Greg Young - CQRS/DDD Notes

- Typical architecture
 - o Dto out from remote façade (over ASL or direct to ASL)
 - Client modifies dto
 - Client sends dto back to remove façade
 - This is a big problem for DDD
 - Anemic domain model
 - An anti-pattern if you're trying to do DDD
 - You can't do DDD with this architecture!
 - Your domain is all nouns
 - Or just "crud"
 - Behavior behind façade is all nouns
 - Business logic ends up in client!
 - Or worse, it's only in a manual or heads of users
 - Other issues
 - Doesn't scale
 - Bigger db is only option
 - Bits of accidental complexity creep in
 - Mapping domain objects to dtos makes query optimization difficult
 - Why do we use this architecture
 - Tooling
 - Comfort/inertia
 - Platform agnostic
 - Devs understand it
 - Where did all this guidance come from? MS/Oracle
 - Companies with vested interest in scaling the data storage!!!
 - They benefit from this architecture!
- First incremental change VERBS, imperative
 - Start speaking in verbs, not nouns
 - Requires analysis on how users use the system
 - Not editing data, but performing a task
 - o How do users actually want to use the system?
 - First change
 - as user interacts with the screen, build messages
 - send messages instead of dtos
 - called a "Command"
 - Consider commands as serializeable method calls

- Always imperative, telling the system to DO something
- This language is critical
 - Helps new devs
 - Communication with business users
- Can the command processor say no? YES
- Represents an action the client would LIKE the server to take

CQRS

- Commands request to mutate state
 - Synchronous in most business systems
 - Benefits but also difficulties with async (fire and forget)
- Query request for data
- Originally called CQS (Bertrand Meyer)
 - Side note read Meyer's books
- Not actually the same pattern
 - Beyond the method level
 - Separate objects for commands and queries
- This pattern is not eventual consistency or event sourcing
 - Just a simple separation of concerns
- If you have setters in your domain model, it's an ANTI PATTERN
 - Code calling the setter should be IN the domain object
 - Even getters cause a lot of problems!
 - "we need dtos!"
 - "we need to show them on screen!"
 - "our ORM needs them!"
 - Lots of orm hints
 - Need expertise of
 - o Domain model
 - Mappings
 - o Orm
 - datamodel
 - Near impossible to build an optimal query!
 - This is a symptom of the "simple" architecture!
- Read side of CQRS
 - Thin read layer on top of data store
 - Queries directly return dtos
 - Could be direct to database, doesn't need to be complex
 - Pulling apart read model allows different decisions to be made
 - Don't need to go through domain model
 - Team needs to know less things to be effective
 - Save time, less accidental complexity
- Write side of CQRS

- Domain model is clear since it isn't for reading
- Domain model with private only state true encapsulation
- Domain only exposes behaviors
- Limiting the width of our contract, simplifying testability
- ORM
 - Solves symptoms of problems
 - Without addressing root of the problem
- How our aggregates are structured
 - Represents a transactional boundary
 - Model behavior and dependency within a transaction
 - Represent a intersection of a group of use cases in a transactional context
- Better decisions are available
 - Optimizations can be made on write and read separately
- CAP Theorem
 - o Consistency, availability, partitionability
 - Typical architecture = CA
 - o CQRS architecture
 - Read side = AP
 - Write Side = CA
- Next incremental change
 - Two data stores

- Next incremental change
 - Application service becomes command handlers
 - Single interface
 - public interface Handles<T> where T : message {
 - void handle(T message);
 - }
 - A handle coordinates the domain to complete the process
 - No logic in command handlers!
 - Dispatch to domain object.
 - Very little code
 - Doesn't know when to commit transaction
 - Handlers are built up in a pipeline that will handle transaction boundaries
 - Cross cutting concerns
 - Authorization, logging, transaction wrapping, etc.
 - Not AOP very simple code instead
 - Build up a chain of handlers

