Contents

A Stereotypical Architecture

Before moving into architectures for Domain Driven Design based projects it is important to start off by analyzing what is generally considered to be the standard architecture that many try to apply to projects. We can from that point attempt to improve upon the stereotypical architecture in small rational steps while trying to minimize the cost in terms of productivity for each step towards a better architecture.

Below is shown a diagram of a stereotypical architecture.

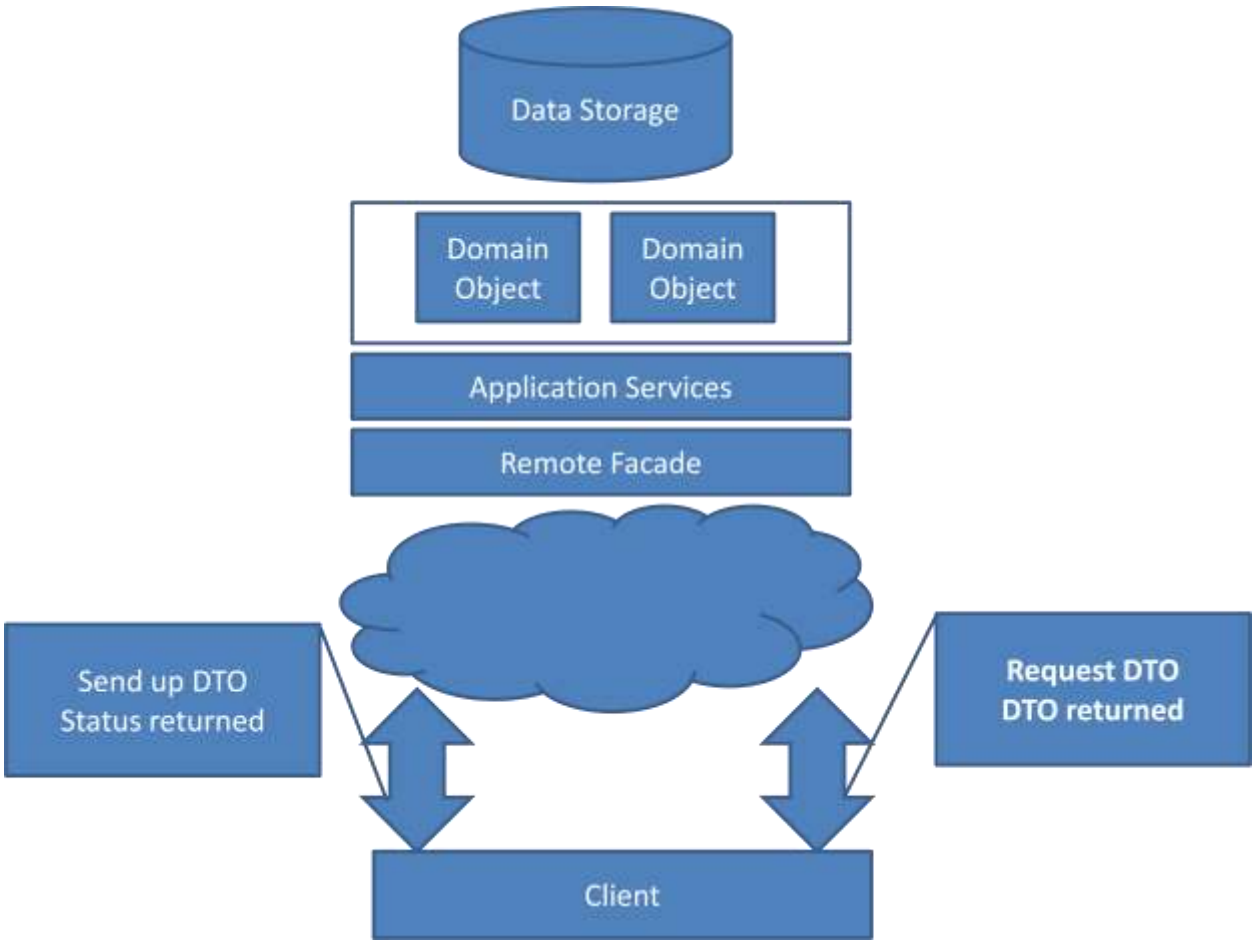

Figure 1 A Stereotypical Architecture

Application Server

The above architecture is centered upon a backing data storage system. This system although typically a RDBMS does not have to be, it could just as easily be a key/value store, and object database, or even plain XML files. The important aspect of the backing store is that it is representing the current state of objects in the domain.

Above the backing data storage lies an Application Server. An area of logic, labeled as the domain in Figure 1 contains the business logic of the system. In this area validation and orchestration logic exists for the processing of requests given to the Application Server.

It is important to note that although Figure 1 is drawn without a data tier one could place a data tier in between the Application Server and the Data Storage. It is also important to note that a "domain" is not necessary to achieve this architecture, one could also use other patterns such as Table Module or Transaction Script. With these only existing as Application Services.

Abstracting the "domain" one will find a facade known as Application Services. Application Services provide a simple interface to the domain and underlying data, they also limit coupling between the consumers of the domain and the domain itself.

On the outside of the Application Server sits some type of Remote Facade. This could be many things such as SOAP, custom TCP/IP, XML over HTTP, TomCat, or even a person manually typing messages that arrive tied to the legs of pigeons. The Remote Facade may or may not be abstracted away from its underlying technology mechanism depending on the situation and tools that are involved.

The overall usage of an Application Server to abstract away the data storage of a system and to provide a centralized location of business logic has become very popular over the years and at the time of this writing is in many circumstances considered to be the default architecture applied to many systems.

Client Interaction

Interacting with the Application Server there is a / are many client(s). The general interaction of the clients can be seen in Figure 2.

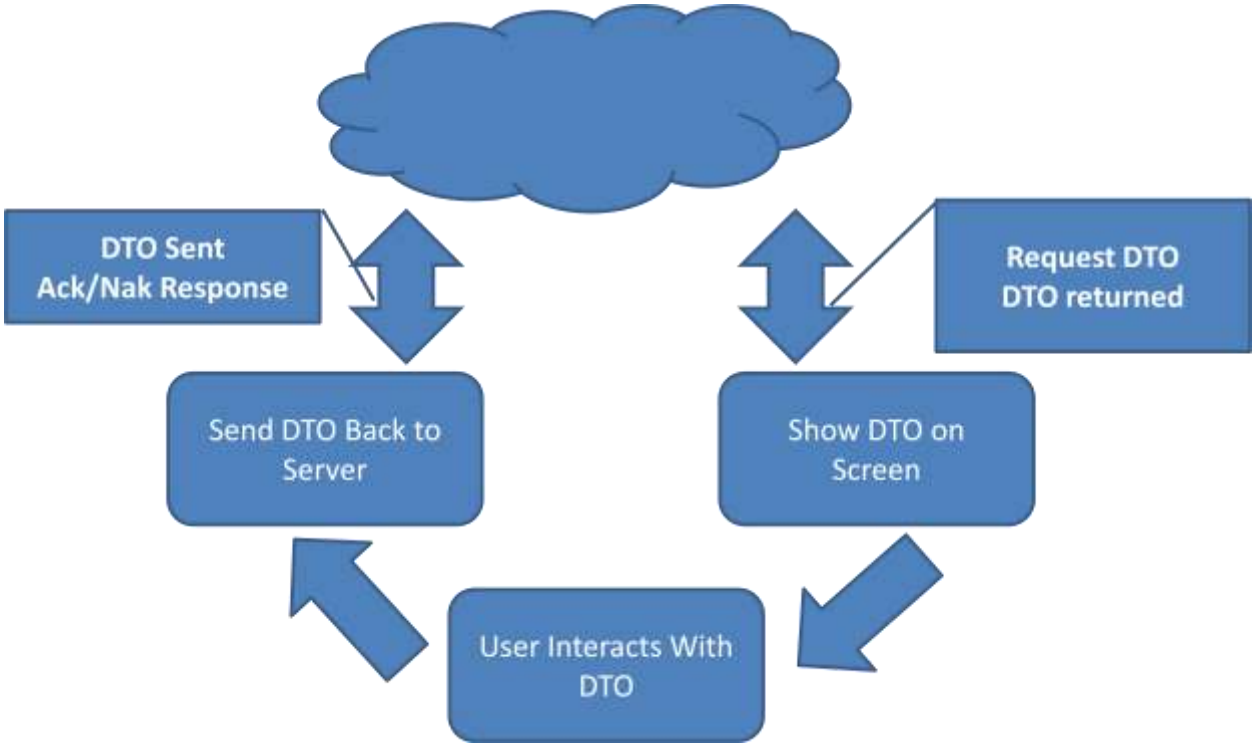

Figure 2 Typical Client Interaction

The basic interaction of the client can be described as a DTO (Data Transfer Object) up/down interaction. Going through the lifecycle of an operation is the easiest way to show the functioning of the API. A user goes to a screen, perhaps to edit a customer. The client sends a request to the remote facade for a DTO representing Customer #id. The Remote facade loads up the domain objects required, and maps the domain objects to a DTO that is then returned to the client. An example of DTO in XML format can be seen in Figure 3 but the basic explanation is that the DTO in this stereotypical architecture contains the current state of the object in questions.

The client will then display the information received from the Remote Facade on the screen allowing the user to interact with it. This is very often done utilizing a view model and/or data binding with the view.

At some point the user will be complete with the editing of the data on the screen and will through some action cause the UI to “Save” the data. Generally this is implemented through a “Save Button” although some User Interfaces will instead just have you leave the current data which forces a save.

```
<Contact id=”1234”>
  <Name>Greg Young</Name>
  <Address>
    <Street>111 Some St.</Street>
    <City>Vernon</City>
    <State>CT</State>
    <Zip>06066</Zip>
    <Country>USA</Country>
  </Address>
</Contact>
```

Figure 3 Example in XML of a DTO

The processing of a Save on the client will take the data that has been edited by the user on the screen, pack it back into a DTO (usually identical to the DTO it requested from the Remote Façade for displaying to the user). It will then send this DTO back up to the Application Server.

The Application Server receiving the DTO will then start a transaction/session, map the DTO back to the domain objects, allow the domain objects to verify any changes, then save the changes within the domain objects back to the data storage likely through the use of something like an Object Relational Mapper that has the ability to distinguish what has changed in the domain objects and update the data storage accordingly. The Application Server will return to the client either an Acknowledgement (Ack) that the change has been persisted or it will return a series of errors as to why it was unable to persist the changes.

Analysis of the Stereotypical Architecture

The architecture provided above as with any architecture has many properties. Some of these properties are good under certain scenarios and other properties can be extremely bad in others. As architects we should really be trying to align many of these properties to best fit our needs.

Simplicity

When looking at properties it is important to note what the most likely property is for a given architecture becoming popular. In the architecture above the most likely property defining its popularity is that it is simple. One could teach a Junior developer how to interact with a system built using this architecture in a very short period of time. Going along with the simplicity, the architecture is completely generic. One could use this architecture on every project. Along with being able to use it on every project, because many people are doing it, its likely that if a team brings on a new member the new member will be intimately familiar with the general architecture of their system again lowering on ramp up costs.

The combination of these two items allows teams to become extremely adept at applying this architecture and more important it allows them to use it as a default architecture. The thought process

of needing to align non-functional requirements really goes away as they know that this architecture will be "good enough" for 80% of the projects that they run into.

Tooling

Many frameworks exist for the optimization of time required to create systems utilizing the architecture provided above. ORM's are the largest single example as they provide valuable services such as change tracking and transaction management with complex object graphs. Other examples would include the automapping frameworks that map from the domain objects to DTOs on both sides resulting in largely removing the amount of "plumbing code" required to map the DTOs back and forth in the Application Server.

Of course there are however also many not-so-good things associated with the architecture provided above. It being that this document is associated with Domain Driven Design the single most important of the not-so-good properties would be that **it is impossible to apply Domain Driven Design with the architecture provided.**

Domain Driven Design

The application of Domain Driven Design is not possible in the above architecture though many who are "practicing" Domain Driven Design use this architecture. The reasoning for why it is impossible can easily be seen when one looks at how the Ubiquitous Language is represented by the object model.

In the architecture above there are only four verbs (and of course synonyms for those four such as edit instead of update). The four verbs are Create, Read, Update, and Delete or CRUD as they are commonly known in the industry. Because the Remote Façade has a data oriented interface the Application Services must necessarily have the same interface.

This means that there are no other verbs within the domain. When however one talks with domain experts in an effort to refine an Ubiquitous Language, it is extremely rare that one ends up with a language that is focused on these four verbs.

There is a related well-known anti-pattern of domain modeling known as an "Anemic Model".

"The basic symptom of an Anemic Domain Model is that at first blush it looks like the real thing. There are objects, many named after the nouns in the domain space, and these objects are connected with the rich relationships and structure that true domain models have. The catch comes when you look at the behavior, and you realize that there is hardly any behavior on these objects, making them little more than bags of getters and setters. Indeed these models often come with design rules that say that you are not to put any domain logic into the domain objects. Instead there are a set of service objects which capture all of the domain logic. These services live on top of the domain model and use the domain model for data" (Fowler, 2003)

The model that is being built in this architecture sounds at first to be an anemic domain model. Because the Application Services map data back and forth to DTO's the domain objects have little behavior and

are littered with getters and setters to be used in the mapping process. There is a structure to the domain showing how objects relate with one another but …

One cannot even create and Anemic Domain Model with this architecture as then all of the business logic would be in services, here the services themselves are really just mapping DTO's to domain objects, there is no actual business logic in them. In this case a large amount of business logic is not existing in the domain at all, nor in the Application Server, it may exist on the client but more likely it exists on either pieces of paper in a manual or in the heads of the people using the system.

Architectures like the one being viewed tend to come with instructions of how to complete complex tasks by editing data in many parts of the system. A stereotypical example of this would be when changing the sex of an employee you must after go edit their health insurance information. This is far worse than the creation of an anemic model, this is the creation of a glorified excel spreadsheet.

Scaling

When one looks at the architecture provided above in the context of scaling one will quickly notice that there is a large bottle neck. The bottleneck in terms of scaling is the data storage. When using a RDBMS as 90%+ currently use this becomes even more of a problem most RDBMS are at this point not horizontally scalable and vertically scaling becomes prohibitively expensive very quickly. It is however also extremely important to remember that most systems do not **need** to scale and as such scalability is really not a grave issue in all cases.

Summary

The DTO up/down architecture employed on many projects is capable of being used for many applications and can offer many benefits in terms of simplicity for teams to work with. It cannot however be used with a Domain Driven Design based project, to attempt so will bring failure to your efforts at applying Domain Driven Design.

This architecture does however make a good baseline and the rest of this document will be focused on improving this architecture in incremental steps while attempting to limit or remove cost while adding business value at each additional step.

Works Cited

Fowler, M. (2003, 11 25). *MF Bliki: AnemicDomainModel.* Retrieved from Bliki: http://martinfowler.com/bliki/anemicdomainmodel

Task Based User Interface

This chapter introduces the concept of a Task Based User Interface and compares it with a CRUD style user interface. It also shows the changes that occur within the Application Server when a more task oriented style is applied to it's API.

One of the largest problems seen in "A Stereotypical Architecture" was that the intent of the user was lost. Because the client interacted by posting data-centric DTOs back and forth with the Application Server, the domain was unable to have any verbs in it. The domain had become a glorified abstraction of the data model. There were no behaviors, the behaviors that existed, existed in the client, on pieces of paper, or in the heads of the users of the software.

Many examples of such applications can be cited. Users have "work flow" information documented for them. Go to screen xyz edit foo to bar, then go to this other screen and edit xyz to abc. For many types of systems this type of workflow is fine. These systems are also generally low value in terms of the business. In an area that is sufficiently complex and high enough ROI in order to use Domain Driven Design these types of workflows become unwieldy.

One reason that is commonly cited for wanting to build a system such as described is that "the business logic and work flows can be changed at any time to anything without need of a change to the software". While this may be true it must be asked at what cost. What happens when someone misses a step in the process they have in their head or you have multiple users who do it differently as is commonly the case? How do you get any reasonable information out of a system in terms of reporting?

One way of dealing with this issue is to move away from the DTO up/down architecture that was illustrated in a "Stereotypical Architecture". Figure 1 shows the client interaction side of a DTO up/down architecture.

