Lecture 23: Domain-Driven Design (Part 1)

Kenneth M. Anderson

Object-Oriented Analysis and Design

CSCI 6448 - Spring Semester, 2005

Goals for this lecture

- Introduce the main concepts of Domain-Driven Design
  - Model-Driven Design
  - Ubiquitous Language
- Present examples that illustrate these concepts

- Goals for the Domain-Driven Design Lectures
  - Cover chapters 1-7 and parts of chapter 10 in 4 lectures!

Domain-Driven Design

- Eric Evans, the author of Domain-Driven Design, has been programming and designing systems for 25 years
- He asserts that for most software projects, the primary focus should be on the domain and domain logic
- "Domain-driven design flows from the premise that the heart of software development is knowledge of the subject matter and finding useful ways of understanding that subject matter. The complexity that we should be tackling is the complexity of the domain itself -- not the technical architecture, not the user interface, not even specific features. This means designing everything around our understanding and conception of the most essential concepts of the business and justifying any other development by how it supports that core."
  From <http://domaindrivendesign.org/articles/blog/evans_eric_ddd_and_mdd.html>

Model-Driven Design

- Domain-Driven Design leads to Model-Driven Design since developers capture their knowledge of a domain in models
- In particular
  - A model-driven design is software structured around a set o domain concepts
    - A model-driven design for a UI framework is one in which UI concepts, such as windows and menus, correspond to software constructs
  - Model-driven design embeds a domain model into the very fabric of a software system.
    - This creates a feedback loop between learning about a domain and implementing a system that addresses a problem in that domain
  - Teams who embrace model-driven design are aware that a change to the code IS a change to the model (and vice versa)

Model Benefits in DDD

- The model and its implementation shape each other
  - The link between model and code makes the model relevant and ensures that the work spent on analysis is not wasted
    - Indeed, the results of your analysis are present in the developed system!
- The model creates a language used by all team members
  - The language spreads domain knowledge throughout a team
  - It allows developers to speak to domain experts (e.g. users) without translation
- The model is distilled knowledge
  - Analysis is hard; we need to capture the information we get from it
  - The model is the team's agreed-upon way of structuring domain knowledge and and distinguishing the elements of most interest

Tackling Complexity

- The heart of software is its ability to solve domain-related problems for its users
  - Software functionality either solves a domain-related problem or performs a domain-related task
    - All other features support these two goals
- When a domain is complex, it becomes difficult to accomplish this
  - To be successful, developers must steep themselves in the domain
- Problem: Most developers are not interested in this!
  - Domain work is messy and demands a lot of knowledge not necessarily related to computer science
  - Developers enjoy quantifiable problems that exercise their technical skills!
- Evans asserts that domain complexity has to be tackled head-on by developers or they risk irrelevance!

Crunching Knowledge

- Domain modeling requires processing (crunching) knowledge
  - In the same way that financial analysts crunch numbers to, e.g., understand the quarterly performance of a corporation
- While speaking with domain experts, a domain modeler will
  - try one organizing idea (set of concepts) after another
  - create models, try them out, reject some, transform others
- Success is achieved when the modeler has created a set of abstract concepts that makes sense of all the details provided by the domain experts
  - Domain experts are a critical part of the process
  - Without them, developers tend to create models with concepts that seem naive and shallow to domain experts

Ingredients of Effective Modeling

- Bind the model and the implementation
  - As a model is developed, create rapid prototypes that test the domain
    - These prototypes contain primarily domain concepts
      - no UI, no persistence, no infrastructure
  - Chapter 3 begins with a story of what happens when you don't do this!
- Cultivate a language based on the model
  - Domain experts teach you their concepts and vocabulary
  - You teach them the basics of class diagrams and sequence diagrams (via the use of lots of examples)
  - Eventually either side "can take terms straight out of the model, organize them into sentences consistent with the structure of the model, and be unambiguously understood without translation"

Ingredients of Effective Modeling, continued

- Develop a knowledge-rich model
  - The objects of your model should have behavior and may follow rules
  - The model should not be just a data schema (think class diagram) but should express behavior, constraints, etc. that help solve domain-related problems
- Distill the model
  - It will be easy to add concepts to a model but more important is learning to remove concepts that are no longer useful
- Be willing to brainstorm and experiement
  - Try out variations of the model with rapid prototypes; think of the model as a laboratory that can enable domain-related experiments
    - Use the results of those experiments to evolve the model

