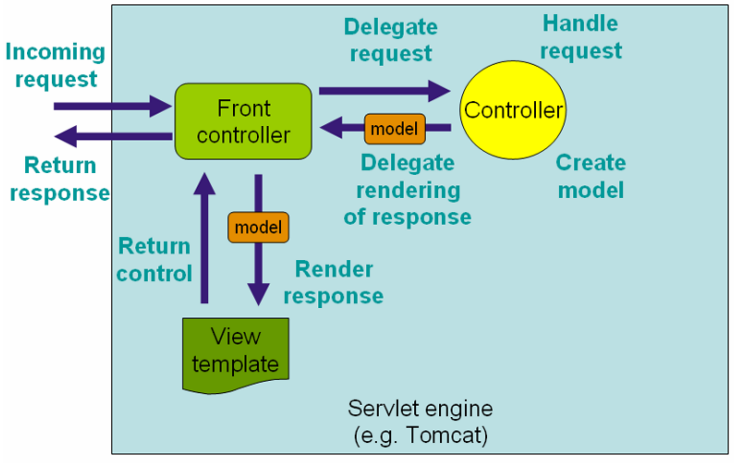
<album> <title>Stormcock</title> <artist>Roy Harper</artist> <trackList> <track><title>Hors d'Oeuvres</title><time>8:37</time></track> <track><title>The Same Old Rock</title><time>12:24</time></track> <track><title>One Man Rock and Roll Band</title><time>7:23</time></track> <track><title>Me and My Woman</title><time>13:01</time></track> </trackList> </album> The translation of the XML document is done by an XSLT program. Each template matches a particular part of the XML and produces the appropriate HTML output for the page. In this case I've kept the formatting to a excessively simple level to show just the essentials. The following template clauses match the basic elements of the XML file.

<xsl:template match="album"> <HTML><BODY bgcolor="white"> <xsl:apply-templates/> </BODY></HTML> </xsl:template> <xsl:template match="album/title"> <h1><xsl:apply-templates/></h1> </xsl:template> <xsl:template match="artist"> <P><B>Artist: </B><xsl:apply-templates/></P> </xsl:template> These template matches handle the table, which here has alternating rows highlighted in different colors. This is a good example of something that isn't possible with cascading style sheets but is reasonable with XML.

<xsl:template match="trackList"> <table><xsl:apply-templates/></table> </xsl:template> <xsl:template match="track"> <xsl:variable name="bgcolor"> <xsl:choose> <xsl:when test="(position() mod 2) = 1">linen</xsl:when> <xsl:otherwise>white</xsl:otherwise> </xsl:choose> </xsl:variable> <tr bgcolor="{$bgcolor}"><xsl:apply-templates/></tr> </xsl:template> <xsl:template match="track/title"> <td><xsl:apply-templates/></td> </xsl:template> <xsl:template match="track/time"> <td><xsl:apply-templates/></td> </xsl:template>

[Team LiB]

spring


Incoming request
Return response
Front controller
Delegate request
model
Controller
Handle request
Delegate rendering of response
Create model
Return control
model
Render response
View template
Servlet engine (e.g. Tomcat)

Thymeleaf

- Thymeleaf is a modern server-side Java template engine for both web and standalone environments.
- Thymeleaf's main goal is to bring elegant natural templates to your development workflow — HTML that can be correctly displayed in browsers and also work as static prototypes, allowing for stronger collaboration in development teams.
- With modules for Spring Framework, a host of integrations with your favourite tools, and the ability to plug in your own functionality, Thymeleaf is ideal for modern-day HTML5 JVM web development — although there is much more it can do.

Thymeleaf

```
<table>
  <thead>
    <tr>
      <th th:text="#{msgs.headers.name}">Name</th>
      <th th:text="#{msgs.headers.price}">Price</th>
    </tr>
  </thead>
  <tbody>
<tr th:each="prod: ${allProducts}">
<td th:text="${prod.name}">Oranges</td>
<td th:text="${#numbers.formatDecimal(prod.price, 1,
2)}">0.99</td>
</tr>
  </tbody>
</table>
```

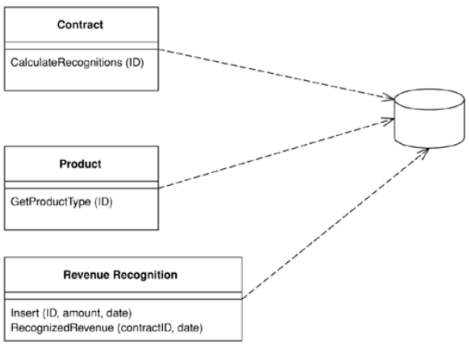
Presentation layer – main tasks

- EE Technolgy
- Formatter
- Formatting date
- Allowing Duplicate Form Submissions – bad
- Localization

Domain layer

Table Module

- A single instance that handles the business logic for all rows in a database table or view.
- A Table Module organizes domain logic with one class per table in the database, and a single instance of a class contains the various procedures that will act on the data.



[Team LiB]

Table Module

A single instance that handles the business logic for all rows in a database table or view.

One of the key messages of object orientation is bundling the data with the behavior that uses it. The traditional object-oriented approach is based on objects with identity, along the lines of Domain Model (116). Thus, if we have an Employee class, any instance of it corresponds to a particular employee. This scheme works well because once we have a reference to an employee, we can execute operations, follow relationships, and gather data on him.

One of the problems with Domain Model (116) is the interface with relational databases. In many ways this approach treats the relational database like a crazy aunt who's shut up in an attic and whom nobody wants to talk about. As a result you often need considerable programmatic gymnastics to pull data in and out of the database, transforming between two different representations of the data.

A Table Module organizes domain logic with one class per table in the database, and a single instance of a class contains the various procedures that will act on the data. The primary distinction with Domain Model (116) is that, if you have many orders, a Domain Model (116) will have one order object per order while a Table Module will have one object to handle all orders.

How It Works

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

Usually you use Table Module with a backing data structure that's table oriented. The tabular data is normally the result of a SQL call and is held in a Record Set (508) that mimics a SQL table. The Table Module gives you an explicit method-based interface that acts on that data. Grouping the behavior with the table gives you many of the benefits of encapsulation in that the behavior is close to the data it will work on.

Often you'll need behavior from multiple Table Modules in order to do some useful work. Many times you see multiple Table Modules operating on the same Record Set (508) (Figure 9.4).

Figure 9.4. Several Table Modules can collaborate with a single Record Set (508).

The most obvious example of Table Module is the use of one for each table in the database. However, if you have interesting queries and views in the database you can have Table Modules for them as well.

The Table Module may be an instance or it may be a collection of static methods. The advantage of an instance is that it allows you to initialize the Table Module with an existing record set, perhaps the result of a query. You can then use the instance to manipulate the rows in the record set. Instances also make it possible to use inheritance, so we can write a rush contract module that contains additional behavior to the regular contract.

The Table Module may include queries as factory methods. The alternative is a Table Data Gateway (144), but the disadvantage of this is having an extra Table Data Gateway (144) class and mechanism in the design. The advantage is that you can use a single Table Module on data from different data sources, since you use a different Table Data Gateway (144) for each data source.

When you use a Table Data Gateway (144) the application first uses the Table Data Gateway (144) to assemble data in a Record Set (508). You then create a Table Module with the Record Set (508) as an argument. If you need behavior from multiple Table Modules, you can create them with the same Record Set (508). The Table Module can then do business logic on the Record Set (508) and pass the modified Record Set (508) to the presentation for display and editing using the table-aware widgets. The widgets can't tell if the record sets came directly from the relational database or if a Table Module manipulated the data on the way out. After modification in the GUI, the data set goes back to the Table Module for validation before it's saved to the database. One of the benefits of this style is that you can test the Table Module by creating a Record Set (508) in memory without going to the database.

Figure 9.5. Typical interactions for the layers around a Table Module.

The word "table" in the pattern name suggests that you have one Table Module per table in the database. While this is true to the first approximation, it isn't completely true. It's also useful to have a Table Module for commonly used views or other queries. Indeed, the structure of the Table Module doesn't really depend on the structure of tables in the database but more on the virtual tables perceived by the application, including views and queries.

When to Use It

Table Module is very much based on table-oriented data, so obviously using it makes sense when you're accessing tabular data using Record Set (508). It also puts that data structure very much in the center of the code, so you also want the way you access the data structure to be fairly straightforward.

However, Table Module doesn't give you the full power of objects in organizing complex logic. You can't have direct instance-to-instance relationships, and polymorphism doesn't work well. So, for handling complicated domain logic, a Domain Model (116) is a better choice. Essentially you have to trade off Domain Model (116)'s ability to handle complex logic against Table Module's easier integration with the underlying table-oriented data structures.

If the objects in a Domain Model (116) and the database tables are relatively similar, it may be better to use a Domain Model (116) that uses Active Record (160). Table Module works better than a combination of Domain Model (116) and Active Record (160) when other parts of the application are based on a common table-oriented data structure. That's why you don't see Table Module very much in the Java environment, although that may change as row sets become more widely used.

The most well-known situation in which I've come across this pattern is in Microsoft COM designs. In COM (and .NET) the Record Set (508) is the primary repository of data in an application. Record sets can be passed to the UI, where data-aware widgets display information. Microsoft's ADO libraries give you a good mechanism to access the relational data as record sets. In this situation Table Module allows you to fit business logic into the application in a well-organized manner, without losing the way the various elements work on the tabular data.

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

Time to revisit the revenue recognition example (page 112) I used in the other domain modeling patterns, this time with a Table Module. To recap, our mission is to recognize revenue on orders when the rules vary depending on the product type. In this example we have different rules for word processors, spreadsheets, and databases.

Table Module is based on a data schema of some kind, usually a relational data model (although in the future we may well see an XML model used in a similar way). In this case I'll use the relational schema from Figure 9.6.

Figure 9.6. Database schema for revenue recognition.

The classes that manipulate this data are in pretty much the same form; there's one Table Module class for each table. In the .NET architecture a data set object provides an in-memory representation of a database structure. It thus makes sense to create classes that operate on this data set. Each Table Module class has a data member of a data table, which is the .NET system class corresponding to a table within the data set. This ability to read a table is common to all Table Modules and so can appear in a Layer Supertype (475).

class TableModule... protected DataTable table; protected TableModule(DataSet ds, String tableName) { table = ds.Tables[tableName]; } The subclass constructor calls the superclass constructor with the correct table name.

class Contract... public Contract (DataSet ds) : base (ds, "Contracts") {} This allows you to create a new Table Module just by passing in a data set to Table Module's constructor

contract = new Contract(dataset); which keeps the code that creates the data set away from the Table Modules, following the guidelines of ADO.NET.

A useful feature is the C# indexer, which gets to a particular row in the data table given the primary key.

class Contract... public DataRow this [long key] { get { String filter = String.Format("ID = {0}", key); return table.Select(filter)[0]; } } The first piece of functionality calculates the revenue recognition for a contract, updating the revenue recognition tables accordingly. The amount recognized depends on the kind of product we have. Since this behavior mainly uses data from the contract table, I decided to add the method to the contract class.

class Contract... public void CalculateRecognitions (long contractID) { DataRow contractRow = this[contractID]; Decimal amount = (Decimal)contractRow["amount"]; RevenueRecognition rr = new RevenueRecognition (table.DataSet); Product prod = new Product(table.DataSet); long prodID = GetProductId(contractID); if (prod.GetProductType(prodID) == ProductType.WP) { rr.Insert(contractID, amount, (DateTime) GetWhenSigned(contractID)); }else if (prod.GetProductType(prodID) == ProductType.SS) { Decimal[] allocation = allocate(amount,3); rr.Insert(contractID, allocation[0], (DateTime) GetWhenSigned(contractID)); rr.Insert(contractID, allocation[1], (DateTime) GetWhenSigned(contractID). AddDays(60)); rr.Insert(contractID, allocation[2], (DateTime) GetWhenSigned(contractID). AddDays(90)); }else if (prod.GetProductType(prodID) == ProductType.DB) { Decimal[] allocation = allocate(amount,3); rr.Insert(contractID, allocation[0], (DateTime) GetWhenSigned(contractID)); rr.Insert(contractID, allocation[1], (DateTime) GetWhenSigned(contractID). AddDays(30)); rr.Insert(contractID, allocation[2], (DateTime) GetWhenSigned(contractID). AddDays(60)); }else throw new Exception("invalid product id"); } private Decimal[] allocate(Decimal amount, int by) { Decimal lowResult = amount / by; lowResult = Decimal.Round(lowResult,2); Decimal highResult = lowResult + 0.01m; Decimal[] results = new Decimal[by]; int remainder = (int) amount % by; for (int i = 0; i < remainder; i++) results[i] = highResult; for (int i = remainder; i < by; i++) results[i] = lowResult; return results; } Usually I would use Money (488) here, but for variety's sake I'll show this using a decimal. I use an allocation method similar to the one I use for Money (488).

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

public enum ProductType {WP, SS, DB}; class Product... public ProductType GetProductType (long id) { String typeCode = (String) this[id]["type"]; return (ProductType) Enum.Parse(typeof(ProductType), typeCode); } GetProductType encapsulates the data in the data table. There's an argument for doing this for all columns of data, as opposed to accessing them directly as I did with the amount on the contract. While encapsulation is generally a Good Thing, I don't use it here because it doesn't fit with the assumption of the environment that different parts of the system access the data set directly. There's no encapsulation when the data set moves over to the UI, so column access functions only make sense when there's some additional functionality to be done, such as converting a string to a product type.

This is also a good time to mention that, although I'm using an untyped data set here because these are more common on different platforms, there's a strong argument (page 509) for using .NET's strongly typed data set.

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

class RevenueRecognition... public long Insert (long contractID, Decimal amount, DateTime date) { DataRow newRow = table.NewRow(); long id = GetNextID(); newRow["ID"] = id; newRow["contractID"] = contractID; newRow["amount"] = amount; newRow["date"]= String.Format("{0:s}", date); table.Rows.Add(newRow); return id; } Again, the point of this method is less to encapsulate the data row and more to have a method instead of several lines of code that are repeated.

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

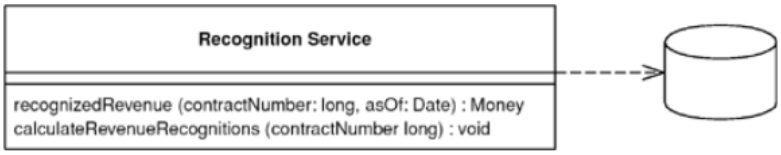
class RevenueRecognition... public Decimal RecognizedRevenue (long contractID, DateTime asOf) { String filter = String.Format("ContractID = {0}AND date <= #{1:d}#", contractID, asOf); DataRow[] rows = table.Select(filter); Decimal result = 0m; foreach (DataRow row in rows) { result += (Decimal)row["amount"]; } return result; } This fragment takes advantage of the really nice feature of ADO.NET that allows you to define a where clause and then select a subset of the data table to manipulate. Indeed, you can go further and use an aggregate function.

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

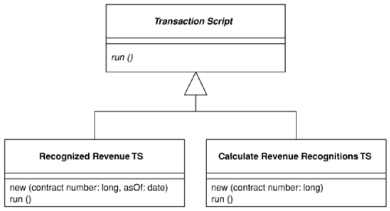
[Team LiB]

Transaction Script

- System transaction vs. Business transaction



Using commands for Transaction Script.



[Team LiB]

Transaction Script

Organizes business logic by procedures where each procedure handles a single request from the presentation.

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

A Transaction Script organizes all this logic primarily as a single procedure, making calls directly to the database or through a thin database wrapper. Each transaction will have its own Transaction Script, although common subtasks can be broken into subprocedures.

How It Works

With Transaction Script the domain logic is primarily organized by the transactions that you carry out with the system. If your need is to book a hotel room, the logic to check room availability, calculate rates, and update the database is found inside the Book Hotel Room procedure.

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

Where you put the Transaction Script will depend on how you organize your layers. It may be in a server page, a CGI script, or a distributed session object. My preference is to separate Transaction Scripts as much as you can. At the very least put them in distinct subroutines; better still, put them in classes separate from those that handle presentation and data source. In addition, don't have any calls from the Transaction Scripts to any presentation logic; that will make it easier to modify the code and test the Transaction Scripts.

You can organize your Transaction Scripts into classes in two ways. The most common is to have several Transaction Scripts in a single class, where each class defines a subject area of related Transaction Scripts. This is straightforward and the best bet for most cases. The other way is to have each Transaction Script in its own class (Figure 9.1), using the Command pattern [Gang of Four]. In this case you define a supertype for your commands that specifies some execute method in which Transaction Script logic fits. The advantage of this is that it allows you to manipulate instances of scripts as objects at runtime, although I've rarely seen a need to do this with the kinds of systems that use Transaction Scripts to organize domain logic. Of course, you can ignore classes completely in many languages and just use global functions. However, you'll often find that instantiating a new object helps with threading issues as it makes it easier to isolate data.

Figure 9.1. Using commands for Transaction Script.

I use the term Transaction Script because most of the time you'll have one Transaction Script for each database transaction. This isn't a 100 percent rule, but it's true to the first approximation.

When to Use It

The glory of Transaction Script is its simplicity. Organizing logic this way is natural for applications with only a small amount of logic, and it involves very little overhead either in performance or in understanding.

As the business logic gets more complicated, however, it gets progressively harder to keep it in a well-designed state. One particular problem to watch for is its duplication between transactions. Since the whole point is to handle one transaction, any common code tends to be duplicated.

Careful factoring can alleviate many of these problems, but more complex business domains need to build a Domain Model (116). A Domain Model (116) will give you many more options in structuring the code, increasing readability and decreasing duplication.

It's hard to quantify the cutover level, especially when you're more familiar with one pattern than the other. You can refactor a Transaction Script design to a Domain Model (116) design, but it's a harder change than it otherwise needs to be. Therefore, an early shot is often the best way to move forward.

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

The Revenue Recognition Problem

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

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

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

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

Figure 9.2. A conceptual model for simplified revenue recognition. Each contract has multiple revenue recognitions that indicate when the various parts of the revenue should be recognized.

Example: Revenue Recognition (Java)

This example uses two transaction scripts: one to calculate the revenue recognitions for a contract and one to tell how much revenue on a contract has been recognized by a certain date. The database structure has three tables: one for the products, one for the contracts, and one for the revenue recognitions.

CREATE TABLE products (ID int primary key, name varchar, type varchar) CREATE TABLE contracts (ID int primary key, product int, revenue decimal, dateSigned date) CREATE TABLE revenueRecognitions (contract int, amount decimal, recognizedOn date, PRIMARY KEY (contract, recognizedOn)) The first script calculates the amount of recognition due by a particular day. I can do this in two stages: In the first I select the appropriate rows in the revenue recognitions table; in the second I sum up the amounts.

Many Transaction Script designs have scripts that operate directly on the database, putting SQL code in the procedure. Here I'm using a simple Table Data Gateway (144) to wrap the SQL queries. Since this example is so simple, I'm using a single gateway rather than one for each table. I can define an appropriate find method on the gateway.

```
class Gateway... public ResultSet findRecognitionsFor(long contractID, MfDate asof) throws SQLException{ PreparedStatement stmt = db.prepareStatement(findRecognitionsStatement); stmt = db.prepareStatement(findRecognitionsStatement); stmt.setLong(1, contractID); stmt.setDate(2, asof.toSqlDate()); ResultSet result = stmt.executeQuery(); return result; } private static final String findRecognitionsStatement = "SELECT amount " + "FROM revenueRecognitions " + "WHERE contract = ? AND recognizedOn <= ?"; private Connection db; I then use the script to sum up based on the result set passed back from the gateway.
class RecognitionService... public Money recognizedRevenue(long contractNumber, MfDate asOf) { Money result = Money.dollars(0); try { ResultSet rs = db.findRecognitionsFor(contractNumber, asOf); while (rs.next()) { result = result.add(Money.dollars(rs.getBigDecimal("amount"))); } return result; }catch (SQLException e) {throw new ApplicationException (e); } } When the calculation is as simple as this, you can replace the in-memory script with a call to a SQL statement that uses an aggregate function to sum the amounts.
```

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

```
class RecognitionService... public void calculateRevenueRecognitions(long contractNumber) { try { ResultSet contracts = db.findContract(contractNumber); contracts.next(); Money totalRevenue = Money.dollars(contracts.getBigDecimal("revenue")); MfDate recognitionDate = new MfDate(contracts.getDate("dateSigned")); String type = contracts.getString("type"); if (type.equals("S")){ Money[] allocation = totalRevenue.allocate(3); db.insertRecognition (contractNumber, allocation[0], recognitionDate); db.insertRecognition (contractNumber, allocation[1], recognitionDate.addDays(60)); db.insertRecognition (contractNumber, allocation[2], recognitionDate.addDays(90)); }else if (type.equals("W")){ db.insertRecognition(contractNumber, totalRevenue, recognitionDate); }else if (type.equals("D")) { Money[] allocation = totalRevenue.allocate(3); db.insertRecognition (contractNumber, allocation[0], recognitionDate); db.insertRecognition (contractNumber, allocation[1], recognitionDate.addDays(30)); db.insertRecognition (contractNumber, allocation[2], recognitionDate.addDays(60)); } }catch (SQLException e) {throw new ApplicationException (e); } } Notice that I'm using Money (488) to carry out the allocation. When splitting an amount three ways it's very easy to lose a penny.
```

The Table Data Gateway (144) provides support on the SQL. First there's a finder for a contract.