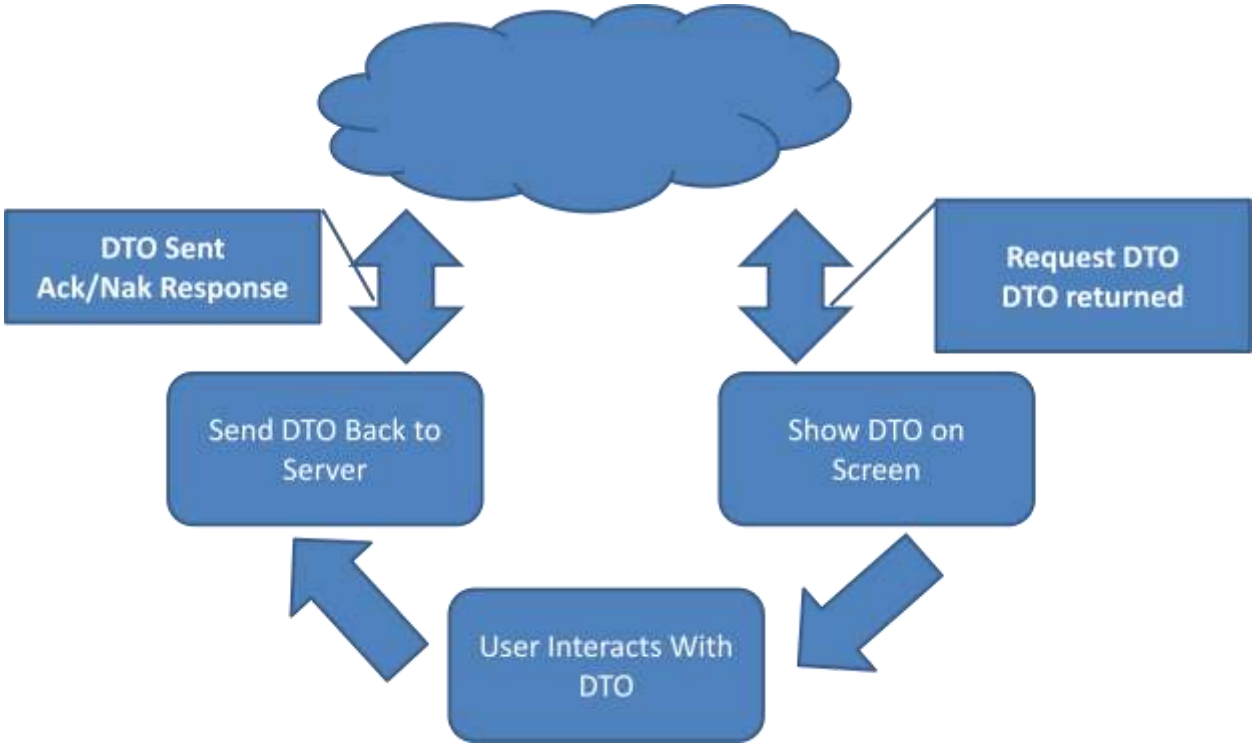

Figure 4 Interaction in a DTO Up/Down Architecture

The basic explanation of the operation is that the UI will request a DTO, say for Customer 1234 from the Application Server. This DTO will be returned to the client and then shown on the screen. The user will interact with the DTO in some way (likely either directly or through a View Model). Eventually the client will click Save or some other trigger will occur and the client will take the DTO and send it back up to the Application Server. The Application Server will then internally map the data back to the domain model and save the changes returning a success or failure.

As discussed the intention of the user is being lost because a DTO is being sent up that just represents the current state of the object after the client's actions are completed. It is possible to bring forward the intention of the user; this will allow the Application Server to process behaviors as opposed to saving data. Figure shows an interaction capturing intent.

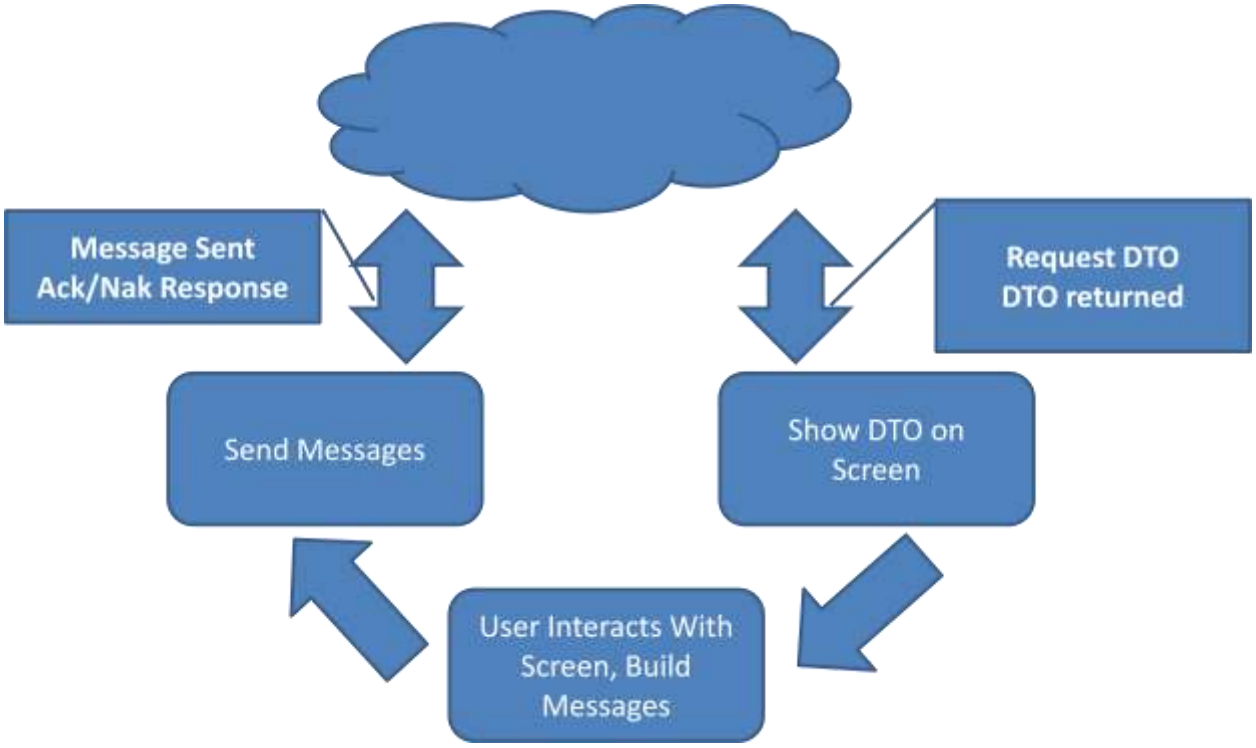

Figure 5 Behavioral Interface

Capturing intent the client interaction is very similar to the DTO up/down methodology in terms of interactions. The client first quests a DTO from the Application Server for instance Customer 1234. The Application Server returns a DTO representing the customer that is then shown on the screen for the user to interact with usually either directly or through a View Model. The similarities however stop at this point.

Instead of simply sending the same DTO back up when the user is completed with their action the client needs to send a message to the Application Server telling it to **do something**. It could be to "Complete a Sale", "Approve a Purchase Order", "Submit a Loan Application". Said simply the client needs to send a message to the Application Server to have it complete the task that the user would like to complete. By telling the Application Server what the user would like to do, it is possible to know the intention of the user.

Commands

The method through which the Application Server will be told what to do is through the use of a Command. A command is a simple object with a name of an operation and the data required to perform that operation. Many think of Commands as being Serializable Method Calls. Listing 1 includes the code of a basic command.

```
public class DeactivateInventoryItemCommand {
   public readonly Guid InventoryItemId;
   public readononly string Comment;

   public DeactivateInventoryItemCommand (Guid id, string comment) {
   InventoryItemId = id;
   Comment = comment;
}
```

Listing 1 A Simple Command

As a side note the example in Listing 1 includes the pattern name after the name of the Command. This is a decision that has many positives and negatives both linguistically and operationally. The choice over whether to use a pattern name in a class name is one that should not be taken lightly by a development team.

One important aspect of Commands is that they are always in the imperative tense; that is they are telling the Application Server to do something. The linguistics with Commands are important. A situation could for with a disconnected client where something has already happened such as a sale and could want to send up a "SaleOccurred" Command object. When analyzing this, is the domain allowed to say no that this thing did not happen? Placing Commands in the imperative tense linguistically shows that the Application Server is allowed to reject the Command, if it were not allowed to, it would be an Event for more information on this see "Events".

Occasionally there exist funny examples of language in English. A perfect example of this would be "Purchase" which can be used either as a verb in the imperative or as a noun describing the result of its usage in the imperative. When dealing with these situations, ensure that the concept being pushed forward represents the imperative of the verb and not the noun. As an example a purchase should be including what to purchase and expecting the domain to possibly fill in some information like when the item was purchased as opposed to sending up a purchase DTO that fully describes the purchase.

The simple Command in Listing 1 includes two data properties. It includes an Id which represents the InventoryItem it will apply to and it includes a comment as to why the item is being deactivated. The comment is quite typical of an attribute associated with a Command, it is a piece of data that is required in order to process the behavior. There should only exist on a Command data points that are required to process the given behavior. This contrasts greatly with the typical architecture where the entire data of the object is passed back to the Application Server.

Most importantly of the data is the Id of the associated inventory item. At least one Id must exist for all commands that are updating state in some way, as all commands are intended to be routed to an object. When issuing a Create Command it is not necessary though valuable to include an Id. Having the client originate Ids normally in the form of UUIDs is extremely valuable in distributed systems.

It is quite common for developers to learn about Commands and to very quickly start creating Commands using vocabulary familiar to them such as “ChangeAddress”, “CreateUser”, or “DeleteClass”. This should be avoided as a default. Instead a team should be focused on what the use case really is.

Is it “ChangeAddress”? Is there a difference between “Correcting an Address” and “Relocating the Customer”? It likely will be if the domain in question is for a telephone company that sends the yellow pages to a customer when they move to a new location.

Is it “CreateUser” or is it “RegisterUser”? “DeleteClass” or “DeregisterStudent”. This process in naming can lead to great amounts of domain insight. To begin defining Commands, the best place to begin is in defining use cases, as generally a Command and a use case align.

It is also important to note that sometimes the only use case that exists for a portion of data is to “create”, “edit”, “update”, “change”, or “delete” it. All applications carry information that is simply supporting information. It is important though to not fall into the trap of mistaking places where there are use cases associated with intent for these CRUD only places.

Commands as a concept are not difficult but are different for many developers. Many developers see the creation of the Commands as a lot of work. If the creation of Commands is a bottleneck in the workflow, many of the ideas being discussed are likely being applied in an incorrect location.

User Interface

In order to build up Commands the User Interface will generally work a bit differently than in a DTO up/down system. Because the UI must build Command objects it needs to be designed in such a way that the user intent can be derived from the actions of the user.

The way to solve this is to lean more towards a “Task Based User Interface” also known as an “Inductive User Interface” in the Microsoft world. This style of UI is not by any means new and offers a quite different perspective on the design of user interfaces. Microsoft identified three major problems with Deductive UIs when researching Inductive UIs.

Users don't seem to construct an adequate mental model of the product*. The interface design for most current software products assumes that users will understand a conceptual model that the designers carefully crafted. Unfortunately, most users don't seem to ever acquire a mental model that is thorough and accurate enough to guide their navigation. These users aren't dumb — they are just very busy and overloaded with information. They do not have the time, energy, or desire to wonder about a conceptual model for their software.*

Even many long-time users never master common procedures. *Designers know that new users may have trouble at first, but expect these problems to vanish as users learn common tasks. Usability data indicates this often doesn't happen. In one study, researchers set up automated equipment to videotape users at home. The tapes showed that users focusing on the task at hand do not necessarily notice the procedure they are following and do not learn from the experience. The next time users perform the same operation, they may stumble through it in exactly the same way.*

***Users must work hard to figure out each feature or screen**. Most software products are designed for (the few) users who understand its conceptual model and have mastered common procedures. For the majority of customers, each feature or procedure is a frustrating, unwanted puzzle. Users might assume these puzzles are an unavoidable cost of using computers, but they would certainly be happier without this burden. (Microsoft Corporation, 2001)*

The basic idea behind a Task Based or Inductive UI is that its important to figure out how the users want to use the software and to make it guide them through those processes.

Many commercial software applications include user interfaces in which a screen presents a set of controls, but leaves it to the user to deduce the page's purpose and how to use the controls to accomplish that purpose. (Microsoft Corporation, 2001)

The goal is to guide the user through the process. An example of the differences can be seen in the DeactivateInventoryItem example previously shown. A typical deductive UI might have an editable data grid containing all of the inventory items. It would have editable fields for various data and perhaps a drop down for the status of the inventory item, deactivated being one of them. In order to deactivate an inventory item the user would have to go to the item in the grid, type in a comment as to why they were deactivating it and then change the drop down to the status of deactivated. A similar example could be where you click to a screen to edit an inventory item but go through the same process as seen in Figure 3.

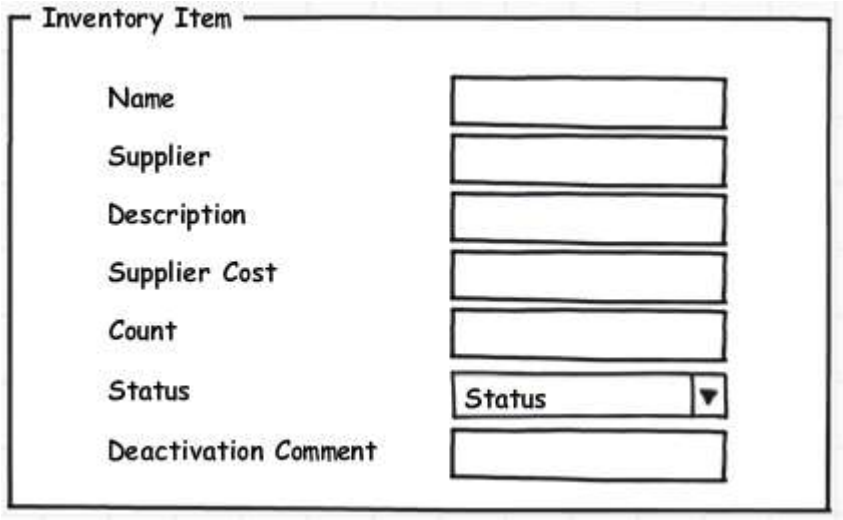

Figure 6 A CRUD screen for an Inventory Item

If the user attempts to submit an item that is “deactivated” and has not entered a comment they will receive an error saying that they must enter a comment as it is a mandatory field for a deactivated item. Some UIs might be a bit more user friendly, they may not show the comment field until the user selects deactivated from the drop down at which point it would appear on the screen. This is far more intuitive to the user as it is a cue that they should be putting data in that field but one can do even better.

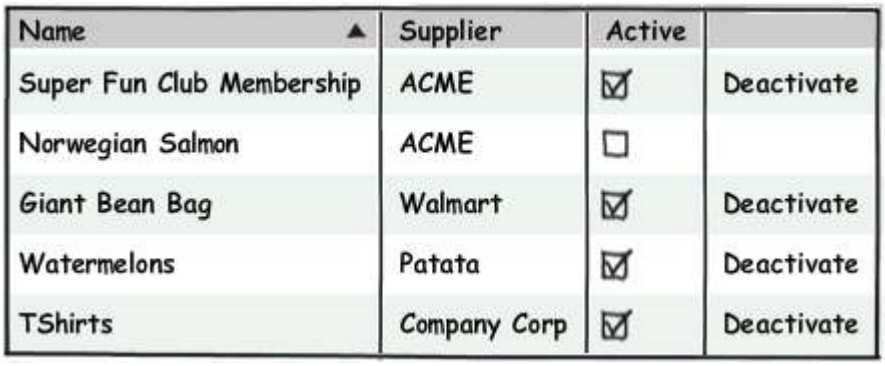

Figure 7 Listing Screen with Link

A Task Based UI would take a different approach, likely it would show a list of inventory items, next to an inventory item there might be a link to "deactivate" the item as seen in Figure 4. This link would take them to a screen that would then ask them for a comment as to why they are deactivating the items which is shown in Figure 5. The intent of the user is clear in this case and the software is guiding them through the process of deactivating an inventory item. It is also very easy to build Commands representing the user's intentions with this style of interface.

