Design of Service Layer and Application Logic

This article is about organizing Application Logic and designing the Service Layer, Use Case, CQRS, Event Sourcing, MVC, etc.

Contents

Layers of logic

Before digging deep into it, it would be nice to understand what Application Logic is and how it differs from Business Logic.

Layered Architecture

One of the most cited definitions of key conceptual layers gives Eric Evans:

> *User Interface (or Presentation Layer)*
>
> *Responsible for showing information to the user and interpreting the user's commands. The external actor might sometimes be another computer system rather than a human user.*
>
> *Application Layer*
>
> *Defines the jobs the software is supposed to do and directs the expressive domain objects to work out problems. The tasks this layer is responsible for are meaningful to the business or necessary for interaction with the application layers of other systems. This layer is kept thin. It* ***does not contain business rules*** *or knowledge, but only coordinates tasks and delegates work to collaborations of domain objects in the next layer down. It does not have state reflecting the business situation, but it can have state that reflects the progress of a task for the user or the program.*

Domain Layer (or Model Layer)

Responsible for representing concepts of the business, information about the ***business situation, and business rules****. State that reflects the business situation is controlled and used here, even though the technical details of storing it are delegated to the infrastructure. This layer is the heart of business software.*

Infrastructure Layer

Provides generic technical capabilities that support the higher layers: message sending for the application, persistence for the domain, drawing widgets for the UI, and so on. The infrastructure layer may also support the pattern of interactions between the four layers through an architectural framework.

- "Domain-Driven Design: Tackling Complexity in the Heart of Software" [4] by Eric Evans

Ward Cunningham gives us the next definitions:

Factor your application classes into four layers in the following way (see Figure 1: FourLayerArchitecture):

The View layer. This is the layer where the physical window and widget objects live. It may also contain Controller classes as in classical MVC. Any new user interface widgets developed for this application are put in this layer. In most cases today this layer is completely generated by a window-builder tool.

The ApplicationModel layer. This layer mediates between the various user interface components on a GUI screen and translates the messages that they understand into messages understood by the objects in the domain model. It is responsible for the flow of the application and controls navigation from window to window. This layer is often partially generated by a window-builder and partially coded by the developer.

The DomainModel layer. This is the layer where most objects found in an OO analysis and design will reside. Examples of the types of objects found in this layer may be Orders, Employees, Sensors, or whatever is appropriate to the problem domain.

The Infrastructure layer. This is where the objects that represent connections to entities outside the application (specifically those outside the object world) reside. Examples of objects in this layer would include SQLTables, 3270Terminals, SerialPorts, SQLBrokers and the like.

> - *Four Layer Architecture*, *Ward Cunningham*

But what does the term Business itself mean? Misunderstanding of this term often leads to significant design problems.

What is Business Logic?

The most authoritative explanation of the term Business can be found, as usual, on the Ward Cunningham website:

> *Software intersects with the Real World. Imagine that.*

There you can find a definition of Business Rule:

> *A Business Rule (in a programming context) is knowledge that gets applied to a set of data to create new value. Or it may be a rule about how to create, modify, or remove data. Or perhaps it is a rule that specifies when certain processes occur.*
>
> *For example, we have a rule about email addresses – when the Driver Name field on our object identifier changes, we erase the email address. When we receive a new email address, we make sure that it contains an "@" sign and a valid domain not on our blacklist.*

Business Logic Definition:

> *Business logic is that portion of an enterprise system which determines how data is:*
>
> - *Transformed and/or calculated. For example, business logic determines how a tax total is calculated from invoice line items.*
> - *Routed to people or software systems, aka workflow.*

The term Business should be distinguished from the term Business Domain:

> *A category about the business domain, such as accounting, finance, inventory, marketing, tracking, billing, reporting, charting, taxes, etc.*

You should also distinguish Business from Business Process:

> *A Business Process is some reproduceable process within an organization. Often it is a something that you want to setup once and reuse over and over again.*
>
> *Companies spend a lot of time and money identifying Business Processes, designing the software that captures a Business Process and then testing and documenting these processes.*
>
> *One example of a Business Process is "Take an order on my web site". It might involve a customer, items from a catalog and a credit card. Each of these things is represented by business objects and together they represent a Business Process.*

Wikipedia gives us the following definition of the term Business Logic:

> *In computer software, business logic or domain logic is the part of the program that encodes the real-world Business Rules that determine how data can be created, stored, and changed. It is contrasted with the remainder of the software that might be concerned with lower-level details of managing a database or displaying the user interface, system infrastructure, or generally connecting various parts of the program.*

Let me summarize this in my own words:

Business Logic
: is a modeling of objects and processes of the domain (i.e., the real world).

Application Logic
: is what provides and coordinates the operation of the business logic.

Subtypes of Business Rules

In “Clean Architecture,” Robert Martin divides Business Rules into two types:

- Application-specific Business Rules
- Application-independent Business Rules

> *Thus we find the system divided into decoupled horizontal layers–the UI, application-specific Business Rules, application-independent Business Rules, and the database, just to mention a few.*
>
> *- “Clean Architecture” by Robert Martin*

Chapters 16, 20 and 22 of Clean Architecture explain in detail the types of Business Rules. Although, Robert Martin allocate the separate category UseCase (Interactor) classes for Application-specific Business Rules, in practice, this level is often rounded to Application Logic level. For example, Martin Fowler and Randy Stafford divide Business Logic into two types - Domain Logic and Application 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.”*
>
> *- “Patterns of Enterprise Application Architecture” [3] by Martin Fowler, Randy Stafford*

In some places, he is inclined to refer “Business Rules” to Domain Logic:

> *The problem came with domain logic: business rules, validations, calculations, and the like.*
>
> *- "Patterns of Enterprise Application Architecture" [3] by Martin Fowler*

And even he admits the presence of a certain vagueness:

> *Then there's the matter of what comes under the term "business logic." I find this a curious term because there are few things that are less logical than business logic.*
>
> *- "Patterns of Enterprise Application Architecture" [3] by Martin Fowler*

Why is separation of Business Rules from Application Logic so important?

