Lecture 26: Domain-Driven Design (Part 4)

Kenneth M. Anderson

Object-Oriented Analysis and Design

CSCI 4448/6448 - Spring Semester, 2005

Goals for this lecture

- Review the extended example presented in chapter 7 of Domain-Driven Design
  - Cargo Delivery Example

First Three Lectures

- The last 3 lectures covered the pattern language of Domain Driven Design; Each core concept in the book is presented as a pattern
  - Ubiquitous Language and Model Driven Design
  - Layered Architecture
  - Entities, Value Objects, Services, Modules
  - Aggregates, Factories, Repositories
- Using these patterns, Evans claims, can improve the quality of software development life cycles; leading you to a place where
  - analysis and design information are captured in code
  - domain complexities are partitioned into useful structures
  - the life cycles of domain objects are managed efficiently

Chapter 7

- In chapter 7, Evans tries to bring the pattern language to bear on one example, to show how the parts work together
  - The domain: cargo tracking
  - The requirements
    - Track key handling of customer cargo
    - Book cargo in advance
    - Send invoices to customers automatically
  - The initial model (page 164)
    - Customer, Cargo, Delivery History, Delivery Specification
    - Handling Event, Carrier Movement, Location
  - It would take a while to get to this point in the model's development

Model Benefits

- This model provides several benefits
  - It captures domain knowledge
    - Multiple Customers are involved with a Cargo, each playing some role
    - A series of Carrier Movements satisfying the Specification will fulfill the delivery goal
  - These statements can be constructed directly from observing the model (a feature related to the Ubiquitous Language pattern)
  - Each element in the model has a clear meaning
    - Handling Event is a discrete action taken with the Cargo
    - Delivery Specification defines a delivery goal
      - This keeps this concern out of the Cargo class; keeping cohesion high
    - ...

On to Design

- The requirements and the model are produced by analysis
- The example starts with the transition to the design phase
  - We begin by applying the Layered Architecture pattern
    - The model is placed in the domain layer
    - We now identify classes that will be needed in the application layer
- Application Layer
  - Three functions needed to solve the requirements are
    - Recording what happens to each cargo (Incident Logging Application)
    - Handling requests to book cargo (Booking Application)
    - Searching for the current or past status of a cargo (Tracking Query)
  - These classes are coordinators that use the domain layer

Identifying Entities and Values

- Another important issue is identifying what concepts in the domain have identity that must be tracked by our application
  - After analysis all concepts except Delivery Specifications are identified as Entity objects
    - Customers, Cargo, Handling Events, Carrier Movements, Location, etc.
    - Most will have automatically generated ids; Handling Events are different because a domain expert reveals that each such event can be uniquely identified by combining a cargo's id with its completion time and type
      - The same Cargo cannot be loaded and unloaded at the same time
  - Delivery Specifications are Value Objects because they can be shared by two cargos going to the same place
    - Eventually, we hope, the cargo's Delivery History should end with an "unload" event at the Location indicated in the Delivery Specification

Designing Associations

- We need to revisit the associations in the model to attempt to reduce the implementation complexity and to better match our requirements (see page 170)
  - For instance, the association between Handling Events and Carrier Movements is bi-directional and has multiplicity in one direction
    - If we were tracking the inventory of ships, we would want to traverse from Carrier Movements to Handling Events; but, we are just tracking individual pieces of cargo; add directionality on that relationship to show we only need to implement one direction
  - We have one cycle in the model
    - Cargo → Delivery History → Handling Event → Cargo
    - Cycles are tricky; need to balance complexity of implementation with performance; for instance, a database lookup on one of the associations can break the cycle but may impede performance if done frequently

Designing Aggregates

- Need to examine model for aggregate boundaries
  - Customers, Locations, and Carrier Movements have their own identities and are (potentially) shared by many Cargos
    - they should each (possibly) be the root of their own Aggregates
      - they may also just be Entities that do not have a complex internal structure
  - Cargo requires additional thought (page 171)
    - Delivery History and Delivery Specifications are obvious elements of a Cargo aggregate
    - Handling Event, however, is not because we previously identified the need to search for handling events in two separate instances
      - Looking for handling events related to a particular Cargo
      - Looking for handling events related to a particular Carrier Movement
        - Even though we decided we didn't need this particular query to meet our requirements; requirements can change!

Selecting Repositories

- There are five Entity objects in the design, each a root of an aggregate
- We need to decide which ones deserve repositories
  - Recall that some objects we need to find via queries others we will "find" by traversing associations
- To do this, we return to our three application functions
  - The booking application needs to look up Customers and Locations
    - It creates Cargos
  - The Incident Logging Application needs to look up Carrier Movements and Cargos
  - The Tracking Query needs to look up Cargos (it gets to handling events by traversing associations)
- As such, we add four repositories (page 172)

Walking Through Scenarios

- We must continually check our decisions by "walking through" scenarios (think "use cases") to make sure we can meet our requirements
  - Two Examples
    - Changing a Cargo's Destination
      - Create a new Delivery Specification and update the Cargo object
    - Repeat Business (Cargos from repeat customers are often very similar)
      - Use old Cargos as a template for new Cargos; Copy old Cargo and then
        - Replace Delivery History with a newly created, empty, history
        - Copy collection that maps roles to customers (do not copy customers)
        - Create new tracking id (just as we would when creating a new Cargo)

Identifying Factories

- The book discusses two factories (others would be needed)
- Cargo needs a factory because it has a lot of parts that need to be created before its root can be handed to the application layer
  - We need to make sure that a newly created Cargo has
    - an empty delivery history that points back to Cargo (page 174-175)
    - a null delivery specification (added later by the booking application)
- Handling Event needs a factory since it would be useful to have factory methods that create each of the different types of handling events automatically (page 176)
  - Because of the cycle in our model, adding handling events to a Cargo's delivery history is not a straightforward task (page 176)
    - In a multiuser app, contention for the Cargo aggregate can occur, yet creation of Handling Events needs to happen quickly (page 177-179)
      - Evans replaces Delivery History with query and adds a Repository

The Role of Modules

- The Shipping model is expanded on page 180 and an initial partitioning into modules is attempted
  - The classes are partitioned by pattern type (Entities, Values, etc.)
    - Such partitions are not helpful since they do not support or capture our knowledge of the domain
- A second partitioning is presented on page 181 that expands on our knowledge of the domain
  - We now group classes with the following concepts
    - Customer, Billing, and Shipping
    - Ubiquitous Language: Our company does shipping for customers so we can bill them

Summary and What's Next?

- The example concludes with a discussion of the issues surrounding the introduction of a new requirement (and dealing with domain objects that live in multiple systems)
- Overall the example provides insight into how the various patterns of Domain-Driven Design can be used to build a software system
- What's Next
  - A survey of other object-oriented life cycles
  - Guest Lecture: The use of the Rational Unified Process at Sandia National Labs
  - Chapter 10 of Domain Driven Design
  - Semester project presentations