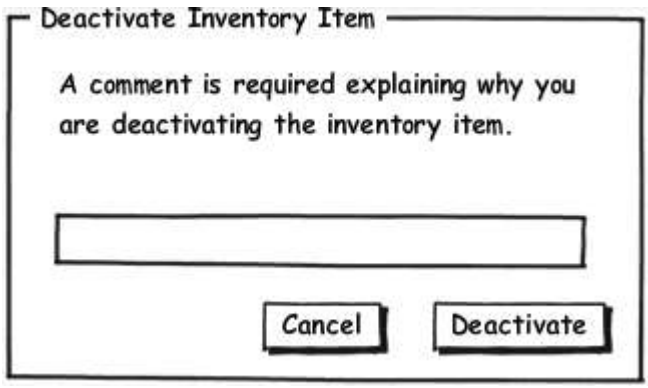

Figure 8 Deactivating an Inventory Item

Web, Mobile, and especially Mac UIs have been trending towards the direction of being task based. The UI guides you through a process and offers you contextually sensitive guidance pushing you in the right direction. This is largely due to the style offering the capability of a much better user experience. **There is a solid focus on how and why the user is using the software; the user's experience becomes an integral part of the process.** Beyond this there is also value on focusing more in general on how the user wants to use the software; this is a great first step in defining some of the verbs of the domain.

Works Cited

Microsoft Corporation. (2001, Feb 9). *Microsoft Inductive User Interface Guidelines.* Retrieved from MSDN: http://msdn.microsoft.com/en-us/library/ms997506

Command and Query Responsibility Segregation

This chapter will introduce the concept of Command and Query Responsibility Segregation. It will look at how the separation of roles in the system can lead towards a much more effective architecture. It will also analyze some of the different architectural properties that exist in systems where CQRS has been applied.

Origins

Command and Query Responsibility Segregation (CQRS) originated with Bertrand Meyer's Command and Query Separation Principle. Wikipedia defines the principle as:

It states that every method should either be a command that performs an action, or a query that returns data to the caller, but not both. In other words, asking a question should not change the answer. More formally, methods should return a value only if they are referentially transparent and hence possess no side effects. (Wikipedia)

Basically it boils down to. If you have a return value you cannot mutate state. If you mutate state your return type must be void. There can be some issues with this. Martin Fowler shows one example on the bliki with:

Meyer likes to use command-query separation absolutely, but there are exceptions. Popping a stack is a good example of a modifier that modifies state. Meyer correctly says that you can avoid having this method, but it is a useful idiom. So I prefer to follow this principle when I can, but I'm prepared to break it to get my pop. (Fowler)

Command and Query Responsibility Segregation was originally considered just to be an extension of this concept. For a long time it was discussed simply as CQS at a higher level. Eventually after much confusion between the two concepts it was correctly deemed to be a different pattern.

Command and Query Responsibility Segregation uses the same definition of Commands and Queries that Meyer used and maintains the viewpoint that they should be pure. The fundamental difference is that in CQRS objects are split into two objects, one containing the Commands one containing the Queries.

The pattern although not very interesting in and of itself becomes extremely interesting when viewed from an architectural point of view.

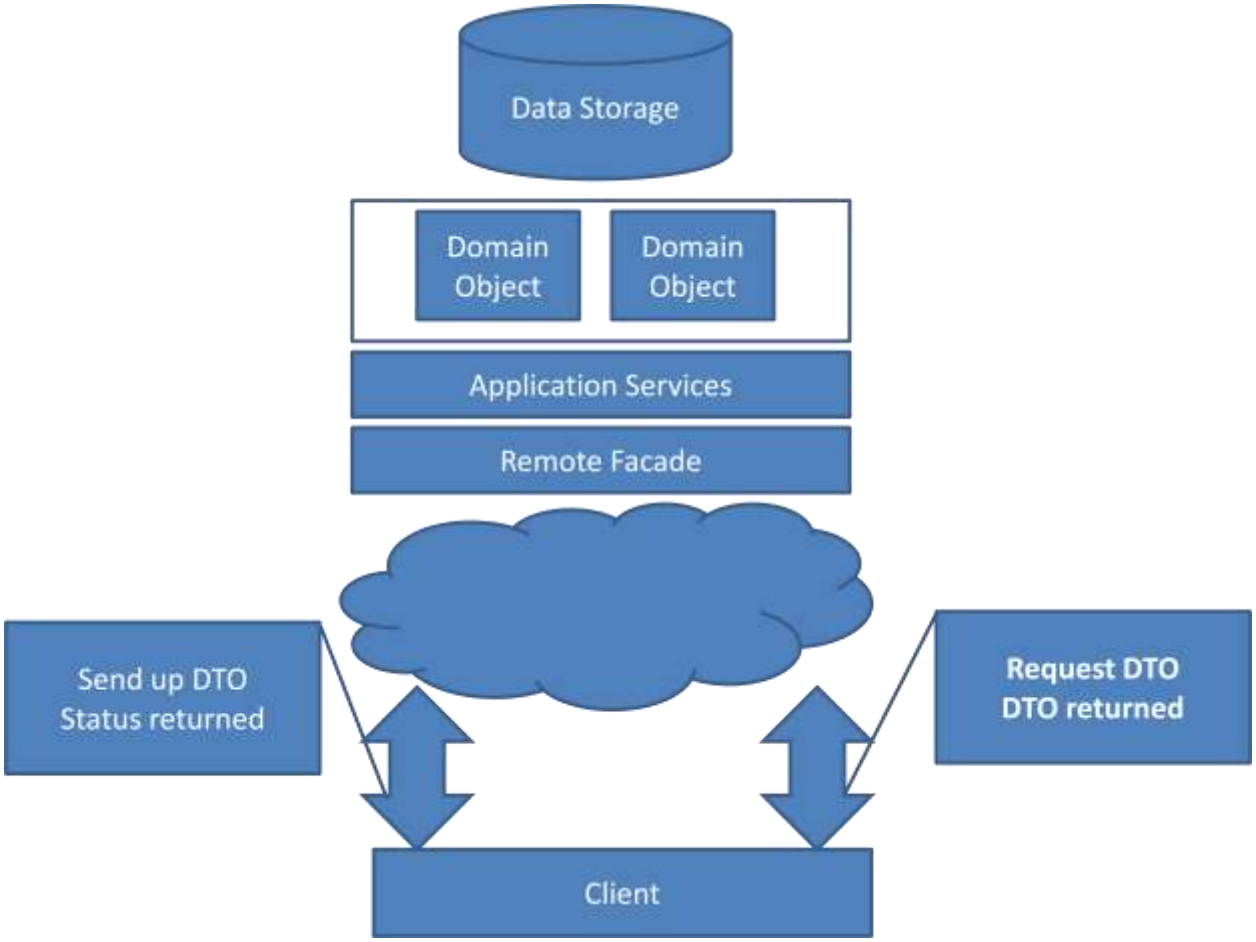

Figure 9 Stereotypical Architecture

Figure 1 contains the stereotypical architecture discussed in the first chapter. One key aspect of the architecture is that the service handles both commands and queries. More often than not the domain is also being used for both commands and queries. The application of CQRS to this architecture although quite simple in definition will drastically change architectural opportunities. A simple service to transform is in Listing 1.

```
CustomerService
void MakeCustomerPreferred(CustomerId)
Customer GetCustomer(CustomerId)
CustomerSet GetCustomersWithName(Name)
CustomerSet GetPreferredCustomers()
void ChangeCustomerLocale(CustomerId, NewLocale)
void CreateCustomer(Customer)
void EditCustomerDetails(CustomerDetails)
```

Listing 2 Original Customer Service

Applying CQRS on the CustomerService would result in two services as shown in Listing 2.

```
CustomerWriteService
void MakeCustomerPreferred(CustomerId)
void ChangeCustomerLocale(CustomerId, NewLocale)
void CreateCustomer(Customer)
void EditCustomerDetails(CustomerDetails)

CustomerReadService
Customer GetCustomer(CustomerId)
CustomerSet GetCustomersWithName(Name)
CustomerSet GetPreferredCustomers()
```

Listing 3 Customer Service after CQRS

While a relatively simple process, this will solve many of the problems that existed in the stereotypical architecture. The service has been split into two separate services, a read side and a write side or the Command side and the Query side.

This separation enforces the notion that the Command side and the Query side have very different needs. The architectural properties associated with use cases on each side are tend to be quite different. Just to name a few:

Consistency

Command: It is far easier to process transactions with consistent data than to handle all of the edge cases that eventual consistency can bring into play.

Query: Most systems can be eventually consistent on the Query side.

Data Storage

Command: The Command side being a transaction processor in a relational structure would want to store data in a normalized way, probably near 3rd Normal Form (3NF)

Query: The Query side would want data in a denormalized way to minimize the number of joins needed to get a given set of data. In a relational structure likely in 1st Normal Form (1NF)

Scalability

Command: In most systems, especially web systems, the Command side generally processes a very small number of transactions as a percentage of the whole. Scalability therefore is not always important.

Query: In most systems, especially web systems, the Query side generally processes a very large number of transactions as a percentage of the whole (often times 2 or more orders of magnitude). Scalabilityis most often needed for the query side.

It is not possible to create an optimal solution for searching, reporting, and processing transactions utilizing a single model.

The Query Side

As stated, the Query side will only contain the methods for getting data. From the original architecture these would be all of the methods that return DTOs that the client consumes to show on the screen.

In the original architecture the building of DTOs was handled by projecting off of domain objects. This process can lead to a lot of pain. A large source of the pain is that the DTOs are a different model than the domain and as such require a mapping.

DTOs are optimally built to match the screens of the client to prevent multiple round trips with the server. In cases with many clients it may be better to build a canonical model that all of the clients use. In either case the DTO model is very different than the domain model that was built in order to represent and process transactions.

Common smells of the problems can be found in many domains.

- Large numbers of read methods on repositories often also including paging or sorting information.
- Getters exposing the internal state of domain objects in order to build DTOs.
- Use of prefetch paths on the read use cases as they require more data to be loaded by the ORM.
- Loading of multiple aggregate roots to build a DTO causes non-optimal querying to the data model. Alternatively aggregate boundaries can be confused because of the DTO building operations

The largest smell though is that the optimization of queries is extremely difficult. Because queries are operating on an object model then being translated to a data model, likely by an ORM it can become difficult to optimize these queries. A developer needs to have intimate knowledge of the ORM and the database. The developer is dealing with a problem of Impedance Mismatch (for more discussion see “Events as a Storage Mechanism”).

After CQRS has been applied there is a natural boundary. Separate paths have been made explicit. It makes a lot of sense now to **not** use the domain to project DTOs. Instead it is possible to introduce a new way of projecting DTOs. Figure

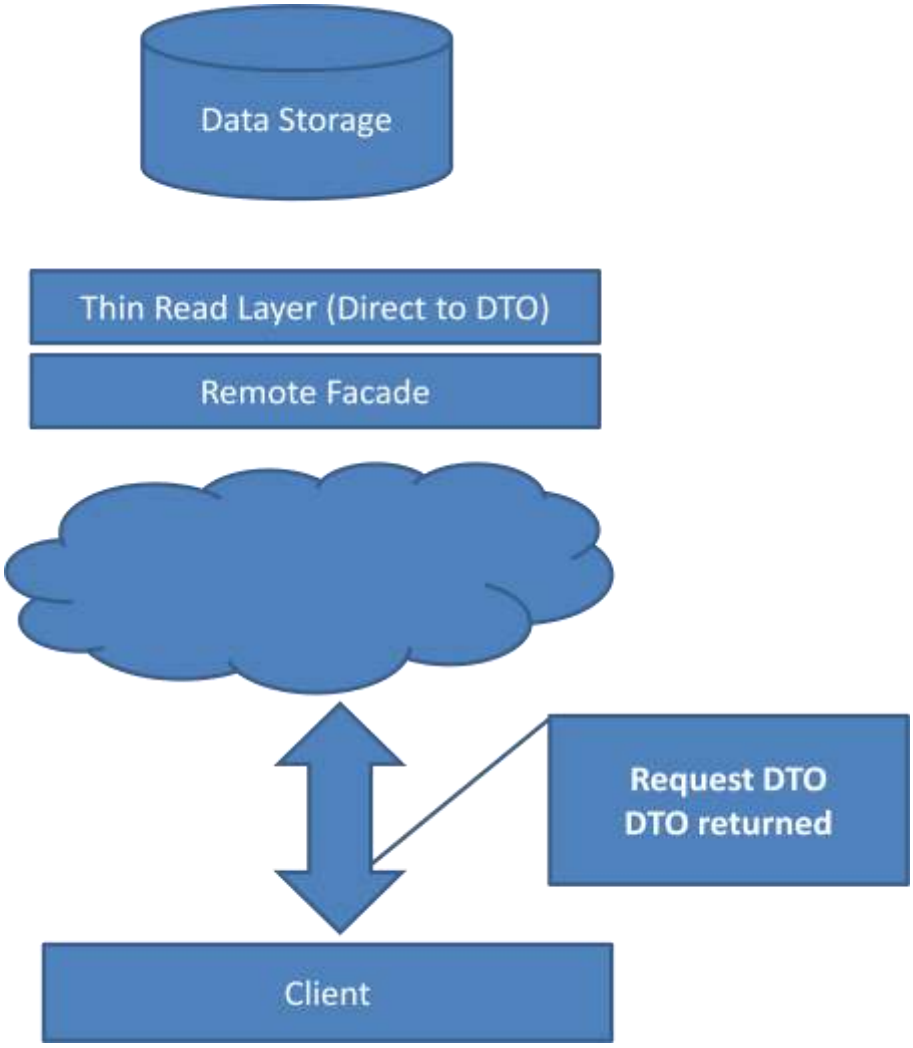

Figure 10 The Query Side

The domain has been bypassed. There is now a new concept called a “Thin Read Layer”. This layer reads directly from the database and projects DTOs. There are many ways that this can be done with handwritten ADO.NET and mapping code and a full blown ORM on the high end. Which choice is right for a team depends largely on the team itself and what they are most comfortable with. Likely the best solution is something in the middle as much of what an ORM provides is not needed and large amounts of time will be lost manually creating mapping code. A possible solution would be to use a small convention based mapping utility.

The Thin Read Layer need not be isolated from the database, it is not necessarily a bad thing to be tied to a database vendor from the read layer. It is also not necessarily bad to use stored procedures for reading, it again depends on the team and the non-functional requirements of the system.

The Thin Read Layer is not a complex piece of code although it can be tedious to maintain. One benefit of the separate read layer is that it will not suffer from an impedance mismatch. It is connected directly to the data model, this can make queries much easier to optimize. Developers working on the Query

side of the system also do not need to understand the domain model nor whatever ORM tool is being used. At the simplest level they would need to understand only the data model.

The separation of the Thin Read Layer and the bypassing of the domain for reads allows also for the specialization of the domain.

The Command Side

Overall the Command side remains very similar to the "Stereotypical Architecture". The illustration in Figure 3 should look nearly identical to the previously discussed architecture. The main differences are that it now has a behavioral as opposed to a data centric contract which was needed in order actually use Domain Driven Design and it has had the reads separated out of it.