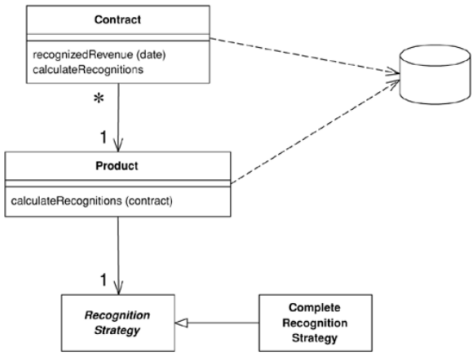
```
class Gateway... public ResultSet findContract (long contractID) throws SQLException{ PreparedStatement stmt = db.prepareStatement(findContractStatement); stmt.setLong(1, contractID); ResultSet result = stmt.executeQuery(); return result; } private static final String findContractStatement = "SELECT * " + "FROM contracts c, products p " + "WHERE ID = ? AND c.product = p.ID"; And secondly there's a wrapper for the insert.
class Gateway... public void insertRecognition (long contractID, Money amount, MfDate asof) throws SQLException { PreparedStatement stmt = db.prepareStatement(insertRecognitionStatement); stmt.setLong(1, contractID); stmt.setBigDecimal(2, amount.amount()); stmt.setDate(3, asof.toSqlDate()); stmt.executeUpdate(); } private static final String insertRecognitionStatement = "INSERT INTO revenueRecognitions VALUES (?, ?, ?)"; In a Java system the recognition service might be a regular class or a session bean.
```

As you compare this to the example in Domain Model (116), unless your mind is as twisted as mine, you'll probably be thinking that this is much simpler. The harder thing to imagine is what happens as the rules get more complicated. Typical revenue recognition rules get very involved, varying not just by product but also by date (if the contract was signed before April 15 this rule applies …). It's difficult to keep a coherent design with Transaction Script once things get that complicated, which is why object bigots like me prefer using a Domain Model (116) in these circumstances.

[Team LiB]

Domain Model

- An object model of the domain that incorporates both behavior and data.
- An OO domain model will often look similar to a database model, yet it will still have a lot of differences. A Domain Model mingles data and process, has multivalued attributes and a complex web of associations, and uses inheritance.



[Team LiB]

Domain Model

An object model of the domain that incorporates both behavior and data.

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

How It Works

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

An OO domain model will often look similar to a database model, yet it will still have a lot of differences. A Domain Model mingles data and process, has multivalued attributes and a complex web of associations, and uses inheritance.

As a result I see two styles of Domain Model in the field. A simple Domain Model looks very much like the database design with mostly one domain object for each database table. A rich Domain Model can look different from the database design, with inheritance, strategies, and other [Gang of Four] patterns, and complex webs of small interconnected objects. A rich Domain Model is better for more complex logic, but is harder to map to the database. A simple Domain Model can use Active Record (160), whereas a rich Domain Model requires Data Mapper (165).

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

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

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

If you need the same object graph between calls to the server, you have to save the server state somewhere, which is the subject of the section on saving server state (page 81).

A common concern with domain logic is bloated domain objects. As you build a screen to manipulate orders you'll notice that some of the order behavior is only needed only for it. If you put these responsibilities on the order, the risk is that the Order class will become too big because it's full of responsibilities that are only used in a single use case. This concern leads people to consider whether some responsibility is general, in which case it should sit in the order class, or specific, in which case it should sit in some usage-specific class, which might be a Transaction Script (110) or perhaps the presentation itself.

The problem with separating usage-specific behavior is that it can lead to duplication. Behavior that's separated from the order is harder to find, so people tend to not see it and duplicate it instead. Duplication can quickly lead to more complexity and inconsistency, but I've found that bloating occurs much less frequently than predicted. If it does occur, it's relatively easy to see and not difficult to fix. My advice is not to separate usage-specific behavior. Put it all in the object that's the natural fit. Fix the bloating when, and if, it becomes a problem.

Java Implementation

There's always a lot of heat generated when people talk about developing a Domain Model in J2EE. Many of the teaching materials and introductory J2EE books suggest that you use entity beans to develop a domain model, but there are some serious problems with this approach, at least with the current (2.0) specification.

Entity beans are most useful when you use Container Managed Persistence (CMP). Indeed, I would say there's little point in using entity beans without CMP. However, CMP is a limited form of object-relational mapping, and it can't support many of the patterns that you need in a rich Domain Model.

Entity beans can't be re-entrant. That is, if you call out from one entity bean into another object, that other object (or any object it calls) can't call back into the first entity bean. A rich Domain Model often uses re-entrancy, so this is a handicap. It's made worse by the fact that it's hard to spot re-entrant behavior. As a result, some people say that one entity bean should never call another. While this avoids re-entrancy, it very much cripples the advantages using a Domain Model.

A Domain Model should use fine-grained objects with fine-grained interfaces. Entity beans may be remotable (prior to version 2.0 they had to be). If you have remote objects with fine-grained interfaces you get terrible performance. You can avoid this problem quite easily by only using local interfaces for your entity beans in a Domain Model.

To run with entity beans you need a container and a database connected. This will increase build times and also increase the time to do test runs since the tests have to execute against a database. Entity beans are also tricky to debug.

The alternative is to use normal Java objects, although this often causes a surprised reaction—it's amazing how many people think that you can't run regular Java objects in an EJB container. I've come to the conclusion that people forget about regular Java objects because they haven't got a fancy name. That's why, while preparing for a talk in 2000, Rebecca Parsons, Josh Mackenzie, and I gave them one: POJOs (plain old Java objects). A POJO domain model is easy to put together, is quick to build, can run and test outside an EJB container, and is independent of EJB (maybe that's why EJB vendors don't encourage you to use them).

My view on the whole is that using entity beans as a Domain Model works if you have pretty modest domain logic. If so, you can build a Domain Model that has a simple relationship with the database: where there's mostly one entity bean class per database table. If you have a richer domain logic with inheritance, strategies, and other more sophisticated patterns, you're better off with a POJO domain model and Data Mapper (165), using a commercial tool or with a homegrown layer.

The biggest frustration for me with the use of EJB is that I find a rich Domain Model complicated enough to deal with, and I want to keep as independent as possible from the details of the implementation environment. EJB forces itself into your thinking about the Domain Model, which means that I have to worry about both the domain and the EJB environment.

When to Use It

If the how for a Domain Model is difficult because it's such a big subject, the when is hard because of both the vagueness and the simplicity of the advice. It all comes down to the complexity of the behavior in your system. If you have complicated and everchanging business rules involving validation, calculations, and derivations, chances are that you'll want an object model to handle them. On the other hand, if you have simple not-null checks and a couple of sums to calculate, a Transaction Script (110) is a better bet.

One factor that comes into this is comfortable used the development team is with domain objects. Learning how to design and use a Domain Model is a significant exercise—one that has led to many articles on the "paradigm shift" of objects use. It certainly takes practice and coaching to get used to a Domain Model, but once used to it I've found that few people want to go back to a Transaction Script (110) for any but the simplest problems.

If you're using Domain Model, my first choice for database interaction is Data Mapper (165). This will help keep your Domain Model independent from the database and is the best approach to handle cases where the Domain Model and database schema diverge.

When you use Domain Model you may want to consider Service Layer (133) to give your Domain Model a more distinct API.

Further Reading

Almost any book on OO design will talk about Domain Models, since most of what people refer to as OO development is centered around their use.

If you're looking for an introductory book on OO design, my current favorite is [Larman]. For examples of Domain Model take a look at [Fowler AP]. [Hay] also gives good examples in a relational context. To build a good Domain Model you should have an understanding of conceptual thinking about objects. For this I've always liked [Martin and Odell]. For an understanding of the patterns you'll see in a rich Domain Model, or any other OO system, you must read [Gang of Four].

Eric Evans is currently writing a book [Evans] on building Domain Models. As I write this I've seen only an early manuscript, but it looks very promising.

Example: Revenue Recognition (Java)

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

But even if the example can't do justice to why you would want a Domain Model, at least it will give you a sense of what one can look like. Therefore, I'm using the same example (page 112) that I used for Transaction Script (110), a little matter of revenue recognition.

An immediate thing to notice is that every class, in this small example (Figure 9.3) contains both behavior and data. Even the humble Revenue Recognition class contains a simple method to find out if that object's value is recognizable on a certain date.

class RevenueRecognition... private Money amount; private MfDate date; public RevenueRecognition(Money amount, MfDate date) { this.amount = amount; this.date = date; } public Money getAmount() { return amount; } boolean isRecognizableBy(MfDate asOf) { return asOf.after(date) || asOf.equals(date); }

Figure 9.3. Class diagram of the example classes for a Domain Model.

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

class Contract... private List revenueRecognitions = new ArrayList(); public Money recognizedRevenue(MfDate asOf) { Money result = Money.dollars(0); Iterator it = revenueRecognitions.iterator(); while (it.hasNext()) { RevenueRecognition r = (RevenueRecognition) it.next(); if (r.isRecognizableBy(asOf)) result = result.add(r.getAmount()); } return result; } A common thing you find in domain models is how multiple classes interact to do even the simplest tasks. This is what often leads to the complaint that with OO programs you spend a lot of time hunting around from class to class trying to find them. There's a lot of merit to this complaint. The value comes as the decision on whether something is recognizable by a certain date gets more complex and as other objects need to know. Containing the behavior on the object that needs to know avoids duplication and reduces coupling between the different objects.

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

class Contract... private Product product; private Money revenue; private MfDate whenSigned; private Long id; public Contract(Product product, Money revenue, MfDate whenSigned) { this.product = product; this.revenue = revenue; this.whenSigned = whenSigned; } class Product... private String name; private RecognitionStrategy recognitionStrategy; public Product(String name, RecognitionStrategy recognitionStrategy) { this.name = name; this.recognitionStrategy = recognitionStrategy; } public static Product newWordProcessor(String name) { return new Product(name, new CompleteRecognitionStrategy()); } public static Product newSpreadsheet(String name) { return new Product(name, new ThreeWayRecognitionStrategy(60, 90)); } public static Product newDatabase(String name) { return new Product(name, new ThreeWayRecognitionStrategy(30, 60)); } class RecognitionStrategy... abstract void calculateRevenueRecognitions(Contract contract); class CompleteRecognitionStrategy... void calculateRevenueRecognitions(Contract contract) { contract.addRevenueRecognition(new RevenueRecognition(contract.getRevenue(), contract.getWhenSigned())); } class ThreeWayRecognitionStrategy... private int firstRecognitionOffset; private int secondRecognitionOffset; public ThreeWayRecognitionStrategy(int firstRecognitionOffset, int secondRecognitionOffset) { this.firstRecognitionOffset = firstRecognitionOffset; this.secondRecognitionOffset = secondRecognitionOffset; } void calculateRevenueRecognitions(Contract contract) { Money[] allocation = contract.getRevenue().allocate(3); contract.addRevenueRecognition(new RevenueRecognition (allocation[0], contract.getWhenSigned())); contract.addRevenueRecognition(new RevenueRecognition (allocation[1], contract.getWhenSigned().addDays(firstRecognitionOffset))); contract.addRevenueRecognition(new RevenueRecognition (allocation[2], contract.getWhenSigned().addDays(secondRecognitionOffset))); } The great value of the strategies is that they provide well-contained plug points to extend the application. Adding a new revenue recognition algorithm involves creating a new subclass and overriding the calculateRevenueRecognitions method. This makes it easy to extend the algorithmic behavior of the application.

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

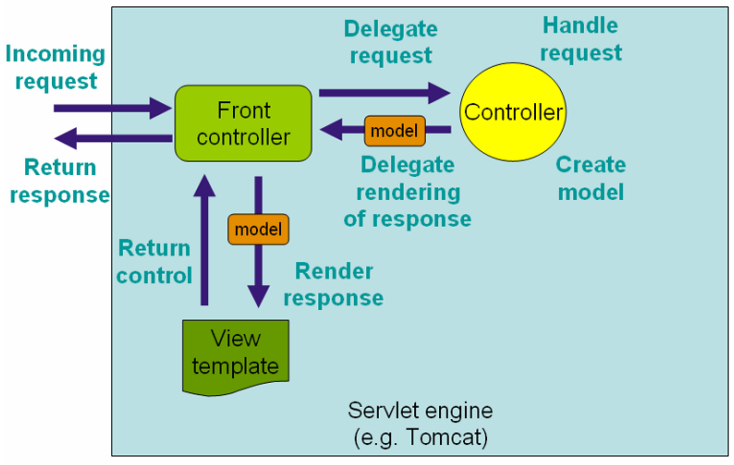
class Tester... private Product word = Product.newWordProcessor("Thinking Word"); private Product calc = Product.newSpreadsheet("Thinking Calc"); private Product db = Product.newDatabase("Thinking DB"); Once everything is set up, calculating the recognitions requires no knowledge of the strategy subclasses.

class Contract... public void calculateRecognitions() { product.calculateRevenueRecognitions(this); } class Product... void calculateRevenueRecognitions(Contract contract) { recognitionStrategy.calculateRevenueRecognitions(contract); } The OO habit of successive forwarding from object to object moves the behavior to the object most qualified to handle it, but it also resolves much of the conditional behavior. You'll notice that there are no conditionals in this calculation. You set up the decision path when you create the products with the appropriate strategy. Once everything is wired together like this, the algorithms just follow the path. Domain models work very well when you have similar conditionals because the similar conditionals can be factored out into the object structure itself. This moves complexity out of the algorithms and into the relationships between objects. The more similar the logic, the more you find the same network of relationships used by different parts of the system. Any algorithm that's dependent on the type of recognition calculation can follow this particular network of objects.

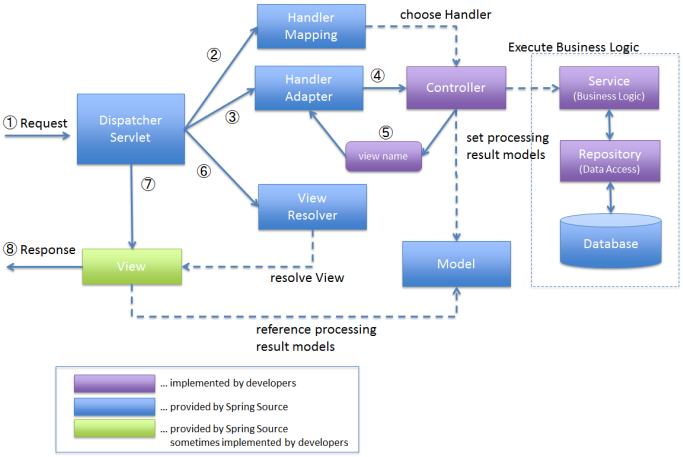
Notice in this example that I've shown nothing about how the objects are retrieved from, and written to, the database. This is for a couple of reasons. First, mapping a Domain Model to a database is always somewhat hard, so I'm chickening out and not providing an example. Second, in many ways the whole point of a Domain Model is to hide the database, both from upper layers and from people working the Domain Model itself. Thus, hiding it here reflects what it's like to actually program in this environment.

[Team LiB]

spring


Incoming request
Return response
Front controller
Delegate request
Handle request
Controller
model
Delegate rendering of response
Create model
model
Return control
Render response
View template
Servlet engine (e.g. Tomcat)

Spring – Request processing

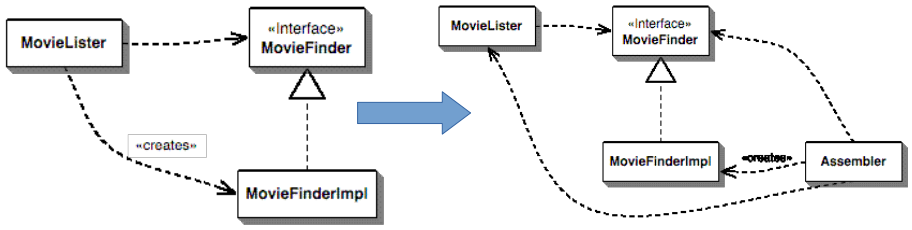


Inversion of Control – IoC

- Inversion of control (IoC)
- Dependency injection

  Dependency injection is a specific type of IoC

  Inversion of Control is a key part of what makes a framework different to a library.



Background

Inversion of control is not a new term in computer science. Martin Fowler traces the etymology of the phrase back to 1988,[5] but it is closely related to the concept of program inversion described by Michael Jackson in his Jackson Structured Programming methodology in the 1970s.[6] A bottom-up parser can be seen as an inversion of a top-down parser: in the one case, the control lies with the parser, in the other case, it lies with the receiving application.

Dependency injection is a specific type of IoC.[4] A service locator such as the Java Naming and Directory Interface (JNDI) is similar. In an article by Loek Bergman,[7] it is presented as an architectural principle.

In an article by Robert C. Martin,[8] the dependency inversion principle and abstraction by layering come together. His reason to use the term "inversion" is in comparison with traditional software development methods. He describes the uncoupling of services by the abstraction of layers when he is talking about dependency inversion. The principle is used to find out where system borders are in the design of the abstraction layers.

Description

In traditional programming, the flow of the business logic is determined by objects that are statically bound to one another. With inversion of control, the flow depends on the object graph that is built up during program execution. Such a dynamic flow is made possible by object interactions that are defined through abstractions. This run-time binding is achieved by mechanisms such as dependency injection or a service locator. In IoC, the code could also be linked statically during compilation, but finding the code to execute by reading its description from external configuration instead of with a direct reference in the code itself.

In dependency injection, a dependent object or module is coupled to the object it needs at run time. Which particular object will satisfy the dependency during program execution typically cannot be known at compile time using static analysis. While described in terms of object interaction here, the principle can apply to other programming methodologies besides object-oriented programming.

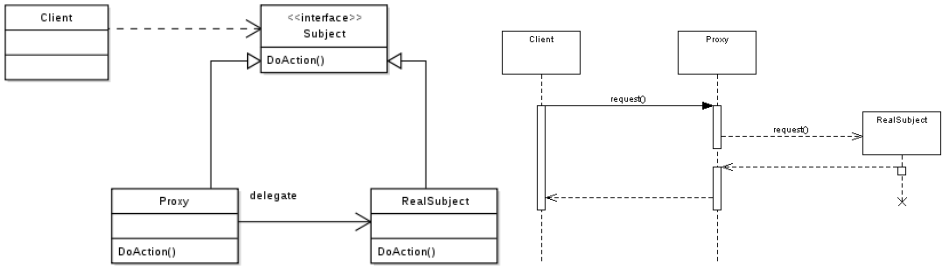
In order for the running program to bind objects to one another, the objects must possess compatible interfaces. For example, class A may delegate behavior to interface I which is implemented by class B; the program instantiates A and B, and then injects B into A.

Proxy Design Pattern

What problems can the Proxy design pattern solve?

- The access to an object should be controlled .
- Additional functionality should be provided when accessing an object.

When accessing sensitive objects, for example, it should be possible to check that clients have the needed access rights.

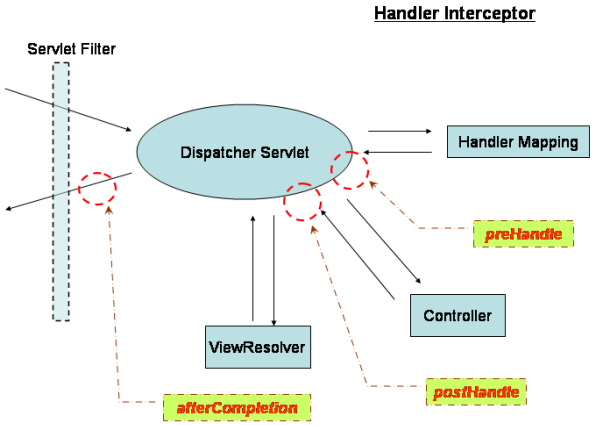


@Controller

- The entry point of your application, this is where Spring passes control to your code.

```
@Controller
public class MyControler {

@RequestMapping(value="/")
public String hello(Model m){
m.addAttribute("person", new Person("David", 10));
return "edit";
}
```

HandlerInterceptor



Validation And Layers

Spring - scope

- **Singleton** - (Default) Scopes a single bean definition to a single object instance per Spring IoC container.
- **Prototype** - Scopes a single bean definition to any number of object instances.
- **Request** - Scopes a single bean definition to the lifecycle of a single HTTP request; that is, each HTTP request has its own instance of a bean created off the back of a single bean definition. Only valid in the context of a web-aware Spring ApplicationContext.
- **Session** - Scopes a single bean definition to the lifecycle of an HTTP Session. Only valid in the context of a web-aware Spring ApplicationContext.
- **global session** - Scopes a single bean definition to the lifecycle of a global HTTP Session. Typically only valid when used in a portlet context. Only valid in the context of a web-aware Spring ApplicationContext.
- **Application** - Scopes a single bean definition to the lifecycle of a ServletContext. Only valid in the context of a web-aware Spring ApplicationContext.

Aspect Oriented Programming

- In computing, aspect-oriented programming (AOP) is a patented[1] programming paradigm that aims to increase modularity by allowing the separation of cross-cutting concerns. It does so by adding additional behavior to existing code (an advice) without modifying the code itself, instead separately specifying which code is modified via a "pointcut" specification, such as "log all function calls when the function's name begins with 'set'". This allows behaviors that are not central to the business logic (such as logging) to be added to a program without cluttering the code core to the functionality. AOP forms a basis for aspect-oriented software development.

- **Aspect, Pointcut, Join point, Advice**

In computing, **aspect-oriented programming** (**AOP**) is a patented[1] programming paradigm that aims to increase modularity by allowing the separation of cross-cutting concerns. It does so by adding additional behavior to existing code (an advice) *without* modifying the code itself, instead separately specifying which code is modified via a "pointcut" specification, such as "log all function calls when the function's name begins with 'set'". This allows behaviors that are not central to the business logic (such as logging) to be added to a program without cluttering the code core to the functionality. AOP forms a basis for aspect-oriented software development.

AOP includes programming methods and tools that support the modularization of concerns at the level of the source code, while "aspect-oriented software development" refers to a whole engineering discipline.