Since the purpose of creating an application is precisely the implementation of Business Rules, it is important to ensure their portability and to separate them from the Application Logic. These two different kinds of rules will change at different times, at different rates, and for different reasons - so they should be separated so that they can be independently changed [2] . Grady Booch said that "Architecture represents the significant design decisions that shape a system, where significant is measured by cost of change [2] ."

Ways to Organize Application Logic

Four ways to organize Application Logic are widespread:

1. Orchestration Service ("request/response", i.e. the service is aware of the interface of other services) aka Service Layer.

2. Choreography Service (Event-Driven, т.е. loosely coupled), which is a derivative of Command pattern and is used commonly in Event-Driven Architecture (in particular, in CQRS and Event Sourcing applications; a reducer in Redux is a good example), and in DDD applications (a subscriber of Domain/Integration Event).
3. Front Controller and Application Controller (which are also kinds of Command pattern).

> *"A Front Controller handles all calls for a Web site, and is usually structured in two parts: a Web handler and a command hierarchy."*
>
> *- "Patterns of Enterprise Application Architecture" [3] by Martin Fowler and others.*

> *"For both the domain commands and the view, the application controller needs a way to store something it can invoke. A Command [Gang of Four] is a good choice, since it allows it to easily get hold of and run a block of code."*
>
> *- "Patterns of Enterprise Application Architecture" [3] by Martin Fowler and others.*

4. Use Case (see also), which also is a kind of Command pattern. At 15:50 Robert C. Martin points to a parallel between Use Case and Command pattern.

In fact, even Method Object is a derivative of Command pattern.

Use Case is necessary because there is Application-specific Business Logic which does not make sense outside the context of the application. It ensures that these application-specific Business Rules are independent of the Application Logic using inverse control (IoC).

If the Use Case did not contain Business Logic, then there would be no sense in separating it from Page Controller, otherwise the application would try to abstract itself from itself.

As you can see, varieties of the Command pattern are widely used to organize the Application Logic.

The listed methods organize, first of all, Application Logic, and only then - Business Logic, which is not obligatory for them, except for Use Case, because otherwise it would have no reason to exist.

With proper organization of the Business Logic, and high quality of ORM (if used, of course), the dependence of the Business Logic of the application will be minimal. The main difficulty of any ORM is to provide access to related objects without mixing Application Logic (and data access logic) into Domain Models, this topic we will discuss in one of the next posts.

Understanding the common features of the methods of organizing Application Logic allows us to design more flexible applications, and, as a result, more painlessly change the architectural style, for example, from Layered to Event-Driven. This topic is covered in part in Chapter 16 “Independence” of “Clean Architecture” by Robert C. Martin, and in section “Premature Decomposition” of Chapter 3 “How to Model Services” of “Building Microservices” by Sam Newman.

What is Service?

> *SERVICE - An operation offered as an interface that stands alone in the model, with no encapsulated state.*
>
> *- “Domain-Driven Design: Tackling Complexity in the Heart of Software”* [4]

> *In some cases, the clearest and most pragmatic design includes operations that do not conceptually belong to any object. Rather than force the issue, we can follow the natural contours of the problem space and include SERVICES explicitly in the model.*
>
> *There are important domain operations that can’t find a natural home in an ENTITY or VALUE OBJECT . Some of these are intrinsically activities or actions, not things, but since our modeling paradigm is objects, we try to fit them into objects anyway...*
>
> *A SERVICE is an operation offered as an interface that stands alone in the model, without encapsulating state, as ENTITIES and VALUE OBJECTS do. S ERVICES are a common pattern in technical frameworks, but they can also apply in the domain layer.*
>
> *The name service emphasizes the relationship with other objects. Unlike ENTITIES and VALUE OBJECTS , it is defined purely in terms of what it can do for a client. A SERVICE tends to be named for an activity, rather than an entity—a verb rather than a noun. A*

> *SERVICE can still have an abstract, intentional definition; it just has a different flavor than the definition of an object. A SERVICE should still have a defined responsibility, and that responsibility and the interface fulfilling it should be defined as part of the domain model. Operation names should come from the UBIQUITOUS LANGUAGE or be introduced into it. Parameters and results should be domain objects.*
>
> *SERVICES should be used judiciously and not allowed to strip the ENTITIES and VALUE OBJECTS of all their behavior. But when an operation is actually an important domain concept, a SERVICE forms a natural part of a MODEL-DRIVEN DESIGN . Declared in the model as a SERVICE, rather than as a phony object that doesn't actually represent anything, the standalone operation will not mislead anyone.*
>
> *A good SERVICE has three characteristics.*
>
> *1. The operation relates to a domain concept that is not a natural part of an ENTITY or VALUE OBJECT . 2. The interface is defined in terms of other elements of the domain model. 3. The operation is stateless.*
>
> *Statelessness here means that any client can use any instance of a particular SERVICE without regard to the instance's individual history. The execution of a SERVICE will use information that is accessible globally, and may even change that global information (that is, it may have side effects). But the SERVICE does not hold state of its own that affects its own behavior, as most domain objects do.*
>
> *When a significant process or transformation in the domain is not a natural responsibility of an ENTITY or VALUE OBJECT, add an operation to the model as a standalone interface declared as a SERVICE. Define the interface in terms of the language of the model and make sure the operation name is part of the UBIQUITOUS LANGUAGE. Make the SERVICE stateless.*
>
> *- "Domain-Driven Design: Tackling Complexity in the Heart of Software"* [4]

Service types by layers of logic

Eric Evans divides Services into three layers of logic:

> *Partitioning Services into Layers*

Application

Funds Transfer App Service

- *Digests input (such as an XML request).*
- *Sends message to domain service for fulfillment.*
- *Listens for confirmation.*
- *Decides to send notification using infrastructure service.*

Domain

Funds Transfer Domain Service

- *Interacts with necessary Account and Ledger objects, making appropriate debits and credits.*
- *Supplies confirmation of result (transfer allowed or not, and so on).*

Infrastructure Send Notification Service

Sends e-mails, letters, and other communications as directed by the application.