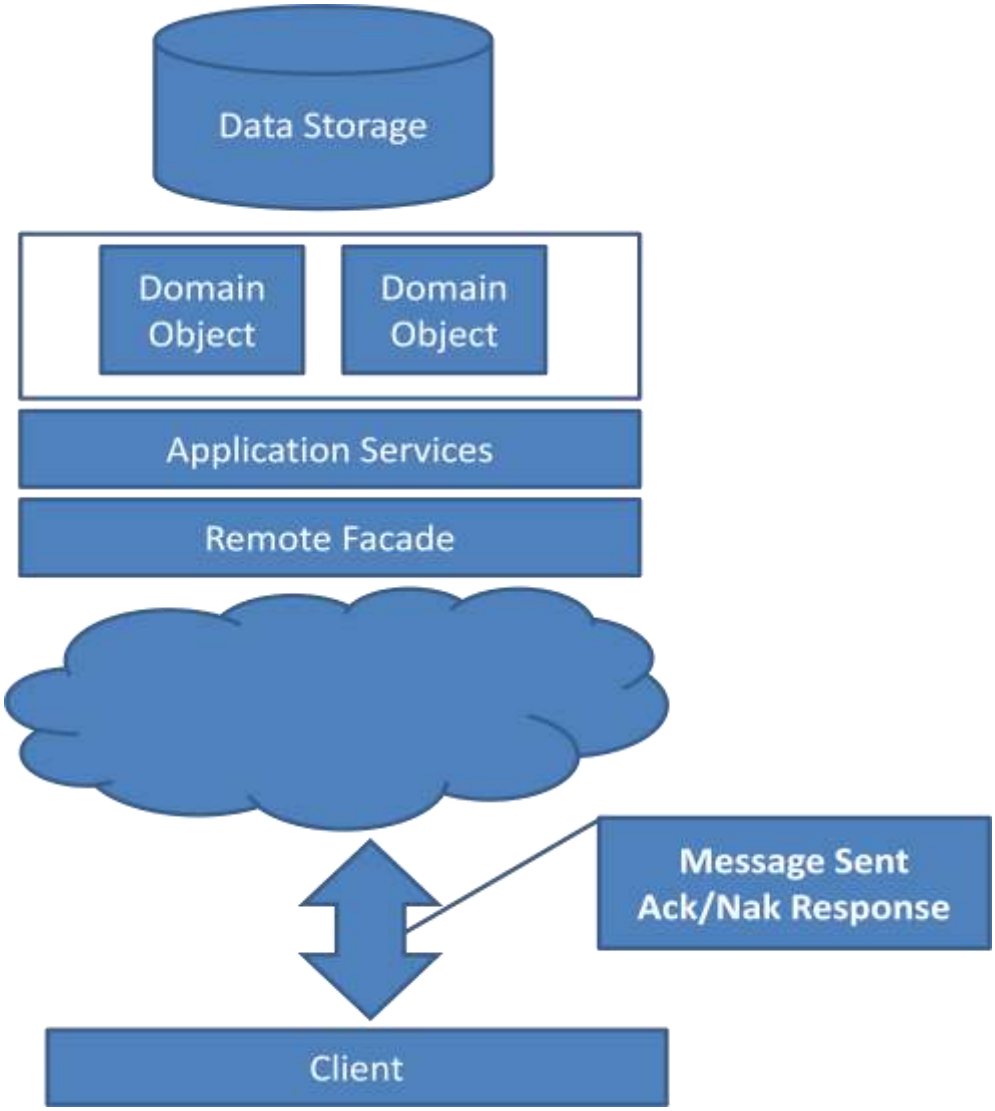

Figure 11 The Command Side

In the “Stereotypical Architecture” the domain was handling both Commands and Queries, this caused many issues within the domain itself. Some of those issues were:

- Large numbers of read methods on repositories often also including paging or sorting information.
- Getters exposing the internal state of domain objects in order to build DTOs.
- Use of prefetch paths on the read use cases as they require more data to be loaded by the ORM.
- Loading of multiple aggregates to build a DTO causes non-optimal querying to the data model. Alternatively aggregate boundaries can be confused because of the DTO building operations

Once the read layer has been separated the domain will only focus on the processing of Commands. These issues also suddenly go away. Domain objects suddenly no longer have a need to expose internal state, repositories have very few if any query methods aside from GetById, and a more behavioral focus can be had on Aggregate boundaries.

This change has been done at a lower or no cost in comparison to the original architecture. In many cases the separation will actually lower costs as the optimization of queries is simpler in the thin read layer than it would be if implemented in the domain model. The architecture also carries lower conceptual overhead when working with the domain model as the querying is separated; this can also lead towards a lower cost. In the worst case, the cost should work out to be equal; all that has really been done is the moving of a responsibility, it is feasible to even have the read side still use the domain.

By applying CQRS the concepts of Reads and Writes have been separated. It really begs the question of whether the two should exist reading the same data model or perhaps they can be treated as if they were two integrated systems, Figure 5 illustrates this concept. There are many well known integration patterns between multiple data sources in order to maintain synchronisity either in a consistent or eventually consistent fashion. The two distinct data sources allow the data models to be optimized to the task at hand. As an example the Read side can be modeled in 1NF and the transactional model could be modeled in 3nf.

The choice of integration model though is very important as translation and synchronization between models can be become a very expensive undertaking. The model that is best suited is the introduction of events, events are a well known integration pattern and offer the best mechanism for model synchronization.

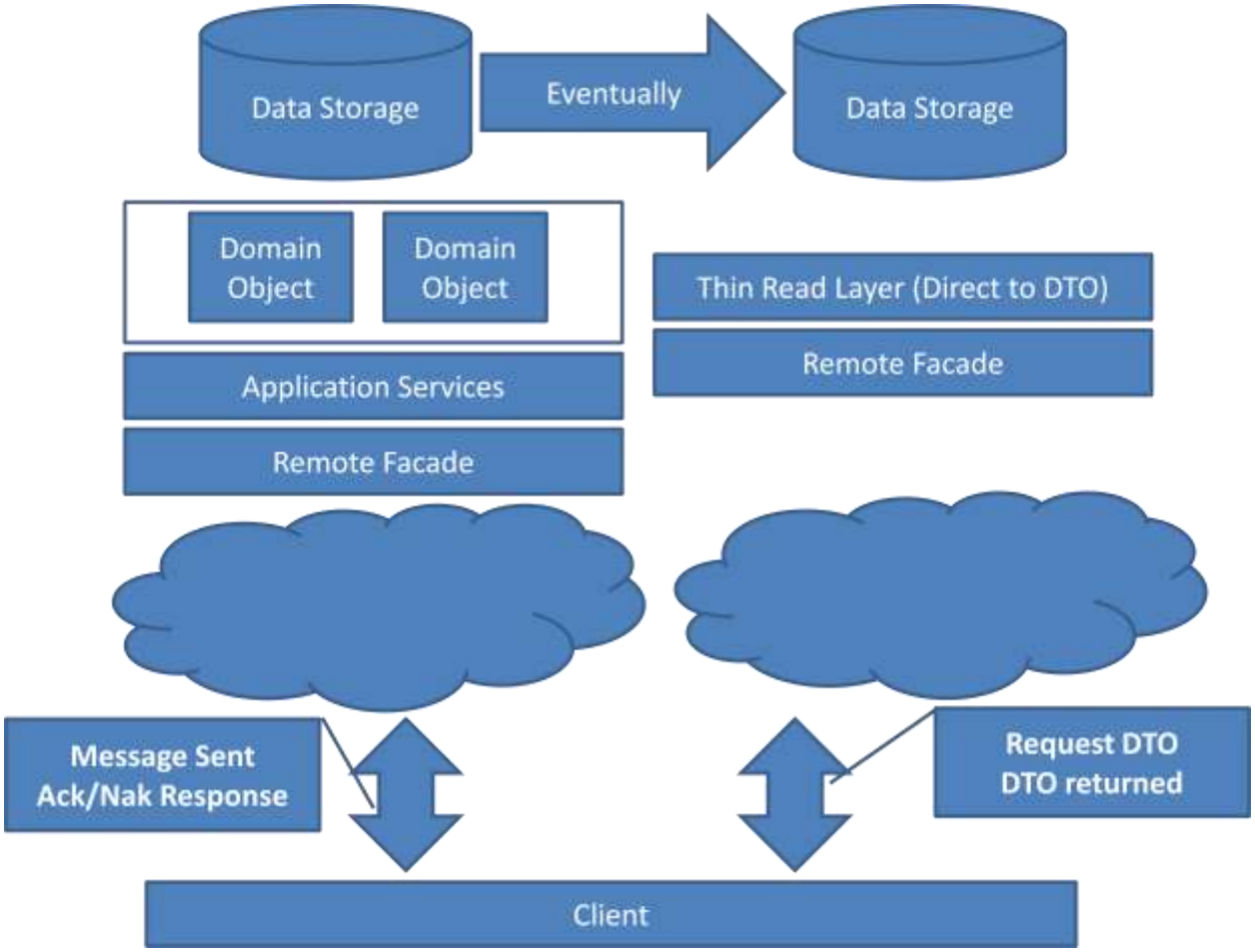

Figure 12 Separated Data Models with CQRS

Events as a Storage Mechanism

Most systems in production today rely on the storing of current state in order to process transactions. In fact it is rare to meet a developer who has worked on a system that maintains current state in any other way. It has not always been like this.

Before the general acceptance of the RDBMS as the center of the architecture many systems did not store current state. This was especially true in high performance, mission critical, and/or highly secure systems. In fact if we look at the inner workings of a RDBMS we will find that most RDBMSs themselves not actually work by managing current state!

The goal of this section is to introduce the concept of event sourcing, to show the benefits, to show how a simple event storage system can be created utilizing a Relational Database for underlying data management.

What is a Domain Event?

An event is something that has happened in the past.

All events should be represented as verbs in the pas t tense such as CustomerRelocated, CargoShipped, or InventoryLossageRecorded. For those who speak French, it should be in Passé Composé, they are things that have **completed** in the past. There are interesting examples in the English language where it is tempting to use nouns as opposed to verbs in the past tense, an example of this would be "Earthquake" or "Capsize", as a congressman recently worried about Guam, but avoid the temptation to use names like this and stick with the usage of verbs in the past tense when creating Domain Events.

It is absolutely imperative that events always be verbs in the past tense as they are part of the Ubiquitous Language. Consider the differences in the Ubiquitous Language when we discuss the side effects from relocating a customer, the event makes the concept explicit where as previously the changes that would occur within an aggregate or between multiple aggregates were left as an implicit concept that needed to be explored and defined. As an example, in most systems the fact that a side effect occurred is simply found by a tool such as Hibernate or Entity Framework, if there is a change to the side effects of a use case, it is an implicit concept. The introduction of the event makes the concept explicit and part of the Ubiquitous Language; relocating a customer does not just change some stuff, relocating a customer produces a CustomerRelocatedEvent which is explicitly defined within the language.

In terms of code, an event is simply a data holding structure as can be seen in Listing 1.

```
public class InventoryItemDeactivatedEvent {
   public readonly Guid InventoryItemId;
   public readononly string Comment;

   public InventoryItemDeactivatedEvet(Guid id, string comment) {
   InventoryItemId = id;
   Comment = comment;
}
```

Listing 4 A Simple Event

The code listing looks very similar to the code listing that was provided for a Command the main differences exist in terms of significance and intent. Commands have an intent of asking the system to perform an operation where as events are a recording of the action that occurred.

Other Definitions and Discussion

There is a related concept to a Domain Event in this description that is defined in Streamlined Object Modeling (SOM). Many people use the term "Domain Event" In SOM when discussing "The Event Principle"

Model the event of people interacting at a place with a thing with a transaction object. Model a point-in-time interaction as a transaction with a single timestamp; model a time-interval interaction as a transaction with multiple timestamps. (Jill Nicola, 2002ll, p. 23)

Although many people use the terminology of a Domain Event to describe this concept the terminology is not having the same definition as a Domain Event in the context of this document. SOM uses another terminology for the concept that better describes what the object is, a Transaction. The concept of a transaction object is an important one in a domain and absolutely deserves to have a name. An example of such a transaction might be a player swinging a bat, this is an action that occurred at a given point in time and should be modeled as such in the domain, this is not however the same as a Domain Event.

This also differs from Martin Fowler's example of what a Domain Event is.

"Example: I go to Babur's for a meal on Tuesday, and pay by credit card. This might be modeled as an event, whose type is "Make Purchase", whose subject is my credit card, and whose occurred date is Tuesday. If Babur's uses and old manual system and doesn't transmit the transaction until Friday, then the noticed date would be Friday." (Fowler)

Further along

"By funneling inputs of a system into streams of Domain Events you can keep a record of all the inputs to a system. This helps you to organize your processing logic, and also allows you to keep an audit log of the system" (Fowler)

The astute reader may pick up that what Martin is actually describing here is a Command as was discussed previously when discussing Task Based UIs. The language of "Make Purchase" is wrong. A purchase was made. It makes far more sense to introduce a PurchaseMade event. Martin did actually make a purchase at the location, they did actually charge his credit card, and he likely ate and enjoyed his food. All of these things are in the past tense, they have already happened and cannot be undone.

An example such as the sales example given also tends to lead towards a secondary problem when built within a system. The problem is that the domain may be responsible for filling in parts of the event. Consider a system where the sale is processed by the domain itself, how much is the sales tax? Often the domain would be calculating this as part of its calculations. This leads to a dual definition of the event, there is the event as is sent from the client without the sales tax then the domain would receive that and add in the sales tax, it causes the event to have multiple definitions, as well as forcing mutability on some attributes. Dual events can sidestep this issue (one for the client with just what it provides and another for the domain including what it has enriched the event from the client with) but this is basically the command event model and the linguistic problems still exist.

A further example of the linguistic problems involved can be shown in error conditions. How should the domain handle the fact that a client told it to do something that it cannot? This condition can exist for many reasons but let's imagine a simple one of the client simply not having enough information to be able to source the event in a known correct way. Linguistically the command/event separation makes much more sense here as the command arrives in the imperative "Place Sale" while the event is in the past tense "SaleCompleted". It is quite natural for the domain to reject a client attempting to "Place a sale"; it is not natural for the domain to tell the client that something in the past tense no longer happened. Consider the discussion with a domain expert; does the domain have a time machine? Parallel realities are far too complex and costly to model in most business systems.

These are exactly the problems that have led to the separation of the concepts of Commands and Events. This separation makes the language much clearer and although subtle it tends to lead developers towards a clearer understanding of context based solely on the language being used. Dual definitions of a concept force the developer to recognize and distinguish context, this weight can translate into both ramp up time for new developers on a project and another thing a member of the team needs to "remember". Anytime a team member needs to remember something to distinguish context there is a higher probability that it will be overlooked or mistook for another context. Being explicit in the language and avoiding dual definitions helps make things clearer both for domain experts, the developers, and anyone who may be consuming the API.

Events as a Mechanism for Storage

When most people consider storage for an object they tend to think about it in a structural sense. That is when considering how the "sale" discussed above should be stored they think about it as being stored as a "Sale" that has "Line Items" and perhaps some "Shipping Information" associated with it. This is not however the only way that the problem can be conceptualized and other solutions offer different and often interesting architectural properties.

Consider for a moment the creation of a small Order object for a web based sale system. Most developers would envision something similar to what is represented in Figure 1. That is a structural viewpoint of what the Order is. An Order has n Line Items and Shipping Information. Of course this is an overly simplified view of what an Order is but it can be seen that the focus is upon the structure of the order and its parts.

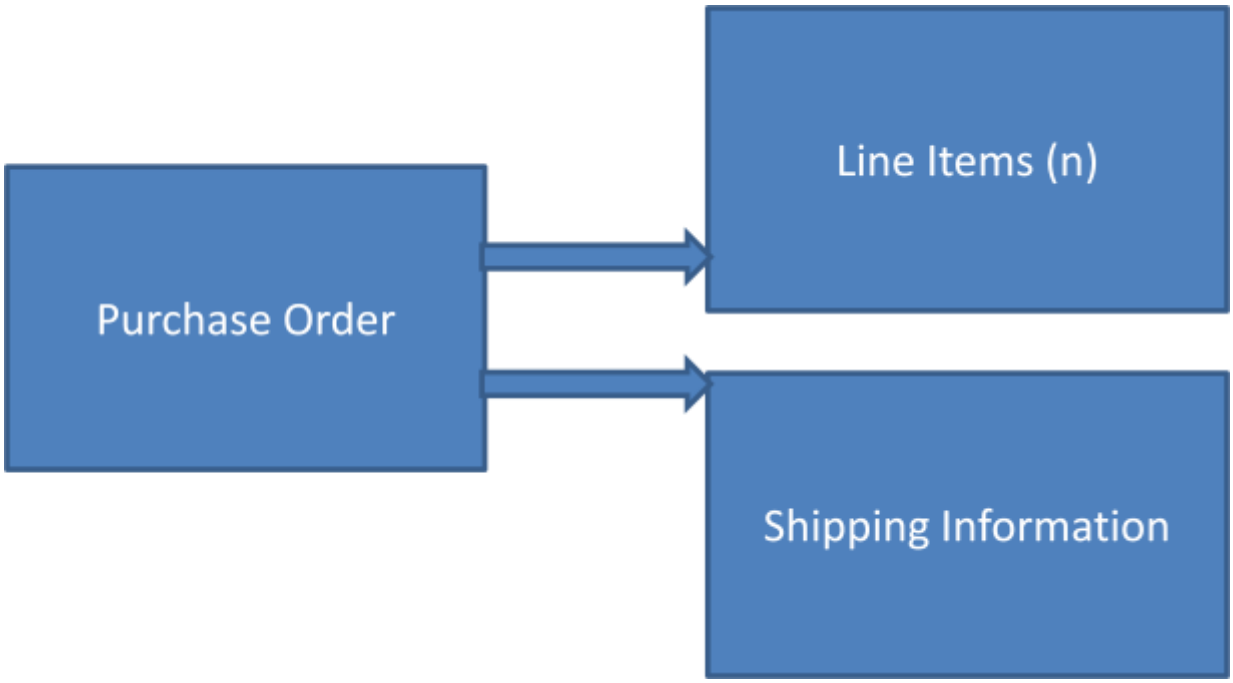

Figure 13 A Structural View of an Order

This is not the only way that this data can be viewed. Previously in the area of discussions there was a discussion about the concept of a transaction. Developers deal with the concept of transactions regularly, they can be viewed as representing the change between a point and the next subsequent point. They are also regularly called “Deltas”. The delta is between two static states can always be defined but more often than not this is left to be an implicit concept, usually relegated to a framework such as Hibernate in the Java world or Entity Framework in the Microsoft world. These frameworks save the original state and then calculate the differences with the new state and update the backing data model accordingly. The making of these deltas explicit can be highly valuable both in terms of technical benefits and more importantly in business benefits.