Aspect-oriented programming entails breaking down program logic into distinct parts (so-called *concerns*, cohesive areas of functionality). Nearly all programming paradigms support some level of grouping and encapsulation of concerns into separate, independent

Spring - AspectJ

```
import org.aspectj.lang.annotation.Aspect;
import org.aspectj.lang.annotation.Before;

@Aspect
@Component
public class LoggingAspect {

@Before("execution(*
vea2015.GreetingController.sayHello3(..))")
public void logBefore(JoinPoint joinPoint){
LogFactory.getLog(joinPoint.getTarget().getClass()).info("Befor
e " + joinPoint.getSignature());

}
}
```

Spring - AspectJ

- @Before, @After, @AfterReturning, @AfterThrowing, @Around - ProceedingJoinPoint

- execution(* com.xyz.myapp.service.*.*(..)) and @annotation(com.xyz.myapp.service.Idempotent)
- execution, within, @annotation

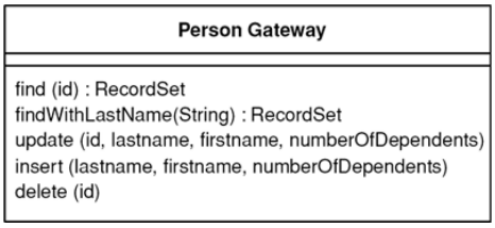
Spring - AspectJ

- execution(* set*(..))
- execution(public * *(..))
- execution(* com.xyz.service..*.*(..))
- within(com.xyz.service.*)
- within(com.xyz.service..*)
- args(java.io.Serializable)
- target(com.xyz.service.AccountService)
- @target(org.springframework.transaction.annotation.Transactional)
- @within(org.springframework.transaction.annotation.Transactional)
- @annotation(org.springframework.transaction.annotation.Transactional)
- @args(com.xyz.security.Classified)
- bean(tradeService)
- bean(*Service)

Data sources

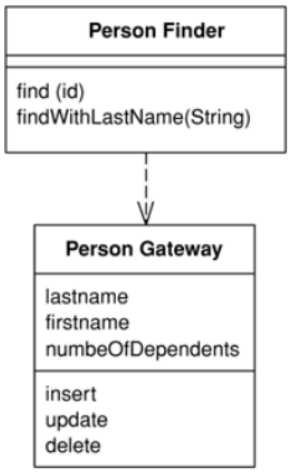
Table Data Gateway

- An object that acts as a Gateway (466) to a database table. One instance handles all the rows in the table.
- A Table Data Gateway holds all the SQL for accessing a single table or view: selects, inserts, updates, and deletes. Other code calls its methods for all interaction with the database.



Row Data Gateway

- An object that acts as a Gateway (466) to a single record in a data source. There is one instance per row.
- Embedding database access code in in-memory objects can leave you with a few disadvantages. For a start, if your in-memory objects have business logic of their own, adding the database manipulation code increases complexity.
- A Row Data Gateway gives you objects that look exactly like the record in your record structure but can be accessed with the regular mechanisms of your programming language.



[Team LiB]

Row Data Gateway

An object that acts as a Gateway (466) to a single record in a data source. There is one instance per row.

Embedding database access code in in-memory objects can leave you with a few disadvantages. For a start, if your in-memory objects have business logic of their own, adding the database manipulation code increases complexity. Testing is awkward too since, if your in-memory objects are tied to a database, tests are slower to run because of all the database access. You may have to access multiple databases with all those annoying little variations on their SQL.

A Row Data Gateway gives you objects that look exactly like the record in your record structure but can be accessed with the regular mechanisms of your programming language. All details of data source access are hidden behind this interface.

How It Works

A Row Data Gateway acts as an object that exactly mimics a single record, such as one database row. In it each column in the database becomes one field. The Row Data Gateway will usually do any type conversion from the data source types to the in-memory types, but this conversion is pretty simple. This pattern holds the data about a row so that a client can then access the Row Data Gateway directly. The gateway acts as a good interface for each row of data. This approach works particularly well for Transaction Scripts (110).

With a Row Data Gateway you're faced with the questions of where to put the find operations that generate this pattern. You can use static find methods, but they preclude polymorphism should you want to substitute different finder methods for different data sources. In this case it often makes sense to have separate finder objects so that each table in a relational database will have one finder class and one gateway class for the results (Figure 10.2).

Figure 10.2. Interactions for a find with a row-based Row Data Gateway.

It's often hard to tell the difference between a Row Data Gateway and an Active Record (160). The crux of the matter is whether there's any domain logic present; if there is, you have an Active Record (160). A Row Data Gateway should contain only database access logic and no domain logic.

As with any other form of tabular encapsulation, you can use a Row Data Gateway with a view or query as well as a table. Updates often turn out to be more complicated this way, as you have to update the underlying tables. Also, if you have two Row Data Gateways that operate on the same underlying tables, you may find that the second Row Data Gateway you update undoes the changes on the first. There's no general way to prevent this; developers just have to be aware of how virtual Row Data Gateways are formed. After all, the same thing can happen with updatable views. Of course, you can choose not to provide update operations.

Row Data Gateways tend to be somewhat tedious to write, but they're a very good candidate for code generation based on a Metadata Mapping (306). This way all your database access code can be automatically built for you during your automated build process.

When to Use It

The choice of Row Data Gateway often takes two steps: first whether to use a gateway at all and second whether to use Row Data Gateway or Table Data Gateway (144).

I use Row Data Gateway most often when I'm using a Transaction Script (110). In this case it nicely factors out the database access code and allows it to be reused easily by different Transaction Scripts (110).

I don't use a Row Data Gateway when I'm using a Domain Model (116). If the mapping is simple, Active Record (160) does the same job without an additional layer of code. If the mapping is complex, Data Mapper (165) works better, as it's better at decoupling the data structure from the domain objects because the domain objects don't need to know the layout of the database. Of course, you can use the Row Data Gateway to shield the domain objects from the database structure. That's a good thing if you're changing the database structure when using Row Data Gateway and you don't want to change the domain logic. However, doing this on a large scale leads you to three data representations: one in the business logic, one in the Row Data Gateway, and one in the database—and that's one too many. For that reason I usually have Row Data Gateways that mirror the database structure.

Interestingly, I've seen Row Data Gateway used very nicely with Data Mapper (165). Although this seems like extra work, it can be effective iff the Row Data Gateways are automatically generated from metadata while the Data Mappers (165) are done by hand.

If you use Transaction Script (110) with Row Data Gateway, you may notice that you have business logic that's repeated across multiple scripts; logic that would make sense in the Row Data Gateway. Moving that logic will gradually turn your Row Data Gateway into an Active Record (160), which is often good as it reduces duplication in the business logic.

Example: A Person Record (Java)

Here's an example for Row Data Gateway. It's a simple person table.

create table people (ID int primary key, lastname varchar, firstname varchar, number_of_dependents int) PersonGateway is a gateway for the table. It starts with data fields and accessors.

class PersonGateway... private String lastName; private String firstName; private int numberOfDependents; public String getLastName() { return lastName; } public void setLastName(String lastName) { this.lastName = lastName; } public String getFirstName() { return firstName; } public void setFirstName(String firstName) { this.firstName = firstName; } public int getNumberOfDependents() { return numberOfDependents; } public void setNumberOfDependents(int numberOfDependents) { this.numberOfDependents = numberOfDependents; } The gateway class itself can handle updates and inserts.

class PersonGateway... private static final String updateStatementString = "UPDATE people " + " set lastname = ?, firstname = ?, number_of_dependents = ? " + " where id = ?"; public void update() { PreparedStatement updateStatement = null; try { updateStatement = DB.prepare(updateStatementString); updateStatement.setString(1, lastName); updateStatement.setString(2, firstName); updateStatement.setInt(3, numberOfDependents); updateStatement.setInt(4, getID().intValue()); updateStatement.execute(); } catch (Exception e) { throw new ApplicationException(e); } finally {DB.cleanUp(updateStatement); } } private static final String insertStatementString = "INSERT INTO people VALUES (?, ?, ?, ?)"; public Long insert() { PreparedStatement insertStatement = null; try { insertStatement = DB.prepare(insertStatementString); setID(findNextDatabaseId()); insertStatement.setInt(1, getID().intValue()); insertStatement.setString(2, lastName); insertStatement.setString(3, firstName); insertStatement.setInt(4, numberOfDependents); insertStatement.execute(); Registry.addPerson(this); return getID(); } catch (SQLException e) { throw new ApplicationException(e); } finally {DB.cleanUp(insertStatement); } } To pull people out of the database, we have a separate PersonFinder. This works with the gateway to create new gateway objects.

class PersonFinder... private final static String findStatementString = "SELECT id, lastname, firstname, number_of_dependents " + " from people " + " WHERE id = ?"; public PersonGateway find(Long id) { PersonGateway result = (PersonGateway) Registry.getPerson(id); if (result != null) return result; PreparedStatement findStatement = null; ResultSet rs = null; try { findStatement = DB.prepare(findStatementString); findStatement.setLong(1, id.longValue()); rs = findStatement.executeQuery(); rs.next(); result = PersonGateway.load(rs); return result; } catch (SQLException e) { throw new ApplicationException(e); } finally {DB.cleanUp(findStatement, rs); } } public PersonGateway find(long id) { return find(new Long(id)); } class PersonGateway... public static PersonGateway load(ResultSet rs) throws SQLException { Long id = new Long(rs.getLong(1)); PersonGateway result = (PersonGateway) Registry.getPerson(id); if (result != null) return result; String lastNameArg = rs.getString(2); String firstNameArg = rs.getString(3); int numDependentsArg = rs.getInt(4); result = new PersonGateway(id, lastNameArg, firstNameArg, numDependentsArg); Registry.addPerson(result); return result; } To find more than one person according to some criteria we can provide a suitable finder method.

class PersonFinder... private static final String findResponsibleStatement = "SELECT id, lastname, firstname, number_of_dependents " + " from people " + " WHERE number_of_dependents > 0"; public List findResponsibles() { List result = new ArrayList(); PreparedStatement stmt = null; ResultSet rs = null; try { stmt = DB.prepare(findResponsibleStatement); rs = stmt.executeQuery(); while (rs.next()) { result.add(PersonGateway.load(rs)); } return result; } catch (SQLException e) { throw new ApplicationException(e); } finally {DB.cleanUp(stmt, rs); } } The finder uses a Registry (480) to hold Identity Maps (195).

We can now use the gateways from a Transaction Script (110)

PersonFinder finder = new PersonFinder(); Iterator people = finder.findResponsibles().iterator(); StringBuffer result = new StringBuffer(); while (people.hasNext()) { PersonGateway each = (PersonGateway) people.next(); result.append(each.getLastName()); result.append(" "); result.append(each.getFirstName()); result.append(" "); result.append(String.valueOf(each.getNumberOfDependents())); result.append(" } return result.toString(); **Example: A Data Holder for a Domain Object (Java)**

I use Row Data Gateway mostly with Transaction Script (110). If we want to use the Row Data Gateway from a Domain Model (116), the domain objects need to get at the data from the gateway. Instead of copying the data to the domain object we can use the Row Data Gateway as a data holder for the domain object.

class Person... private PersonGateway data; public Person(PersonGateway data) { this.data = data; } Accessors on the domain logic can then delegate to the gateway for the data.

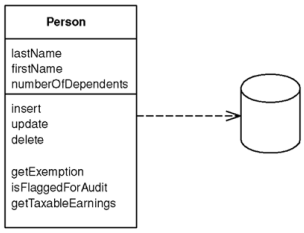
class Person... public int getNumberOfDependents() { return data.getNumberOfDependents(); } The domain logic uses the getters to pull the data from the gateway.

class Person... public Money getExemption() { Money baseExemption = Money.dollars(1500); Money dependentExemption = Money.dollars(750); return baseExemption.add(dependentExemption.multiply(this.getNumberOfDependents())); }

[Team LiB]

Active record

- An object that wraps a row in a database table or view, encapsulates the database access, and adds domain logic on that data.



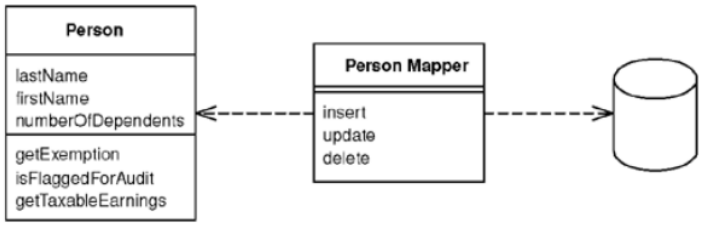
- An object carries both data and behavior. Much of this data is persistent and needs to be stored in a database. Active Record uses the most obvious approach, putting data access logic in the domain object. This way all people know how to read and write their data to and from the database.

Row Data Gateway vs. Active Record

- Active Record is very similar to Row Data Gateway (152). The principal difference is that a Row Data Gateway (152) contains only database access while an Active Record contains both data source and domain logic. Like most boundaries in software, the line between the two isn't terribly sharp, but it's useful.

Data Mapper

- A layer of Mappers that moves data between objects and a database while keeping them independent of each other and the mapper itself.



[Dawn Lift]

Data Mapper

A layer of Mappers (473) that moves data between objects and a database while keeping them independent of each other and the mapper itself.

Objects and relational databases have different mechanisms for structuring data. Many parts of an object, such as collections and inheritance, aren't present in relational databases. When you build an object model with a lot of business logic it's valuable to use these mechanisms to better organize the data and the behavior that goes with it. Doing so leads to variant schemas; that is, the object schema and the relational schema don't match up.

You still need to transfer data between the two schemas, and this data transfer becomes a complexity in its own right. If the in-memory objects know about the relational database structure, changes in one tend to ripple to the other.

The Data Mapper is a layer of software that separates the in-memory objects from the database. Its responsibility is to transfer data between the two and also to isolate them from each other. With Data Mapper the in-memory objects needn't know even that there's a database present; they need no SQL interface code, and certainly no knowledge of the database schema. (The database schema is always ignorant of the objects that use it.) Since it's a form of Mapper (473), Data Mapper itself is even unknown to the domain layer.

How it Works

The separation between domain and data source is the main function of a Data Mapper, but there are plenty of details that have to be addressed to make this happen. There's also a lot of variety in how mapping layers are built. Many of the comments here are pretty broad, because I try to give a general overview of what you need to separate the cat from its skin.

We'll start with a very basic Data Mapper example. This is the simplest style of this layer that you can have and might not seem worth doing. With simple database mapping examples other patterns usually are simpler and thus better. If you are going to use Data Mapper at all you usually need more complicated cases. However, it's easier to explain the ideas if we start simple at a very basic level.

A simple case would have a Person and Person Mapper class. To load a person from the database, a client would call a find method on the mapper (Figure 10.3) The mapper uses an Identity Map (195) to see if the person is already loaded; if not, it loads it.

Figure 10.3. Retrieving data from a database.

Updates are shown in Figure 10.4. A client asks the mapper to save a domain object. The mapper pulls the data out of the domain object and shuttles it to the database.

Figure 10.4. Updating data.

The whole layer of Data Mapper can be substituted, either for testing purposes or to allow a single domain layer to work with different databases.

A simple Data Mapper would just map a database table to an equivalent in-memory class on a field-to-field basis. Of course, things aren't usually simple. Mappers need a variety of strategies to handle classes that turn into multiple fields, classes that have multiple tables, classes with inheritance, and the joys of connecting together objects once they've been sorted out. The various object-relational mapping patterns in this book are all about that. It's usually easier to deploy these patterns with a Data Mapper than it is with the other organizing alternatives.

When it comes to inserts and updates, the database mapping layer needs to understand what objects have changed, which new ones have been created, and which ones have been destroyed. It also has to fit the whole workload into a transactional framework. The Unit of Work (184) pattern is a good way to organize this.

Figure 10.3 suggests that a single request to a find method results in a single SQL query. This isn't always true. Loading a typical order with multiple order lines may involve loading the order lines as well. The request from the client will usually lead to a graph of objects being loaded, with the mapper designer deciding exactly how much to pull back in one go. The point of this is to minimize database queries, so the finders typically need to know a fair bit about how clients use the objects in order to make the best choices for pulling data back.

This example leads to cases where you load multiple classes of domain objects from a single query. If you want to load orders and order lines, it will usually be faster to do a single query that joins the orders and order line tables. You then use the result set to load both the order and the order line instances (page 243).

Since objects are very interconnected, you usually have to stop pulling the data back at some point. Otherwise, you're likely to pull back the entire database with a request. Again, mapping layers have techniques to deal with this while minimizing the impact on the in-memory objects, using Lazy Load (200). Hence, the in-memory objects can't be entirely ignorant of the mapping layer. They may need to know about the finders and a few other mechanisms.

An application can have one Data Mapper or several. If you're hardcoding your mappers, it's best to use one for each domain class or root of a domain hierarchy. If you're using Metadata Mapping (306), you can get away with a single mapper class. In the latter case the limiting problem is your find methods. With a large application it can be too much to have a single mapper with lots of find methods, so it makes sense to split these methods up by each domain class or head of the domain hierarchy. You get a lot of small finder classes, but it's easy for a developer to locate the finder she needs.

As with any database find behavior, the finders need to use an Identity Map (195) in order to maintain the identity of the objects read from the database. Either you can have a Registry (480) of Identity Maps (195), or you can have each finder hold an Identity Map (195) (providing there is only one finder per class per session).

Handling Finders

In order to work with an object, you have to load it from the database. Usually the presentation layer will initiate things by loading some initial objects. Then control moves into the domain layer, at which point the code will mainly move from object to object using associations between them. This will work effectively providing that the domain layer has all the objects it needs loaded into memory or that you use Lazy Load (200) to load in additional objects when needed.

On occasion you may need the domain objects to invoke find methods on the Data Mapper. However, I've found that with a good Lazy Load (200) you can completely avoid this. For simpler applications, though, may not be worth trying to manage everything with associations and Lazy Load (200). Still, you don't want to add a dependency from your domain objects to your Data Mapper. You can solve this dilemma by using Separated Interface (476). Put any find methods needed by the domain code into an interface class that you can place in the domain package.

Mapping Data to Domain Fields

Mappers need access to the fields in the domain objects. Often this can be a problem because you need public methods to suppot the mappers you don't want for domain logic. (I'm assuming that you won't commit the cardinal sin of making fields public.) There's no easy to answer to this. You could use a lower level of visibility by packaging the mappers closer to the domain objects, such as in the same package in Java, but this confuses the bigger dependency picture because you don't want other parts of the system that know the domain objects to know about the mappers. You can use reflection, which can often bypass the visibility rules of the language. It's slower, but the slower speed may end up as just a rounding error compared to the time taken by the SQL call. Or you can use public methods, but guard them with a status field so that they throw an exception if they're used outside the context of a database load. If so, name them in such a way that they're not mistaken for regular getters and setters.

Tied to this is the issue of when you create the object. In essence you have two options. One is to create the object with a rich constructor so that it's at least created with all its mandatory data. The other is to create an empty object and then populate it with the mandatory data. I usually prefer the former since it's nice to have a well-formed object from the start. This also means that, if you have an immutable field, you can enforce it by not providing any method to change its value.

The problem with a rich constructor is that you have to be aware of cyclic references. If you have two objects that reference each other, each time you try to load one it will try to load the other, which will in turn try to load the first one, and so on, until you run out of stack space. Avoiding this requires special case code, often using Lazy Load (200). Writing this special case code is messy, so it's worth trying to do without it. You can do this by creating an empty object. Use a no-arg constructor to create a blank object and insert that empty object immediately into the Identity Map (195). That way, if you have a cycle, the Identity Map (195) will return an object to stop the recursive loading.

Using an empty object like this means you may need some setters for values that are truly immutable when the object is loaded. A combination of a naming convention and perhaps some status-checking guards can fix this. You can also use reflection for data loading.

Metadata-Based Mappings

One of the decisions you need to make concerns storing the information about how fields in domain objects are mapped to columns in the database. The simplest, and often best, way to do this is with explicit code, which requires a mapper class for each domain object. The mapper does the mapping through assignments and has fields (usually constant strings) to store the SQL for database access. An alternative is to use Metadata Mapping (306), which stores the metadata as data, either in a class or in a separate file. The great advantage of metadata is that all the variation in the mappers can be handled through data without the need for more source code, either by use of code generation or reflective programming.

When to Use It

The primary occasion for using Data Mapper is when you want the database schema and the object model to evolve independently. The most common case for this is with a Domain Model (116). Data Mapper's primary benefit is that when working on the domain model you can ignore the database, both in design and in the build and testing process. The domain objects have no idea what the database structure is, because all the correspondence is done by the mappers.

This helps you in the code because you can understand and work with the domain objects without having to understand how they're stored in the database. You can modify the Domain Model (116) or the database without having to alter either. With complicated mappings, particularly those involving existing databases, this is very valuable.

The price, of course, is the extra layer that you don't get with Active Record (160), so the test for using these patterns is the complexity of the business logic. If you have fairly simple business logic, you probably won't need a Domain Model (116) or a Data Mapper. More complicated logic leads you to Domain Model (116) and therefore to Data Mapper.

I wouldn't choose Data Mapper without Domain Model (116), but can I use Domain Model (116) without Data Mapper? If the domain model is pretty simple, and the database is under the domain model developers' control, then it's reasonable for the domain objects to access the database directly with Active Record (160). Effectively this puts the mapper behavior discussed here into the domain objects themselves. As things become more complicated, it's better to refactor the database behavior out into a separate layer.

Remember that you don't have to build a full-featured database-mapping layer. It's a complicated beast to build, and there are products available that do this for you. For most cases I recommend buying a database-mapping layer rather than building one yourself.

Example: A Simple Database Mapper (Java)

Here's an absurdly simple use of Data Mapper to give you a feel for the basic structure. Our example is a person with an isomorphic people table.

class Person... private String lastName; private String firstName; private int numberOfDependents; The database schema looks like this:

create table people (ID int primary key, lastname varchar, firstname varchar, number_of_dependents int) We'll use the simple case here, where the Person Mapper class also implements the finder and Identity Map (195). However, I've added an abstract mapper Layer Supertype (475) to indicate where I can pull out some common behavior. Loading involves checking that the object isn't already in the Identity Map (195) and then pulling the data from the database.