- "Domain-Driven Design: Tackling Complexity in the Heart of Software" [4]

Most SERVICES discussed in the literature are purely technical and belong in the infrastructure layer. Domain and application SERVICES collaborate with these infrastructure SERVICES. For example, a bank might have an application that sends an e-mail to a customer when an account balance falls below a specific threshold. The interface that encapsulates the e-mail system, and perhaps alternate means of notification, is a SERVICE in the infrastructure layer.

It can be harder to distinguish application SERVICES from domain SERVICES. The application layer is responsible for ordering the notification. The domain layer is responsible for determining if a threshold was met–though this task probably does not call for a SERVICE, because it would fit the responsibility of an "account" object. That banking application could be responsible for funds transfers. If a SERVICE were devised to make appropriate debits and credits for a funds transfer,that capability would belong in the domain layer. Funds transfer has a meaning in the banking domain language, and it involves fundamental business logic. Technical SERVICES should lack any business meaning at all.

> *Many domain or application SERVICES are built on top of the populations of ENTITIES and VALUES, behaving like scripts that organize the potential of the domain to actually get something done. ENTITIES and VALUE OBJECTS are often too fine-grained to provide a convenient access to the capabilities of the domain layer. Here we encounter a very fine line between the domain layer and the application layer. For example, if the banking application can convert and export our transactions into a spreadsheet file for us to analyze, that export is an application SERVICE. There is no meaning of "file formats" in the domain of banking, and there are no business rules involved.*
>
> *On the other hand, a feature that can transfer funds from one account to another is a domain SERVICE because it embeds significant business rules (crediting and debiting the appropriate accounts, for example) and because a "funds transfer" is a meaningful banking term. In this case, the SERVICE does not do much on its own; it would ask the two Account objects to do most of the work. But to put the "transfer" operation on the Account object would be awkward, because the operation involves two accounts and some global rules.*
>
> - *"Domain-Driven Design: Tackling Complexity in the Heart of Software"* [4]

> *Domain Models (116) are preferable to Transaction Scripts (110) for avoiding domain logic duplication and for managing complexity using classical design patterns. But putting application logic into pure domain object classes has a couple of undesirable consequences. First, domain object classes are less reusable across applications if they implement application-specific logic and depend on application-specific packages. Second, commingling both kinds of logic in the same classes makes it harder to reimplement the application logic in, say, a workflow tool if that should ever become desirable. For these reasons Service Layer factors each kind of business logic into a separate layer, yielding the usual benefits of layering and rendering the pure domain object classes more reusable from application to application.*
>
> - *"Patterns of Enterprise Application Architecture"* [3]

Domain Service

Higher-level policies belong to Domain Logic, so we start with it. Fortunately, this is not the most numerous type of Services.

In detail, the topic of Domain Services and the reasons for their existence are revealed Vaughn Vernon:

> *Further, don't confuse a Domain Service with an Application Service. We don't want to house business logic in an Application Service, but we do want business logic housed in a Domain Service. If you are confused about the difference, compare with Application. Briefly, to differentiate the two, an Application Service, being the natural client of the domain model, would normally be the client of a Domain Service. You'll see that demonstrated later in the chapter. Just because a Domain Service has the word service in its name does not mean that it is required to be a coarse-grained, remote-capable, heavyweight transactional operation.*
>
> ...
>
> *You can use a Domain Service to*
>
> - *Perform a significant business process*
> - *Transform a domain object from one composition to another*
> - *Calculate a Value requiring input from more than one domain object*
>
> *- "Implementing Domain-Driven Design" by Vaughn Vernon*

Application Service

This is the most numerous type of Services. Application Services are also known as Service Layer.

Infrastructure Service

Infrastructure Service should be separate of other types of Service.

> *The infrastructure layer usually does not initiate action in the domain layer. Being "below" the domain layer, it should have no specific knowledge of the domain it is serving. Indeed, such technical capabilities are most often offered as SERVICES . For example, if an application needs to send an e-mail, some message-sending interface can be located in the infrastructure layer and the application layer*

elements can request the transmission of the message. This decoupling gives some extra versatility. The message-sending interface might be connected to an e-mail sender, a fax sender, or whatever else is available. But the main benefit is simplifying the application layer, keeping it narrowly focused on its job: knowing when to send a message, but not burdened with how.

The application and domain layers call on the SERVICES provided by the infrastructure layer. When the scope of a SERVICE has been well chosen and its interface well designed, the caller can remain loosely coupled and uncomplicated by the elaborate behavior the SERVICE interface encapsulates.

But not all infrastructure comes in the form of SERVICES callable from the higher layers. Some technical components are designed to directly support the basic functions of other layers (such as providing an abstract base class for all domain objects) and provide the mechanisms for them to relate (such as implementations of MVC and the like). Such an "architectural framework" has much more impact on the design of the other parts of the program.

- "Domain-Driven Design: Tackling Complexity in the Heart of Software" [4]

Infrastructure Layer - Provides generic technical capabilities that support the higher layers: message sending for the application, persistence for the domain, drawing widgets for the UI, and so on. The infrastructure layer may also support the pattern of interactions between the four layers through an architectural framework.