Example: PCB Design

- Chapter 1 contains a brief example of domain-driven design when the author was working with a team of printed-circuit board designers
  - Evans knew nothing about electronic hardware
  - Initial conversations with the designers were difficult
    - Evans was not an electrical engineer and didn't understand the designers concepts or skills
    - The designers were not software developers and could not explain the functionality they needed very well
  - The first breakthrough came with Evans noticing that the concept of a "Net" kept appearing in the reports that the designers wanted from the software system
    - A net on a PCB is a wire conductor that connects a set of components and can carry an electric signal to each component its connected to

Example continued

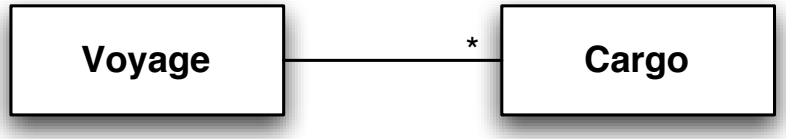
- This “breakthrough” led to a sequence of class diagrams and interaction diagrams that eventually led to a greater understanding of the domain PLUS identified a problem within the domain that needed solving
  - See pages 7 - 11 of domain-driven design
- The concepts included
  - component type (pin map), component instance, pin, net
- The problem was detecting the number of hops a signal caused across the components of a PCB
  - One concept, net topology, was dropped when the developer discovered it wasn’t needed to solve this problem

Knowledge-Rich Design

- To repeat, models capture more information than just the “classes” of a domain
  - As Evans says, it goes beyond the “find the nouns and verbs” heuristic of some object-oriented models
- In particular, models capture behavior, business activities, and business rules (policies, constraints, etc.)
- This makes knowledge crunching difficult since there may be inconsistencies between business rules
  - Domain experts (workers) can often detect these inconsistencies and apply “common sense” or create policies to deal with inconsistencies when they arise
    - Software can’t do this! (At least not easily!)

Example: Modeling Business Rules

- Consider a simple domain: Booking cargo on ships



- The associated application might have code that looks like this:

```
public int makeBooking(Cargo cargo, Voyage voyage) {
   int confirmation = getOrderNumber();
   voyage.addCargo(cargo, confirmation);
   return confirmation;
}
```

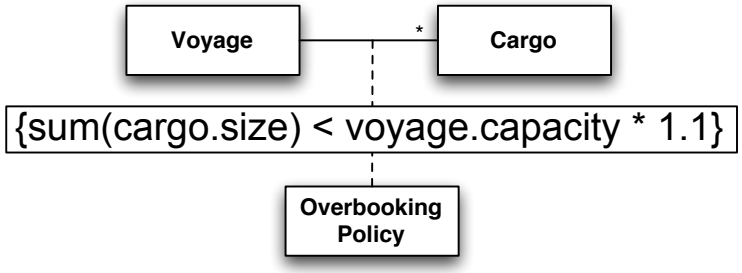

Example, continued

- A standard practice in the shipping industry is to accept more cargo than a particular ship can carry on voyage
  - This is know as “overbooking” and is used because there are typically always last-minute cancellations
- One way of specifying the overbooking policy is via a percentage of the ship’s total capacity
- So, if a requirements documents says “Allow 10% overbooking” our code might change to look like this

```
public int makeBooking(Cargo cargo, Voyage voyage) {
   double maxBooking = voyage.capacity() * 1.1;
   if ((voyage.bookedCargoSize() + cargo.size()) > maxBooking)
      return -1;
   ...
}
```

Problem: Hidden Knowledge

- The problem with this approach is that the overbooking policy is not reflected in our model
  - The policy has been recorded in a requirements document and hidden away inside one method in our application
- Domain-Driven Design recommends making that knowledge a part of the model: **change the model, change the code (and vice versa)**



Example, continued

- Our code would now look like this

```
public int makeBooking(Cargo cargo, Voyage voyage) {
   if (!overbookingPolicy.isAllowed(cargo, voyage)) return -1
   ...
}
```

