Lecture 25: Domain-Driven Design (Part 3)

Kenneth M. Anderson

Object-Oriented Analysis and Design

CSCI 4448/6448 - Spring Semester, 2005

Goals for this lecture

- Review (most of) the material presented in chapter 6 of Domain-Driven Design
- Aggregates
- Factories
- Repositories
- Present examples that illustrate these concepts
Domain Object Life Cycle

- Every object has a life cycle (see page 123)
  - It is created
  - It moves through various states
  - It is then deleted or archived
    - If the latter, it can eventually be restored and live again
- For transient objects, this life cycle is simple to manage
- But for domain objects, this life cycle can be complicated
  - You need to keep track of state changes and each object may have complex relationships with other objects

Life Cycles and MDD

- Two challenges occur in Model-Driven Design with respect to managing object life cycles
  - Maintaining object integrity throughout the life cycle
    - making sure constraints/rules/invariants are maintained
  - Preventing the model from getting “swamped” by the complexity of managing the life cycle
- We do this via the use of three patterns
  - Aggregates: which provide clear boundaries within the model and thereby reduce complexity
  - Factories: used to encapsulate the complexities of creating and reconstituting complex objects (aggregates)
  - Repositories: use to encapsulate the complexities of dealing with persistent complex objects (aggregates)
Aggregates

- Model objects often participate in complex relationships
  - Managing the consistency of these relationships can be difficult
- Compounding this problem is the fact that the “real world” often gives no hints as to the location of sharp boundaries in this web of concepts and relationships
  - Quick Example: Deleting a Person from a Database
    - Do you delete the Person’s associated value objects?
    - What if other Person objects need those value objects
      - If so, they may end up pointing to garbage
      - If not, you may litter the database with unreferenced value objects
- Further compounding this problem is that these objects are often accessed by multiple users concurrently
  - We need to prevent simultaneous changes to interdependent objects

More on Aggregates

- An Aggregate is a cluster of associated objects that we treat as a unit for the purpose of data changes
  - Each aggregate has a root and a boundary
    - The root is a single, specific entity object contained in the aggregate
    - The boundary defines what is IN the aggregate and what is OUT
  - Clients can only hold references to the root object of an aggregate
  - Objects within the aggregate are allowed to hold references to one another
  - Objects within the aggregate have local identity but not global identity
    - Thus a Car object may have global identity but its tires do not (see page 127)
  - Aggregates can have invariants associated with them (page 128)
    - These invariants are typically enforced/maintained by the root object
  - A delete operation on an aggregate must delete everything within the boundary at once
Example

The book presents an extended aggregate example on pages 130 to 134 looking at a purchase order aggregate and examining issues surrounding updates to purchase orders and parts.

It highlights the consistency issues that can come into play when dealing with multiuser updates to shared objects.

Factories

Factories are key elements in the domain layer that manage the creation of complex domain objects.

- Car engines are hard to build; humans and robots are used to accomplish the task.
- Once built, the car engine can focus on what it does best.
  - It doesn’t need to know how to build itself.
  - Furthermore, you don’t need the humans/robots that created it, in order to use it.

The same is true of complex domain objects.

- We can create them with factories and then use them for their purpose without need for the factory.
- This approach can keep complex object construction and rule invariant code out of the domain objects themselves.
Factories, continued

While they are considered part of the domain layer, Factories typically do NOT belong to the model

Basic interactions (page 138)
- The factory defines a method with all the parameters needed to create a particular class of domain object
- A client invokes the method providing the required parameters
- The factory creates the new object and makes sure that all class invariants are valid
- The factory returns the newly created object to the client
Factories are thus ideal for creating Entities and Aggregates
- They are less necessary for Value Objects

Implementing Factories

Three Factory-related methods appear in the Design Patterns book
- Abstract Factory, Factory Method (see Lecture 13), Builder