- "Domain-Driven Design: Tackling Complexity in the Heart of Software" [4]

Service types by collaboration

By collaboration of interaction the Services are divided into Orchestration Service ("request/response", i.e. the service is aware of the interface of other Services) and Choreography Service (Event-Driven, т.е. loosely coupled) [8]. Them are two idiomatic styles of collaboration. The main drawback of the first one is a high awareness of the interface of other Services, i.e. High coupling, which reduces its reuse. The last one is a variation of the Command pattern, and is used commonly in Event-Driven Architecture (in particular, in CQRS and Event Sourcing applications; a reducer in Redux is a good example), and in DDD applications (a subscriber of Domain/Integration Event).

Orchestration Service

Orchestration Service is known as Service Layer and is cosidered in more detail below.

Choreography Service

There is an interesting article "Clarified CQRS" by Udi Dahan, cited by Martin Fowler in his article "CQRS".

And this article has an interesting point.

> *The reason you don't see this layer explicitly represented in CQRS is that it isn't really there...*
>
> *- "Clarified CQRS" by Udi Dahan*

In fact, a command handler is a Service, but an event-driven one that follows a given interface. It should contain Application Logic (not Business Logic).

> *Our command processing objects in the various autonomous components actually make up our service layer.*
>
> *- "Clarified CQRS" by Udi Dahan*

Choreography Services can only be at Application Logic, even if it is subscribed to a Domain Event.

Common mistakes of design Choreography Service

Sometimes, Orchestration Services proxy calls to Choreographic Services. This often happens with frontend-developers, for example, when Redux/NgRx is used together with an Angular based application, which uses Services a lot.

Having Low Coupling Event-Driven Choreography Services as Command handlers, it would be a design mistake to try to bind them with the High Coupling classic Orchestration Service (with the only purpose of helping the Application Logic hide them from itself).

> *Each command is independent of the other, so why should we allow the objects which handle them to depend on each other?*
>
> *- "Clarified CQRS" by Udi Dahan*

However, there is a question of awareness of Command handlers and applications about the interface of a specific CQRS implementation. To align the interfaces, there is an Adapter pattern, which can be provided if necessary.

Another common mistake is placing Business Logic in a Choreography Service and degenerating the behavior of Domain Models.

This leads to the problem Eric Evans talked about:

> *"If the framework's partitioning conventions pull apart the elements implementing the conceptual objects, the code no longer reveals the model.*
>
> *There is only so much partitioning a mind can stitch back together, and if the framework uses it all up, the domain developers lose their ability to chunk the model into meaningful pieces."*
>
> *- "Domain-Driven Design: Tackling Complexity in the Heart of Software" by Eric Evans*

In an application with extensive Business Logic, this can significantly degrade the quality of business modeling, and complicate the process of Model Distillation in a process of Knowledge Crunching [4]. Also, such code acquires signs of "Divergent Change" [7] and

"Shotgun Surgery" [7], which greatly complicates elimination of domain modeling mistakes in a process of Evolutionary Design. Ultimately, this leads to a rapid increase in the cost of code change.

Udi Dahan in his article allows the use of Transaction Script to organize Business Logic. In this case, the choice between Transaction Script и Domain Model is considered in detail in "Patterns of Enterprise Application Architecture" by M. Fowler and others. Transaction Script may be appropriate when an application uses Redux together with GraphQL to minimize network traffic. If an application uses the REST-API and has extensive Business Logic, the use of the Domain Model and DDD will be more appropriate.

Service types by communication

By communication, Services are divided into Synchronous and Asynchronous.

Service types by state

Stateless Service

Typically, most Services are stateless. They are well known, and there is nothing to add.

Statefull Service

The UseCases/Interactors [2] classes are a variation of the Command pattern, and can be considered as a Statefull Service.

Eric Evans has a similar idea:

> *We might like to create a Funds Transfer object to represent the two entries plus the rules and history around the transfer. But we are still left with calls to SERVICES in the interbank networks. What's more, in most development systems, it is awkward to make a direct interface between a domain object and external resources. We can dress up such external SERVICES with a FACADE that takes inputs in*

> *terms of the model, perhaps returning a Funds Transfer object as its result. But whatever intermediaries we might have, and even though they don't belong to us, those SERVICES are carrying out the domain responsibility of funds transfer.*
>
> *- "Domain-Driven Design: Tackling Complexity in the Heart of Software" [4]*

And Randy Stafford with Martin Fowler too:

> *The two basic implementation variations are the domain facade approach and the operation script approach. In the domain facade approach a Service Layer is implemented as a set of thin facades over a Domain Model (116). The classes implementing the facades don't implement any business logic. Rather, the Domain Model (116) implements all of the business logic. The thin facades establish a boundary and set of operations through which client layers interact with the application, exhibiting the defining characteristics of Service Layer.*
>
> *In the operation script approach a Service Layer is implemented as a set of thicker classes that directly implement application logic but delegate to encapsulated domain object classes for domain logic. The operations available to clients of a Service Layer are implemented as scripts, organized several to a class defining a subject area of related logic. Each such class forms an application "service," and it's common for service type names to end with "Service." A Service Layer is comprised of these application service classes, which should extend a Layer Supertype (475), abstracting their responsibilities and common behaviors.*
>
> *- "Patterns of Enterprise Application Architecture" [3] by Martin Fowler, Randy Stafford*

Notice, he used the term "Domain Model". These guys are the last of those who can confuse "Domain Model" and "DataMapper", especially with so many editors and reviewers. A client expects an interface from the domain model that it does not implement and should not implement for some reason (usually the Single Responsibility Principle). On the other hand, the client can not implement this behavior itself, as this would lead to "G14: Feature Envy" [1]. There is an Adapter (aka Wrapper) pattern for interface alignment, see "Design Patterns Elements of Reusable Object-Oriented Software" [6]. Statefull Service differs from the usual Adapter pattern only in that it contains lower-level logic, i.e. Application Logic, rather than Business Logic of Domain Model.