The find behavior starts in the Person Mapper, which wraps calls to an abstract find method to find by ID.

class PersonMapper... protected String findStatement() { return "SELECT " + COLUMNS + " FROM people" + " WHERE id = ?"; } public static final String COLUMNS = " id, lastname, firstname, number_of_dependents "; public Person find(Long id) { return (Person) abstractFind(id); } public Person find(long id) { return find(new Long(id)); } class AbstractMapper... protected Map loadedMap = new HashMap(); abstract protected String findStatement(); protected DomainObject abstractFind(Long id) { DomainObject result = (DomainObject) loadedMap.get(id); if (result != null) return result; PreparedStatement findStatement = null; try { findStatement = DB.prepare(findStatement()); findStatement.setLong1, id.longValue()); ResultSet rs = findStatement.executeQuery(); rs.next(); result = load(rs); return result; } catch (SQLException e) { throw new ApplicationExcception(e); } finally { DB.cleanUp(findStatement); } } The find method calls the load method, which is split between the abstract and person mappers. The abstract mapper checks the ID, pulling it from the data and registering the new object in the Identity Map (195).

class AbstractMapper... protected DomainObject load(ResultSet rs) throws SQLException { Long id = new Long(rs.getLong(1)); if (loadedMap.containsKey(id)) return (DomainObject) loadedMap.get(id); DomainObject result = doLoad(id, rs); loadedMap.put(id, result); return result; } abstract protected DomainObject doLoad(Long id, ResultSet rs) throws SQLException; class PersonMapper... protected DomainObject doLoad(Long id, ResultSet rs) throws SQLException { String lastNameArg = rs.getString(2); String firstNameArg = rs.getString(3); int numDependentsArg = rs.getInt(4); return new Person(id, lastNameArg, firstNameArg, numDependentsArg); } Notice that the Identity Map (195) is checked twice, once by abstractFind and once by load. There's a reason for this madness.

I need to check the map in the finder because, if the object is already there, I can save myself a trip to the database—I always want to save myself that long hike if I can. But I also need to check in the load because I may have queries that I can't be sure of resolving in the Identity Map (195). Say I want to find everyone whose last name matches some search pattern. I can't be sure that I have all such people already loaded, so I have to go to the database and run a query.

class PersonMapper... private static String findLastNameStatement = "SELECT " + COLUMNS + " FROM people " + " WHERE UPPER(lastname) like UPPER(?)" + " ORDER BY lastname"; public List findByLastName(String name) { PreparedStatement stmt = null; ResultSet rs = null; try { stmt = DB.prepare(findLastNameStatement); stmt.setString(1, name); rs = stmt.executeQuery(); return loadAll(rs); } catch (SQLException e) { throw new ApplicationExcception(e); } finally { DB.cleanUp(stmt, rs); } } class AbstractMapper... protected List loadAll(ResultSet rs) throws SQLException { List result = new ArrayList(); while (rs.next()) result.add(load(rs)); return result; } When I do this I may pull back some rows in the result set that correspond to people I've already loaded. I have to ensure that I don't make a duplicate, so I have to check the Identity Map (195) again.

Writing a find method this way in each subclass that needs it involves some basic, but repetitive, coding, which I can eliminate by providing a general method.

class AbstractMapper... public List findMany(StatementSource source) { PreparedStatement stmt = null; ResultSet rs = null; try { stmt = DB.prepare(source.sql()); for (int i = 0; i < source.parameters().length; i++) stmt.setObject(i+1, source.parameters()[i]); rs = stmt.executeQuery(); return loadAll(rs); } catch (SQLException e) { throw new ApplicationExcception(e); } finally { DB.cleanUp(stmt, rs); } } For this to work I need an interface that wraps both the SQL string and the loading of parameters into the prepared statement.

interface StatementSource... String sql(); Object[] parameters(); I can then use this facility by providing a suitable implementation as an inner class.

class PersonMapper... public List findByLastName2(String pattern) { return findMany(new FindByLastName(pattern)); } static class FindByLastName implements StatementSource { private String lastName; public FindByLastName(String lastName) { this.lastName = lastName; } public String sql() { return "SELECT " + COLUMNS + " FROM people " + " WHERE UPPER(lastname) like UPPER(?)" + " ORDER BY lastname"; } public Object[] parameters() { Object[] result = {lastName}; return result; } } This kind of work can be done in other places where there's repetitive statement invocation code. On the whole I've made the examples here more straight to make them easier to follow. If you find yourself writing a lot of repetitive straight-ahead code you should consider doing something similar.

With the update the JDBC code is specific to the subtype.

class PersonMapper... private static final String updateStatementString = "UPDATE people " + " SET lastname = ?, firstname = ?, number_of_dependents = ? " + " WHERE id = ?"; public void update(Person subject) { PreparedStatement updateStatement = null; try { updateStatement = DB.prepare(updateStatementString); updateStatement.setString(1, subject.getLastName()); updateStatement.setString(2, subject.getFirstName()); updateStatement.setInt(3, subject.getNumberOfDependents()); updateStatement.setInt(4, subject.getID().intValue()); updateStatement.execute(); } catch (Exception e) { throw new ApplicationExcception(e); } finally { DB.cleanUp(updateStatement); } } For the insert some code can be factored into the Layer Supertype (475)

class AbstractMapper... public Long insert(DomainObject subject) { PreparedStatement insertStatement = null; try { insertStatement = DB.prepare(insertStatement()); subject.setID(findNextDatabaseId()); insertStatement.setInt(1, subject.getID().intValue()); doInsert(subject, insertStatement); insertStatement.execute(); loadedMap.put(subject.getID(), subject); return subject.getID(); } catch (SQLException e) { throw new ApplicationExcception(e); } finally { DB.cleanUp(insertStatement); } } abstract protected String insertStatement(); abstract protected void doInsert(DomainObject subject, PreparedStatement insertStatement) throws SQLException; class PersonMapper... protected String insertStatement() { return "INSERT INTO people VALUES (?, ?, ?, ?)"; } protected void doInsert(DomainObject abstractSubject, PreparedStatement stmt) throws SQLException { Person subject = (Person) abstractSubject; stmt.setString(2, subject.getLastName()); stmt.setString(3, subject.getFirstName()); stmt.setInt(4, subject.getNumberOfDependents()); } **Example: Separating the Finders (Java)**

To allow other objects to invoke finder behavior I can use Separated Interface (476) to separate the finder interfaces from the mappers (Figure 10.5). I can put these finder interfaces in a separate package that's visible to the domain layer, or, as in this case, I can put them in the domain layer itself.

Figure 10.5. Defining a finder interface in the domain package.

One of the most common finds is one that finds an object according to a particular surrogate ID. Much of this processing is quite generic, so it can be handled by a suitable Layer Supertype (475). All it needs is a Layer Supertype (475) for domain objects that know about IDs.

The interface for finding lies in the finder interface. It's usually best not made generic because you need to know what the return type is.

interface ArtistFinder... Artist find(Long id); Artist find(long id); The finder interface is best declared in the domain package with the finders held in a Registry (480). In this case I've made the mapper class implement the finder interface.

class ArtistMapper implements ArtistFinder... public Artist find(Long id) { return (Artist) abstractFind(id); } public Artist find(long id) { return find(new Long(id)); } The bulk of the find method is done by the mapper's Layer Supertype (475), which checks the Identity Map (195) to see if the object is already in memory. If not, it completes a prepared statement that's loaded in by the artist mapper and executes it.

class AbstractMapper... abstract protected String findStatement(); protected Map loadedMap = new HashMap(); protected DomainObject abstractFind(Long id) { DomainObject result = (DomainObject) loadedMap.get(id); if (result != null) return result; PreparedStatement stmt = null; ResultSet rs = null; try { stmt = DB.prepare(findStatement()); stmt.setLong(1, id.longValue()); rs = stmt.executeQuery(); rs.next(); result = load(rs); return result; } catch (SQLException e) { throw new ApplicationExcception(e); } finally {cleanUp(stmt, rs); } } class ArtistMapper... protected String findStatement() { return "select * + COLUMN_LIST + " from artists art where ID = ?"; } public static String COLUMN_LIST = "art.ID, art.name"; The find part of the behavior is about getting either the existing object or a new one. The load part is about putting the data from the database into a new object.

class AbstractMapper... protected DomainObject load(ResultSet rs) throws SQLException { Long id = new Long(rs.getLong("id")); if (loadedMap.containsKey(id)) return (DomainObject) loadedMap.get(id); DomainObject result = doLoad(id, rs); loadedMap.put(id, result); return result; } abstract protected DomainObject doLoad(Long id, ResultSet rs) throws SQLException; class ArtistMapper... protected DomainObject doLoad(Long id, ResultSet rs) throws SQLException { String name = rs.getString("name"); Artist result = new Artist(id, name); return result; } Notice that the load method also checks the Identity Map (195). Although redundant in this case, the load can be called by other finders that haven't already done this check. In this scheme all a subclass has to do is develop a doLoad method to load the actual data needed, and return a suitable prepared statement from the findStatement method.

You can also do a find based on a query. Say we have a database of tracks and albums and we want a finder that will find all the tracks on a specified album. Again the interface declares the finders.

interface TrackFinder... Track find(Long id); Track find(long id); List findForAlbum(Long albumID); Since this is a specific find method for this class, it's implemented in a specific class, such as the track mapper class, rather than in a Layer Supertype (475). As with any finder, there are two methods to the implementation. One sets up the prepared statement; the other wraps the call to the prepared statement and interprets the results.

class TrackMapper... public static final String findForAlbumStatement = "SELECT ID, seq, albumID, title " + "FROM tracks " + "WHERE albumID = ? ORDER BY seq"; public List findForAlbum(Long albumID) { PreparedStatement stmt = null; ResultSet rs = null; try { stmt = DB.prepare(findForAlbumStatement); stmt.setLong(1, albumID.longValue()); rs = stmt.executeQuery(); List result = new ArrayList(); while (rs.next()) result.add(load(rs)); return result; } catch (SQLException e) { throw new ApplicationExcception(e); } finally {cleanUp(stmt, rs); } } The finder calls a load method for each row in the result set. This method has the responsibility of creating the in-memory object and loading it with the data. As in the previous example, some of this can be handled in a Layer Supertype (475), including checking the Identity Map (195) to see if something is already loaded.

Example: Creating an Empty Object (Java)

There are two basic approaches for loading an object. One is to create a fully valid object with a constructor, which is what I've done in the examples above. This results in the following loading code:

class AbstractMapper... protected DomainObject load(ResultSet rs) throws SQLException { Long id = new Long(rs.getLong(1)); if (loadedMap.containsKey(id)) return (DomainObject) loadedMap.get(id); DomainObject result = doLoad(id, rs); loadedMap.put(id, result); return result; } abstract protected DomainObject doLoad(Long id, ResultSet rs) throws SQLException; class PersonMapper... protected DomainObject doLoad(Long id, ResultSet rs) throws SQLException { String lastNameArg = rs.getString(2); String firstNameArg = rs.getString(3); int numDependentsArg = rs.getInt(4); return new Person(id, lastNameArg, firstNameArg, numDependentsArg); } The alternative is to create an empty object and load it with the setters later.

class AbstractMapper... protected DomainObjectEL load(ResultSet rs) throws SQLException { Long id = new Long(rs.getLong(1)); if (loadedMap.containsKey(id)) return (DomainObjectEL) loadedMap.get(id); DomainObjectEL result = createDomainObject(); result.setID(id); loadedMap.put(id, result); doLoad (result, rs); return result; } abstract protected DomainObjectEL createDomainObject(); abstract protected void doLoad(DomainObjectEL obj, ResultSet rs) throws SQLException; class PersonMapper... protected DomainObjectEL createDomainObject() { return new Person(); } protected void doLoad(DomainObjectEL obj, ResultSet rs) throws SQLException { Person person = (Person) obj; person.dbLoadLastName(rs.getString(2)); person.setFirstName(rs.getString(3)); person.setNumberOfDependents(rs.getInt(4)); } Notice that I'm using a different kind of domain object Layer Supertype (475) here, because I want to control the use of the setters. Let's say that I want the last name of a person to be an immutable field. In this case I don't want to change the value of the field once it's loaded, so I add a status field to the domain object.

class DomainObjectEL... private int state = LOADING; private static final int ACTIVE = 1; public void beActive() { state = ACTIVE; } } can then check the value of this during a load.

class Person... public void dbLoadLastName(String lastName) { assertStateIsLoading(); this.lastName = lastName; } class DomainObjectEL... void assertStateIsLoading() { Assert.isTrue(state == LOADING); } What I don't like about this is that we now have a method in the interface that most clients of the Person class can't use. This is an argument for the mapper using reflection to set the field, which will completely bypass Java's protection mechanisms.

Is the status-based guard worth the trouble? I'm not entirely sure. On the one hand it will catch bugs caused by people calling update methods at the wrong time. On the other hand is the seriousness of the bugs worth the cost of the mechanism? At the moment I don't have a strong opinion either way.

[Dawn Lift]

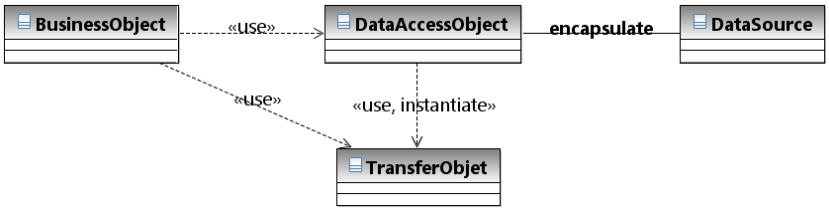
Data Layer

- Datasource, JDBC,
- JDBC Support
- Datasource JPA
- JPA Mapping, QL
- Transaction

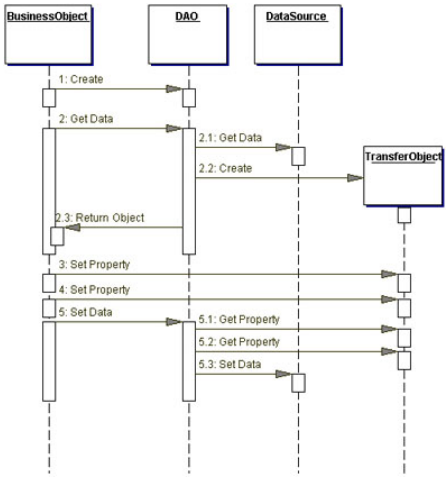
DAO – Data Access Object

Solution

- Use a DAO object to wrap all access to the repository. The DAO is responsible for connecting to the repository and storing or retrieving data.
- The DAO provides a simple and completely implementation-independent storage interface.



DAO – Data Access Object



BusinessObject

The BusinessObject represents the data client. It is the object that requires access to the data source to obtain and store data. A BusinessObject may be implemented as a session bean, entity bean, or some other Java object, in addition to a servlet or helper bean that accesses the data source.

DataAccessObject

The DataAccessObject is the primary object of this pattern. The DataAccessObject abstracts the underlying data access implementation for the BusinessObject to enable transparent access to the data source. The BusinessObject also delegates data load and store operations to the DataAccessObject.

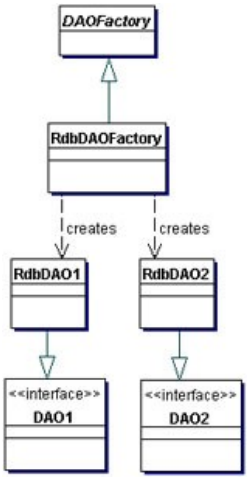
DataSource

This represents a data source implementation. A data source could be a database such as an RDBMS, OODBMS, XML repository, flat file system, and so forth. A data source can also be another system (legacy/mainframe), service (B2B service or credit card bureau), or some kind of repository (LDAP).

TransferObject

This represents a Transfer Object used as a data carrier. The DataAccessObject may use a Transfer Object to return data to the client. The DataAccessObject may also receive the data from the client in a Transfer Object to update the data in the data source.

Factory for DAO



BusinessObject

The BusinessObject represents the data client. It is the object that requires access to the data source to obtain and store data. A BusinessObject may be implemented as a session bean, entity bean, or some other Java object, in addition to a servlet or helper bean that accesses the data source.

DataAccessObject

The DataAccessObject is the primary object of this pattern. The DataAccessObject abstracts the underlying data access implementation for the BusinessObject to enable transparent access to the data source. The BusinessObject also delegates data load and store operations to the DataAccessObject.

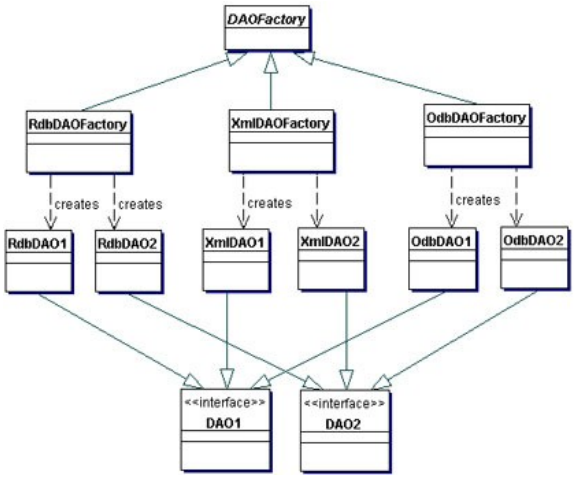
DataSource

This represents a data source implementation. A data source could be a database such as an RDBMS, OODBMS, XML repository, flat file system, and so forth. A data source can also be another system (legacy/mainframe), service (B2B service or credit card bureau), or some kind of repository (LDAP).

TransferObject

This represents a Transfer Object used as a data carrier. The DataAccessObject may use a Transfer Object to return data to the client. The DataAccessObject may also receive the data from the client in a Transfer Object to update the data in the data source.

Factory for DAO



BusinessObject

The BusinessObject represents the data client. It is the object that requires access to the data source to obtain and store data. A BusinessObject may be implemented as a session bean, entity bean, or some other Java object, in addition to a servlet or helper bean that accesses the data source.

DataAccessObject

The DataAccessObject is the primary object of this pattern. The DataAccessObject abstracts the underlying data access implementation for the BusinessObject to enable transparent access to the data source. The BusinessObject also delegates data load and store operations to the DataAccessObject.

DataSource

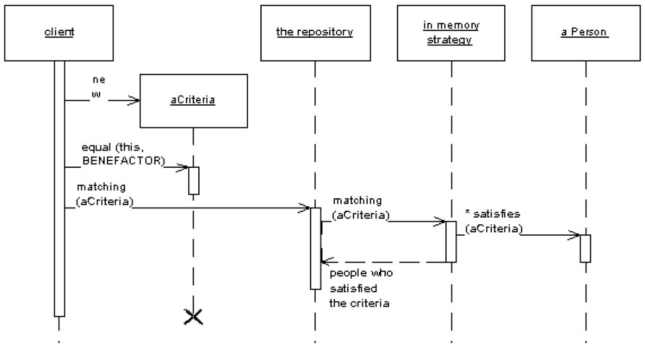
This represents a data source implementation. A data source could be a database such as an RDBMS, OODBMS, XML repository, flat file system, and so forth. A data source can also be another system (legacy/mainframe), service (B2B service or credit card bureau), or some kind of repository (LDAP).

TransferObject

This represents a Transfer Object used as a data carrier. The DataAccessObject may use a Transfer Object to return data to the client. The DataAccessObject may also receive the data from the client in a Transfer Object to update the data in the data source.

Repository pattern

- Mediates between the domain and data mapping layers using a collection-like interface for accessing domain objects.



A system with a complex domain model often benefits from a layer, such as the one provided by Data Mapper (165), that isolates domain objects from details of the database access code. In such systems it can be worthwhile to build another layer of abstraction over the mapping layer where query construction code is concentrated. This becomes more important when there are a large number of domain classes or heavy querying. In these cases particularly, adding this layer helps minimize duplicate query logic.

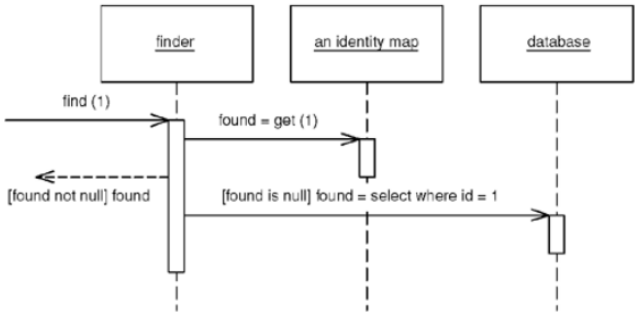
A Repository mediates between the domain and data mapping layers, acting like an in-memory domain object collection. Client objects construct query specifications declaratively and submit them to Repository for satisfaction. Objects can be added to and removed from the Repository, as they can from a simple collection of objects, and the mapping code encapsulated by the Repository will carry out the appropriate operations behind the scenes. Conceptually, a Repository encapsulates the set of objects persisted in a data store and the operations performed over them, providing a more object-oriented view of the persistence layer. Repository also supports the objective of achieving a clean separation

DAO vs. Repository

ORM - behavioral

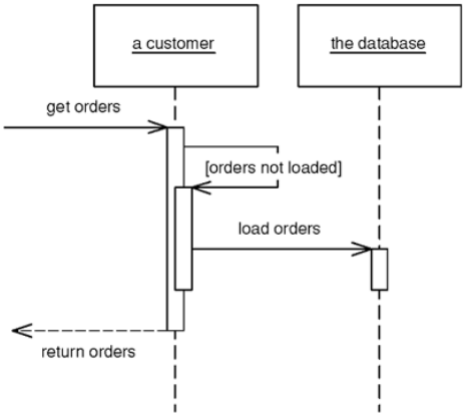
Identity Map

- 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.



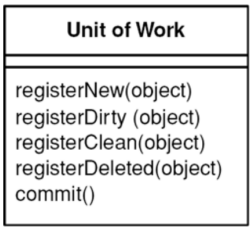
Lazy Load

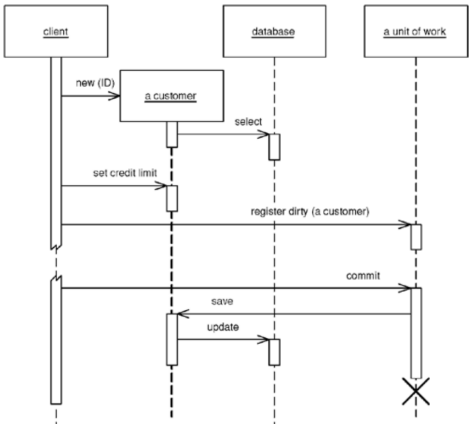
- An object that doesn't contain all of the data you need but knows how to get it.