Two basic requirements on Factories
- Each creation method is atomic (cannot be interrupted in the presence of multiple threads) and enforces all invariants of the created object or Aggregate
  - If something goes wrong during creation, the method should throw an exception than allow an object to be created in an inconsistent state
  - Placing invariant logic in a factory can often save a lot of space in the domain class itself; because typically a domain object’s methods will not allow the object to be transformed into an illegal state after its created
- Factory methods should return abstract types, not concrete classes
  - Thus a Factory for a linked list in Java would return the type List and not the type LinkedList or ArrayList
Locating Factories

- As much as possible, place factories where control for the creation of an object makes sense
  - If you need to add an element to a pre-existing aggregate, provide a factory method on the root object of the aggregate (page 140)
  - If you have two closely related domain objects, consider placing a factory method on one of them to create the other (page 140)
  - When creating aggregates or complex Entity objects, create a dedicated Factory object
    - With respect to aggregates, the factory creates the aggregate all at once and returns a reference to the aggregate's root (page 141)
    - With respect to Entities, the factory will make sure that the entity's identity is globally unique within your application

Factories and Archived Objects

- Factories can be used to reconstitute archived objects (page 146)
  - Reconstituting a persistent object can be a complex process
  - There are two differences in this situation however
    - Entity objects are not given new identities; rather the stored information is used to reconstitute their previous identities
    - Rule violations will be handled differently
      - when first creating an object, a violation can safely cause an exception
      - but if information from an archive causes a violation, it means either
        - there is a bug in our domain class, allowing an object that was valid after creation to enter an invalid state
        - or, there is a bug in our persistence code, that takes a valid object and stores it incorrectly
        - OR, the persistent information was modified outside of our application
      - regardless, rather than throwing an exception, we must attempt repair
Repositories

- To do anything with an object, you must have a reference to it
  - How do you get that reference?
    - You could create the object
    - Or, you could traverse an association from an object you already have
    - OR, you could execute a query to find that object in a database based on its attributes
  - Most designs will use a combination of search and traversal to keep model-driven designs manageable (avoiding the problems discussed at the beginning of chapter 3)
    - You need to be careful with search however, because it becomes easy to think of objects as just “containers” for the information stored in the database; your design can start to lose its OO feel
    - In particular, you can decide not to create aggregates and entities, and just use queries to grab the objects you need directly

Finding a balance

- To limit the scope of the object access problem, we can
  - not worry about transient objects
    - objects used only in the method that created them
  - not provide query access to objects that can more easily be found via traversal
    - thus, we don’t need a search function for, e.g., a person’s address; instead we will traverse an association of the Person class to get that
  - not provide query access to objects that are internal to an aggregate; you need to go through the aggregate’s root
  - Instead, we will use a Repository that provides search capabilities based on object attributes to find, typically, the roots of aggregates that are not convenient to reach by traversal
More on Repositories

- A Repository represents a collection of objects of a certain type
  - Clients can add and remove objects from this set; the repository will take care of adding/removing the corresponding object to/from a particular persistence mechanism
  - Clients provide attributes to a repository's search methods to gain access to particular objects in the domain (page 151); the repository takes care of creating the query needed to retrieve such objects from the persistence mechanism (page 155)

Benefits

- Repositories provide a simple model for accessing persistent objects
- Repositories decouple applications from persistence mechanisms
- Repositories can be defined abstractly and then be implemented in multiple ways (such as in-memory collections, XML, RDMS, etc.)

Additional Issues

- Query Types
  - Repositories can support
    - hard-coded queries (page 153) and
    - specification-based queries (page 153)

- Implementation Concerns
  - Client code ignores a repository's implementation, developers do not
    - Developers need to understand how queries bring objects into memory and how that memory is reclaimed

- Keep factories and repositories distinct (page 158)
  - Have repositories use factories to reconstitute objects
  - With new objects, have clients add the object to a repository; do not have a factory create and then add an object directly

- What's Next? Covering the extended example of chapter 7