The usage of such deltas can be seen in many mature business models. The canonical example of delta usage is in the field of accounting. When looking at a ledger such as in Figure 2 each transaction or delta is being recorded. Next to it is a denormalized total of the state of the account at the end of that delta. In order to calculate this number the current delta is applied to the last known value. The last known value can be trusted because at any given point the transactions from the “beginning of time” for that account could be re-run in order to reconcile the validity of that value. In there exists a verifiable audit log.

Date	Comment	Change	Current Balance
1/1/2000	Deposit from 1372	+10000.00	10000.00
1/3/2000	Check 1	-4000.00	6000.00
1/4/2000	Purchase Coffee	-3.00	5997.00
1/6/2000	Purchase Internet	-5.00	5992.00
1/8/2000	Deposit from 1373	+1000.00	6992.00

Figure 14 A Simplified Ledger

Because all of the transactions or deltas associated with the account exist, they can be stepped through verifying the reult. The “Current Balance” at any point can be derived either by looking at the “Current Balance” or by adding up all of the “Changes” since the beginning of time for the account. The second property is obviously valuable in a domain such as accounting as accountants are dealing with money and the ability to check that calculations were performed correctly is extremely valuable, it was even more valuable before computers when it was common place to have an exhausted accountant make a mistake in a calculation at 3 am when they should be sleeping instead of working with the books.

There are however some other interesting properties to this mechanism of representing state, as an example, it is possible to go back and look at what a state was at a given point in time. Consider for that the account was allowed to reach a balance of below zero and there is a rule that says it is not supposed to. It is possible and relatively easy, to view the account as it was just prior to processing that transaction that put it into the invalid state and see what state it was in, making it far easier to reproduce what often times end up as heisenbugs in other circumstances.

These types of benefits are not only limited to naturally transaction based domains though. In fact **every** domain is a naturally transaction based domain when Domain Driven Design is being applied. When applying Domain Driven Design there is a heavy focus on behaviors, normally coinciding with use cases, Domain Driven Design is interested in how users **use** the system.

Returning to the Order example from Figure 1, the same order could be represented in the form of a transactional model as shown in Figure 3.

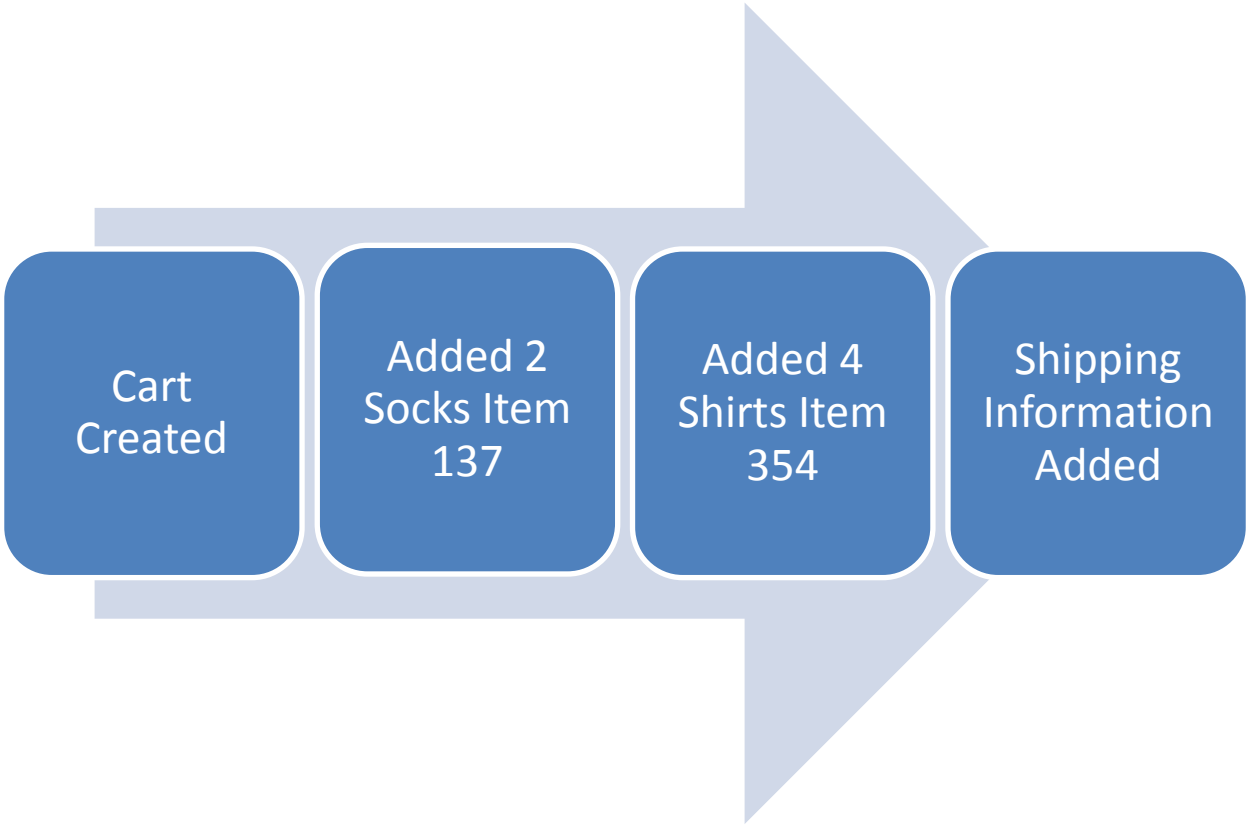

Figure 15 Transactional View of Order

This can be applied to any type of object. By replaying through the events the object can be returned to the last known state. It is mathematically equivalent to store the end of the equation or the equation that represents it. There is a structural representation of the object, but it exists only by replaying previous transactions to return the structure to its last known state, data is not persisted in a structure but as a series of transactions. One very interesting possibility here is that unlike when storing current state in a structural way there is no coupling between the representation of current state in the domain and in storage, the representation of current state in the domain can vary without thought of the persistence mechanism.

It is vitally important to note the language in Figure 3. All of the verbs are in the past tense. These are Domain Events. Consider what would happen if the language were in the imperative tense, “Add 2 socks item 137”, “Create Cart”. What if there were behaviors associated with adding and item (such as reserving it from an inventory system via a webservice call), should these behaviors be when reconstituting an object? What if logic has changed so that this item could no longer be added given the context? This is one of many examples where dual contexts between Commands and Events are required, there is a contextual difference between returning to a given state and attempting to transition to a new one.

There is no Delete

A common question that arises is how to delete information. It is not possible as previously jump into the time machine and say that an event never happened (eg: delete a previous event). As such it is necessary to model a delete explicitly as a new transaction as shown in Figure 4. Further discussion on the business value of handling deletes in this mechanism can be found in “Business Value of the Event Log”.

Is

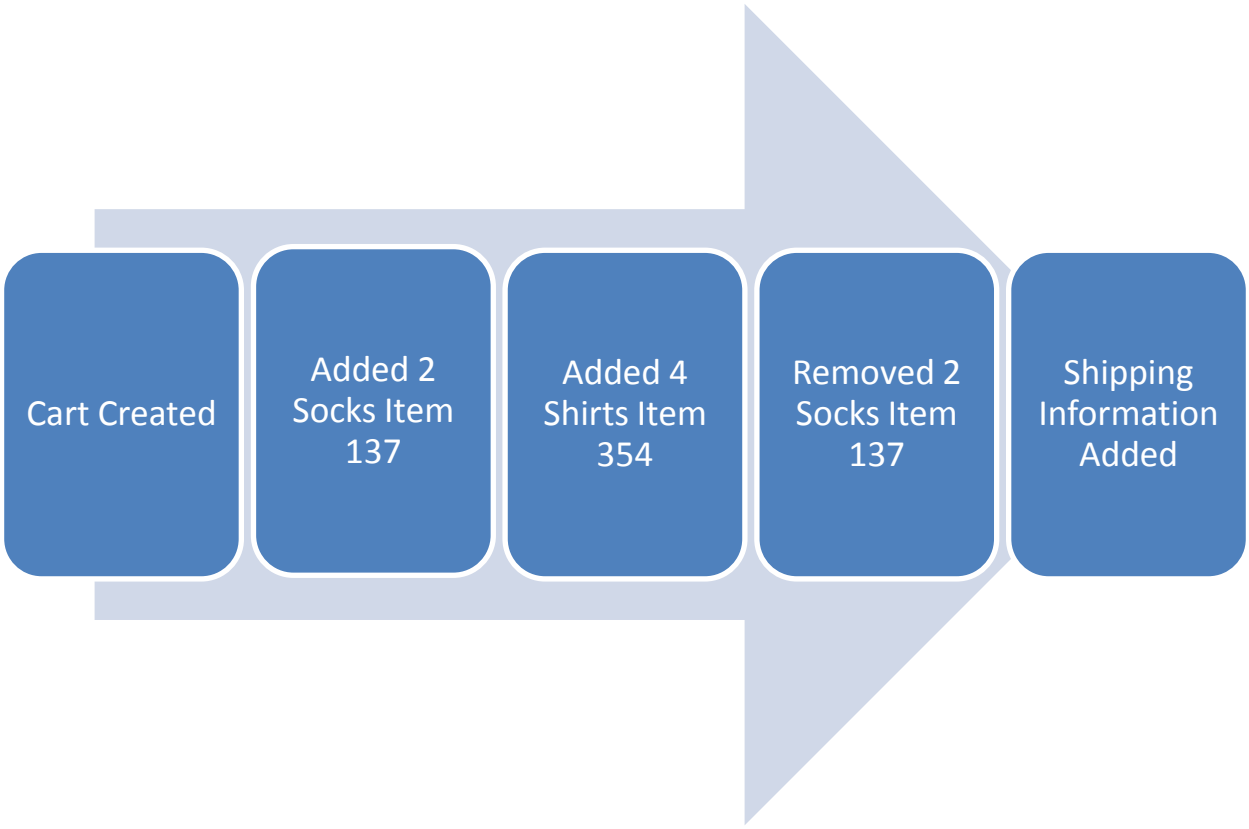

Figure 16 Transactional View of Order with Delete

In the event stream in Figure 4 the two pairs of socks were added then later removed. The end state is equivalent to not having added the two pairs of socks. The data has not however been deleted, new data has been added to bring the object to the state as if the first event had not happened, this process is known as a Reversal Transaction.

By placing a Reversal Transaction in the event stream not is the object returned to the state as if the item had not been added, the reversal leaves a trail that shows that the object had been in that state at a given point in time.

There are also architectural benefits to not deleting data. The storage system becomes an additive only architecture, it is well known that append-only architectures distribute more easily than updating architectures because there are far fewer locks to deal with.

Performance and Scalability

As an append-only model storing events is a far easier model to scale. There is however other benefits in terms of performance and scalability especially compared with a stereotypical relational model. As an example, the storage of events offers a much simpler mechanism to optimize as it is limited to a single append-only model. There are many other benefits.

Partitioning

A very common performance optimization in today's systems is the use of Horizontal Partitioning. With Horizontal Partitioning the same schema will exist in many places and some key within the data will be used to determine in which of the places the data will exist. Some have renamed the term to "Sharding" as of late. The basic idea is that you can maintain the same schema in multiple places and based on the key of a given row place it in one of many partitions.

One problem when attempting to use Horizontal Partitioning with a Relational Database it is necessary to define the key with which the partitioning should operate. This problem goes away when using events. **Aggregate IDs are the only partition point in the system.** No matter how many aggregates exist or how they may change structures, the Aggregate Id associated with events is the only partition point in the system.

Horizontally Partitioning an Event Store is a very simple process.

Saving Objects

When dealing with a stereotypical system utilizing a relational data storage it can be quite complex to figure out what has changed within the Aggregate. Again many tools have been built to help alleviate the pain that arises from this often complex task but **is the need for a tool a sign of a bigger problem**?

Most ORMs can figure out the changes that have occurred within a graph. They do this generally by maintaining two copies of a given graph, the first they hold in memory and the second they allow other code to interact with. When it becomes time to save a complex bit of code is run, walking the graph the code has interacted with and using the copy of the original graph to determine what has changed while the graph was in use by the code. These changes will then be saved back to the data storage system.

In a system that is Domain Event centric, the aggregates are themselves tracking strong events as to what has changed within them. There is no complex process for comparing to another copy of a graph, instead simply ask the aggregate for its changes. The operation to ask for changes is far more efficient than having to figure out what has changed.

Loading Objects

A similar issue exists when loading objects. Consider the work that is involved with loading a graph of objects in a stereotypical relational database backed system. Very often there are many queries that must be issued to build the aggregate. In order to help minimize the latency cost of these queries many ORMs have introduced a heuristic of Lazy Loading also known as Delayed Loading where a proxy is given in lieu of the real object. The data is only loaded when some code attempts to use that particular object.

Lazy Loading is useful because quite often a given behavior will only use a certain portion of data out of the aggregate and it prevents the developer from having to explicitly represent which data that is while amortizing the cost of the loading of the aggregate. It is this need for amortization of cost that shows a problem.

Aggregates are considered as a whole represented by the Aggregate Root. Conceptually an Aggregate is loaded and saved in its entirety. (Evans, 2001).

Conceptually it is much easier to deal with the concept of an Aggregate being loaded and saved in its entirety. The concept of Lazy Loading is not a trivial one when added and is especially not trivial when optimizing use cases. The heuristic is needed because loading full aggregates from a relational database is operationally too slow.

When dealing with events as a storage mechanism things are quite different. There is but one thing being stored, events. Simply load all of the events for an Aggregate and replay them. There can only ever be a single query on the system, there is no need to attempt to implement things like Lazy Loading. This is bad for people who want to build complex and quite often impressive frameworks for managing things like Lazy Loading but it is good for development teams who no longer need to learn these frameworks.

Many would quickly point out that although it requires more queries in a relational system, when storing events there may be a huge number of events for some aggregates. This can happen quite often and a relatively simple solution exists for the problem.

Rolling Snapshots

A Rolling Snapshot is a denormalization of the current state of an aggregate at a given point in time. It represents the state when all events to that point in time have been replayed. Rolling Snapshots are used as a heuristic to prevent the need to load all events for the entire history of an aggregate. Figure 5 shows a typical Event Stream. One way of process thing the event stream is to replay the events from the beginning of time until the end of the event stream is reached.

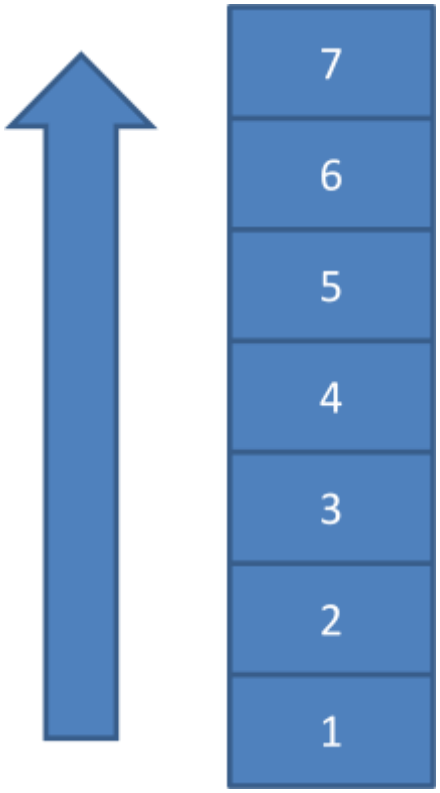

Figure 17 An Event Stream

The problem that exists is that there may be a very large number of events between the beginning of time and the current point. It can be easily imagined that there is an event stream with a million or more events that have occurred, such an event stream would be quite inefficient to load.

The solution is to use a Rolling Snapshot, to place a denormalization of the state at a given point in time. It would then be possible to only play the events from that point in time forward in order to load the Aggregate.