Unit of Work

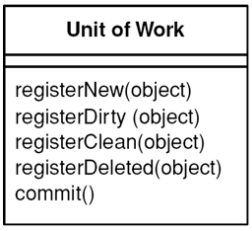
- Maintains a list of objects affected by a business transaction and coordinates the writing out of changes and the resolution of concurrency problems.





Design Patterns - Unit of Work

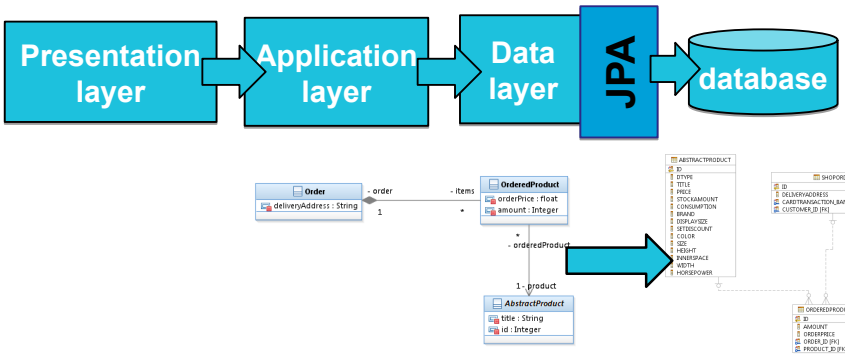
- Maintains a list of objects affected by a business transaction and coordinates the writing out of changes and the resolution of concurrency problems.
- When you're pulling data in and out of a database, it's important to keep track of what you've changed; otherwise, that data won't be written back into the database. Similarly you have to insert new objects you create and remove any objects you delete.



- You can change the database with each change to your object model, but this can lead to lots of very small database calls, which ends up being very slow. Furthermore it requires you to have a transaction open for the whole interaction, which is impractical if you have a business transaction that spans multiple requests. The situation is even worse if you need to keep track of the objects you've read so you can avoid inconsistent reads.
- A Unit of Work keeps track of everything you do during a business transaction that can affect the database. When you're done, it figures out everything that needs to be done to alter the database as a result of your work.

JPA – Java Persistent API

- API for persistence using object-relational mapping
- Only interface implementation needs to be connected



JPA - Entity

- **Entities** - are lightweight objects from the persistent domain. They typically represent a table in a database.
  - Each single object corresponds to one record in the database.
- **The persistent state** of an entity is represented by persistent class variables or persistent properties.
  - The mapping between database (tables/columns) and properties is determined by annotations

JPA – entity class

- The class must be annotated with the annotation **javax.persistence.Entity**
- The class must have a public or protected constructor without parameters (it can have other constructors)
- Neither the class nor any method or class variable may be declared as **final**
-
-

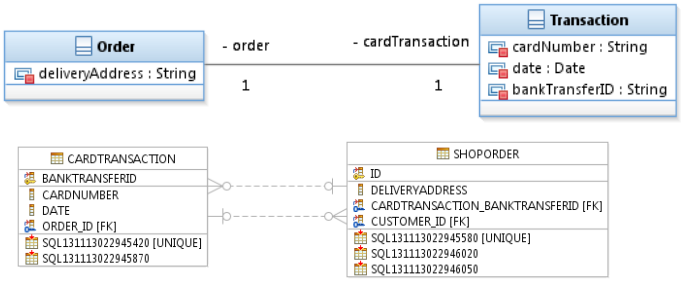
JPA – entity class

- Entity classes can be children of entity and non-entity classes. Non-entity classes can be children of entity classes.
- Persistent class variables must be defined as private, protected, or package-private. They should only be accessed using **set, get methods.**

JPA – multiplicity of references 1-1

```
@Entity
public class Order {
@OneToOne
private Transaction
cardTransaction;
…
```

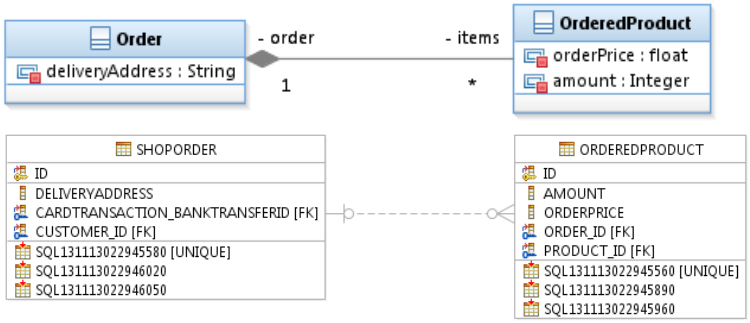
```
@Entity
public class Transaction {
@OneToOne
private Order order;
…
```



JPA – multiplicity of references 1-N

```
@Entity
public class Order {
@OneToMany(mappedBy="order")
private Set<OrderedProduct>
items;
…
```

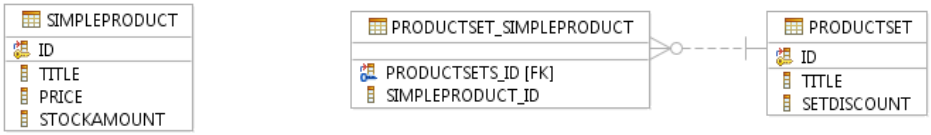
```
@Entity
public class OrderedProduct {
@ManyToOne
private Order order;
…
```



JPA – multiplicity of references M-N

```
@Entity
public class SimpleProduct
extends AbstractProduct {
@ManyToMany(mappedBy="simpleP
roduct")
private List<ProductSet>
produc
}
```

```
@Entity
public class ProductSet
extends @ManyToMany
private List<SimpleProduct>
simpleProduct;
private float setDiscount;
```



5

JPA – inheritance mapping strategies

- One table per class hierarchy
- One table per specific entity
- Join strategy

```
public enum InheritanceType {
   SINGLE_TABLE,
   JOINED,
   TABLE_PER_CLASS
};
@Inheritance(strategy=JOINED)
```

JPA – inheritance mapping strategies

One table per class hierarchy

@Inheritance(strategy=SINGLE_TABLE)

@DiscriminatorColumn(

String name

DiscriminatorType discriminatorType

String columnDefinition

String length)

public enum DiscriminatorType {

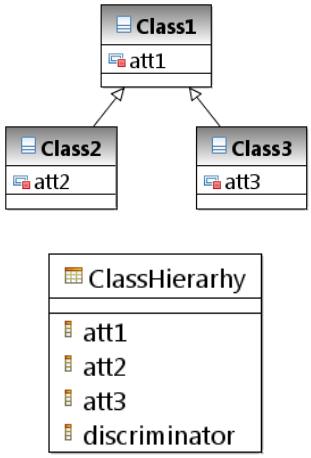
STRING,

CHAR,

INTEGER

};

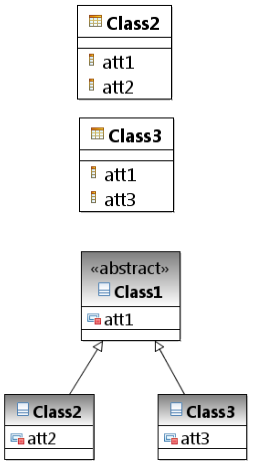
@DiscriminatorValue



JPA – inheritance mapping strategies

One table for a specific entity

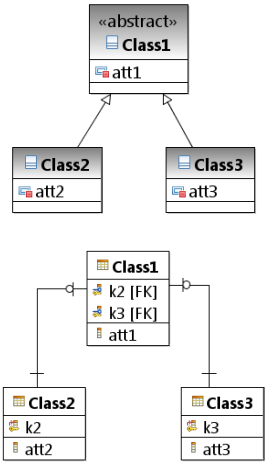
@Inheritance(strategy=TABLE_PER_CLASS)



JPA – inheritance mapping strategies

Join strategy

@Inheritance(strategy=JOINED)



JPA – MappedSuperclass

```
@MappedSuperclass
public class Person {
@Column(length=50)
private String name;
@Column(length=50)
private String surename;
@Column(length=50)
private String email;
@Column(length=50)
private String password;
}
```

```
@Entity
public class Customer extends Person
{
@Id
@GeneratedValue(strategy=GenerationT
ype.IDENTITY)
private int id;
@OneToMany(mappedBy="customer")
private Set<Order> orders;
```

```
@Entity
public class Employee extends Person {
@Id
@Column(length=50)
private String login;
private float sallary;
@Column(length=50)
private String depsrtment;
}
```


Mapped Superclasses

Entities may inherit from superclasses that contain persistent state and mapping information,
but are not entities. That is, the superclass is not decorated with the @Entity annotation, and is
not mapped as an entity by the Java Persistence provider. These superclasses are most often
used when you have state and mapping information common to multiple entity classes.
Mapped superclasses are specified by decorating the class with the javax.persistence.MappedSuperclass annotation.
Mapped superclasses are not queryable, and can’t be used in EntityManager or Query operations. You must use entity subclasses of the mapped superclass in EntityManager or Query operations.Mapped superclasses can’t be targets of entity relationships.Mapped superclasses can be abstract or concrete.
Mapped superclasses do not have any corresponding tables in the underlying datastore. Entities
that inherit from the mapped superclass define the table mappings. For instance, in the code
sample above the underlying tables would be FULLTIMEEMPLOYEE and PARTTIMEEMPLOYEE, but
there is no EMPLOYEE table.

JPA – the life cycle of an entity

- New
- Managed
- Detached
- Removed

```
@PersistenceContext
EntityManager em;
...
public LineItem createLineItem(Order
    order, Product product, int quantity) {
    LineItem li = new LineItem(order,
    product, quantity);
    order.getLineItems().add(li);
    em.persist(li);
    return li;
}

em.remove(order);
em.flush();
```

JPA – cascade operations

- Applied to attributes/relationships with other entities
  - ALL
  - PERSIST
  - MERGE
  - REMOVE
  - REFRESH
  - DETACH
- Take with extreme caution!!!!!!
- Better replaced by using JPA Entity Graph

JPA – queries

Example

- SELECT DISTINCT p FROM Player AS p, IN (p.teams) AS t WHERE t.league.sport = :sport

- Advantages
- Disadvantages

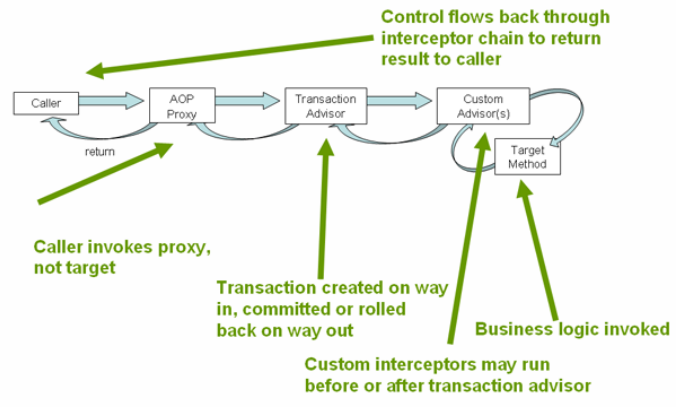
Spring - Transaction

- consistent programming model

```
public interface PlatformTransactionManager {
public TransactionStatus getTransaction(
TransactionDefinition paramTransactionDefinition)
throws TransactionException;

public void commit(TransactionStatus paramTransactionStatus)
throws TransactionException;

public void rollback(TransactionStatus paramTransactionStatus)
throws TransactionException;
}
```

Spring - Transaction

- Spring Framework provides both declarative and programmatic transaction management



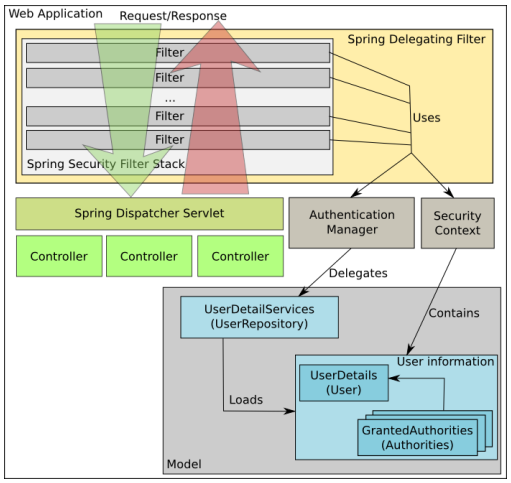
Spring - Transaction

- **Enum<Propagation>** annotation parameter for **@Transactional**
- **MANDATORY** - Support a current transaction, throw an exception if none exists.
- **NESTED** - Execute within a nested transaction if a current transaction exists.
- **NEVER** - Execute non-transactionally, throw an exception if a transaction exists.
- **NOT_SUPPORTED** - Execute non-transactionally, suspend the current transaction if one exists.
- **REQUIRED** - Support a current transaction, create a new one if none exists.
- **REQUIRES_NEW** - Create a new transaction, and suspend the current transaction if one exists.
- **SUPPORTS** - Support a current transaction, execute non-transactionally if none exists.

Discusion

- Data mapper, Lazy load, Identity map
- Transaction - where

Security


Web Application
Request/Response
Spring Delegating Filter
Filter
Filter
...
Filter
Filter
Uses
Spring Security Filter Stack
Spring Dispatcher Servlet
Authentication Manager
Security Context
Controller
Controller
Controller
Delegates
UserDetailServices (UserRepository)
Contains
User information
UserDetails (User)
Loads
GrantedAuthorities (Authorities)
Model

What are Web Services

- Interface to the application accessible via a computer network based on standard Internet technologies.
- In general: if an application is accessible over a network using protocols such as HTTP, XML, SMTP, or Jabber, it is a web service.
- The layer between the application program and the client.

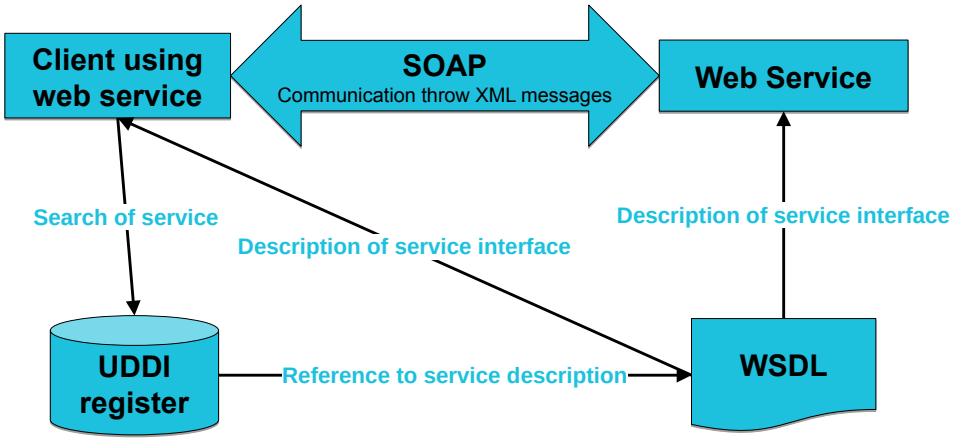
What are Web Services

- The functionality of the service does not depend on the language in which the client or server is implemented (Java, C++, PHP, C#, ...).
- Example:
- server=WWW server, client=browser
- Nowadays we don't understand web services in this general way, a web service is a set of concrete specifications (W3C).
- Available services: stock exchange, stock market, search services (Google), maps, weather.
- Components of a distributed application?

Web Services Architecture

- Group of protocols, http://www.w3.org/2002/ws/:
  - Transport of messages – SOAP,
    - http://www.w3.org/2000/xp/Group/.
  - Description of service – WSDL,
    - http://www.w3.org/2002/ws/desc/.
  - Search of service – UDDI.

Web Services Architecture



Web Services Description Language (WSDL)

- XML-based web service description.
- IBM, Microsoft, today W3C.
- WSDL file with service interface definition is an XML document, contains the definition:
  - Method,
  - Parameters.

Example WSDL

```
<wsdl:definitions
    targetNamespace=" http://tempur i.org / ">
<wsdl : types>
    <s:schema elementFormDefault="qualified "
    targetNamespace=" http://tempuri.org / ">
. . .
    <s:element name="Query">
    <s:complexType><s:sequence>
    <s:element minOccurs="1" maxOccurs="1"
    name="dbId" type="s:int" / >
    <s:element minOccurs="0" maxOccurs="1"
    name="query" type="s:string"/>
    </s:sequence></s:complexType>
    </s:element>
. . .
```

Simple Object Access Protocol (SOAP)

- Standard protocol for wrapping messages shared between applications (envelope + set of rules for representing data in XML).
- SOAP messages can be wrapped in various protocols, such as HTTP. However, we can use it for RPC (Remote Procedure Call).
- It consists of three parts:
  - envelope - defines what the message contains and how to process it.
  - A set of encoding rules - e.g. serializing primitive data types for RPC, sending messages using HTTP.
  - Conventions for representing remote procedure calls.

Simple Object Access Protocol (SOAP)

- SOAP is based on XML.
- SOAP is relatively simple
- It does not address transactions and security.
- The message contains an Envelope element that contains:
  - header - information,
  - body - meta-information.

Example SOAP 1.2, request 1/2

```
POST /AmphorAWS/AmphorAWS.asmx HTTP/1.1
Host : localhost
Content−Type: application/soap+xml;charset=utf−8
Content−Length: length

<?xml version="1.0" encoding="utf−8" ?>
<soap12:Envelope
xmlns:xsi="http://www.w3.org/2001/XMLSchema−instance"
xmlns:xsd="http://www.w3. org/2001/XMLSchema"
xmlns:soap12="http://www.w3.org/2003/05/ soap−envelope">
```

Example SOAP 1.2, request 2/2

```
<soap12:Body>
    <Query xmlns="http://tempuri.org/">
    <dbId>1</dbId>
    <query>
    doc('books.xml')/books/book[author/last='Fernadez']
    </query>
    </Query>
</soap12:Body>
</soap12:Envelope>
```

Example SOAP 1.2, response 1/2

```
HTTP/1.1 200OK
Content−Type: application/soap+xml ; charset=utf−8
Content−Length: length

<?xml version="1.0" encoding="utf−8" ?>
<soap12:Envelope
xmlns:xsi="http://www.w3.org/2001/XMLSchema−instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap12="http://www.w3.org/2003/05/soap−envelope">
```

Example SOAP 1.2, response 2/2

```
<soap12:Body>
   <QueryResponse xmlns="http://tempuri.org/">
      <QueryResult>string</QueryResult>
   </QueryResponse>
</soap12:Body>
</soap12:Envelope>
```

Universal Description, Discovery and Integration (UDDI)

- Register and search for web services.
- Offers a public database (registries). For example, the two largest databases are maintained by ~~IBM~~ and ~~Microsoft~~.
- The UDDI registry contains four types of entities:
  - business entities.
  - business services.
  - binding templates, e.g. description using WSDL.
  - service types.

Java web services

- Standard JavaEE web application
- Definition of class:

```
@WebService(name="TestWS")
public class MyWebService {
   @WebMethod
   public String sayHallo(int nTimes){
      String ret = "";
      for(int i=0; i<nTimes; i++){
         ret += "Ahoj ";
      }
      return ret;
   }
}
```

Java WS – automatically generated client

- Use in normal JavaSE application:

```
public class WebServiceClient {
   public static void main(String[] args){
      MyWebService ws = new
               MyWebServiceLocator().getMyWebServicePort();
      String response = ws.sayHallo(5);
      System.out.println("Web service response:" +
                                          response);
   }
}
```


Jboss server nesmí být spuštěn z eclipse ale z příkazové řádky, aby nechyběla definice -Djava.endorsed.dirs=/<JBOSS_HOME>/lib/endorsed

Representational State Transfer (REST)

- REST gives a coordinated set of constraints to the design of components in a distributed hypermedia system that can lead to a higher-performing and more maintainable architecture.
- To the extent that systems conform to the constraints of REST they can be called RESTful.

