Lecture 24: Domain-Driven Design (Part 2)

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 chapters 4 and 5 of Domain-Driven Design
  - Layered Architecture
  - Associations, Entities, Value Objects, Services, Modules
- Present examples that illustrate these concepts

Layered Architecture

- Modern applications are often implemented using a layered architecture; different layers support separation of concerns
- A typical breakdown consists of the following layers
  - User Interface
  - Application
  - Domain
  - Infrastructure
- The key is to isolate domain concepts from system concepts
  - "To apply our best thinking, we need to be able to look at the elements of our model and see them as a system. We must not be forced to pick [domain concepts] out of a much larger mix of objects, like trying to identify constellations in the night sky"

Quick Example

- Shipping Application: Select Destination City for Cargo
  - We need code that
    - places a list selection widget on screen (UI)
    - queries the database for all possible cities (Infrastructure)
    - interprets user events and validates them (Application)
    - associates the selected city with the cargo (Domain)
    - commits change to database (Infrastructure)
- The domain layer constitutes only a small portion of the entire software system, yet its importance is disproportionate to its size
  - (for reasons covered in lecture 23)

Basic Principles of Layers

- Dependencies between layers should exist in only one direction
- As such, within a layer, an object can depend on
  - other objects in its layer
  - and objects in layers “below” it
  - See example on page 72
- If an object in a lower layer needs to communicate with an object in a layer above it needs to use indirect mechanisms, such as
  - callbacks
    - an “upper” object passes itself as a parameter to a “lower” object after implementing a predefined callback interface; the “lower” object uses this reference to communicate back up
  - the Observer pattern

Domain Layer

- Objects within the domain layer are elements of “the model”
  - They should be isolated from the UI, Application, and Infrastructure layers as much as possible
    - “The domain objects, free of the responsibility of displaying themselves, storing themselves, managing application tasks, and so forth, can be focused on expressing the domain model. This allows a model to evolve... to capture essential business knowledge and put it to work.”
  - The domain layer is where all of the concepts, behaviors, and rules specified for the model are implemented; the other layers should be devoid of “domain logic” as much as possible
    - Rather than implementing a domain rule in the application layer, have it call the domain layer and respond appropriately
      - e.g., a violation of a business rule might raise an exception in the domain layer, that is caught by the application layer, and displayed by the UI layer.

Expressing the Model in Software

- Chapter 5 looks at issues that arise when establishing the link between a model and the software that implements it
  - "[Associations and Objects] are simple to conceive and to draw, but implementing them is a potential quagmire. [!]" — Page 81
- We will look at the following model-related concepts
  - Associations: Relationships between model concepts
  - Entities: Objects with identity that need to be tracked
  - Value Objects: Serve as attributes to describe other objects
  - Services: Something that is done for a client on request; services will mainly live within the technical layers of your software system ("display this domain concept") but the domain layer will also need services to model domain-related activities

Associations

- For every traversable association in the model, there is a mechanism in the software with the same properties
  - Ex.: an association between a customer and a sales representative
    - Represents, on one hand, "domain knowledge"
    - On the other hand, it also represents a pointer between two objects, or the result of a database lookup, etc.
  - Associations can be implemented in many ways
    - A one-to-many association can be implemented as
      - a collection class pointed to by an instance variable
      - it might be a getter method that queries a database
- Associations in the "real world"
  - Lots of many-to-many relationships, with many being bidirectional
    - Really hard to implement!

Dealing with Associations

- There are three techniques for making associations managable
  - Impose a traversal direction
  - Add a qualifier, effectively eliminating or reducing multiplicity
  - Eliminate nonessential associations (as dictated by the problem you are trying to solve)
- See examples of the first two techniques on pages 84-88

Entities

- Some objects are not defined primarily by their attributes. They represent a thread of identity that runs through time and often across distinct representations
  - Consider the notion of “customer” in a typical business system
    - Customer may have a payment history
      - if its good, “status” will accrue; if its bad, the customer’s information may be transferred to a bill-collection agency
    - The same customer may be in the contact management software used by your company’s sales force
    - The customer may be “squashed flat” for storage in a database
    - If business stops, the customer may be placed in an archive
  - Each aspect of the customer may be implemented in multiple ways, using different representations and/or programming languages
    - They all represent the SAME customer however, and some means must exist to match them even though their attributes may be different

Entities, continued

- An object defined primarily by its identity is called an Entity
  - They have life cycles that can radically change their form and content
  - Their identities must be defined so that they can be effectively tracked
    - This notion of identity is DIFFERENT from the identity mechanisms of programming languages; i.e., it is different from “a == b” and “a.equals(b)” that OO languages provide
  - Example
    - Two deposits of the same amount made to the same bank account on the same day are NOT identical; they are two separate entity objects in the banking domain
      - the objects representing the amounts ARE identical, however, and are most likely Value Objects (discussed next)

Modeling Entities

- The key to modeling an entity object is to include only those attributes that are used to establish its identity or are commonly used to find or match it; include only those behaviors that support the task of maintaining its identity
  - All other behaviors and attributes should be placed in separate objects (some of which may also be Entities)
- See example page 94

Designing the Identity Operation

- Each Entity must have a way of establishing its identity
  - Such that two instances of the same entity can be distinguished from one another, even if they both contain the same descriptive attributes (like our bank deposits from slide 11)
- Identity is often operationally established by
  - ensuring that a single attribute has a unique id
  - or ensuring that some combination of attribute values always produce a unique key
- Often the means for establishing identity require a careful study of the domain; what is it that humans do to distinguish the real-world counterparts of the entity object?

Value Objects (aka Values)

- Some objects have no conceptual identity; these objects describe some aspect of a thing
  - A person may be modeled as an Entity with an identity, but that person's NAME is a Value object
- Values are instantiated to represent elements of a design that we care about only for WHAT they are, not WHO they are
  - Example values
    - Colors, Dates, Numbers, Strings, etc.
- Values are immutable; once created their values do not change
  - create values via factory methods; do not provide setter methods
  - operations that manipulate values produce new values as a result
  - Benefits: such objects can be easily shared
- See example on page 99

© University of Colorado, Boulder, 2005

Services

- In some situations, the clearest and most pragmatic design includes operations that do not conceptually belong to a single object; Rather than force the issue, we can follow the natural contours of the problem space and include SERVICES explicitly in the model
  - Slippery slope: if you give up too often on finding a home for an operation, you will end up with a procedural programming solution
  - On the other hand, if you force an operation into an object that doesn't fit that object's definition, you weaken that object's cohesion and make it more difficult to understand

Services, continued

- A service is an operation offered as an interface that stands alone in the model; it is defined purely in terms of what it can do for a client
- Services tend to be named for what they can do (verbs rather than nouns)
- A good service has three characteristics
  - The operation relates to a domain concept that is not a natural part of an entity or value object
  - The interface is defined in terms of other elements of the domain model
  - The operation is stateless (does not maintain or update its own internal state in response to being invoked)

Modules

- Modules are groupings of model elements; They provide two views on a model
  - one view provides details within an individual module
  - the second view provides information about relationships between modules
- We shoot for modules with high cohesion and low coupling
  - high cohesion: elements within a module all support the same purpose
  - low coupling: elements within a module primarily reference themselves; references to objects outside the module are kept to a minimum

What's Next

- Review the material of Chapter 6 of Domain-Driven Design
  - Life Cycles of Domain Objects
    - Aggregates
    - Factories
    - Repositories