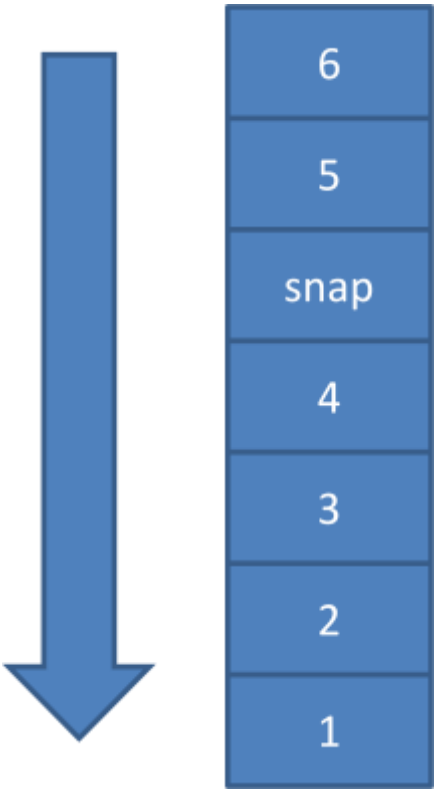

Figure 18 Event Stream with Snapshot

Figure 6 shows an Event Stream with a Rolling Snapshot placed within it. The process for rebuilding an Aggregate changes when using Rolling Snapshots. Instead of reading from the beginning of time forward, it is read backwards putting the events on to a stack until either there were no more events left or a snapshot was found. The snapshot would then if found be applied and the events would be popped off the stack and applied until the stack was empty.

It is important to note that although this is an easy way to conceptualize how Rolling Snapshots work, that this is a less than ideal solution in a production system for various reasons. Further discussion on the implementation of Rolling Snapshots can be found in "Building an Event Storage".

The snapshot itself is nothing more than a serialized form of the graph at that given point in time. **By having the state of that graph at that point in time replaying all the events prior to that snapshot can be avoided.** Snapshots can be taken asynchronously by a process monitoring the Event Store.

Introducing Rolling Snapshots allows control of the worst case when loading from events. The maximum number of events that would be processed can be tuned to optimize performance for the system in question. With the introduction of Rolling Snapshots it is a relatively trivial process to achieve one to two orders of magnitude of performance gain on the two operations that the Event Storage supports. It is important though to remember that Rolling Snapshots are just a heuristic and that conceptually the event stream is still viewed in its entirety.

Impedance Mismatch

Using events as a storage mechanism also offers very different properties when compared to a typical relational model when the impedance mismatch that exists between a typical relational model and the object oriented domain model is analyzed. Scott Ambler describes the problem in an essay on agiledata.org as

"Why does this impedance mismatch exist? The object-oriented paradigm is based on proven software engineering principles. The relational paradigm, however, is based on proven mathematical principles. Because the underlying paradigms are different the two technologies do not work together seamlessly. The impedance mismatch becomes apparent when you look at the preferred approach to access: With the object paradigm you traverse objects via their relationships whereas with the relational paradigm you join the data rows of tables. This fundamental difference results in a non-ideal combination of object and relational technologies, although when have you ever used two different things together without a few hitches?" (Ambler)

The impedance mismatch between the domain model and the relational database has a large cost associated with it. There are many tools that aim to help minimize the effects of the impedance mismatch such as Object Relational Mappers (ORM). They tend to work well in most situations but there is still a fairly large cost associated to the impedance mismatch even when using tools such as ORMs.

The cost is that a developer really needs to be intimately with both the relational model and the object oriented model. They also need to be familiar with the many subtle differences between the two models. Scott identifies this with

"To succeed using objects and relational databases together you need to understand both paradigms, and their differences, and then make intelligent tradeoffs based on that knowledge." (Ambler)

Some of these subtle differences can be found in Wikipedia under the "Object-Relational Impedance Mismatch" page but to include some of the major differences.

Declarative vs. imperative interfaces — Relational thinking tends to use data as interfaces, not behavior as interfaces. It thus has a declarative tilt in design philosophy in contrast to OO's behavioral tilt. (Some relational proponents propose using triggers, stored procedures, etc. to provide complex behavior, but this is not a common viewpoint.) (Object-Relational Impedance Mismatch)

Structure vs. behaviour — OO primarily focuses on ensuring that the structure of the program is reasonable (maintainable, understandable, extensible, reusable, safe), whereas relational systems focus on what kind of behaviour the resulting run-time system has (efficiency, adaptability, fault-tolerance, liveness, logical integrity, etc.). Object-oriented methods generally assume that the primary user of the object-oriented code and its interfaces are the application developers. In relational systems, the end-users' view of the behaviour of the system is sometimes considered to be more important. However, relational queries and "views" are common techniques to re-represent information in application- or

task-specific configurations. Further, relational does not prohibit local or application-specific structures or tables from being created, although many common development tools do not directly provide such a feature, assuming objects will be used instead. This makes it difficult to know whether the stated non-developer perspective of relational is inherent to relational, or merely a product of current practice and tool implementation assumptions. (Object-Relational Impedance Mismatch)

Set vs. graph relationships - The relationship between different items (objects or records) tend to be handled differently between the paradigms. Relational relationships are usually based on idioms taken from set theory, while object relationships lean toward idioms adopted from graph theory (including trees). While each can represent the same information as the other, the approaches they provide to access and manage information differ. (Object-Relational Impedance Mismatch)

There are many other subtle differences such as data types, identity, and how transactions work. The object-relational impedance mismatch can be quite a pain to deal with and it requires a very large amount of knowledge to deal with effectively.

There is not an impedance mismatch between events and the domain model. The events are themselves a domain concept, the idea of replaying events to reach a given state is also a domain concept. The entire system becomes defined in domain terms. Defining everything in domain terms not only lowers the amount of knowledge that developers need to have, it also limits the number of representations of the model needed as the events are directly tied to the domain model itself.

Business Value of the Event Log

It needs to be made clear at the very start of this section that the value of the Event Log is directly correlated with places that you would want to use Domain Driven Design in the first place. Domain Driven Design should be used in places where the business derives competitive advantage. Domain Driven Design itself is very difficult and expensive to apply; a company will however receive high ROI on the effort if the domain is complex and if they derive competitive advantage from it. Using an Event Log similarly will have high ROI when dealing with an area of competitive advantage but may have negative ROI in other places.

Storing only current state only allows to ask certain kinds of questions of the data. For example consider orders in the stock market. They can change for a few reasons, an order can change the amount of volume that they would like to buy/sell, the trading system can automatically adjust the volume of an order, or a trade could occur lowering the volume available on the current order.

If posed with a question regarding current liquidity such as the price for a given number of shares in the market, it really does not matter which of these changes occurred, it does not really matter **how** the data got the way it was, it matters **what** it is at a given point in time. A vast majority of queries even in the business world are focused on the **what**, labels to send customers mails, how much was sold in April, how many widgets are in the warehouse.

There are however other types of queries that are becoming more and more popular in business, they focus on the **how**. Examples can commonly be seen in the buzzword “Business Intelligence”. Perhaps

there is a correlation between people having done an action and their likelyhood of purchasing some product? These types of questions generally focus on how something came into being as opposed to what it came out to be.

It is best to go through an example. There is a development team at a large online retailer. In an iteration planning meeting a domain expert comes up with an idea. He believes that there is a correlation between people having added then removed an item from their cart and their likelihood of responding to suggestions of that product by purchasing it at a later point. The feature is added to the following iteration.

The first hypothetical team is utilizing a stereotypical current state based mechanism for storing state. They plan that in this iteration they will add tracking of items via a fact table that are removed from carts. They plan for the next iteration that they will then build a report. The business will receive after the second iteration a report that can show them information back to the previous iteration when the team released the functionality that began tracking items being removed from carts.

This is a very stereotypical process, at some organizations the report and the tracking may be released simultaneously but this is a relatively small detail in the handling. From a business perspective the domain experts are happy, they made a request of the team and the team was able to quickly fulfill the request, new functionality has been added in a quick and relatively painless way. The second team will however have quite a different result.

The second team has been storing events; they represent their current state by building up off of a series of events. They just like the first team go through and add tracking of items removed from carts via a fact table but they also run this handler from the beginning of the event log to back populate all of the data from the time that the business started. They release the report in the same iteration and the report has data that dates back for years.

The second team can do this because they have managed to store what the system actually did as opposed to what the current state of data is. It is possible to go back and look and interpret the old data in new and interesting ways. It was never considered to track what items were removed from carts or perhaps the number of times a user removes and items from their cart was considered important. These are both examples of new and interesting ways of looking at data.

As the events represent every action the system has undertaken any possible model describing the system can be built from the events.

Businesses regularly come up with new and interesting ways of looking at data. It is not possible with any level of confidence to predict how a business will want to look at today's data in five years. The ability for the business to look at the data in the way that it wants in five years is of an unknown but possibly extremely high value; it has already been stated that this should be done in areas where the business derives its competitive advantage so it is relatively easy to reason that the ability to look at today's data in an unexpected way could be a competitive advantage for the business. How do you value the possible success or failure of a company based upon an architectural decision now?

How do software teams justify looking at their Magic 8 Ball to predict what the business will need in five or even ten years? Many try to use YAGNI (You Ain't Gonna Need It) (Wikipedia) but YAGNI only applies when you actually know that you won't need it, how can the dynamic world of business and how they may want to look at data in five or ten years be predicted?

- Is it more expensive to actually model every behavior in the system? Yes.
- Is it more expensive in terms of disk cost and thought process to store every event in the system? Yes.
- **Are these costs worth the ROI when the business derives a competitive advantage from the data?**

Works Cited

Ambler, S. W. (n.d.). *The Object Relational Mismatch*. Retrieved from agiledata.org: http://www.agiledata.org/essays/impedanceMismatch.html

Evans, E. (2001). *Domain Driven Design.* Addisson Wesley.

Fowler, M. (n.d.). *Domain Event.* Retrieved from EAA Dev: http://martinfowler.com/eeaDev/DomainEvent.html

Jill Nicola, M. M. (2002ll). *Streamlined Object Modelling.* Prentice H.

Object-Relational Impedance Mismatch. (n.d.). Retrieved from Wikipedia: http://en.wikipedia.org/wiki/Object-relational_impedance_mismatch

Wikipedia. (n.d.). *You ain't gonna need it*. Retrieved from wikipedia: http://en.wikipedia.org/wiki/You_ain't_gonna_need_it

Building an Event Storage

In “Events as a Storage Mechanism” the concept of rebuilding state from a series of events was looked at from a conceptual viewpoint. This chapter will focus on the implementation of an actual Event Storage and some of the issues that come up in producing an implementation.

The implementation discussed in this chapter is not intended to be a production quality Event Storage, more so it is provided as a discussion point around how to build an Event Storage. The implementation here although not highly performant could meet the needs of a large percentage of applications that are built today.

For the explanatory implementation it is easiest to build the Event Storage in an existing technology such as a RDBMS. This will alleviate many of the technical issues that can arise that are out of the scope of a basic discussion on how to build an event storage such as transaction commit models or data locality for read performance.

Structure

A basic Event Storage can be represented in a Relational Database utilizing only two tables.

Column Name	Column Type
AggregateId	Guid
Data	Blob
Version	Int

Figure 19 Table Layout for Events Table

This table represents the actual Event Log. There will be one entry per event in this table. The event itself is stored in the [Data] column. The event is stored using some form of serialization, for the rest of this discussion the mechanism will assumed to be built in serialization although the use of the memento pattern can be highly advantageous.

The table is shown with the minimum amount of information possible, most organizations would want to add a few columns such as the time that the change was made or context information associated with the change. Examples of context information might include the user that initiated the change, the ip address they sourced the change from, or their level of permission when they sourced the change.

A version number is also stored with each event in the Events Table. This can generally be thought of as an increasing integer for most cases. Each event that is saved has an incremented version number. The version number is unique and sequential only within the context of a given aggregate. This is because Aggregate Root boundaries are consistency boundaries.

The [AggregateId] column is a foreign key that should be indexed; it points to the next table which is the Aggregates table.

Column Name	Column Type
AggregateId	Guid
Type	Varchar
Version	Int

Figure 20 Table Layout for Aggregates Table

Author comment: I have gone back and forth between calling this concept "Aggregate" in the Event Storage in lieu of another name such as "Event Provider" as "Aggregate" is really a domain concept and an Event Storage could work without a domain.

The Aggregates table is representing the aggregates currently in the system, every aggregate must have an entry in this table. Along with the identifier there is a denormalization of the current version number. This is primarily an optimization as it could be derived from the Events table but it is much faster to query the denormalization that it would be to query the Events table directly. This value is also used in the optimistic concurrency check.

Also included is a [Type] column for this example, this would be the fully qualified name of the type of aggregate being stored. This can be useful for various purposes not the least of which is debugging, it is however unnecessary for the creation of a basic Event Storage.

Operations

Event Storages are far simpler that most data storage mechanisms as they do not support general purpose querying. An Event Storage at its simplest level has only two operations. Having only two operations makes an Event Storage simpler than most data storage mechanisms as well as easier to optimize.

The first operation is to get all of the events for an aggregate. It is extremely important that the events are ordered in the same order that they were written, the version number can be used for this purpose. This can all be done quite simply using an underlying RDBMS.

```
SELECT * FROM EVENTS WHERE AGGREGATEID='' ORDER BY VERSION
```

This is the only query that should be executed by a production system against the Event Storage. A possible secondary query that can be useful is to limit this result set by an actual date to see the state of an object at a point in time, but generally a production system should not be doing this.

The other operation an Event Storage must support is the writing of a set of events to an aggregate root. This can be done either in code or in a stored procedure. A stored procedure or dynamically generated SQL containing if statements is preferred as without the insert process will take multiple round trips. The pseudo-code for the insert process can be seen in Listing 1.

```
Begin
   version = SELECT version from aggregates where AggregateId = ‘’
   if version is null
       Insert into aggregates
       version = 0
   end
   if expectedversion != version
      raise concurrency problem
   foreach event
      insert event with incremented version number
   update aggregate with last version number
End Transaction
```

Listing 5 Write Operation in Event Storage

The write operation is also relatively simple though there are a few subtleties to be found within it. The basic narrative is that it first checks to see if an aggregate exists with the unique identifier it is to use, if there is not one it will create it and consider the current version to be zero. It will then attempt to do an optimistic concurrency test on the data coming in if the expected version does not match the actual version it will raise a concurrency exception. Providing the versions are the same, it will then loop through the events being saved and insert them into the events table, incrementing the version number by one for each event. Finally it will update the Aggregates table to the new current version number for the aggregate. It is important to note that these operations are in a transaction as it is required to insure that optimistic concurrency amongst other things works in a distributed environment.

The contract for an Event Storage in code can be defined with the following interface.

```
public interface IEventStore {
   void SaveChanges(Guid AggregateId, int OriginatingVersion, IEnumerable<Event> events);
  IEnumerable<Event> GetEventsFor(Guid AggregateId);
}
```

Listing 6 Interface for an Event Store

Although not a trivial exercise to create a production quality Event Storage the overall concepts behind an Event Storage are relatively easy. Likely in the future there will be many off the shelf Event Storage systems available as either products or open source projects. There is however one very important optimization that was discussed in “Events as a Storage Mechanism” that really should exist in most systems and that is the concept of a “Rolling Snapshot”.

Rolling Snapshots

Rolling Snapshots are a heuristic to prevent the need to load all of the events when issuing a query to rebuild an Aggregate. They are a denormalization of the aggregate at a given point in time. A change to the query logic and an additional table are all that is necessary to add the heuristic to the basic Event Storage. Further discussion on Rolling Snapshots at a conceptual level can be found in the "Events as a Storage Mechanism" chapter.

Column Name	Column Type
AggregateId	Guid
SerializedData	Blob
Version	Int

Figure 21 Definition of Snapshots Table

The Snapshots table is relatively basic. It's primary data in the blob that contains the serialized version of the aggregate at a given point in time. The serialized data could be in any one of a host of possible schemas, binary, XML, raw text, etc. The decision on how to serialize the snapshots is really dependent upon the system being built. A version number is included with the snapshot, it represents which version of the aggregate the snapshot represents.

In order to have snapshots being created a process that handles the task of creating the snapshots needs to be introduced. This process can live outside of the Application Server as a background process. There can be a single process running or many depending on needs due to throughput. All snapshots happen asynchronously. Figure 4 shows a conceptual architecture with a [SnapShotter] process introduced.