Similar to FUBU MVC pipeline

Events

- Structure and purpose
 - Look a lot like commands
 - Class structure is exactly the same
 - Difference is intention
 - The big difference in code is LANGUAGE class name in this case
 - Crucial that events are verbs in the past tense
 - A completed action in the past
 - Favorite things to say to juniors can you use that in a sentence please?
- Event Handlers
 - Also implements the "Handles" interface
 - Produce events in the domain
 - These events can write data from domain, shipped into read storage
 - Just insert some data into the denormalized table(s) for querying
 - These are so dumb and simple, should be hiring low paid folks to do this
 - Can even keep database Nazis happy
 - Call a stored procedure from your event handler!
 - Then DBAs can go live in their own little world
 - Also helps get buy in from the DBA, even though we're going to subvert them in the long run
 - Same as command handlers, again difference is intention
 - Can also build up a pipeline of these just like commands
 - Can use same Transactional wrappers, logging, etc.
- Serious problem with ORM architecture
 - ORM is the problem
 - What does it do?
 - Ask orm for a graph, which gets mapped into objects
 - In the event of an error, Which model is correct? The read model? Or the write model?
 - NHibernate + Events gives us two sources of truth
 - Delta generated by NH
 - And our events
 - How to get around this?
 - o Get rid of NHibernate
 - o And still make database Nazis happy
 - Alternative because of events
 - The event makes explicit what a change inside our domain means
 - Events are deltas, so we don't need and ORM to calculate them for us
 - Event handlers on both sides (read and write model)
 - Update write model and read model from events

- Handlers on both sides
- Solves model synchronization problem
 - Save events
 - In event of bug, drop or fix event then replay
- The ONE source of truth is the events sent to both models

- Event Sourcing and benefits of the Event Log
 - Event sourcing lets us get rid of one of the 2 data sources
 - Relational models versus Behavior models
 - Mature industries use a behavior model
 - Banking
 - No "account" object
 - Your "balance" is represented as a summation of transactions
 - Example: shopping cart
 - Cart Created
 - Add Item to cart
 - Add Item to cart
 - Add Item to cart
 - Remove Item from cart
 - Shipping Information Added
 - All events can be replayed at any time
 - Main reason to use events and event sourcing
 - If I have an event stream, I can generate any possible structural model from that event stream
 - This is huge value from a business perspective
 - You don't know what business questions will be asked in 5-10 years
 - Just store all of the data and you will be able to answer any question they have at any point in the future!
 - Structural models lose information
 - Event sources don't lose ANY information
 - Other interesting things
 - Append only
 - No locks
 - Immutable once written
 - Easy to scale
 - Can always create new contexts for this information
 - A structural model can be built to answer any question
 - Rewind/replay
 - Audit log
 - State is derived from the audit log

- O What about when there are millions of events?
 - Rolling snapshots
 - This is definitely place for a framework
 - Instead of starting from the beginning and playing forward, instead start end and add each event to a stack until you hit a snapshot. Then load domain object from snapshot, then pop off stack to replay forward from the snapshot
- Aggregate root
 - No transactional guarantees outside of an aggregate root
 - Aggregates are loaded up from Events
- Structure Two Tables
 - Event Table (no primary key because no updating or deleting ever!)
 - AggregateId Guid
 - Date Blob
 - Version int (only per aggregate)
 - Aggregate Table (or Event Provider table)
 - Aggregate Id Guid
 - Type varchar
 - Version int (represents the max version from the event table)
- Only query in the whole system (without snapshots):
 - select * from events where aggregateid=x order by version
- Write (probably in a stored procedure)

```
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
```

- Snapshots (could even go in aggregate table if you only want one snapshot per aggregate – historical snapshots are rarely needed)
 - Structure
 - AggregateId Guid
 - Serialized Data blob
 - Version int

- Just storing the serialized data is not the best way to go what if you change a field name?
 - Use Memento pattern
 - o A separate object is serialized instead of the aggregate
 - Smaller system can work with a simple serialization mechanism for snapshotting because it's fast to rebuild the snapshots in the small system
- Queueing on the Read side
 - Read model as a queue of events from the write side
 - Use event storage as the queue for the read model
 - No need for distributed transactions
 - Idempotency
 - read model event handlers just drop events they have already heard before
- Business Driven! Not technology driven
- o No more model on the Write side
 - No more costs
 - It's just events
- Don't build your own just grab one of the open source event stores
 - There are some that work with a sql backend