Representational State Transfer (REST)

- Communicate over HTTP with the same HTTP verbs (GET, POST, PUT, DELETE, etc.)
- REST interfaces with external systems using resources identified by URI
- DELETE /people/tom
- Roy Thomas Fielding in his 2000 PhD dissertation "Architectural Styles and the Design of Network-based Software Architectures"

Representational State Transfer (REST)

Architectural constraints

- **Client–server**
- **Stateless**
- **Cacheable**
- **Layered system**
- **Code on demand (optional)**
- **Uniform interface**
  - Identification of resources
  - Manipulation of resources through these representations
  - Self-descriptive messages
  - Hypermedia as the engine of application state (HATEOAS)

Architectural constraints
The architectural properties of REST are realized by applying specific interaction constraints to components, connectors, and data elements.[4][6] One can characterise applications conforming to the REST constraints described in this section as "RESTful".[2] If a service violates any of the required constraints, it cannot be considered RESTful. Complying with these constraints, and thus conforming to the REST architectural style, enables any kind of distributed hypermedia system to have desirable non-functional properties, such as performance, scalability, simplicity, modifiability, visibility, portability, and reliability.[4]
The formal REST constraints are:
Client–server
See also: Client–server model
A uniform interface separates clients from servers. This separation of concerns means that, for example, clients are not concerned with data storage, which remains internal to each server, so that the portability of client code is improved. Servers are not concerned with the user interface or user state, so that servers can be simpler and more scalable. Servers and clients may also be replaced and developed independently, as long as the interface between them is not altered.
Stateless
See also: Stateless protocol
The client–server communication is further constrained by no client context being stored on the server between requests. Each request from any client contains all the information necessary to service the request, and session state is held in the client. The session state can be transferred by the server to another service such as a database to maintain a persistent state for a period and allow authentication. The client begins sending requests when it is ready to make the transition to a new state. While one or more requests are outstanding, the client is considered to be *in transition*. The representation of each application state contains links that may be used the next time the client chooses to initiate a new state-transition.[8]
Cacheable
See also: Web cache
As on the World Wide Web, clients and intermediaries can cache responses. Responses must therefore, implicitly or explicitly, define themselves as cacheable, or not, to prevent clients from reusing stale or inappropriate data in response to further requests. Well-managed caching partially or completely eliminates some client–server interactions, further improving scalability and performance.
Layered system
See also: Layered system
A client cannot ordinarily tell whether it is connected directly to the end server, or to an intermediary along the way. Intermediary servers may improve system scalability by enabling load balancing and by providing shared caches. They may also enforce security policies.
Code on demand (optional)
See also: Client-side scripting
Servers can temporarily extend or customize the functionality of a client by the transfer of executable code. Examples of this may include compiled components such as Java applets and client-side scripts such as JavaScript. "Code on demand" is the only optional constraint of the REST architecture.
Uniform interface
The uniform interface constraint is fundamental to the design of any REST service.[4] The uniform interface simplifies and decouples the architecture, which enables each part to evolve independently. The four constraints for this uniform interface are:
Identification of resources Individual resources are identified in requests, for example using URIs in web-based REST systems. The resources themselves are conceptually separate from the representations that are returned to the client. For example, the server may send data from its database as HTML, XML or JSON, none of which are the server's internal representation. Manipulation of resources through these representations When a client holds a representation of a resource, including any metadata attached, it has enough information to modify or delete the resource. Self-descriptive messages Each message includes enough information to describe how to process the message. For example, which parser to invoke may be specified by an Internet media type (previously known as a MIME type). Responses also explicitly indicate their cacheability.[4] Hypermedia as the engine of application state (HATEOA

What Are RESTful Web Services?
RESTful web services are built to work best on the Web. Representational State Transfer (REST) is an architectural style that specifies constraints, such as the uniform interface, that if applied to a web service induce desirable properties, such as performance, scalability, and modifiability, that enable services to work best on the Web. In the REST architectural style, data and functionality are considered resources and are accessed using **Uniform Resource Identifiers (URIs)**, typically links on the Web. The resources are acted upon by using a set of simple, well-defined operations. The REST architectural style constrains an architecture to a client/server architecture and is designed to use a stateless communication protocol, typically HTTP. In the REST architecture style, clients and servers exchange representations of resources by using a standardized interface and protocol.
The following principles encourage RESTful applications to be simple, lightweight, and fast:
Resource identification through URI: A RESTful web service exposes a set of resources that identify the targets of the interaction with its clients. Resources are identified by URIs, which provide a global addressing space for resource and service discovery. See The @Path Annotation and URI Path Templates for more information.
Uniform interface: Resources are manipulated using a fixed set of four create, read, update, delete operations: PUT, GET, POST, and DELETE. PUT creates a new resource, which can be then deleted by using DELETE. GET retrieves the current state of a resource in some representation. POST transfers a new state onto a resource. See Responding to HTTP Methods and Requests for more information.
Self-descriptive messages: Resources are decoupled from their representation so that their content can be accessed in a variety of formats, such as HTML, XML, plain text, PDF, JPEG, JSON, and others. Metadata about the resource is available and used, for example, to control caching, detect transmission errors, negotiate the appropriate representation format, and perform authentication or access control. See Responding to HTTP Methods and Requests and Using Entity Providers to Map HTTP Response and Request Entity Bodies for more information.
Stateful interactions through hyperlinks: Every interaction with a resource is stateless; that is, request messages are self-contained. Stateful interactions are based on the concept of explicit state transfer. Several techniques exist to exchange state, such as URI rewriting, cookies, and hidden form fields. State can be embedded in response messages to point to valid future states of the interaction. See Using Entity Providers to Map HTTP Response and Request Entity Bodies and “Building URIs” in the JAX-RS Overview document for more information.

RESTfull – API rules

- **Hypermedia Controls** - The objective of hypermedia controls is to advise the client of what can be done next and to supply the URIs necessary to perform the next action.
- **Resource Naming** - RESTful APIs are written for clients and should have meaning for the clients of those APIs. When choosing nouns to name the resources, you should be familiar with the structure of the application’s data and how your clients are likely to use them. There are no defined rules as to how you should name your resources, but there are conventions that, if followed, can help you create a set of self-descriptive resource names that others intuitively understand.

RESTfull – API rules

- **Nouns Not Verbs** - You must name the resources after nouns, not verbs or actions. The purpose of the resource name is to represent the resource. The HTTP method describes the action to be performed.
- To represent a single user resource, you would use the noun users to represent all users and the user's ID to identify the specific user, like so:

  users/123456

- An example of a **non REST** and badly formed URI would be

  users/123456/update ,

  users/123456?action=update

RESTfull – API rules

- The nature of data is that it is **hierarchical**. So imagine that you want to represent all the posts of the user with ID 123456 . You would use the noun posts to represent all posts and create the URI

  users/123456/posts

- **different ways** - To represent all posts by a specified user, you can use the URI

  posts/users/123456

RESTfull – API rules

- **Self Descriptive** - As you have seen, the nouns chosen should reflect the resource they represent. Combining these representations with identifiers makes the URI easy to interpret and intuitive to understand. If you read a URI in combination with its HTTP method and it is not immediately obvious what resource it represents, it has failed as a RESTful URI.

RESTfull – API rules

- **Plural Not Singular** - Resource names should be plural because they represent collections of data. The resource name users represents a collection of users, and the resource name posts represents a collection of posts.
- The idea is that plural nouns represent a collection in the service, and the ID refers to one instance within that collection.
- It may be justifiable to use a singular noun if there is only one instance of that data type in the entire application, but this is quite uncommon.

RESTfull – API rules – HTTP methods

- GET - You use this method to get resource representations from the service. You should never use it to update, delete, or create a resource. Calling it once should have the same effect as calling it 100 times.
- If the resource requested is successful, the representation of the resource is returned in the body of the HTTP response in the requested data format, which commonly is either JSON or XML. The HTTP response code returned is 200 (OK) . If the resource is not found, it should return 404 (NOT FOUND) , and if the resource request is badly formed, it should return 400 (BAD REQUEST) .
- A well formed URI that you might use in your forum application could be GET users/123456/ followers , which represents all the followers of the user 123456 .

RESTfull – API rules – HTTP methods

- **POST** - You use the POST method to create a new resource within the given context. For example, to create a new user, you would post to the users resource the data necessary for a new user to be created. The service takes care of creating the new resource, associating it to the context, and assigning an ID.
- On successful creation, the HTTP response is 201 (CREATED) , and a link to the newly created resource is returned either in the Location header of the response or in the JSON payload of the response body. The resource representation may be returned in the response body. This is often preferable to avoid making an additional call to the API to retrieve a representation of the data that had been just created. This reduces the chattiness of the API.
- In addition to the HTTP response codes to a GET request, a POST can return 204 (NO CONTENT) if the body of the request is empty. A well formed URI that you might use in your forum application could be POST users , with a request body containing the new user's details or POST users/123456/ posts to create a new post for the user 123456 from the data in the request body.

RESTfull – API rules – HTTP methods

- **PUT** - The PUT method is most commonly used to update a known resource. The URI includes enough information to identify the resource, such as a context and an identifier. The request body contains the updated version of the resource.
- If the update is successful, it returns the HTTP response code 200 . A URI that updates a user's information is PUT users/123456 . Less commonly, you can use the PUT method to create a resource if the client creates the identifier of the resource. However, this way of creating a resource is a little confusing. Why use a PUT when a POST works just as well and is commonly known?
- An important point to note about updating a resource is that the entire representation of the resource is passed to the service in the HTTP body request, not just the information that has changed.

RESTfull – API rules – HTTP methods

- **DELETE** - Surprisingly, you use this method to delete a resource from a service. The URI contains the context and the identifier of the resource. To delete a user with the ID 123456, you use the URI

  DELETE users/123456

- The response body may include a representation of the deleted resource. A successful deletion results in a 200 (OK) HTTP response code being returned; if the resource is not found, a 400 code is returned.

RESTfull – API rules

- A REST API must not define fixed resource names or hierarchies (an obvious coupling of client and server). Servers must have the freedom to control their own namespace. Instead, allow servers to instruct clients on how to construct appropriate URIs, such as is done in HTML forms and URI templates, by defining those instructions within media types and link relations.
- *[Failure here implies that clients are assuming a resource structure due to out-of band information, such as a domain-specific standard, which is the data-oriented equivalent to RPC's functional coupling].*

RESTfull – API HATEOAS

```xml
<account>
   <account_number>12345</account_number>
   <balance currency="usd">100.00</balance>
   <link rel="deposit" href="https://somebank.org/account/12345/deposit" />
   <link rel="withdraw" href="https://somebank.org/account/12345/withdraw" />
   <link rel="transfer" href="https://somebank.org/account/12345/transfer" />
   <link rel="close" href="https://somebank.org/account/12345/close" />
</account>
```

```json
{
    "name": "Alice",
    "links": [ {
        "rel": "self",
        "href": "http://localhost:8080/customer/1"
    } ]
}
```

RESTfull – API CoD

Code on Demand

- REST allows client functionality to be extended by downloading and executing code in the form of applets or scripts. This simplifies clients by reducing the number of features required to be pre-implemented. Allowing features to be downloaded after deployment improves system extensibility. However, it also reduces visibility, and thus is only an optional constraint within REST.
- At the time this was written, the web was mostly just static documents and the only "web client" was the browser itself. Now it's commonplace for JavaScript-powered web apps to be consuming REST APIs. This is an example of code on demand - the browser grabs an initial HTML document and supports <script> tags inside that document so that an application can be loaded on-demand.

JSON

```
{
   "glossary": {
       "title": "example glossary",
"GlossDiv": {
          "title": "S",
"GlossList": {
             "GlossEntry": {
                "ID": "SGML",
   "SortAs": "SGML",
   "GlossTerm": "Standard Generalized Markup Language",
   "Acronym": "SGML",
   "Abbrev": "ISO 8879:1986",
   "GlossDef": {
                  "para": "A meta-markup language, used to create markup languages such as DocBook.",
       "GlossSeeAlso": ["GML", "XML"]
                },
   "GlossSee": "markup"
            }
         }
      }
   }
}
```

JSON

- Object
  {}
  { members }
- members
  pair
  pair , members
- pair
  string : value
- array
  []
  [elements]
- elements
  value
  value , elements

- value
  string
  number
  object
  array
  true
  false
  null

- string
  ""
  " chars "
- chars
  char
  char chars
- char
  any-Unicode-character-
  except-"-or-\-or-
  control-character
  \" \\ \/ \b \f \n \r \t
  \u four-hex-digits
- number
  int
  int frac
  int exp
  int frac exp
  int digit
  digit1-9 digits
  - digit
  - digit1-9 digits frac . digits exp
  e digits digits digit
  digit digits e e
  e+ e- E E+ E-

CORBA – Common Object Request Broker Architecture (1991)

IDL - Interface Definition Language

- Platform-independent language for describing interfaces and data types

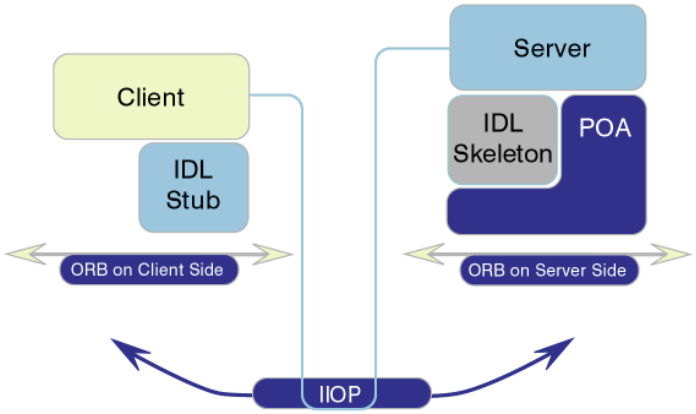
```
Module StockObjects {
  struct Quote {
   string symbol;
   long at_time;
   double price;
   long volume;
 };

 exception Unknown{};
```

```
interface Stock {
Quote get_quote() raises(Unknown);
void set_quote(in Quote stock_quote);
// Provides the stock description,
readonly attribute string description;
  };
interface Stock  Factory {
     Stock create_stock(
      in string symbol,
      in string description
    );  };};
```

CORBA

- Distributed object architecture
- Platform and language independent
- Specification only
- No reference implementation

CORBA – Architecture



CORBA 1.0 (October 1991)

Implementace objektu: Kód objektu, který implementuje zveřejněné služby objektu. Implementace může být napsána v libovolném podporovaném jazyce (obvykle C, C++ nebo Java). Rozhraní služeb objektu je definováno v jazyce IDL (*Interface Definition Language*).

Klient: Program využívající vzdálené objekty. Pro použití objektu musí mít dostupnou definici rozhraní objektu v jazyce IDL buď vv době překladu nebo při*dynamickém volání* za chodu programu a jednoznačnou adresu objektu (IOR).

IDL stubs (spojky): Kód vygenerovaný kompilátorem jazyka IDL, který propojuje uživatelský kód s agentem ORB. V jazyce C++ má spojka formu*zástupné třídy, jejíž metody může klientský kód přímo volat.*

DII (Dynamic Invocation Interface): Klient může používat také objekty, ke kterým získá definici rozhraní za běhu programu. Rozhraní pro *dynamické volání* metod dovoluje generovat dynamické požadavky.

ORB (Object Request Broker): *Zprostředkovatel objektových služeb zahrnuje veškeré vnitřní mechanismy pro vyhledání požadovaného objektu, generování a přenos požadavků, parametrů a výsledků na úrovni komunikace mezi systémy. ORB může používat různé me*tody komunikace, včetně přímé aktivace objektů v rámci jednoho adresového prostoru.

Přenosný Objektový adaptér (POA): Objektový adaptér propojuje implementaci objektu se agentem ORB, demultiplexuje přicházející požadavky, aktivuje objekty a předává jim požadavky prostřednictvím volání metod kostry objektu.

Kostra objektu: Je vygenerována kompilátorem jazyka IDL, slouží jako bázová třída odpovídající definici objektu v jazyce IDL.

DSI (Dynamic Skeleton Interface): Dynamicky vytvořená kostra objektu, obdoba DII na straně klienta. Typickým použitím je *most* pro transformaci požadavků z jednoho komunikačního protokolu do jiného nebo *firewall*.

GIOP (General Inter ORB Protocol): Protokol komunikace mezi různými ORB. Je definován nad běžným spojovaným transportním protokolem. Konkrétní implementace nad protokolem TCP/IP je definována jako IIOP (Internet Inter-ORB Protocol).

SOA – Service Oriented Architecture

The OASIS group[4] and the Open Group[5] have both created formal definitions. OASIS defines SOA as:

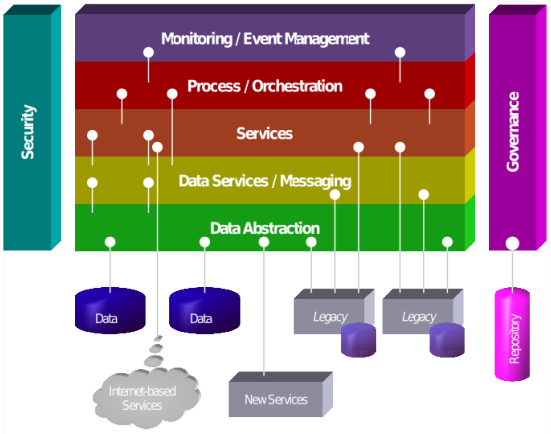
- A paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains. It provides a uniform means to offer, discover, interact with and use capabilities to produce desired effects consistent with measurable preconditions and expectations.

SOA – Service Oriented Architecture

The Open Group's definition is:

- Service-Oriented Architecture (SOA) is an architectural style that supports service-orientation. Service-orientation is a way of thinking in terms of services and service-based development and the outcomes of services.
- A service: Is a logical representation of a repeatable business activity that has a specified outcome (e.g., check customer credit, provide weather data, consolidate drilling reports) Is self-contained May be composed of other services Is a "black box" to consumers of the service

SOA – Service Oriented Architecture



SOA – Service Oriented Architecture

- **Principles**: There are no industry standards relating to the exact composition of a service-oriented architecture, although many industry sources have published their own principles. Some of these [12][13][14][15] include the following:
- **Standardized service contract**: Services adhere to a communications agreement, as defined collectively by one or more service-description documents.
- **Service loose coupling**: Services maintain a relationship that minimizes dependencies and only requires that they maintain an awareness of each other.
- **Service abstraction**: Beyond descriptions in the service contract, services hide logic from the outside world.
- **Service reusability**: Logic is divided into services with the intention of promoting reuse.

SOA – Service Oriented Architecture

- **Service autonomy**: Services have control over the logic they encapsulate, from a Design-time and a Run-time perspective.
- **Service statelessness**: Services minimize resource consumption by deferring the management of state information when necessary[16]
- **Service discoverability**: Services are supplemented with communicative meta data by which they can be effectively discovered and interpreted.
- **Service composability**: Services are effective composition participants, regardless of the size and complexity of the composition.
- **Service granularity**: A design consideration to provide optimal scope and right granular level of the business functionality in a service operation.
- **Service normalization**: Services are decomposed or consolidated to a level of normal form to minimize redundancy. In some cases, services are denormalized for specific purposes, such as performance optimization, access, and aggregation. [17]

SOA – Service Oriented Architecture

- **Service optimization**: All else being equal, high-quality services are generally preferable to low-quality ones.
- **Service relevance**: Functionality is presented at a granularity recognized by the user as a meaningful service.
- **Service encapsulation**: Many services are consolidated for use under the SOA. Often such services were not planned to be under SOA.
- **Service location transparency**: This refers to the ability of a service consumer to invoke a service regardless of its actual location in the network. This also recognizes the discoverability property (one of the core principle of SOA) and the right of a consumer to access the service. Often, the idea of service virtualization also relates to location transparency. This is where the consumer simply calls a logical service while a suitable SOA-enabling runtime infrastructure component, commonly a service bus, maps this logical service call to a physical service.

SOA – Orchestration

- In system administration, **orchestration** is the automated configuration, coordination, and management of computer systems and software.
- Orchestration in this sense is about aligning the business request with the applications, data, and infrastructure.
- Orchestration includes a workflow and provides a directed action towards larger goals and objectives.

SOA – Choreography

- **Service choreography** in business computing is a form of service composition in which the interaction protocol between several partner services is defined from a global perspective. The idea underlying the notion of service choreography can be summarised as follows:
- "Dancers dance following a global scenario without a single point of control"
- That is, at run-time each participant in a service choreography executes its part according to the behavior of the other participants. A choreography's role specifies the expected messaging behavior of the participants that will play it in terms of the sequencing and timing of the messages that they can consume and produce.
- Choreography describes the sequence and conditions in which the data is exchanged between two or more participants in order to meet some useful purpose.

Microservice Architecture

Context

- You are developing a server-side enterprise application. It must support a variety of different clients including desktop browsers, mobile browsers and native mobile applications. The application might also expose an API for 3rd parties to consume. It might also integrate with other applications via either web services or a message broker. The application handles requests (HTTP requests and messages) by executing business logic; accessing a database; exchanging messages with other systems; and returning a HTML/JSON/XML response. There are logical components corresponding to different functional areas of the application.

Problem

- What’s the application’s deployment architecture?

https://martinfowler.com/articles/microservices.html
https://microservices.io/patterns/microservices.html
https://en.wikipedia.org/wiki/Microservices

Microservice Architecture

Forces

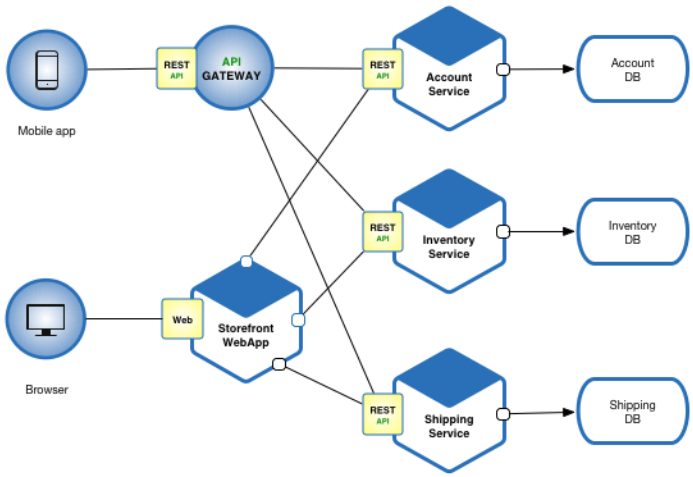
- There is a team of developers working on the application
- New team members must quickly become productive
- The application must be easy to understand and modify
- You want to practice continuous deployment of the application
- You must run multiple copies of the application on multiple machines in order to satisfy scalability and availability requirements
- You want to take advantage of emerging technologies (frameworks, programming languages, etc)

Microservice Architecture

Solution

- Define an architecture that structures the application as a set of loosely coupled, collaborating services. This approach corresponds to the Y-axis of the Scale Cube. Each service implements a set of narrowly, related functions. For example, an application might consist of services such as the order management service, the customer management service etc.
- Services communicate using either synchronous protocols such as HTTP/REST or asynchronous protocols such as AMQP. Services can be developed and deployed independently of one another. Each service has its own database in order to be decoupled from other services.