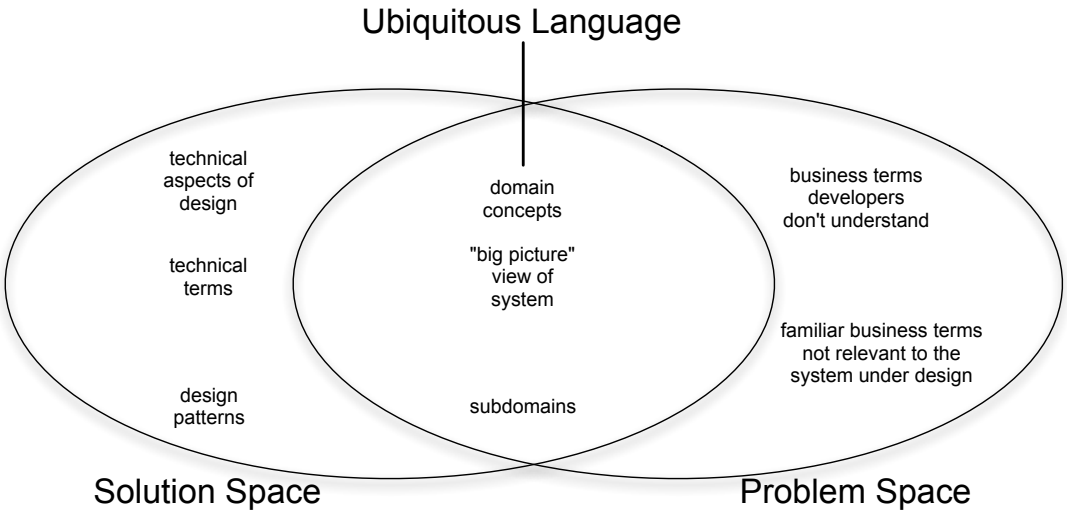
- and the OverbookingPolicy class would have this method:

```
public boolean isAllowed(Cargo cargo, Voyage voyage) {
   return (cargo.size() + voyage.bookedCargoSize()) <=
      (voyage.capacity() * 1.1);
}
```

Ubiquitous Language

- A core piece of domain-driven design is the symbiotic nature of the model being created for your software system and the language used by developers to discuss the system under design
  - Evans states
    - Use the model as the backbone of a language. Commit the team to exercising that language relentlessly in all communication within the team and in the code. Use the same language in diagrams, writing, and especially speech
    - Iron out difficulties by experimenting with alternative expressions, which reflect alternative models. Then refactor the code, renaming classes, methods, and modules to conform to the new model
  - This allows developers to use their linguistic abilities to aid software design, resolving ambiguity or confusion over terms in conversation
    - See example on pages 27 to 30

Merging Jargons



Using Language to Refine a Model

- When working on a model, speak the relationships out loud until you've captured requirements in a concise way
- Three examples (from bad to good)
  - If we give the **Routing Service** an origin, destination, and arrival time, it can look up the stops the cargo will have to make and, well... stick them in the database.
  - The origin, destination, and so on... it all feeds into the **Routing Service**, and we get back an Itinerary that has everything we need in it
  - A **Routing Service** finds an **Itinerary** that satisfies a **Route Specification**
- Once you've got a statement like the last, you can supplement with additional statements
  - A **Routing Service** needs an origin, a destination, and an arrival time

One Team, One Language

- Developers will want to hide the model from the domain experts
  - The model is too abstract for them
  - They don't understand objects
  - We have to collect requirements in their terminology
- Recall that the model is supposed to describe the user's domain!
  - If sophisticated domain experts don't understand the model, something is wrong with the model!
  - Indeed, the domain experts will be your best resource for testing the adequacy of a model
    - They will quickly find problems, notice ambiguity, identify naive assumptions, etc.

Documents and Diagrams

- Diagrams are good for
  - anchoring discussions about the model
  - brainstorming changes to the model
- But diagrams can't capture everything
  - attributes and relationships tell half the story because behavior and constraints are not easily illustrated
    - You can diagram some behaviors but you can't diagram all of them!
- "The vital detail about the design is captured in the code!"
- Documents are needed to capture decisions in large groups but beware entropy! Documents often get left behind!
  - Good documents provide background and record the thinking behind decisions; they should not try to replace code's ability to provide detail
  - See example page 42

What's Next

- Review the contents of Part II of Domain-Driven Design
  - The Building Blocks of a Model-Driven Design
    - Layered Architecture
    - Entities, Value Objects, Services, Modules
    - Life Cycles of Domain Objects
      - Aggregates
      - Factories
      - Repositories