This approach reminds me of "Cross-Cutting Concerns" [1], with the only difference being that "Cross-Cutting Concerns" implements the interface of the original object (delegate), while domain facade complements it. When a wrapper implements the

interface of the original object, it is usually called Aspect or Decorator. Often in such cases the term Proxy is used, but, in fact, the Proxy pattern has a slightly different purpose. This approach is often used to provide the Domain Model with the logic of access to related objects, while keeping the Domain Model completely "clean", i.e. separated from the behavior of lower level logic.

When I was working with legacy code, I saw swollen Domain Models with a huge number of methods (I met up to several hundred methods). If you analyze such models, you can often find extrinsical responsibilities in the class. As you know, size of a class is measured by amount of its responsibilities. Statefull Service and Adapter pattern are a good alternative to remove extrinsical responsibilities from a model and make swollen models lose weight.

Destination of Service Layer

> *A 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.*
>
> *- "Patterns of Enterprise Application Architecture"* [3]

> *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 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.*
>
> *- "Patterns of Enterprise Application Architecture"* [3]

> *The benefit of Service Layer is that it defines a common set of application operations available to many kinds of clients and it coordinates an application's response in each operation. The response may involve application logic that needs to be transacted atomically across multiple transactional resources. Thus, in an application with more than one kind of client of its business logic, and complex responses in its use cases involving multiple transactional resources, it makes a lot of sense to include a Service Layer with container-managed transactions, even in an undistributed architecture.*
>
> *- "Patterns of Enterprise Application Architecture"* [3]

> *A common approach in handling domain logic is to split the domain layer in two. A Service Layer (133) is placed over an underlying Domain Model (116) or Table Module (125). Usually you only get this with a Domain Model (116) or Table Module (125) since a domain layer that uses only Transaction Script (110) isn't complex enough to warrant a separate layer. The presentation logic interacts with the domain purely through the Service Layer (133), which acts as an API for the application.*
>
> *As well as providing a clear API, the Service Layer (133) is also a good spot to place such things as transaction control and security. This gives you a simple model of taking each method in the Service Layer (133) and describing its transactional and security characteristics. A separate properties file is a common choice for this, but .NET's attributes provide a nice way of doing it directly in the code.*
>
> *- "Patterns of Enterprise Application Architecture"* [3]

Traditionally Service Layer is an Application layer logic. This implies that level of Service Layer is lower than level of Domain Layer (i.e. layer of real world objects, which is also called "business rules"). This means that the objects of the Domain Layer should not be aware of the Service Layer.

In addition to the above, the Service Layer can carry the following responsibilities:

- To combine the parts of an atomic operation (for example, application should save the data to several storages, e.g. database, redis, file system within a single business transaction or should roll back all).
- To hide the data source (here it duplicates the responsibility of the pattern Repository) and can be omitted if there are no other reasons.

- To aggregate the application level operations that are being reused by several clients (for example, some part of application-level logic is used in several different controllers).
- As basis for implementation of Remote Facade.
- When you have a large controller method, you have to do decomposition. Thus, you apply Extract Method to separate each responsibility into own method. When you did it, you found that the class lost its focus. The quantity of methods has been increased that means the Cohesion (i.e. coefficient of sharing the class' properties by the class' methods) has been reduced. To restore the Cohesion you have to extract these methods into separate Method Object, which can be used as a Service Layer.
- The Service Layer can be used as an aggregator for queries if it is over the Repository pattern and uses the Query object pattern. The fact is that the Repository pattern limits its interface using the Query Object interface. And since class does not have to make assumptions about its clients, it is impossible to accumulate pre-defined queries in the Repository class, because it can not be aware about the all needs of all clients. Clients should take care of themselves. But the Service Layer was created for client service. Therefore, it's a responsibility of the Service Layer.

In other cases, the logic of the Service Layer can be placed directly at the application level (usually a controller).

When not to use Service Layer?

> *The easier question to answer is probably when not to use it. You probably don't need a Service Layer if your application's business logic will only have one kind of client say, a user interface and its use case responses don't involve multiple transactional resources. In this case your Page Controllers can manually control transactions and coordinate whatever response is required, perhaps delegating directly to the Data Source layer. But as soon as you envision a second kind of client, or a second transactional resource in use case responses, it pays to design in a Service Layer from the beginning.*
>
> *- "Patterns of Enterprise Application Architecture"* [3]

However, the widely held view that access to the model should always be made through the Service Layer:

> *My preference is thus to have the thinnest Service Layer (133) you can, if you even need one. My usual approach is to assume that I don't need one and only add it if it seems that the application needs it. However, I know many good designers who always use a Service Layer (133) with a fair bit of logic, so feel free to ignore me on this one.*
>
> *- "Patterns of Enterprise Application Architecture"* [3]

> *The idea of splitting a services layer from a domain layer is based on a separation of workflow logic from pure domain logic. The services layer typically includes logic that's particular to a single use case and also some communication with other infrastructures, such as messaging. Whether to have separate services and domain layers is a matter some debate. I tend to look as it as occasionally useful rather than mandatory, but designers I respect disagree with me on this.*
>
> *- "Patterns of Enterprise Application Architecture"* [3]

Service is not a wrapper for Data Mapper

Often Service Layer is mistakenly made in the for of wrapper over DataMapper. This is not quite the right decision. A Data Mapper serves a Domain Model, a Repository serves an Aggregate [9], but a Service serves a client (or a client group). The Service Layer can manipulate multiple Data Mappers, Repositories, other Services within a business transaction and in favour of a client. Therefore, Service's methods usually contain name of the returned Domain Model as a suffix (for example, getUser()), while methods of a Data Mapper (or a Repository) do not need such suffix (since the Domain name is already present in name of the Data Mapper class, and the Data Mapper serves only one Domain Model).