Microservice Architecture



Microservice Architecture Benefits

- Enables the continuous delivery and deployment of large, complex applications.
  - Better testability - services are smaller and faster to test
  - Better deployability - services can be deployed independently
  - It enables you to organize the development effort around multiple, auto teams. It enables you to organize the development effort around multiple teams. Each (two pizza) team is owns and is responsible for one or more single service. Each team can develop, deploy and scale their services independently of all of the other teams.
- Each microservice is relatively small
  - Easier for a developer to understand
  - The IDE is faster making developers more productive
  - The application starts faster, which makes developers more productive, and speeds up deployments

Microservice Architecture Benefits

- Improved fault isolation. For example, if there is a memory leak in one service then only that service will be affected. The other services will continue to handle requests. In comparison, one misbehaving component of a monolithic architecture can bring down the entire system.
- Eliminates any long-term commitment to a technology stack. When developing a new service you can pick a new technology stack. Similarly, when making major changes to an existing service you can rewrite it using a new technology stack

Microservice Architecture Drawbacks

- Developers must deal with the additional complexity of creating a distributed system.
  - Developer tools/IDEs are oriented on building monolithic applications and don’t provide explicit support for developing distributed applications.
  - Testing is more difficult
  - Developers must implement the inter-service communication mechanism.
  - Implementing use cases that span multiple services without using distributed transactions is difficult
  - Implementing use cases that span multiple services requires careful coordination between the teams
- Deployment complexity. In production, there is also the operational complexity of deploying and managing a system comprised of many different service types.
- Increased memory consumption. The microservice architecture replaces N monolithic application instances with NxM services instances. If each service runs in its own JVM (or equivalent), which is usually necessary to isolate the instances, then there is the overhead of M times as many JVM runtimes. Moreover, if each service runs on its own VM (e.g. EC2 instance), as is the case at Netflix, the overhead is even higher.

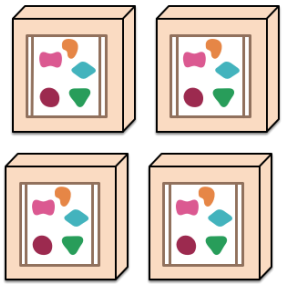
Microservice Architecture

- The microservice architectural style [1] is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies.

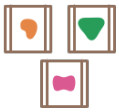
Microservice Architecture

A monolithic application puts all its functionality into a single process...

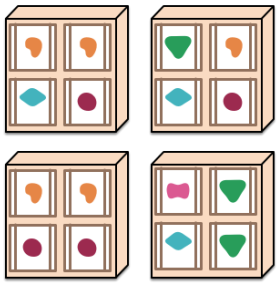
... and scales by replicating the monolith on multiple servers



A microservices architecture puts each element of functionality into a separate service...



... and scales by distributing these services across servers, replicating as needed.

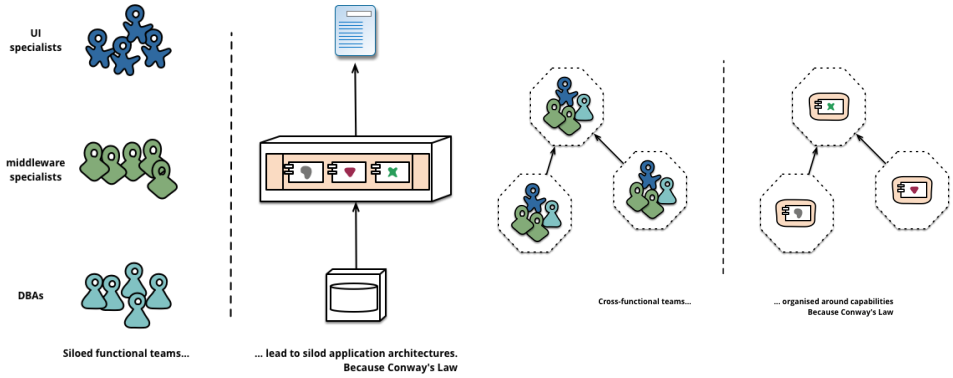


Characteristics of a Microservice Architecture
Componentization via Services

- Microservice architectures will use libraries, but their primary way of componentizing their own software is by breaking down into services. We define libraries as components that are linked into a program and called using in-memory function calls, while services are out-of-process components who communicate with a mechanism such as a web service request, or remote procedure call. (This is a different concept to that of a service object in many OO programs [3].)

Characteristics of a Microservice Architecture

Organized around Business Capabilities



Characteristics of a Microservice Architecture
Products not Projects

- Microservice proponents tend to avoid this model, preferring instead the notion that a team should own a product over its full lifetime. A common inspiration for this is Amazon's notion of "you build, you run it" where a development team takes full responsibility for the software in production. This brings developers into day-to-day contact with how their software behaves in production and increases contact with their users, as they have to take on at least some of the support burden.

Characteristics of a Microservice Architecture

Smart endpoints and dumb pipes

- When building communication structures between different processes, we've seen many products and approaches that stress putting significant smarts into the communication mechanism itself. A good example of this is the Enterprise Service Bus (ESB), where ESB products often include sophisticated facilities for message routing, choreography, transformation, and applying business rules.
- The microservice community favours an alternative approach: smart endpoints and dumb pipes. Applications built from microservices aim to be as decoupled and as cohesive as possible - they own their own domain logic and act more as filters in the classical Unix sense - receiving a request, applying logic as appropriate and producing a response. These are choreographed using simple RESTish protocols rather than complex protocols such as WS-Choreography or BPEL or orchestration by a central tool.
- Microservice teams use the principles and protocols that the world wide web (and to a large extent, Unix) is built on. Often used resources can be cached with very little effort on the part of developers or operations folk.

Characteristics of a Microservice Architecture

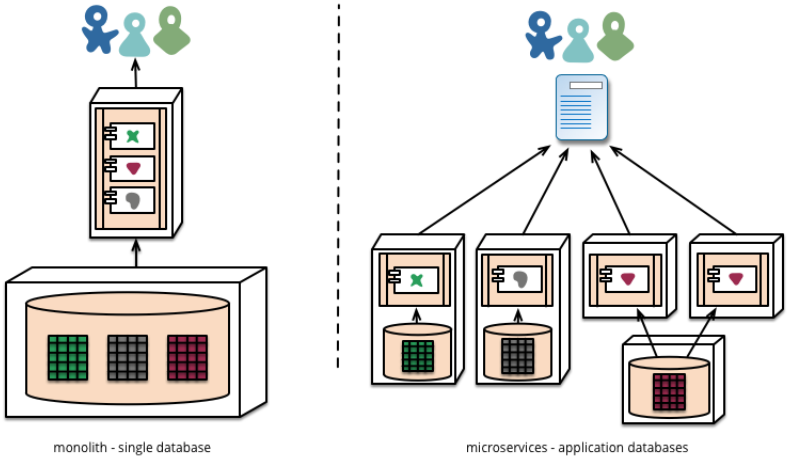
Decentralized Governance

- One of the consequences of centralised governance is the tendency to standardise on single technology platforms. Experience shows that this approach is constricting - not every problem is a nail and not every solution a hammer. We prefer using the right tool for the job and while monolithic applications can take advantage of different languages to a certain extent, it isn't that common.
- Splitting the monolith's components out into services we have a choice when building each of them. You want to use Node.js to standup a simple reports page? Go for it. C++ for a particularly gnarly near-real-time component? Fine. You want to swap in a different flavour of database that better suits the read behaviour of one component? We have the technology to rebuild him.
- **Of course, just because you can do something, doesn't mean you should - but partitioning your system in this way means you have the option.**

Characteristics of a Microservice Architecture
Decentralized Data Management

- Decentralization of data management presents in a number of different ways. At the most abstract level, it means that the conceptual model of the world will differ between systems. This is a common issue when integrating across a large enterprise, the sales view of a customer will differ from the support view. Some things that are called customers in the sales view may not appear at all in the support view. Those that do may have different attributes and (worse) common attributes with subtly different semantics.

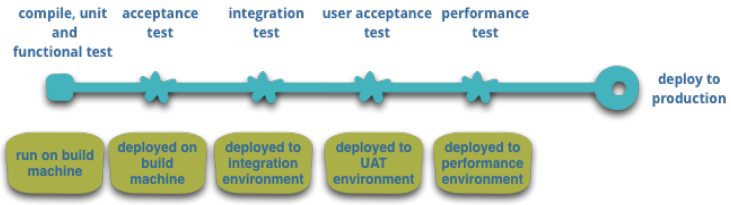
Characteristics of a Microservice Architecture
Decentralized Data Management



Characteristics of a Microservice Architecture

Infrastructure Automation

- Infrastructure automation techniques have evolved enormously over the last few years - the evolution of the cloud and AWS in particular has reduced the operational complexity of building, deploying and operating microservices.
- Many of the products or systems being build with microservices are being built by teams with extensive experience of Continuous Delivery and it's precursor, Continuous Integration. Teams building software this way make extensive use of infrastructure automation techniques. This is illustrated in the build pipeline shown below.



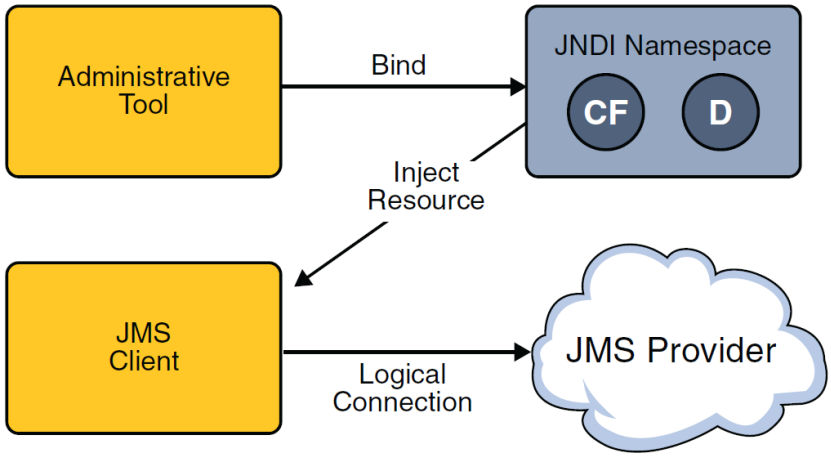
Characteristics of a Microservice Architecture
Design for failure

- A consequence of using services as components, is that applications need to be designed so that they can tolerate the failure of services. Any service call could fail due to unavailability of the supplier, the client has to respond to this as gracefully as possible. This is a disadvantage compared to a monolithic design as it introduces additional complexity to handle it. The consequence is that microservice teams constantly reflect on how service failures affect the user experience. Netflix's Simian Army induces failures of services and even datacenters during the working day to test both the application's resilience and monitoring.

JMS – Java Message Services

- Asynchronous messaging between two components
- loosely coupled
- Guarantees message delivery

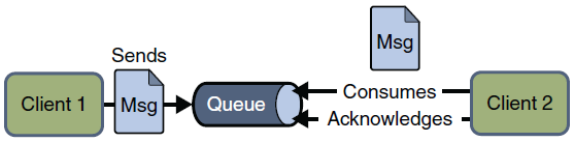
JMS – Architecture



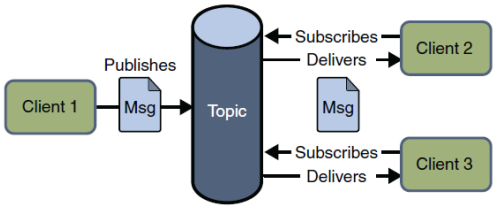
A JMS application is composed of the following parts.
■ A *JMS provider is a messaging system that implements the JMS interfaces and provides* administrative and control features. An implementation of the Java EE platform includes a JMS provider.
■ *JMS clients are the programs or components, written in the Java programming language,*
that produce and consume messages. Any Java EE application component can act as a JMS
client.
■ *Messages are the objects that communicate information between JMS clients.*
■ *Administered objects are preconfigured JMS objects created by an administrator for the use*
of clients. The two kinds of JMS administered objects are destinations and connection factories, which are described in “JMS Administered Objects” on page 903.
Figure 31–2 illustrates the way these parts interact. Administrative tools allow you to bind destinations and connection factories into a JNDI namespace. A JMS client can then use resource injection to access the administered objects in the namespace and then establish a
logical connection to the same objects through the JMS provider.

JMS - domains

- Point-to-Point Messaging Domain



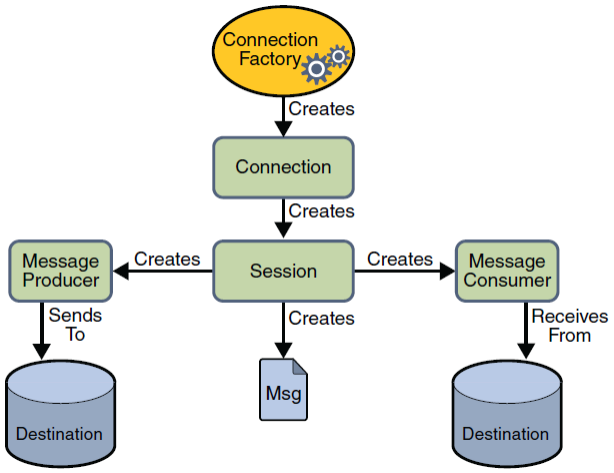
- Publish/Subscribe Messaging Domain



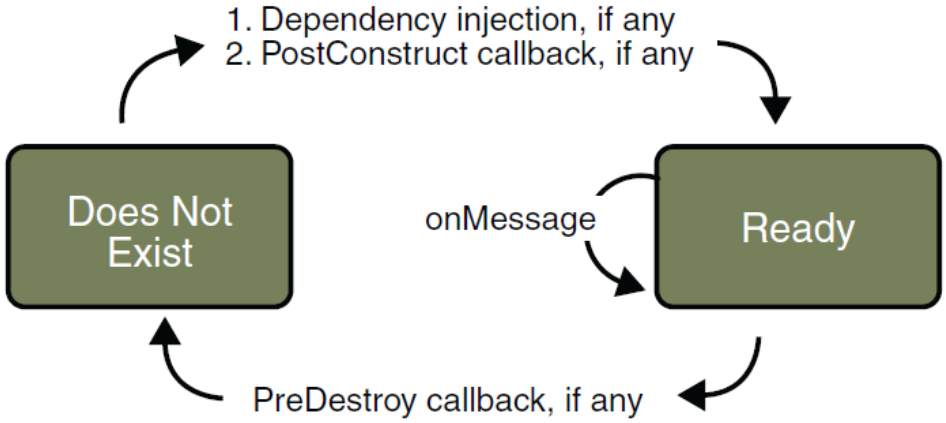
Point-to-Point Messaging Domain
A point-to-point (PTP) product or application is built on the concept of message queues, senders, and receivers. Each message is addressed to a specific queue, and receiving clients extract messages from the queues established to hold their messages. Queues retain all
messages
sent to them until the messages are consumed or until the messages expire.
PTP messaging has the following characteristics and is illustrated in Figure 31–3.
■ Each message has only one consumer.
■ A sender and a receiver of a message have no timing dependencies. The receiver can fetch the message whether or not it was running when the client sent the message.
■ The receiver acknowledges the successful processing of a message.
Use PTP messaging when every message you send must be processed successfully by one consumer.

Publish/Subscribe Messaging Domain
In a publish/subscribe (pub/sub) product or application, clients address messages to a *topic,* which functions somewhat like a bulletin board. Publishers and subscribers are generally anonymous and can dynamically publish or subscribe to the content hierarchy. The system takes care of distributing the messages arriving from a topic's multiple publishers to its multiple subscribers. Topics retain messages only as long as it takes to distribute them to current subscribers.
Pub/sub messaging has the following characteristics.
■ Each message can have multiple consumers.
■ Publishers and subscribers have a timing dependency. A client that subscribes to a topic can consume only messages published after the client has created a subscription, and the subscriber must continue to be active in order for it to consume messages.
The JMS API relaxes this timing dependency to some extent by allowing subscribers to create *durable subscriptions, which receive messages sent while the subscribers are not active. Durable* subscriptions provide the flexibility and reliability of queues but still allow clients to send messages to many recipients. For more information about durable subscriptions, see "Creating Durable Subscriptions" on page 944.
Use pub/sub messaging when each message can be processed by zero, one, or many consumers. Figure 31–4 illustrates pub/sub messaging.

JMS – Model


Connection Factory
Creates
Connection
Creates
Message Producer
Creates
Session
Creates
Message Consumer
Sends To
Creates
Receives From
Destination
Msg
Destination

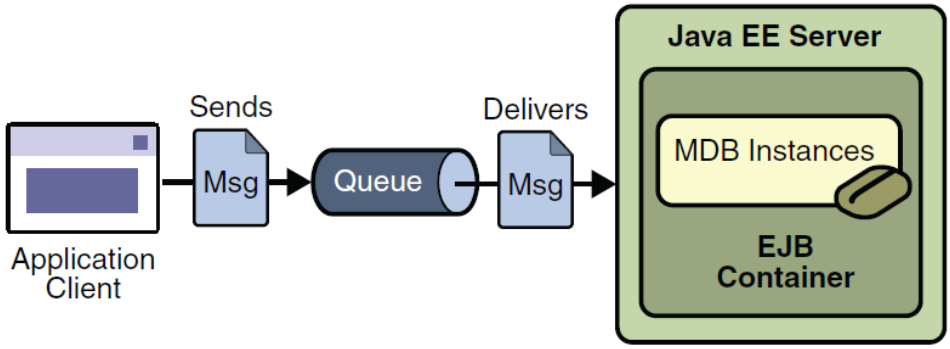
MDB – Message Driven Bean



The EJB container usually creates a pool of message-driven bean instances. For each instance,
the EJB container performs these tasks:
1. If the message-driven bean uses dependency injection, the container injects these references
before instantiating the instance.
2. The container calls the method annotated @PostConstruct, if any.

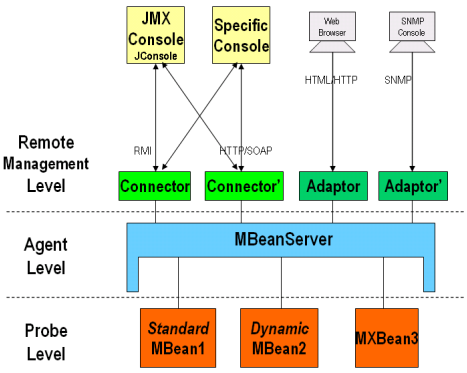
Like a stateless session bean, a message-driven bean is never passivated, and it has only two
states: nonexistent and ready to receive messages.
At the end of the life cycle, the container calls the method annotated @PreDestroy, if any. The
bean's instance is then ready for garbage collection.

MDB - example


Java EE Server
Sends
Delivers
MDB Instances
Msg
Queue
Msg
EJB
Container
Application
Client

Java Management Extensions (JMX)

- JMX is a Java technology that supplies tools for managing and monitoring applications, system objects, devices (e.g. printers) and service-oriented networks. Those resources are represented by objects called MBeans (for Managed Bean). In the API, classes can be dynamically loaded and instantiated.



Java EE Connector Architecture (JCA)

- Java EE Connector Architecture (JCA) is a Java-based technology solution for connecting application servers and enterprise information systems (EIS) as part of enterprise application integration (EAI) solutions. While JDBC is specifically used to connect Java EE applications to databases, JCA is a more generic architecture for connection to legacy systems.
- The Java EE Connector Architecture defines a standard for connecting a compliant application server to an EIS. It defines a standard set of system-level contracts between the Java EE application server and a resource adapter. The system contracts defined by the Java EE Connector Architecture are described by the specification as follows:

Java EE Connector Architecture (JCA)

- **Connection management** — Connection management enables an application server to pool connections.
- **Transaction management** — Transaction management enables an application server to use a transaction manager.
- **Security management** — Security management reduces security threats to the EIS and protects valuable information resources managed by the EIS.
- **Life cycle management** — Life cycle management enables an application server to manage the life cycle of a resource adapter from initiation through upgrades to obsolescence.
- **Work management** — Work management enables a resource adapter to do work (monitor network endpoints, invoke application components, and so on) by submitting work instances to an application server for execution.
- **Transaction inflow management** — Transaction inflow management enables a resource adapter to propagate an imported transaction to an application server.
- **Message inflow management** — Message inflow management enables a resource adapter to asynchronously deliver messages to message endpoints.

Portal

What is Portal?

- A portal is a web application that commonly provides personalization, authentication, aggregation of content from multiple sources, and provides a presentation layer for IS.
- Aggregation is the integration of content from multiple sources into a web page.
- A portal may have sophisticated personalization to provide personalized content.
  - A portal site may have different sets of portlets for different users

Portlet

What is portlet?

- The portlet provides a specific piece of content as part of the portal page.
- A portlet pluggable is a UI component, managed and displayed on a web portal.
- A portlet produces a piece (fragment) of HTML (XHTML, WML) that is aggregated into a portal page.

Portlet

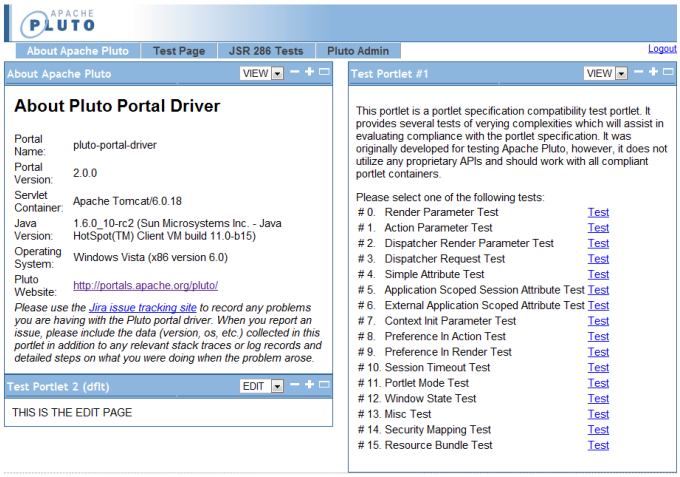
Portal page

- A portal page is a collection of non-overlapping portlet windows

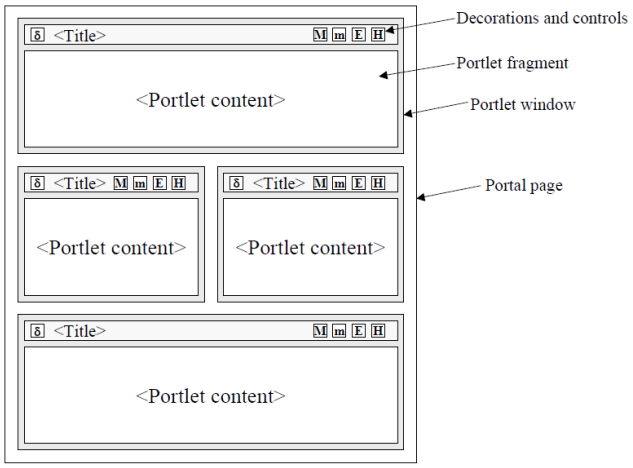
What is a portlet container

- A portlet container launches portlets and provides them with an environment and manages their lifecycle.
- It provides a persistent repository for portlet settings.
- The container does not aggregate the contents of the portlets into the page, that is the responsibility of the portal.