0

- Implementing Aggregates
 - Command is public method on aggregate
 - Check/guard/throw if command not allowed
 - Call ApplyEvent to do actual state change
 - Event is private method on aggregate
 - No longer allowed to say no
 - Cannot fail, no exceptions
 - NO BEHAVIOR in state changing methods, no conditional logic, no exceptions
 - AggregateBase.LoadFromHistory
 - Given an IEnumerable of events, apply each event

```
public class AggregateRoot : IAggregateRoot {
   private Dictionary<Type, appliesEvent> lookup = new Dictionary<Type, appliesEvent>();
   protected void RegisterHandler<T>(appliesEvent handler) {
       lookup.Add(typeof(T), handler);
   public IEnumerable<Event> GetChanges() {
       foreach(Event e in events) yield return e;
   protected void ApplyEvent(Event e) { ApplyEvent(e, true); }
   private_void ApplyEvent(Event e, bool add) {
       appliesEvent handler;
       if(!lookup.TryGetValue(e.GetType(), out handler)) {
               throw new HandlerNotFoundException();
       handler(e):
       if(add) UnitOfWork.GetCurrent().Enregister(e);
   }
   public void LoadFromHistory(IEnumerable<Event> events) {
       foreach(Event e in events) {
           ApplyEvent(e, false);
}
 protected void ApplyEvent(Event e) { ApplyEvent(e, true); }
 private void ApplyEvent(Event e, bool add) {
    appliesEvent handler;
     if(!lookup.TryGetValue(e.GetType(), out handler)) {
              throw new HandlerNotFoundException();
     if(add) UnitOfWork.GetCurrent().Enregister(e);
 public void LoadFromHistory(IEnumerable<Event> events) {
     foreach(Event e in events) {
         ApplyEvent(e, false);
```

- Use a convention based approach to register event handlers on your domain objects
- See "Greg young delegate adjuster" on Google
 - Allows you to treat handlers generically
 - A little trick, but a lot of people get caught up on it

- Introduces a dependency on your backend storage
- Introduces cross aggregate transactional boundaries
- How do I test when Aggregate is totally encapsulated?
 - Write behavioral tests
- Testing
 - Given/When/Then
 - Test Fixture per context (not per class)
 - Given
 - Prepare the scenario
 - When
 - One and only one line of code
 - Greg's test framework ONLY lets you return an IEnumerabe<Event> from the Given method
 - Because all state transition are in the form of events, I can make assertions not only that what happened should happen, but that would should not have happened did not happen

```
public abstract class AggregateRootTestFixture<T> : where T:IAggregateRoot, new() {
           protected IAggregateRoot root;
           protected Exception caught = null;
           protected T sut;
           protected List<Event> events;
           protected IEnumerable<Event> Given();
           protected abstract void When();
           [SetUp]
           public void Setup() {
                   root = new T();
                   root.LoadFromHistory(Given());
                   try {
                           When();
                           events = new List<Event>(root.GetChanges());
                   }catch(Exception Ex) {
                           caught = Ex;
                   }
           }
   }
0
```

- The test framework isn't what makes the testing easy, but the way we're creating our domain model! Event driven is what makes it work.
- When building systems like this, started walking away from testing in this fashion
 - Why? Lots of noise to describe the scenario
 - Already have something that can run the scenario event handlers!
 - New approach
 - Given
 - A set of events
 - When I
 - o Issue a command
 - Expect
 - A set of events or exception

- This is why Greg is a language nazi
 - Given
 - o inventory item created
 - When I
 - deactive inventory item
 - Expect
 - inventory item deactivated
- Commands and Events are stupid little serializable objects
 - Now I can represent my tests in plain old English
 - Meta data / JSON / Etc.
 - One test to rule them all
- Actually gave end users a tool for setting up scenarios, taking and action, and stating their expectations
 - Built their tests and domain from the user results! Cool.
- No more focus on how an object implements something, but its external behaviors

SESSION 5

Creating read model objects

0

Multiple handles, listening to multiple events

- Can have many handlers for a given event
- Only one handler for a given command
- The big problem that most people run in to VERSIONING
 - Event Versioning
 - Make new version of the event
 - Make new "apply" version on domain object
 - Modify domain object command to raise new event version
 - What about cleaning up old "apply" methods from old versions?
 - Introduce a "convert" method in the aggregate root
 - Run all converters in aggregate root "LoadFromHistory"
 - When to version?
 - Only when the first event is written to production
 - Can't ever delete old version of the event, but can remove behavior from aggregate root object once all events can be up converted
 - If you can't convert from one version to the next, then you actually have a new event, not a new version of the old one.
 - Command Versioning

