Domain-Driven Design

By Bill Irelan

CSCI 5448

Spring 2011

First, a little background

- A domain is a collection of knowledge and ways to solve problems.
- For example: a bridge-building domain may consist of knowledge about properties of various materials, stresses, strains, compressive forces, and vectors.
- Over many years, bridge builders have developed good ways of solving common problems and complying with government safety regulations.

What is domain-driven design?

- Domain-Driven Design (DDD) is the idea of a very tight coupling between a model of the domain, such as an activity diagram or use case, and the software.
- Another central idea of DDD is learning the vocabulary of a domain and using the vocabulary to communicate.

Relationship between the model and the software

- A change in one implies a change in the other: if the model of the domain changes, the software also changes.
- The reverse is also true: if the software changes, the model must change.

Changing one without the other

- Changing only the model means the software gets left behind as the understanding of the domain evolves.
- Changing only the software calls into question the relevance of the software to the user's needs.
- The model and software influence each other through several iterations and need to evolve together.

A word about vocabulary….

- In DDD, the vocabulary of the domain is used everywhere: conversations about the software, in diagrams, class names, method names, and variable names.
- When a set of words means the same thing to everyone on a project, communication is easier. Misunderstandings happen less often.
- Concepts in the domain map directly into the code – it's easy to look at the code and understand what's going on.
- Code maintenance is easier because you can figure out where to look to do a bug fix or an enhancement.

Why should computer scientists bother to learn about a domain?

- Writing good software requires an understanding of what the user needs and the problems they're trying to solve.
- Forcing users to map their domain into something different raises their learning curve and makes them less productive. In the worst-case scenario, they get frustrated with your software, stop using it, and tell all their friends and bosses about their bad experience.

I’m a computer scientist, not a ….

- On Star Trek: The Original Series, one of Dr. McCoy’s catchphrases was “I’m a doctor, not a (bricklayer, physicist, moon-shuttle conductor, engineer, coal miner, ….)”
- That’s true – you’re a computer scientist and not an expert in another domain. But if you learn enough to meet a user on their turf and understand how to solve the problems they face in a way that makes sense to them, you will have written good software that helps them. They will want to keep using you! And continued employment is a good thing.

Wait! I didn’t sign up for this.

- Learning about a domain can feel messy, not very relevant, and frustrating.
- That’s normal – but it’s also a great opportunity.
- If you like to learn, writing software is a way to sample what other domains are like. And maybe you’ll discover the domain is more interesting than you thought!

A personal experience using DDD concepts

- My experience happened before Eric Evans published his book in 2004, but in retrospect we used DDD design principles.
- While at IBM in the mid 1990's, I was selected to lead a team of five software developers to fulfill a $10,000,000 contract with a business partner (let's call them Company XYZ.)
- XYZ ran printing shops all over the United States, much like Kinko's, only they were for large corporate customers. Need 360,000 utility bills printed? No problem!

What was the goal?

- XYZ needed a software system to manage their printing.
- The system needed to be very simple so it was easy for their print shops to manage the workload and meet deadlines.

Immersion in the domain

- To gain an understanding of the domain, I talked with lots of people at XYZ who knew how the print shops worked.
- I also talked with operators that ran the print shops. Some of them didn't even have a high school education, but that wasn't important. What **was** important is that they knew their domain very well.
- I needed to respect that expertise, learn from it, and work with them to make their jobs easier.

How the story ended

- After a year of development, the first version of the software was delivered on-time, on-budget, and XYZ was very satisfied.
- XYZ hired IBM again with a follow-on contract for another $5 million to add more features, which my team also delivered on-time and on-budget.
- I was quickly promoted at IBM and traveled to many places (even Europe!) to give presentations. Very cool!

How do I do DDD?

- First, find a domain expert. Talk with them and learn the vocabulary of the domain.
- Next, work with the domain experts to develop a model of the concepts in the domain and how they interact. The model can be textual, like a use case, but diagrams are often easier for people to understand. Use the vocabulary you have learned in the first step.
- Keep iterating with the domain experts on the model until you have a clear understanding of the problems the user needs you to solve.
- Translate the model into code with DDD practices. If you need to implement something different from the model, revisit the thinking for the model and change one or the other to keep them in agreement.