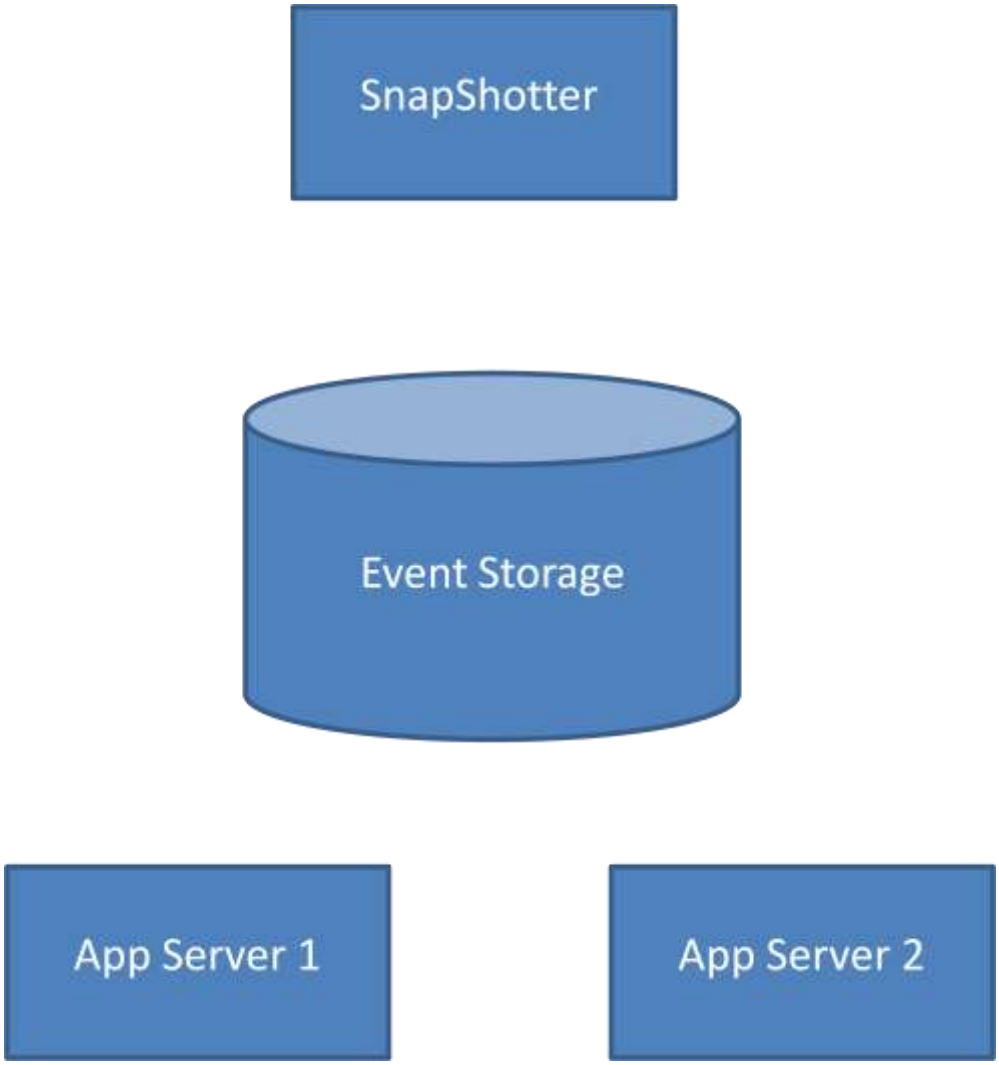

Figure 22 Introduction of a Snapshotter

The [SnapShotter] sits behind the Event Storage and periodically queries for any Aggregates that need to have a snapshot taken because they have gone past the allowed number of events. This query can be done quite easily in the simple Event Storage discussed by joining the Aggregates table to the Snapshots table on the Aggregate identifier. The difference is calculated by subtracting the last snapshot version from the current version with a where clause that only returned the aggregates with a difference greater than some number. This query will return all of the Aggregates that a snapshot to be created. The snapshotter would then iterate through this list of Aggregates to create the snapshots (if using multiple snapshotters the competing consumer pattern works well here).

The process of creating a snapshot involves having the domain load up the current version of the Aggregate then take a snapshot of it. The creation of the snapshot can be done in many ways. Once the

snapshot has been taken, it is saved back to the snapshot table so that queries will have the snapshot available.

Many use the default serialization package available with their platform with good results though the Memento pattern is quite useful when dealing with snapshots. The Memento pattern (or custom serialization) better insulates the domain over time as the structure of the domain objects change. The default serializer has versioning problems when the new structure is released (the existing snapshots must either deleted and recreated or updated to match the new schema). The use of the Memento pattern allows the separated versioning of the snapshot schema from the domain object itself.

In "Events as a Storage Mechanism" a different, simpler mechanism was shown for the storage of snapshots. That system had the snapshots in line in the Event Log, this other mechanism although conceptually simpler has a few issues that can come up in a production system. The issues revolve around the need of ordering of the snapshot within the event log.

Consider that the Snapshotter has realized that an Aggregate Root needs to have a snapshot taken. It loads up the Aggregate and takes the snapshot. Unfortunately while it was doing this, one of the Application Servers made a change to the same Aggregate. As the snapshot is position dependent within the Event Log, it would receive an optimistic concurrency failure. The easy answer would be to simply repeat the process but what if it failed again? The snapshotter on a very busy Aggregate could end up in a situation where it would have a very low probability of actually writing the snapshot successfully.

By separating the snapshots into their own table and associating them to a version of the aggregate this problem is solved. Ordering of snapshots is not needed, the snapshot does not even need to be at the latest version, the snapshot that is taken is valid **at the version it was taken.**

Snapshots are a heuristic that will dramatically improve the performance of many systems, though not all systems need snapshotting. It is generally recommended to handle development without snapshotting as it can always be introduced later as a simple performance enhancement for the system.

Event Storage as a Queue

It has been previously discussed that the events coming out of a domain are also an [Integration Model]. Very often these events are not only saved but also published to queue where they are dispatched asynchronously to listeners either within the same system (the reporting model is a good example) or to other applications. An issue that exists with many systems publishing events is that they require a two-phase commit between whatever storage they are using (Relational or otherwise) and the publishing of their events to the queue.

The reason that the two-phase commit is needed is that a catastrophe could occur during the small period of time between when the write to the data storage commits and when the write to the queue commits. If a failure were to happen during this period the message would not be published on the queue (or if the other direction it may be published but the change may not be saved). If either case were to happen the listeners of the events would be out of sync with the producer.

The two-phase commit can be expensive but for low latency systems there is a larger problem when dealing with this situation. Generally the queue itself is persistent so the event becomes written on disk twice in the two-phase commit, once to the Event Storage and once to the persistent queue. Given for most systems having dual writes is not that important but if you have low latency requirements it can become quite an expensive operation as it will also force seeks on the disk. Figure 5 illustrates the two-phase commit between data storage and a publishing queue.

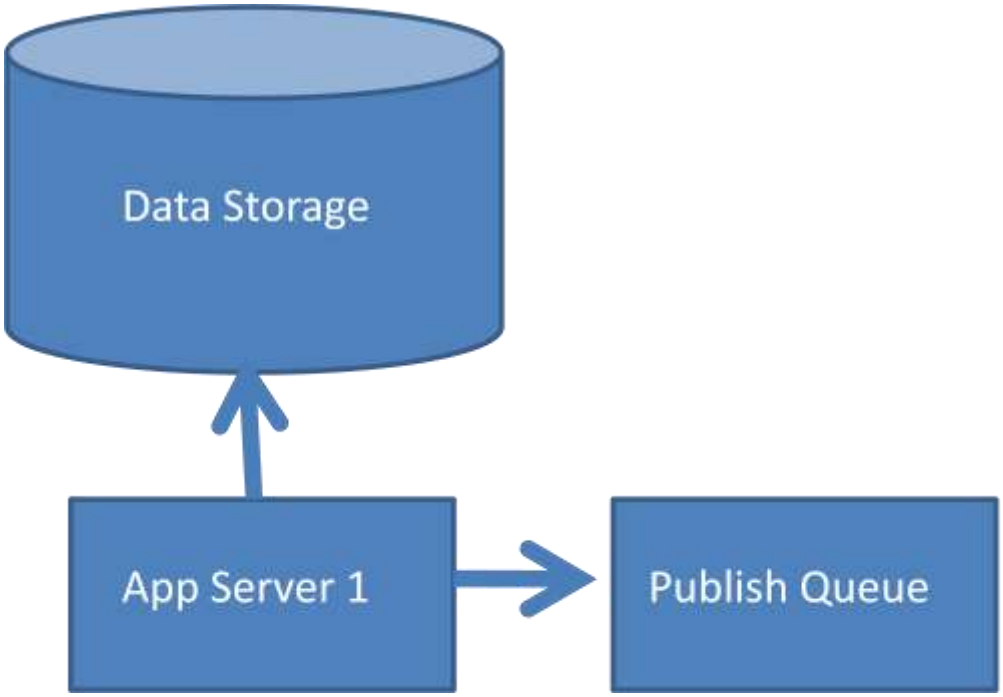

Figure 23 Two Phase Commit with Queue

Some try to get around this problem by only writing to a queue then have something on the other side of the queue update the data storage with the changes represented by the events, this however has some issues. The largest issue is that not all of the events will be able to be written to the storage, eventual consistency has been introduced and it is possible that an optimistic concurrency problem will occur on the write of the events. Dealing with this problem in a production system is non-trivial.

Many organizations do the opposite, use the event storage as a queue. Adding a sequence number to the Events table previously discussed allows the use the Event Storage as a queue. Figure 5 illustrates the change to the schema of the Events table.

Column Name	Column Type
AggregateId	Guid
Data	Blob
SequenceNumber	Long

Version	Int

Figure 24 Events Table as a Queue

The database would insure that the values of sequence number would be unique and incrementing, this can be easily done using an auto-incrementing type. Because the values are unique and incrementing a secondary process can chase the Events table, publishing the events off to ther queue. The chasing process would simply have to store the value of the sequence number of the last event it had processed, it could even update this value with a two-phase commit bringing the update and the publish to the queue into the same transaction. This process can be seen in Figure 7.

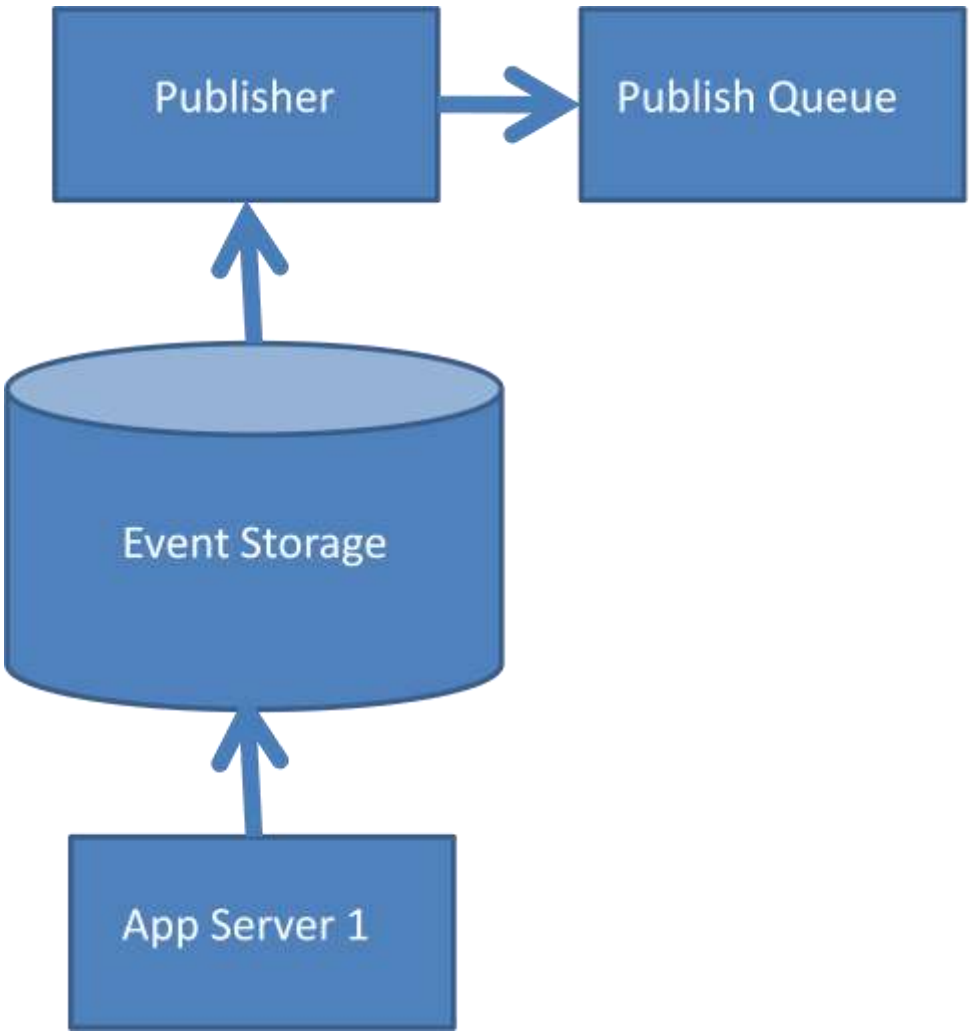

Figure 25 Event Storage as a Queue

The work has been taken off of the initial processing in a known safe way. The publish can happen asynchronously to the actual write. This lowers the latency of completing the initial operation, it also will limit the number of disk writes in the processing of the initial request to one. This strategy can be extremely valuable when dealing with low latency requirements as it allows much of the work on the initial processing to be offloaded to another process asynchronously and in a safe way, there is little difference whether the publish happens as part of the initial processing or asynchronously as generally messages are published asynchronously anyways, using the Event Store as a queue just raises the time until the message is actually published slightly, this can be viewed as slightly raising the SLA.

CQRS and Event Sourcing

CQRS and Event Sourcing become most interesting when combined together. This chapter looks at the intersection of these two concepts within a system where Domain Driven Design has been applied.

CQRS and Event Sourcing have a symbiotic relationship. CQRS allows Event Sourcing to be used as the data storage mechanism for the domain. One of the largest issues when using Event Sourcing is that you cannot ask the system a query such as "Give me all users whose first names are 'Greg'". This is due to not having a representation of current state. With CQRS the only query that exists within the domain is GetById which is supported with Event Sourcing.

Event Sourcing is also very important when building out a non-trivial CQRS based system. The problem with integration between the two models is a large one. The maintaining of say relational models, one for read and the other for write, is quite costly. It becomes especially costly when you factor in that there is also an event model in order to synchronize the two. With Event Sourcing the event model is also the persistence model on the Write side. This drastically lowers costs of development as no conversion between the models is needed.

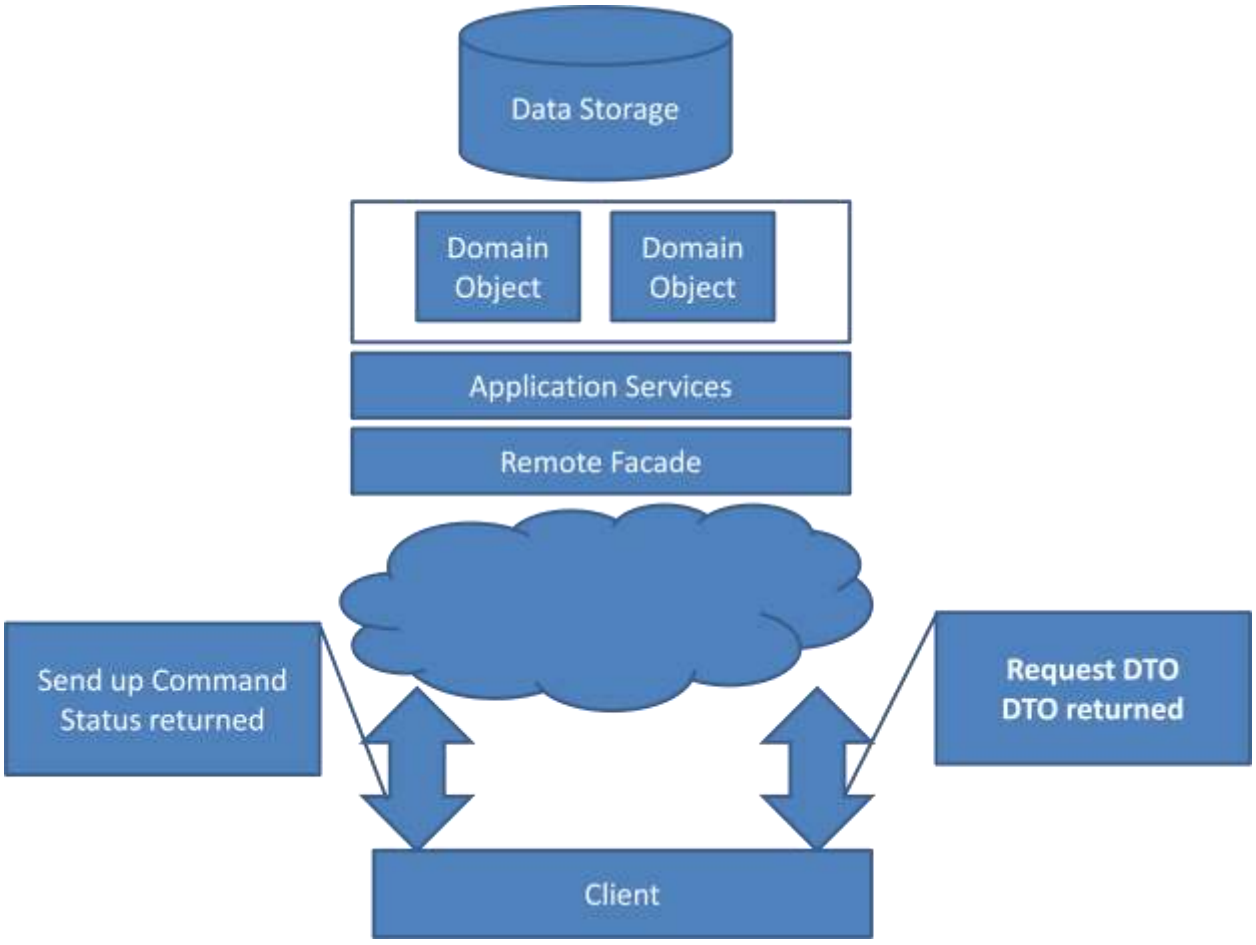

Figure 26 Stereotypical Architecture Sending Commands

The original stereotypical architecture with using commands in Figure 1 can be compared to Figure 2 CQRS with Event Sourcing and found to be roughly equivalent amounts of work.