- Similar, but eventually we won't receive the old command anymore
- We can delete old commands that are no longer sent from any clients
- Snapshot Versioning
 - For most systems, just delete existing snapshots and make new ones
 - In high performance systems, use memento to avoid need for snapshots
- Eventual Consistency a term that really confuses business people!
 - This makes business users think the data is "wrong"
 - o But the data is just "old"
 - And there is ALWAYS old data being showed to users
 - all their data is stale anyway
 - printing for example
 - We want to shorten to length of time of the oldness of course
 - O How do we deal with users not seeing their change?
 - First thing educate the users, explain that changes might take a few seconds
 - Email
 - Trick the user
 - Change around the UI a bit
 - Example: Take user to a "what do you want to do next" screen, instead of a list screen after an add. Buys you enough time for the add to complete.
 - Show the added data even though not committed yet
 - Cost have to duplicate the logic
 - One Way commands Put a queue in front of the domain
 - Client sends a command to the queue
 - Queue tells client "it worked!"
 - Benefit
 - How hard to scale a queue? To make it available?
 - Easy
 - Big spike of traffic
 - o Processing the queue in the background
 - Users don't see that there is slowness
 - Spike doesn't affect user experience or harm availability or introduce latency
 - Ability to take systems down
 - Queue can back up while other systems are down
 - Problem with one way commands
 - Duplication of logic in the client
 - Needs some logic to ensure command will succeed
 - Set validation
 - http://codebetter.com/blogs/gregyoung/archive/2010/08/12/eventualconsistency-and-set-validation.aspx

- Architecture and Team Structure
 - Business Side
 - CQRS isn't just about technology, it's about business
 - Gives us the ability to make different decisions in different places
 - Better optimize decisions
 - Typical architecture (ouch!)
 - View First
 - Start with client
 - Drive down the layers to data
 - The Microsoft Way
 - Start with the data storage
 - Drive up the layers to client
 - The DDD Way
 - Start with the Domain
 - Drive out to client, then back to data storage
 - Remote Façade first
 - Back to data storage, then client at the end
 - Client centric point of view
 - All of these mechanisms involve VERTICAL SLICES!!!
 - Assumes all developers are equal in ability
 - Insane!!!
 - Do I want the same developer working in the domain that is working in the client? Are there different properties I might be interested for those different roles? What about the data store?
 - This all changes with the CQRS Architecture
 - CQRS Architecture
 - Three distinct pieces
 - Read Model
 - Consumes Events
 - Produces Dtos
 - Client
 - Consumes Dtos
 - Produces Commands
 - Write Model
 - Consumes Commands
 - Produces Events
 - Best devs in domain model just commands and events
 - Maximizes their time in the domain model
 - High skill level, know how to ask questions

- Client devs
 - No need to know about the domain model
 - Dtos and how to send a command
- Read model devs
 - Low skilled, Monkey code
 - Outsource/offshoring
 - Why do this in house?
 - May be even cheaper to use devs than to pay to automate
 - Don't pay an American developer to do this!!!
 - Read model is also lower risk
 - Much easier to scale
 - Much cheaper to rewrite (can always rebuild and replay)
- Gives us hard boundaries in our system
 - We can specialize in cost and performance and team optimizations in interesting ways
- Management
 - A team of some size before seeing a drop off of productivity
 - Typical architecture requires a high level of management
 - Most teams diminishing returns around 10-14 developers
 - Extremely Important
 - Without changing your level of management maturity, you can get roughly 2.5X more people working on a CQRS project, given the same level of management
- What does the 2.5X mean to your company?
 - 1 year with 6 developers? vs.
 - 4 months 18 developers?
 - Factor of 3 on Time to Market
 - Business is making money for 8 months instead of bleeding cash those 8 months
- CQRS Process and Workflow for a new feature
 - Feature Estimation
 - Create the commands, events and dtos
 - XSD tool
 - Check in to project planning with the feature
 - Feature bubbles to top of schedule
 - Grab xsds and generate their contracts
 - Client generate dtos and commands from xsd
 - Read generate events and dtos from xsd
 - Write generate commands and events from xsd
- These team structure specializations are one of the biggest benefits of CQRS
- Cost