Diagram examples

- This example relates to shipping things overseas. Which diagram do you think is easier to understand for someone who knows about shipping? This one:

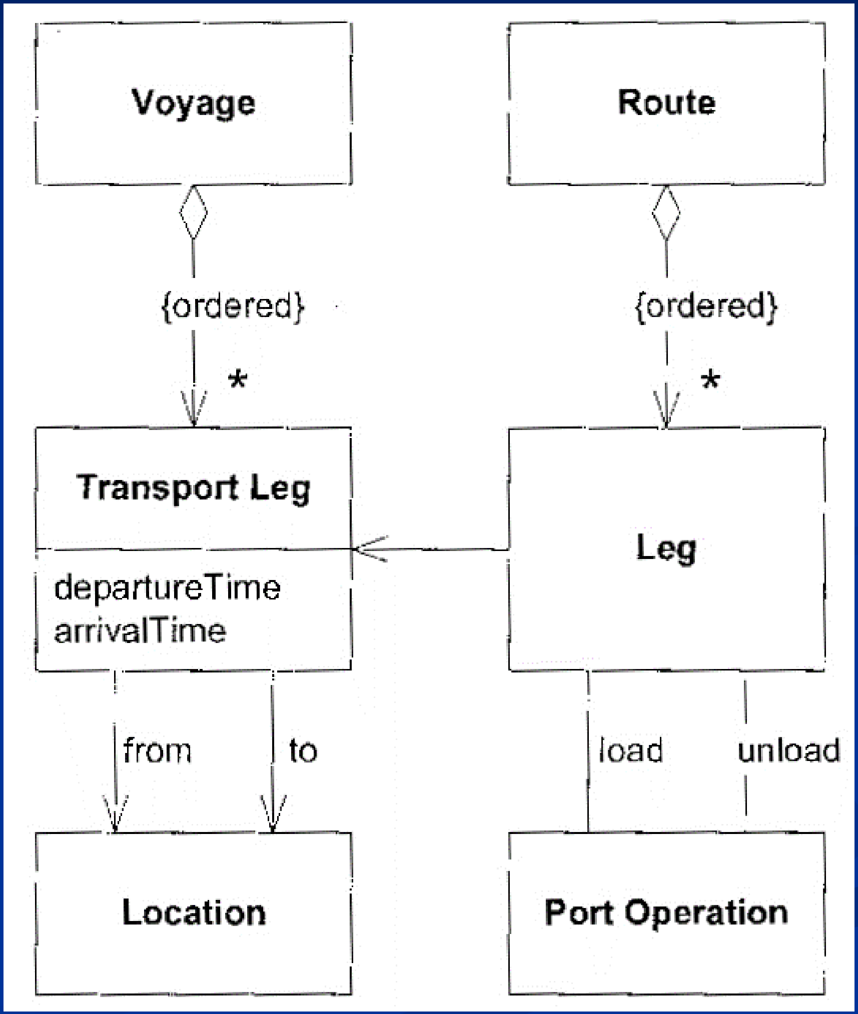

Diagram examples, con't.

- …or this one?