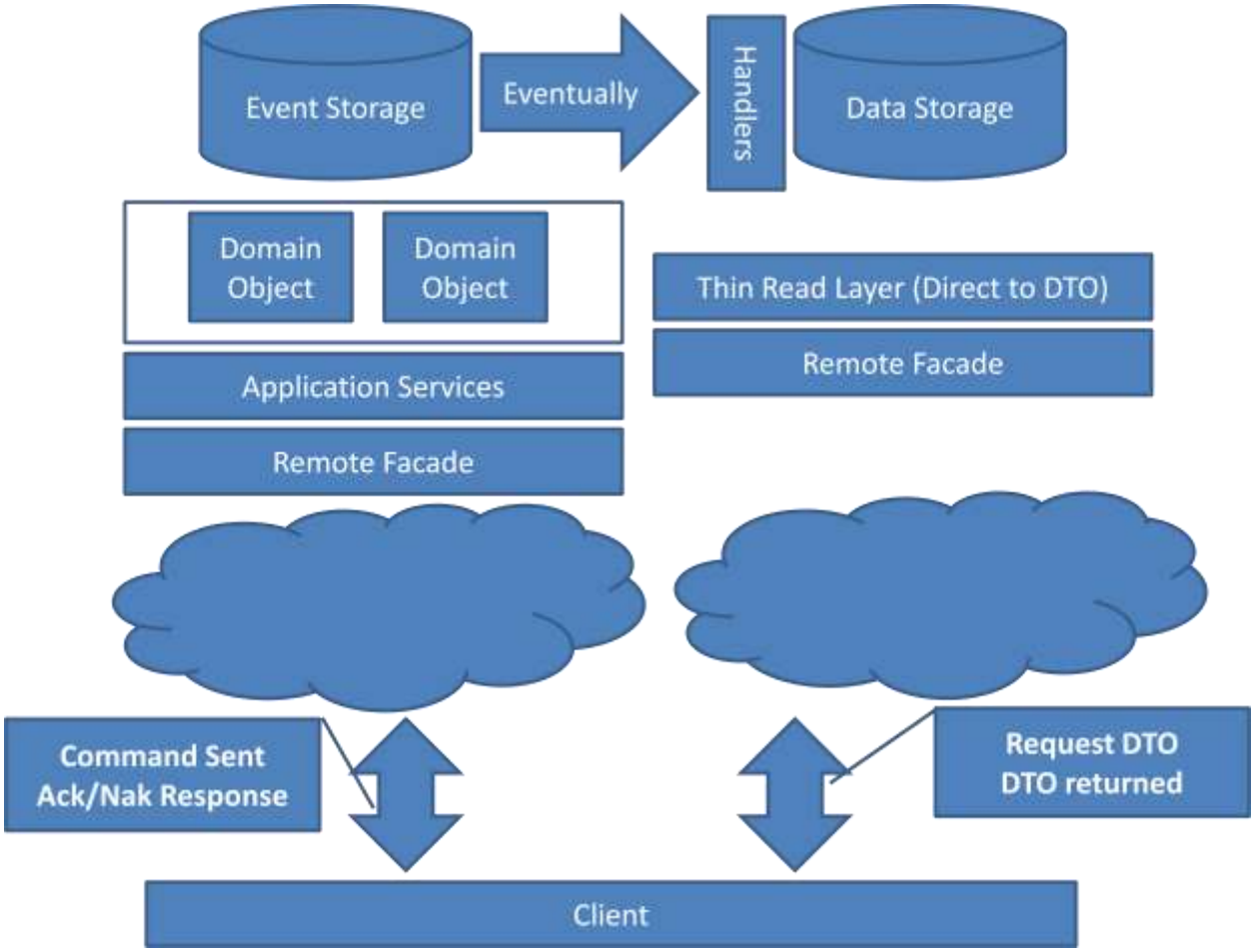

Figure 27 CQRS with Event Sourcing

Cost Analysis

The client will be identical amounts of work between the two architectures. This is because the client operates in the exact same way. In both architectures the client receives DTOs and produces Commands that tell the Application Server to do something.

The queries between the two models will also be very similar in terms of cost. In the stereotypical architecture the queries are being built off of the domain model, in the CQRS based architecture they are being built by the Thin Read Layer projecting directly to DTOs. As was discussed in “Command and Query Responsibility Segregation” the Thin Read Layer should be equally or in some cases less expensive.

The main differentiation between the two architectures when looking at cost is in the domain model and persistence. In the stereotypical architecture an ORM was doing most of the heavy lifting in order to persist the domain model within a Relational Database. This process introduces an Impedance Mismatch between the domain model and the storage mechanism, the Impedance Mismatch as discussed in "Events as a Storage Mechanism" can be highly costly both in productivity and the knowledge that developers need to have.

The CQRS and Event Sourcing based architecture does **not** have an Impedance Mismatch between the domain model and the storage mechanism on the Write side. The domain produces events, these same events are all that is stored. The usage of events is all that the domain model knows about. There is however an impedance mismatch in the read model. The Event Handlers must take events and update the read model to its concept of what the events mean. The Impedance Mismatch here is between the Events and the Relational Model.

The Impedance Mismatch between events and a Relational Model is much smaller than the mismatch between an Object Model and a Relational Model and is much easier to bridge in simple ways. The reason for this is that the Event Model does not have structure, it is representing **actions** that should be taken within the Relational Model.

Looked at from this perspective, the two architectures have roughly the same amount of work being done. Its not that its a lot more work or a lot less work; its just **different** work. The event based model may be slightly more costly due to the need of definition of events but this cost is relatively low and it also offers a smaller Impedance Mismatch to bridge which helps to make up for the cost. The event based model also offers all of the benefits discussed in "Events" that also help to reduce the overall initial cost of the creation of the events.

That said the CQRS and Event Sourcing model is actually less expensive in most cases!

Integration

Everything up until this point has been comparing the systems in isolation. This rarely happens within an organization. More often than not organizations do only rely on systems but on systems of systems that are integrated in some way.

With the stereotypical architecture no integration has yet been supported, except of course perhaps integration through the database which is a well established anti-pattern for most systems. Integration is viewed as an afterthought.

The integration must be custom written. Many organizations choose to build services over the top of their model to allow for integration. Some of these services may be the same services that the clients use but more often than not there is additional work that must be done in order to support integration.

A larger problem exists when the product is being delivered to many customers. It is the teams responsibility to provide hooks for all of the customers and how they would like to integrate with the

system. This often becomes a very large and unwieldy piece of code, especially on systems that are installed at hundreds or thousands of different clients all of which have different needs. The business model here tends to be to bill the client for each piece of custom integration, this can be quite profitable but it is a terrible model in terms of software.

With the CQRS and Event Sourcing based model, integration has been thought of since the very first use case. The Read side needs to integrate and represent what is occurring on the Write Side, it is an integration point. The integration model is "production ready" all throughout the initial building of the system and it is being tested throughout by the integration with the Read Side.

The event based integration model is also known to be complete as all behaviors within the system have events. If the system is capable of doing something, it is by definition automatically integrated. In some circumstances it may be desirable to limit the publishing of events but it is a decision to limit what is published as opposed to needing to write code to publish something.

The event based model is also by nature a push model that contains many advantages over the pull model. If the stereotypical architecture desired a push based model then there would be large amounts of work added to track events and ensure that they were synchronized with what the system recorded in its own data model.

Differences in Work Habits

The two architectures also differ greatly in parallelization of work. In the stereotypical architecture work is generally done in vertical slices. There are four common methodologies used.

- **Data Centric**: Start with database and working out.
- **Client Centric:** Start with client and work in.
- **Façade/Contract First:** Start with façade, then work back to data model then work finally implement client
- **Domain Centric:** Start with the domain model, work out to the client then implement data model

These methodologies all have a commonality; they tend to work in vertical slices. The same developers will work on a feature through these steps. The same can be done with the CQRS and Event Sourcing based architecture but it does not need to be. Consider a very high level view of the systems as contained in Figure 3.

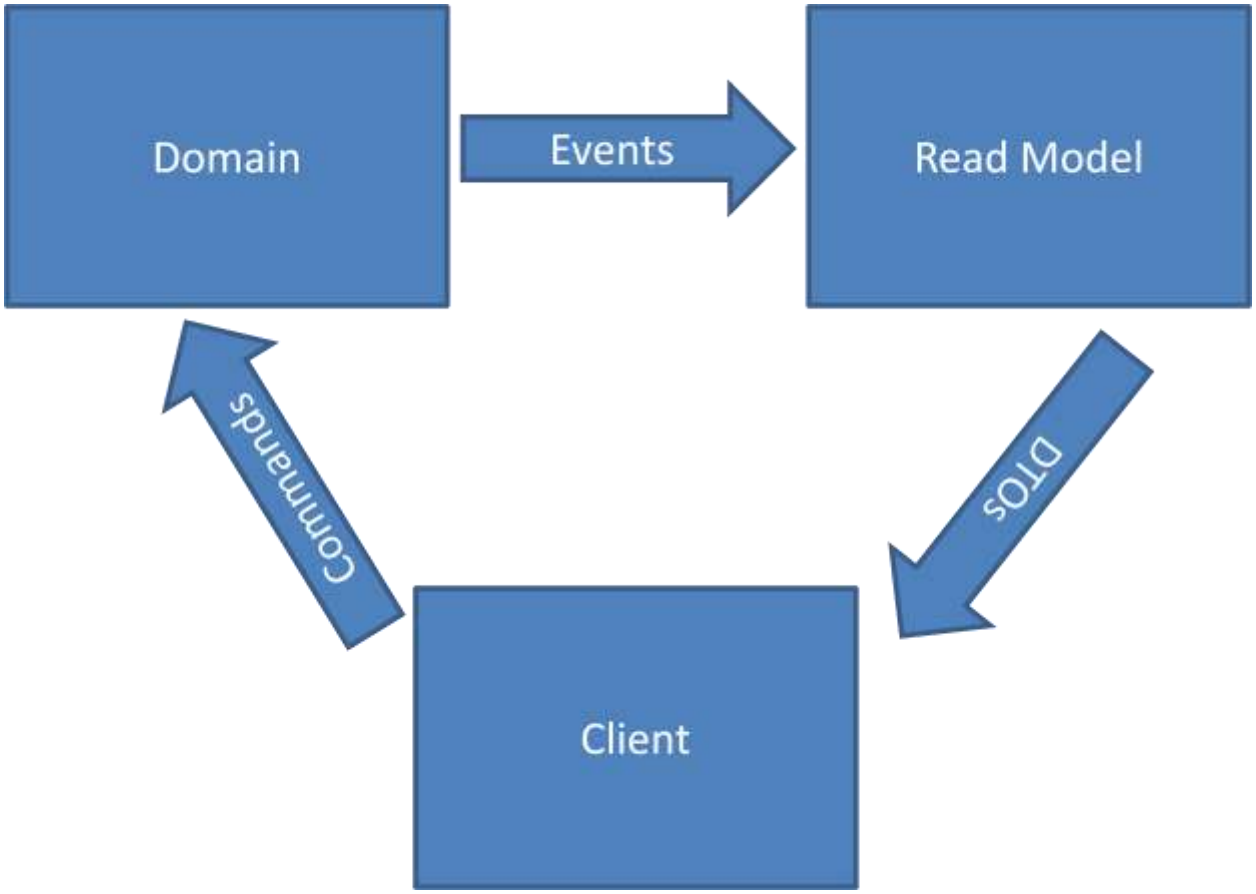

Figure 28 High Level View of CQRS and Event Sourcing

The architecture can be viewed as three distinct decoupled areas. The first is the client; it consumes DTOs and produces Commands. The second is the domain; it consumes commands and produces events. The third is the Read Model; it consumes events and produces DTOs. The decoupled nature of these three areas can be extremely valuable in terms of team characteristics.

Parallelization

It is relatively easy to have five to eight developers working on vertical slices at a given point without running into too many conflicts in what is being done. This is because for a small number of developers it is relatively easy to communicate what each developer is working on and to insure that there are few if any areas where developers overlap. This problem becomes much more difficult as you scale up the number of developers.

Instead of working in vertical slices the team can be working on three concurrent vertical slices, the client, the domain, and the read model. This allows for a much better scaling of the number of developers working on a project as since they are isolated from each other they cause less conflict when making changes. It would be reasonable to nearly triple a team size without introducing a larger amount of conflict due to not needing to introduce more communication for the developers. They still communicate in the same way but they communicate about smaller decoupled pieces. This can be

extremely beneficial when time to market is important as it can drastically lower the amount of calendar time to get a project done.

All Developers are not Created Equally

There, it has been said. On a team there are many different types of developers, some attributes to consider in differences amongst developers include

- Technical Proficiency
- Knowledge of the Business Domain
- Cost
- Soft Skills

The points of decoupling are natural and support the specialization of teams in given areas. As an example in the domain, the best candidate is a person who is high in cost but also has a large amount of business knowledge, technical proficiency, and soft skills to talk with domain experts. When dealing with the read model and the generation of DTOs this is simply not the case, it is a relatively straight forward thing to do. The requirements are different which often leads to the next item.

Outsourcing

It is often not cost effective to keep low cost, medium skilled developers on a team. The overhead of keeping employees in terms of salary costs as well as compliance with various governmental regulations is often not worth the benefits of having the developers as employees. If a company is in a high cost locale, the company can certainly get cheaper developers offshore. Whether offshore or onshore the separation helps with successfully outsourcing part of a project.

Outsourced projects often fail because large amounts of communication are required between the outsourcers and the local team or domain experts. With these communications many problems can come up including time differences, cultural, and linguistic.

The Read Model as an example is an ideal area of the system to outsource. The contracts for the Read Model as well of specifications for how it work are quite concrete and easily described. Little business knowledge is needed and the technical proficiency requirements on most systems will be in the mid-range.

The Domain Model on the other hand is something that will not work at all if outsourced. The developers of the Domain Model need to have large amounts of communications with the domain experts. The developers will also benefit greatly by having initial domain knowledge. These developers are best kept locally within the team and should be highly valued.

A company can save large amounts of capital by outsourcing this area of the system at a low risk, this capital can then be reinvested in other, more important areas of the system. The directed use of capital is very important in reaching a higher quality, lower cost system.

Specialization

A problem exists when working with vertical slices. The "best" developers, with best being defined as most valuable, work with the domain. When working with a vertical slice though anecdotal evidence suggests that they spend roughly 20-30% of their time in this endeavor.

With the secondary architecture, the team of developers working with the domain spend 80+% of their time working with the domain and interacting with Domain Experts. The developers have no concern for how the data model is persisted, or what data needs to be displayed to users. The developers focus on the use cases of the system. They need only know Commands and Events.

This specialization frees them to engage in the far more important activities of reaching a good model and a highly descriptive Ubiquitous Language with the Domain Experts. It also helps to optimize the time of the Domain Experts as opposed to having them sit idly while the "technical" aspects of vertical slices are being worked on.

Only Sometimes

There are many benefits offered through the separation but they do not need to be used. It is also quite common to have a normal sized team still work in vertical slices. There are benefits in terms of risk management amongst other things to having a small to medium sized team work in vertical slices of the whole system.

The real benefit with the CQRS and Event Sourcing based architecture is that the option exists to bring it into three distinct vertical slices with each having its own attributes optimized as opposed to using a one size fits all mechanism

.