> *Identifying the operations needed on a Service Layer boundary is pretty straightforward. They're determined by the needs of Service Layer clients, the most significant (and first) of which is typically a user interface.*
>
> *- "Patterns of Enterprise Application Architecture"* [3]

Implementation of Service Layer

There is a few examples of Service Layer implementations:

- https://github.com/in2it/zfdemo/blob/master/application/modules/user/services/User.php
- https://framework.zend.com/manual/2.4/en/in-depth-guide/services-and-servicemanager.html
- https://framework.zend.com/manual/2.4/en/user-guide/database-and-models.html#using-servicemanager-to-configure-the-table-gateway-and-inject-into-the-albumtable
- https://github.com/zendframework/zf2-tutorial/blob/master/module/Album/src/Album/Model/AlbumTable.php

Inversion of control

Use Inversion of control, desirable in the form of Passive [1] "Dependency Injection" (DI).

> *True Dependency Injection goes one step further. The class takes no direct steps to resolve its dependencies; it is completely passive. Instead, it provides setter methods or constructor arguments (or both) that are used to inject the dependencies. During the con- struction process, the DI container instantiates the required objects (usually on demand) and uses the constructor arguments or setter methods provided to wire together the depen- dencies. Which dependent objects are actually used is specified through a configuration file or programmatically in a special-purpose construction module. "Clean Code: A Handbook of Agile Software Craftsmanship"* [1]

One of the main responsibilities of Service Layer is the hiding of data source. It allows you to use Service Stub for testing. The same approach can be used for parallel development, when the implementation of the Service Layer is not ready yet. Sometimes it is useful to replace the Service with a fake data generator. In general, the Service Layer will be of little use if it is not possible to substitute it (or to substitute the dependencies used by it).

Widespread problem of Django applications

A common mistake is to use the django.db.models.Manager class (and even django.db.models.Model) as a Service Layer. Often you can see how some method of the class django.db.models.Model takes as an argument the HTTP-request object

django.http.request.HttpRequest, for example, to check the permissions.

The HTTP request object is the Application Layer logic, while the model class is the logic of the Domain Layer, i.e. objects of the real world, which are also called business rules. Checking permissions is also the logic of Application Layer.

The lower layer should not be aware of the higher layer. Domain-level logic should not be aware of application-level logic.

The class django.db.models.Manager corresponds most closely to the class Finder described in "Patterns of Enterprise Application Architecture" [3].

> *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.*
>
> *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.*
>
> *- Chapter 10. "Data Source Architectural Patterns : Row Data Gateway", "Patterns of Enterprise Application Architecture" [3]*

Although Django does not use the Repository pattern, it uses an abstraction of the selection criteria in the form similar to the Query Object pattern. Like the Repository pattern, the model class (ActiveRecord) limits its interface using the Query Object interface. Clients should use the provided interface, rather than impose their responsibilities on the Model and its Manager on knowledge of their queries. And since class does not have to make assumptions about its clients, it is impossible to accumulate pre-defined queries in the Model class, because it can not be aware about the all needs of all clients. Clients should take care of themselves. But the Service Layer was created for client service. Therefore, it's a responsibility of the Service Layer.

Attempts to exclude the Serving Layer from Django applications leads to the appearance of Managers with a lot of methods.

A good practice would be to hide the implementation (in the form of ActiveRecord) of Django models by the Service Layer. This will allow painless ORM replace if necessary.

> *Some might also argue that the application logic responsibilities could be implemented in domain object methods, such as Contract.calculateRevenueRecognitions(), or even in the data source layer, thereby eliminating the need for a separate Service Layer. However, I find those allocations of responsibility undesirable for a number of reasons. First, domain object classes are less reusable across applications if they implement application-specific logic (and depend on application-specific Gateways (466), and the like). They should model the parts of the problem domain that are of interest to the application, which doesn't mean all of application's use case responsibilities. Second, encapsulating application logic in a "higher" layer dedicated to that purpose (which the data source layer isn't) facilitates changing the implementation of that layer perhaps to use a workflow engine.*
>
> - *"Patterns of Enterprise Application Architecture"* [3]

Problems of Django annotation

I often observed the problem when a new field was added to the Django Model, and multiple problems started to occur, since this name was already used either with the annotation interface or with Raw-SQL. Also, the implementation of annotations by Django ORM makes it impossible to use the pattern Identity Map. Storm ORM / SQLAlchemy implement annotations more successfully. If you still had to work with Django Model, refrain from using Django annotation mechanism in favor of bare pattern DataMapper.

Peculiar properties of Service Layer on client side

Using the Aggregate concept and reactive programming libraries, such as RxJS, allows us to implement Service Layer using a simplest pattern like Gateway, see, for example, the tutorial of Angular documentation. In this case, Query Object is usually implemented as a simple dictionary, which is then converted to a list of GET parameters for the URL. Such service usually communicates with a server either through JSON-RPC, or through REST-API Actions.

Everything works well until you need to express prioritized queries, for example, using the logical operator "OR" which has a lower priority than the logical operator "AND". This raises the question of who should be responsible for building the query, the Service Layer of the client or the Service Layer of the server?

On the one hand, the server should not make assumptions about its clients, and must limit its interface through the interface Query Object. But this dramatically increases the level of complexity of the client, in particular, the implementation of Service Stub. To facilitate implementation, you can use the library rql mentioned in the article "Implementation of Repository pattern for browser's JavaScript".

On the other hand, the Service Layer, albeit a remote call, is designed to serve clients, so it can concentrate the logic of query building. If the client does not contain complex logic, allowing to interpret the prioritized queries for Service Stub, then no need to complicate it. In this case, it's easier to add a new method to the remote call service, and get rid of the need for prioritized queries.