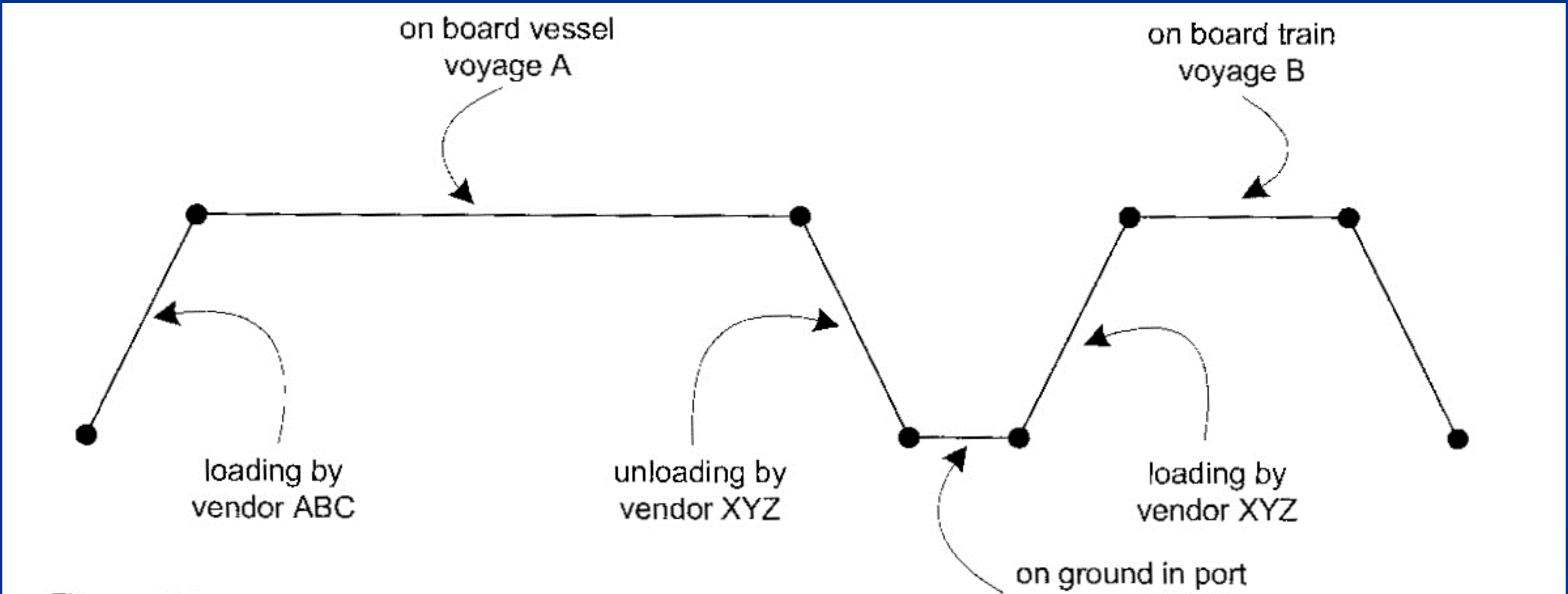

Figure 2.5

Explanatory Models

- The diagram on the previous slide is an explanatory model.
- It doesn’t have a format – the important part is to establish a way that you and the domain expert can connect with a shared vocabulary. Be creative! If the diagram is easily understandable, the diagram is correct.

Architectural patterns

- When you get to the coding phase, consider the Layers and Model-View-Controller (MVC) patterns.
- In the Layers pattern, the logic of the model is encapsulated in one of the levels.
- In the MVC pattern, the logic of the model is separated from its presentation.

Four basic DDD elements

- Once you have a model, start translating the concepts and relationships into DDD elements.
- There are four basic DDD elements: entity objects, value objects, services, and modules.

Entity Objects

- Entity objects are defined by a need for an identity. If the object needs to be distinguished from other instances of the same class, it is an entity object.
- Consider transactions in a checking account. A transaction could be an object, but transactions definitely need to be distinct from one another.

Value Objects

- Value objects are ones that have an unchanging set of properties – they do not need an identity.
- If you are building HTML and create an object to represent a color of red, that can be a value object. Many other objects can reuse the red color object because it doesn’t change.
- The distinguishing feature between an entity object and a value object is whether the object needs a unique identity.

Services

- Sometimes concepts in the domain don’t map to an entity or a value object. The concept is an action, not a thing. These concepts are services.
- Services are declared as a standalone interface – don’t graft it onto an entity or a value object if it doesn’t fit naturally.

Modules

- Modules contain a cohesive set of concepts from the model.
- Strive for low coupling between modules and high cohesion within a module.
- Modules denote a boundary for encapsulation – two separate sets of concepts that are internally cohesive should have a boundary between them.

What comes next?

- Entity and value objects are aggregated according to their roles in the model
- Use the Factory pattern to instantiate and encapsulate the aggregations when appropriate within the context of the model.

How to learn more

- There is much more to DDD - this presentation has covered the first 122 of 500 pages in Eric Evans' book, *Domain-Driven Design*. There are lots of fully fleshed-out examples in the book.
- There is a fully-functional Java example at http://dddsample.sourceforge.net/
- http://www.domaindrivendesign.org/ is also a good resource. This website hosts the DDD community's discussion threads and has a schedule for training sessions.

Parting thoughts

- DDD is a good strategy on how to collaborate with users and write maintainable software.
- Not all DDD concepts will map to the problem you're trying to solve – don't apply a DDD concept where it doesn't seem to fit.
- DDD is another tool to have in your software development toolbox. Have it ready to use when it seems right to use it.

Bibliography

- *Domain-Driven Design*, Eric Evans, Addison-Wesley publishers, 2004, ISBN 0-3121-12521-5
- The DDD community at http://www.domaindrivendesign.org/
- http://memory-alpha.org/wiki/I%27m_a_doctor,_not_a...