- Greenfield vs. Brownfield?
- Brownfield Migrate iteratively
 - Actual analysis (what are the actual commands?)
 - Task based UI
 - Real behavior in domain model instead
 - Can do all this with existing 3rd normal form database
 - Can apply CQRS immediately, but for now, leave queries going over the domain model
 - Don't want to rewrite all our queries, just start with read only remote façade to work against domain (later to migrate to 1st normal form model)
 - Start adding events into the domain objects
 - Just start producing and publishing events
 - o Put in an event handler and stop saving with NHibernate
 - o One aggregate at a time
 - Make it do everything with events
 - Store all of these events
 - Quantify costs of 3rd normal form model
 - O What are the costs?
 - o Resources? Hardware?
 - If costs are too high, drop in event sourcing and get rid of 3rd normal form model

Greenfield

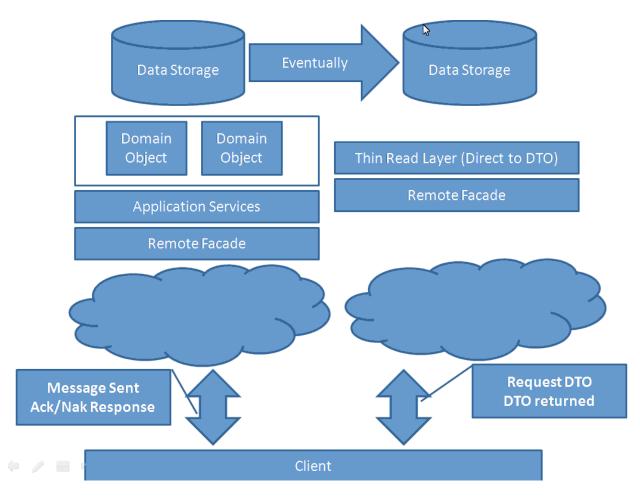
- Separate Read and Write remote façades
 - No add to cost, especially if read façade is communicating with the domain for now
 - Separation of thin read layer will probably reduce costs
- Events coming out of the domain model
 - o If two separate models, a definite cost increase
 - Event sourcing gets rid of the 3rd normal form model
 - Back to cost parity maintaining one model
 - Writes more difficult, but reads simple
 - Versus old architecture, reads difficult but writes simple
- Not saying one is way cheaper, but it's about options for organization.
 Not more or less work, but different work
- The Integration BOMB SHELL
 - However, in Greg's experience, a CQRS architecture is 30-60% cheaper than the first system
 - The big cost difference is that with CQRS the integration model is already completed! And how many systems never communicate with other external systems?

- Integration costs are HUGE
- Integration often costs more than the original project development costs
- PUSH model integration (instead of pull)
- o With push model, I only need to understand my own events
- Inverting the dependencies! ☺
- This architecture allows our initial development to build out our integration model with no additional costs

Summary

- Patterns
 - A Task Based UI
 - Documents forthcoming
 - Command Query Responsibility Segregation
 - Specialization of the read side
 - No more reads off the domain
 - Allowed our systems to diverge and make different decisions
 - SPECIALIZATION is what CQRS is all out about
 - o In code
 - o In teams
 - Placing of building these things
 - Domain Events
 - Two data storages one for read, one for transactions
 - Costs of keeping models synchronized
 - Finally lost our irrational fear of dropping a production DB
 - How we can have the 3rd normal form DB at start, and then talk about maintenance costs with a view to killing it
 - Storing events even if we have a 3rd normal form db
 - The value of the event log
 - o Any possible structural model can be built from it
 - Extremely valuable from a business perspective
 - o Value when it's time for the system to die
 - The new system runs off the event log
 - Event storage is a good place for a framework
- o Business benefits
 - Team Management
 - 3 vertical slices instead of one
 - Specializing in each slice
- Recognized that these systems are pieces of a system of systems
 - The work of creating the system of systems is not non trivial
 - CQRS builds the integration model. It's done and tested.
- Greg's thoughts

- I like that I don't need a framework
 - I don't need a jr. dev to know NHibernate or other tool to be effective
 - I wouldn't put effort into building a framework
 - It's just simple, clear code
 - How long to understand the framework? Versus just reading the code that we've written that does the work?
- Filesystem watcher
 - Look at event handler dll
 - If it changes, respinup event handlers while running
 - Don't have to bring down the whole system
- Two Queues
 - Command Queue
 - Event Storage (also a queue)
 - Storing messages as a queue
 - And sending events to the read model



Open Source

Mark Nihjof

Jonathan Oliver