Concurrent update issue

The advent of the Internet has provided access to a huge amount of data that is excessively large with the capabilities of the server. There was a need for scalability and distributed storage and processing of data.

One of the most acute problems is the concurrent update issue.

> *All race conditions, deadlock conditions, and concurrent update problems are due to mutable variables. You cannot have a race condition or a concurrent update problem if no variable is ever updated. You cannot have deadlocks without mutable locks.*
>
> *- "Clean Architecture: A Craftsman's Guide to Software Structure and Design" [2] by Robert C. Martin*

An order is a correct imposition of restrictions.

CQRS

The concurrent update issue can be significantly reduced by restricting bidirectional state changes by introducing unidirectional changes, i.e. by separating reading from writing. This is exactly the approach used by Redux.

> *"it allows us to host the two services differently eg: we can host the read service on 25 servers and the write service on two. The processing of commands and queries is fundamentally asymmetrical, and scaling the services symmetrically does not make a lot of sense."*
>
> - *"CQRS, Task Based UIs, Event Sourcing agh!" by Greg Young*

Organizing Application Logic and Business Logic is well covered in the article "Clarified CQRS" by Udi Dahan.

The use of CQRS facilitates the use of Functional Programming paradigm.

> *For both theoretical and practical reasons detailed elsewhere [10], the command-query separation principle is a methodological rule, not a language feature, but all serious software developed in Eiffel observes it scrupulously, to great referential transparency advantage. Although other schools of object-oriented programming regrettable do not apply it (continuing instead the C style of calling functions rather than procedures to achieve changes), but in my view it is a key element of the object-oriented approach. It seems like a viable way to obtain the referential transparency goal of functional programming – since expressions, which only involve queries, will not change the state, and hence can be understood as in traditional mathematics or a functional language – while acknowledging, through the notion of command, the fundamental role of the concept of state in modeling systems and computations.*
>
> - *"Software architecture: object-oriented vs functional" by Bertrand Meyer*

Functional Programming inherently cannot produce side effects (since Functional Programming imposes a restriction on assignment (mutability)). This is the reason for its popularity growth in the era of distributed computing. No mutability - no concurrent update issues.

It is necessary to distinguish the Functional Programming paradigm from the languages supporting this paradigm, since quite often the languages supporting this paradigm allow not to follow it.

However, despite the new opportunities to use Functional Programming in code, the data storage itself (IO device) is still prone to the concurrent update issues, since it has mutable rows, and, therefore, has a side effect.

The solution to this problem is usually to replace CRUD (Create, Read, Update, Delete) with CR, i.e. imposing restrictions on Update and Delete rows in the storage, that is widespread with the name Event Sourcing. There are specialized storages that implement Event Sourcing, but it can also be implemented without specialized tools.

Event Sourcing

If CQRS allows working with data storages in the Imperative style, and separates a command (side effect) from a query (reading) data, then Event Sourcing goes even further and imposes a restriction on changing and deleting data, turning CRUD into CR. This pattern allows working with data storages in the Functional style, and provides the same benefits: no mutable state - no concurrent update issues. And the same disadvantages - high memory and processor consumption. This is the reason why this pattern is widely used in distributed systems, where there is an acute need for its advantages, and at the same time, its weaknesses do not appear (since distributed systems are not limited either in memory or in processor power).

A good example of Event Sourcing can be the principle of organizing a bank account in a database, when the account is not a source of truth, but simply reflects the aggregate value of all transactions (i.e. events).

This topic is well covered in Chapter 6 “Functional Programming” of “Clean Architecture” by Robert C. Martin.

> *More importantly, nothing ever gets deleted or updated from such a data store. As a consequence, our applications are not CRUD; they are just CR. Also, because neither updates nor deletions occur in the data store, there cannot be any concurrent update issues.*
>
> *If we have enough storage and enough processor power, we can make our applications entirely immutable–and, therefore,* ***entirely functional****.*
>
> *If this still sounds absurd, it might help if you remembered that this is precisely the way your source code control system works.*
>
> *- “Clean Architecture: A Craftsman’s Guide to Software Structure and Design” [2] by Robert C. Martin*

> *Event Sourcing is naturally functional.* *It's an append only log of facts that have happened in the past. You can say that any projection any state is a left fold over your previous history.*
>
> - Greg Young, "A Decade of DDD, CQRS, Event Sourcing" at 16:44

> *It's actually functional.*
>
> - Greg Young, "Event Sourcing is actually just functional code" at 34:49

> *I have always said that Event Sourcing is "Functional Data Storage". In this talk we will try migrating to a idiomatic functional way of looking at Event Sourcing. Come and watch all the code disappear! By the time you leave you will never want an "Event Sourcing Framework (TM)" ever again!*
>
> - Greg Young, "Functional Data", NDC Conferences

Further Reading

- "Clean Code: A Handbook of Agile Software Craftsmanship" by Robert C. Martin [1], chapters:
    - Dependency Injection ... 157
    - Cross-Cutting Concerns ... 160
    - Java Proxies ... 161
    - Pure Java AOP Frameworks ... 163
- "Clean Architecture: A Craftsman's Guide to Software Structure and Design" [2] by Robert C. Martin
    - Chapter 6 Functional Programming : Event Sourcing

    - Chapter 16 Independence
    - Chapter 18 Boundary Anatomy : Services
    - Chapter 20 Business Rules
    - Chapter 22 The Clean Architecture
    - Chapter 34 The Missing Chapter

- "Patterns of Enterprise Application Architecture" by Martin Fowler [3], chapters:
    - Part 1. The Narratives : Chapter 2. Organizing Domain Logic : Service Layer
    - Part 1. The Narratives : Chapter 8. Putting It All Together
    - Part 2. The Patterns : Chapter 9. Domain Logic Patterns : Service Layer