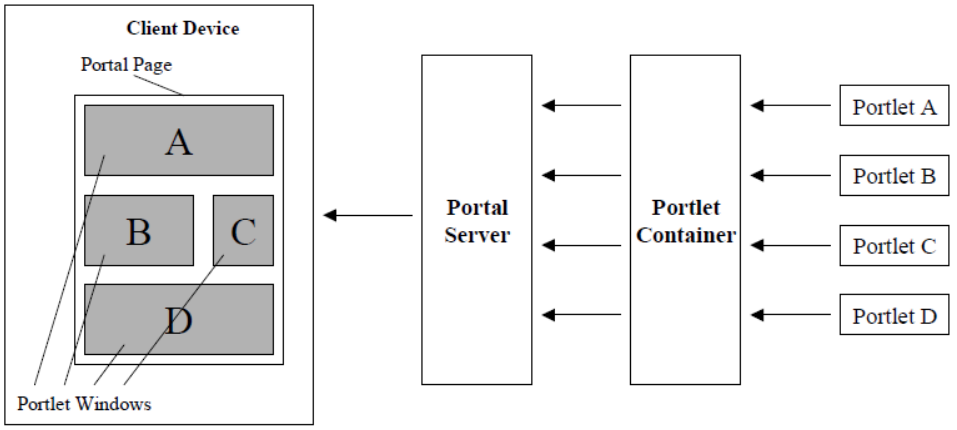
Portlet


APACHE PLUTO
About Apache Pluto
Test Page
JSR 286 Tests
Pluto Admin
Logout
About Apache Pluto
VIEW
About Pluto Portal Driver
Portal Name: pluto-portal-driver
Portal Version: 2.0.0
Servlet Container: Apache Tomcat/6.0.18
Java Version: 1.6.0_10-rc2 (Sun Microsystems Inc. - Java HotSpot(TM) Client VM build 11.0-b15)
Operating System: Windows Vista (x86 version 6.0)
Pluto Website: http://portals.apache.org/pluto/
Please use the Jira issue tracking site to record any problems you are having with the Pluto portal driver. When you report an issue, please include the data (version, os, etc.) collected in this portlet in addition to any relevant stack traces or log records and detailed steps on what you were doing when the problem arose.
Test Portlet 2 (dflt)
EDIT
THIS IS THE EDIT PAGE
Test Portlet #1
VIEW
This portlet is a portlet specification compatibility test portlet. It provides several tests of verying complexities which will assist in evaluating compliance with the portlet specification. It was originally developed for testing Apache Pluto, however, it does not utilize any proprietary APIs and should work with all compliant portlet containers.
Please select one of the following tests:
0. Render Parameter Test Test
1. Action Parameter Test Test
2. Dispatcher Render Parameter Test Test
3. Dispatcher Request Test Test
4. Simple Attribute Test Test
5. Application Scoped Session Attribute Test Test
6. External Application Scoped Attribute Test Test
7. Context Init Parameter Test Test
8. Preference In Action Test Test
9. Preference In Render Test Test
10. Session Timeout Test Test
11. Portlet Mode Test Test
12. Window State Test Test
13. Misc Test Test
14. Security Mapping Test Test
15. Resource Bundle Test Test

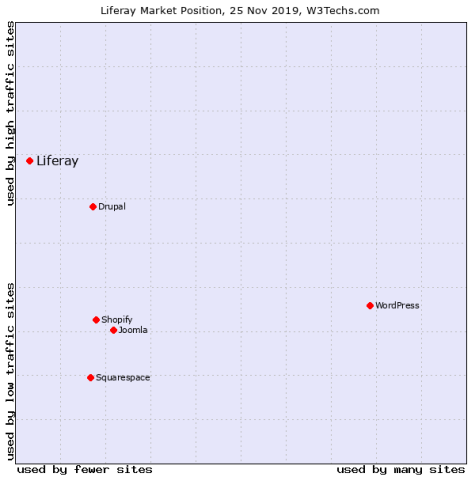
Portlet


<Title>
<Portlet content>
Decorations and controls
Portlet fragment
Portlet window
Portal page

Portlet

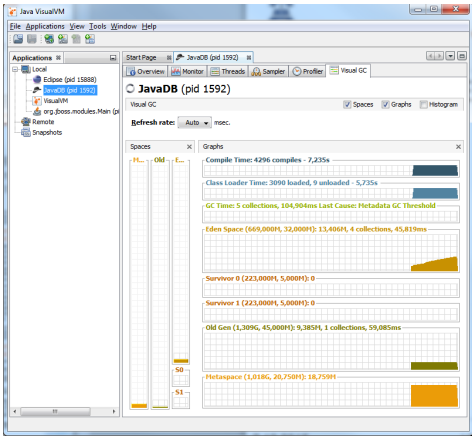

Client Device
Portal Page
A
B
C
D
Portlet Windows
Portal Server
Portlet Container
Portlet A
Portlet B
Portlet C
Portlet D

Liferay porta



Java - Profiling

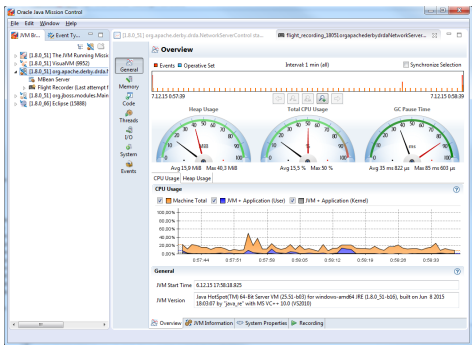
- VisualVM
- https://visualvm.github.io

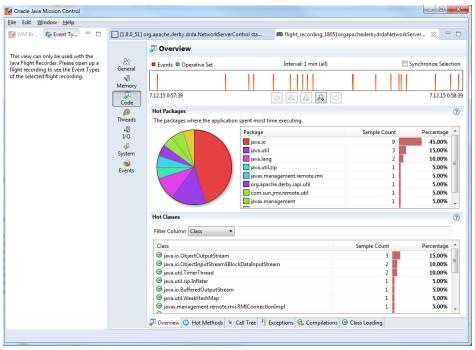


Java Mission Control

- Java Flight Recorder and Java Mission Control together create a complete tool chain to continuously collect low level and detailed runtime information enabling after-the-fact incident analysis. Java Flight Recorder is a profiling and event collection framework built into the Oracle JDK. It allows Java administrators and developers to gather detailed low level information about how the Java Virtual Machine (JVM) and the Java application are behaving. Java Mission Control is an advanced set of tools that enables efficient and detailed analysis of the extensive of data collected by Java Flight Recorder. The tool chain enables developers and administrators to collect and analyze data from Java applications running locally or deployed in production environments.

Java Mision Control

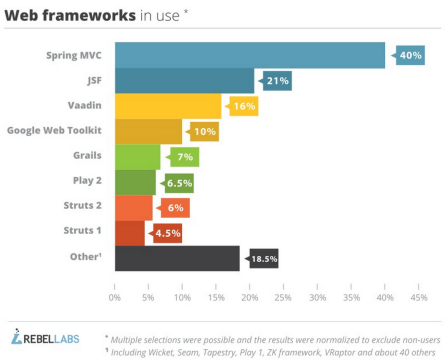


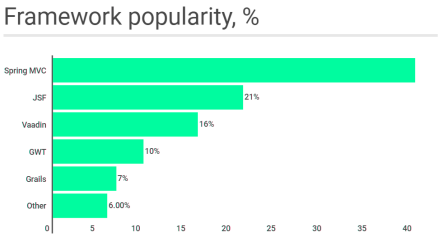


Vaadin Framework

- Vaadin Framework lets you build single page web apps in server-side Java or any other JVM language. All of the browser–server communication and DTOs are automated for you. Your app's state resides on the server, but your end-users use an HTML5 web app in their browsers.

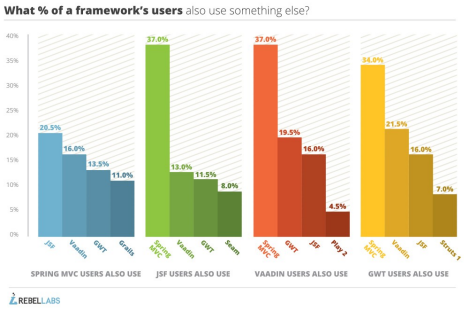
Java Frameworks





Java Frameworks

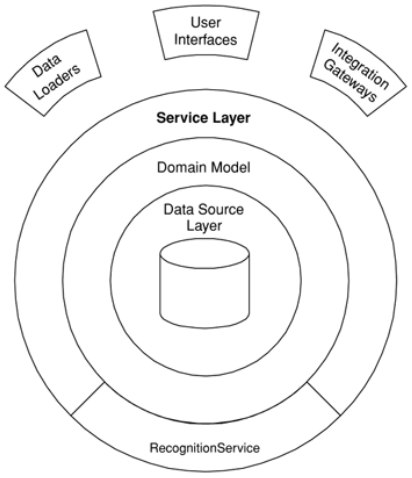
- **What % of specific framework users use more than one framework?**
- Spring MVC – 54%
- JSF – 54%
- Vaadin – 54%
- GWT – 70%



Apache Jackrabbit

- The Apache Jackrabbit™ content repository is a fully conforming implementation of the Content Repository for Java Technology API (JCR, specified in JSR 170 and JSR 283).
- A content repository is a hierarchical content store with support for structured and unstructured content, full text search, versioning, transactions, observation, and more.
- The JCR storage model is a tree of nodes and properties: nodes (addressable by path like in a filesystem) are used to organize the content, and named properties store the actual data, either as simple types (string, boolean, number, etc.) or as binary streams for storing files of arbitrary size.

Service Layer



- 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.
- Enterprise applications typically require different kinds of interfaces to the data they store and the logic they implement: data loaders, user interfaces, integration gateways, and others. Despite their different purposes, these interfaces often need common interactions with the application to access and manipulate its data and invoke its business logic. The interactions may be complex, involving transactions across multiple resources and the coordination of several responses to an action. Encoding the logic of the interactions separately in each interface causes a lot of duplication.
- A Service Layer defines an application's boundary [Cockburn PloP] and its set of available operations from the perspective of interfacing client layers. It encapsulates the application's business logic, controlling transactions and coordinating responses in the implementation of its operations.

How It Works

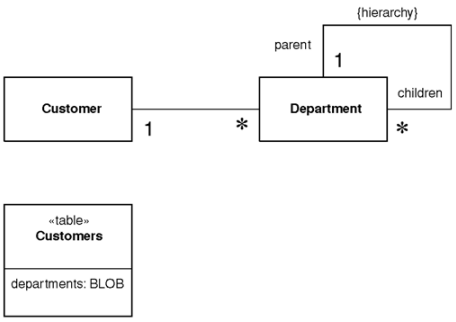
A Service Layer can be implemented in a couple of different ways, without violating the defining characteristics stated above. The differences appear in the allocation of responsibility behind the Service Layer interface. Before I delve into the various implementation possibilities, let me lay a bit of groundwork.

Kinds of "Business Logic"

Like Transaction Script (110) and Domain Model (116), Service Layer is a pattern for organizing business logic. Many designers, including me, like to divide "business logic" into two kinds: "domain logic," having to do purely with the problem domain (such as strategies for calculating revenue recognition on a contract), and "application logic," having to do with application responsibilities [Cockburn UC] (such as notifying contract administrators, and integrated applications, of revenue recognition calculations). Application logic is sometimes referred to as "workflow logic," although different people have different interpretations of "workflow."

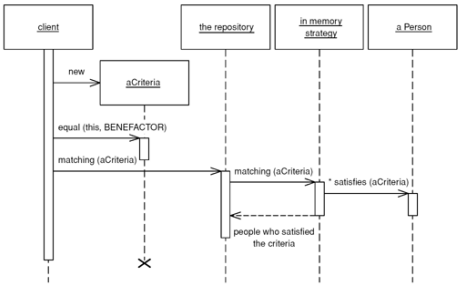
Domain Models (116) are preferable to Transaction Scripts (110) for avoiding domain logic duplication and for managing

Serialized LOB



- Saves a graph of objects by serializing them into a single large object (LOB), which it stores in a database field.
- Object models often contain complicated graphs of small objects. Much of the information in these structures isn't in the objects but in the links between them. Consider storing the organization hierarchy for all your customers. An object model quite naturally shows the composition pattern to represent organizational hierarchies, and you can easily add methods that allow you to get ancestors, siblings, descendents, and other common relationships.

Repository



- Mediates between the domain and data mapping layers using a collection-like interface for accessing domain objects.
- A system with a complex domain model often benefits from a layer, such as the one provided by Data Mapper (165), that isolates domain objects from details of the database access code. In such systems it can be worthwhile to build another layer of abstraction over the mapping layer where query construction code is concentrated. This becomes more important when there are a large number of domain classes or heavy querying. In these cases particularly, adding this layer helps minimize duplicate query logic.
- A Repository mediates between the domain and data mapping layers, acting like an in-memory domain object collection. Client objects construct query specifications declaratively and submit them to Repository for satisfaction. Objects can be added to and removed from the Repository, as they can from a simple collection of objects, and the mapping code encapsulated by the Repository will carry out the appropriate operations behind the scenes. Conceptually, a Repository encapsulates the set of objects persisted in a data store and the operations performed over them, providing a more object-oriented view of the persistence layer. Repository also supports the objective of achieving a clean separation and one-way dependency between the domain and data mapping layers.

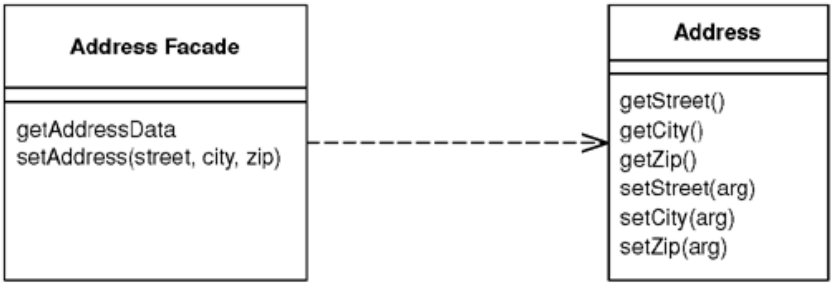
Session State Patterns

- Client Session State
- Stores session state on the client.
- Even the most server-oriented designs need at least a little Client Session State, if only to hold a session identifier. With some applications you can consider putting all of the session data on the client, in which case the client sends the full set of session data with each request and the server sends back the full session state with each response. This allows the server to be completely stateless.

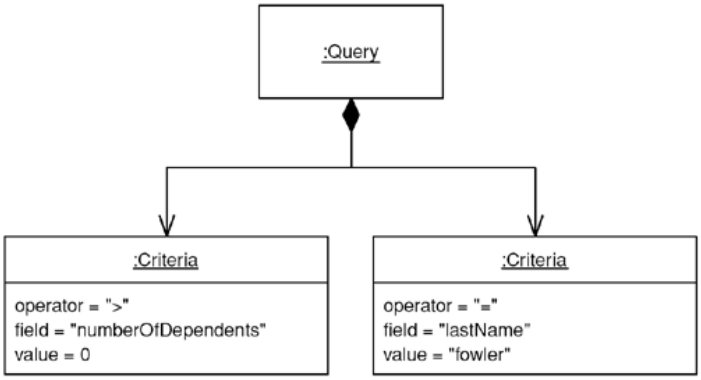
- Server Session State
- Keeps the session state on a server system in a serialized form.
- In the simplest form of this pattern a session object is held in memory on an application server. You can have some kind of map in memory that holds these session objects keyed by a session ID; all the client needs to do is to give the session ID and the session object can be retrieved from the map to process the request.

- Database Session State
- Stores session data as committed data in the database.
- When a call goes out from the client to the server, the server object first pulls the data required for the request from the database. Then it does the work it needs to do and saves back to the database all the data required.
- In order to pull information from the database, the server object will need some information about the session, which requires at least a session ID number to be stored on the client. Usually, however, this information is nothing more than the appropriate set of keys needed to find the appropriate amount of data in the database.

Remote Facade



Query Object

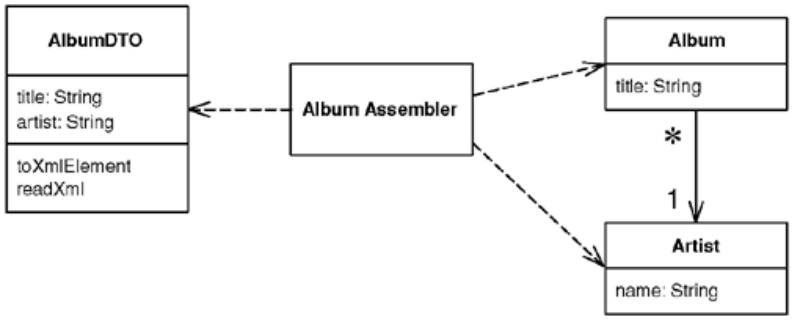


Layer Supertype

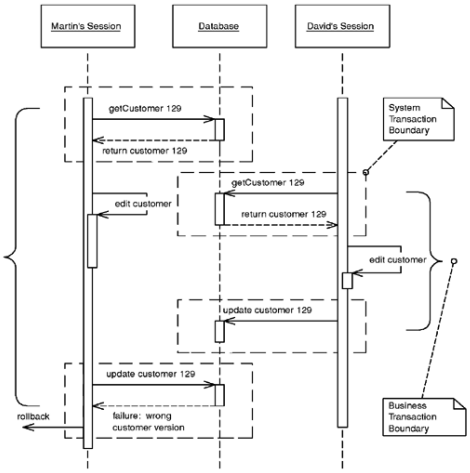
- A type that acts as the supertype for all types in its layer.
- It's not uncommon for all the objects in a layer to have methods you don't want to have duplicated throughout the system. You can move all of this behavior into a common Layer Supertype.

```
class DomainObject...
  private Long ID;
  public Long getID() {
    return ID;
  }
  public void setID(Long ID) {
    Assert.notNull("Cannot set a null ID", ID);
    this.ID = ID;
  }
  public DomainObject(Long ID) {
    this.ID = ID;
  }
  public DomainObject() {
  }
```

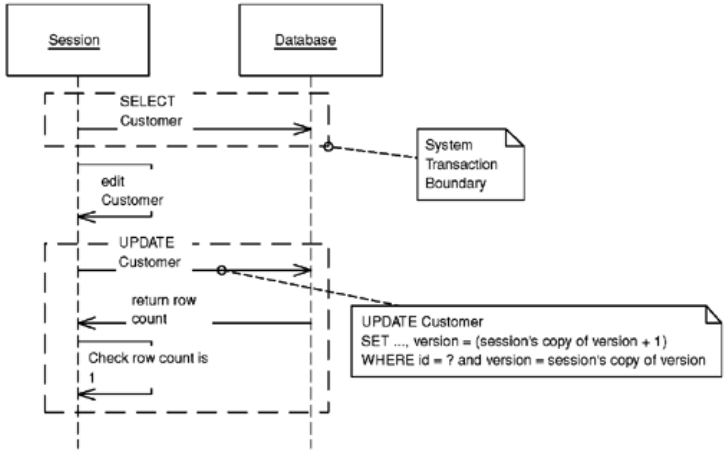
Data Transfer Object



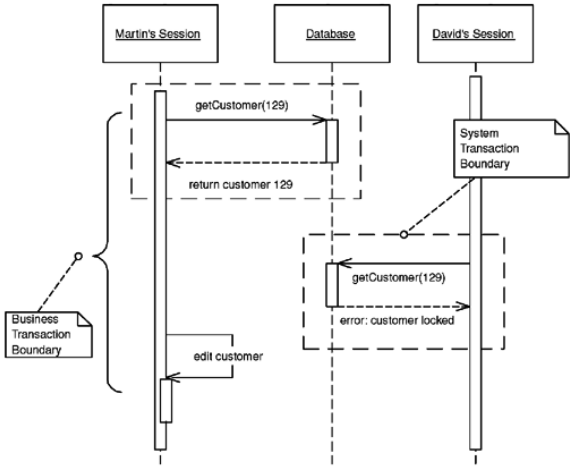
Optimistic Offline Lock



Optimistic Offline Lock



Pessimistic Offline Lock

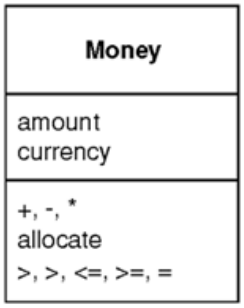


Pessimistic Offline Lock

```
public void acquireLock(Long lockable, String owner) throws ConcurrencyException {
  if (!hasLock(lockable, owner)) {
    Connection conn = null;
    PreparedStatement pstmt = null;
    try {
      conn = ConnectionManager.INSTANCE.getConnection();
      pstmt = conn.prepareStatement(INSERT_SQL);
      pstmt.setLong(1, lockable.longValue());
      pstmt.setString(2, owner);
      pstmt.executeUpdate();
    } catch (SQLException sqlEx) {
      throw new ConcurrencyException("unable to lock " + lockable);
    } finally {
      closeDBResources(conn, pstmt);
    }
  }
}
```

```
public void releaseLock(Long lockable, String owner) {
    Connection conn = null;
    PreparedStatement pstmt = null;
    try {
      conn = ConnectionManager.INSTANCE.getConnection();
      pstmt = conn.prepareStatement(DELETE_SINGLE_SQL);
      pstmt.setLong(1, lockable.longValue());
      pstmt.setString(2, owner);
      pstmt.executeUpdate();
    } catch (SQLException sqlEx) {
      throw new SystemException("unexpected error releasing lock on " +
lockable);
    } finally {
      closeDBResources(conn, pstmt);
    }
  }
```

Money



Special Case