- "Domain-Driven Design: Tackling Complexity in the Heart of Software" by Eric Evans [4], chapters:
    - Part II: The Building Blocks of a Model-Driven Design : Chapter Four. Isolating the Domain : Layered Architecture
    - Part II: The Building Blocks of a Model-Driven Design : Chapter Five. A Model Expressed in Software : Services

- "Implementing Domain-Driven Design" [5] by Vaughn Vernon
    - Chapter 4 Architecture : Command-Query Responsibility Segregation, or CQRS
    - Chapter 4 Architecture : Event-Driven Architecture : Long-Running Processes, aka Sagas
    - Chapter 4 Architecture : Event-Driven Architecture : Event Sourcing
    - Chapter 7 Services
    - Chapter 14 Application : Application Services
    - Appendix A Aggregates and Event Sourcing: A+ES : Inside an Application Service

- "Microsoft Application Architecture Guide" 2nd Edition (Patterns & Practices) by Microsoft Corporation (J.D. Meier, David Hill, Alex Homer, Jason Taylor, Prashant Bansode, Lonnie Wall, Rob Boucher Jr., Akshay Bogawat), chapters:
    - Chapter 5: Layered Application Guidelines ... 55
    - Chapter 5: Layered Application Guidelines : Services and Layers ... 58
    - Chapter 9: Service Layer Guidelines ... 115
    - Chapter 17: Crosscutting Concerns ... 205
    - Chapter 21: Designing Web Applications : Service Layer ... 288
    - Chapter 25: Designing Service Applications : Service Layer ... 371

- "Microsoft .NET: Architecting Applications for the Enterprise" 2nd Edition by Dino Esposito, Andrea Saltarello, chapters:
    - Chapter 5 Discovering the domain architecture : The layered architecture ... 129

    - Chapter 10 Introducing CQRS ... 255
    - Chapter 11 Implementing CQRS ... 291
    - Chapter 12 Introducing event sourcing ... 311
    - Chapter 13 Implementing event sourcing ... 325

- "Design Patterns Elements of Reusable Object-Oriented Software" by Erich Gamma [6], chapters:
    - Design Pattern Catalog : 4 Structural Patterns : Adapter ... 139
    - Design Pattern Catalog : 4 Structural Patterns : Decorator ... 175

- "Building Microservices. Designing Fine-Grained Systems" by Sam Newman, chapters:
    - Chapter 3 How to Model Services : Premature Decomposition ... 33

- "Cloud Design Patterns. Prescriptive architecture guidance for cloud applications" by Alex Homer, John Sharp, Larry Brader, Masashi Narumoto, Trent Swanson, chapters:
    - Command and Query Responsibility Segregation (CQRS) pattern
    - Event Sourcing pattern
    - Compensating Transaction pattern

- ".NET Microservices: Architecture for Containerized .NET Applications" edition v2.2.1 (mirror) by Cesar de la Torre, Bill Wagner, Mike Rousos, chapters:
    - Tackle Business Complexity in a Microservice with DDD and CQRS Patterns
    - Apply simplified CQRS and DDD patterns in a microservice
    - Apply CQRS and CQS approaches in a DDD microservice in eShopOnContainers
    - Implement reads/queries in a CQRS microservice

- "CQRS Journey" by Dominic Betts, Julián Domínguez, Grigori Melnik, Fernando Simonazzi, Mani Subramanian, chapters:
    - Reference 1: CQRS in Context
    - Reference 2: Introducing the Command Query Responsibility Segregation Pattern
    - Reference 3: Introducing Event Sourcing
    - Reference 4: A CQRS and ES Deep Dive
    - Reference 6: A Saga on Sagas

- "Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions" by Gregor Hohpe, Bobby Woolf, chapters:

    - 7. Message routing : Process manager ... 278
- "Microservices Patterns: With examples in Java" 1st Edition by Chris Richardson
    - Pattern: Command Query Responsibility Segregation (CQRS)
    - Pattern: Event sourcing
    - Pattern: Saga
- "CQRS"
- "Command Query Separation
- "Event Sourcing"
- "What do you mean by "Event-Driven"?"
- "Patterns for Accounting"
- "CQRS, Task Based UIs, Event Sourcing agh!" by Greg Young
- "Clarified CQRS" by Udi Dahan
- "CQRS Documents" by Greg Young
- "Sagas" by Hector Garcia-Molina and Kenneth Salem

Эта статья на Русском языке "Проектирование Сервисного Слоя и Логики Приложения".

Footnotes

[1]	(1, 2, 3, 4, 5) "Clean Code: A Handbook of Agile Software Craftsmanship" by Robert C. Martin
[2]	(1, 2, 3, 4, 5, 6) "Clean Architecture: A Craftsman's Guide to Software Structure and Design" by Robert C. Martin
[3]	(1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19) "Patterns of Enterprise Application Architecture" by Martin Fowler, David Rice, Matthew Foemmel, Edward Hieatt, Robert Mee, Randy Stafford
[4]	(1, 2, 3, 4, 5, 6, 7, 8, 9, 10) "Domain-Driven Design: Tackling Complexity in the Heart of Software" by Eric Evans

[5] “Implementing Domain-Driven Design” by Vaughn Vernon

[6] (1, 2) “Design Patterns Elements of Reusable Object-Oriented Software” by Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides, 1994

[7] (1, 2) “Refactoring: Improving the Design of Existing Code” by Martin Fowler, Kent Beck, John Brant, William Opdyke, Don Roberts

[8] “Building Microservices. Designing Fine-Grained Systems” by Sam Newman

[9] “.NET Microservices: Architecture for Containerized .NET Applications” edition v2.2.1 (mirror) by Cesar de la Torre, Bill Wagner, Mike Rousos

Updated on Oct 12, 2019:

How to quickly develop high-quality code. Team work.

Проектирование Сервисного Слоя и Логики Приложения

Comments