STEVENS
INSTITUTE of TECHNOLOGY
THE INNOVATION UNIVERSITY®

SSW 322: Software Engineering Design VI

Domain Driven Design (DDD)
2019 Spring Week 6

Prof. Lu Xiao
lxiao6@stevens.edu
Babbio 513
Office Hour: Monday/Wednesday 2 to 4 pm
Software Engineering
School of Systems and Enterprises

Today's topics

- Domain Driven Design
  - What is domain driven design?
  - Why domain driven design?
  - Core concepts of domain driven design

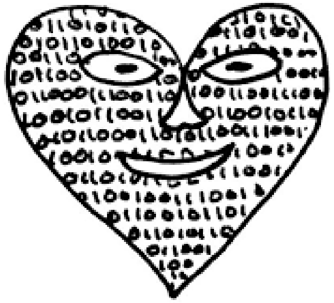
Acknowledgement:
The materials are from book:
Domain---Driven Design, Eric Evans

What is Domain-Driven Design (DDD)?

- Domain: a problem area
- The term DDD was invented by Eric Evans, the author of book Domain-Driven Design.

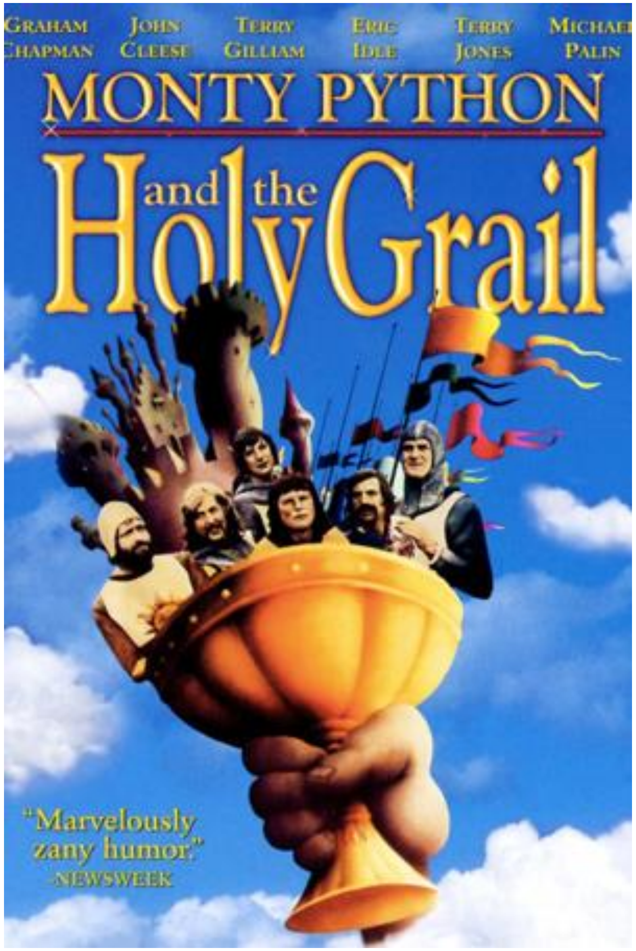
The heart of software is its ability to solve domain-related problems for its user.
---Eric Evans

- Concentrate on a domain and its logic;
- Establish design on domain models;
- Emphasize collaboration between technical developers and domain experts.



Why DDD?

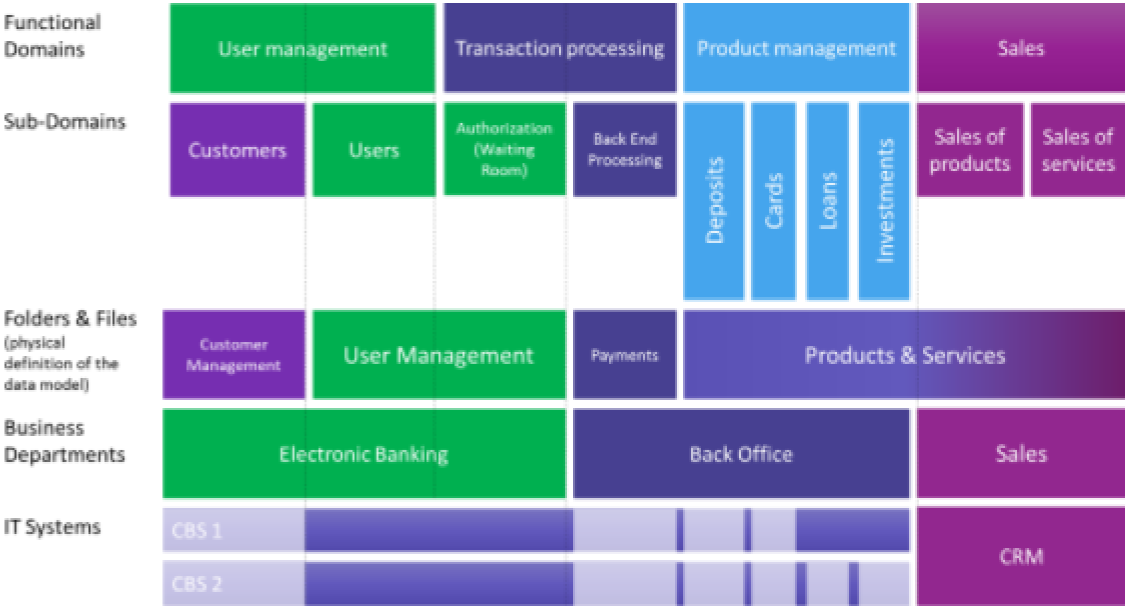
- A motivating example:
  - John Cleese and Michael Palin shot a funny take of a scene after over and over again trying (Comedian Domain: make it funny)
  - The film editor cut the final take because “a coat sleeve that was visible for a moment at the edge of the picture” (Film editor focused on the precise execution of his own specialty)



DDD Examples (1)

Online Banking System:

- User management
- Transaction processing
- Financial product management



http://www.servicetechmag.com/I84/0514-2

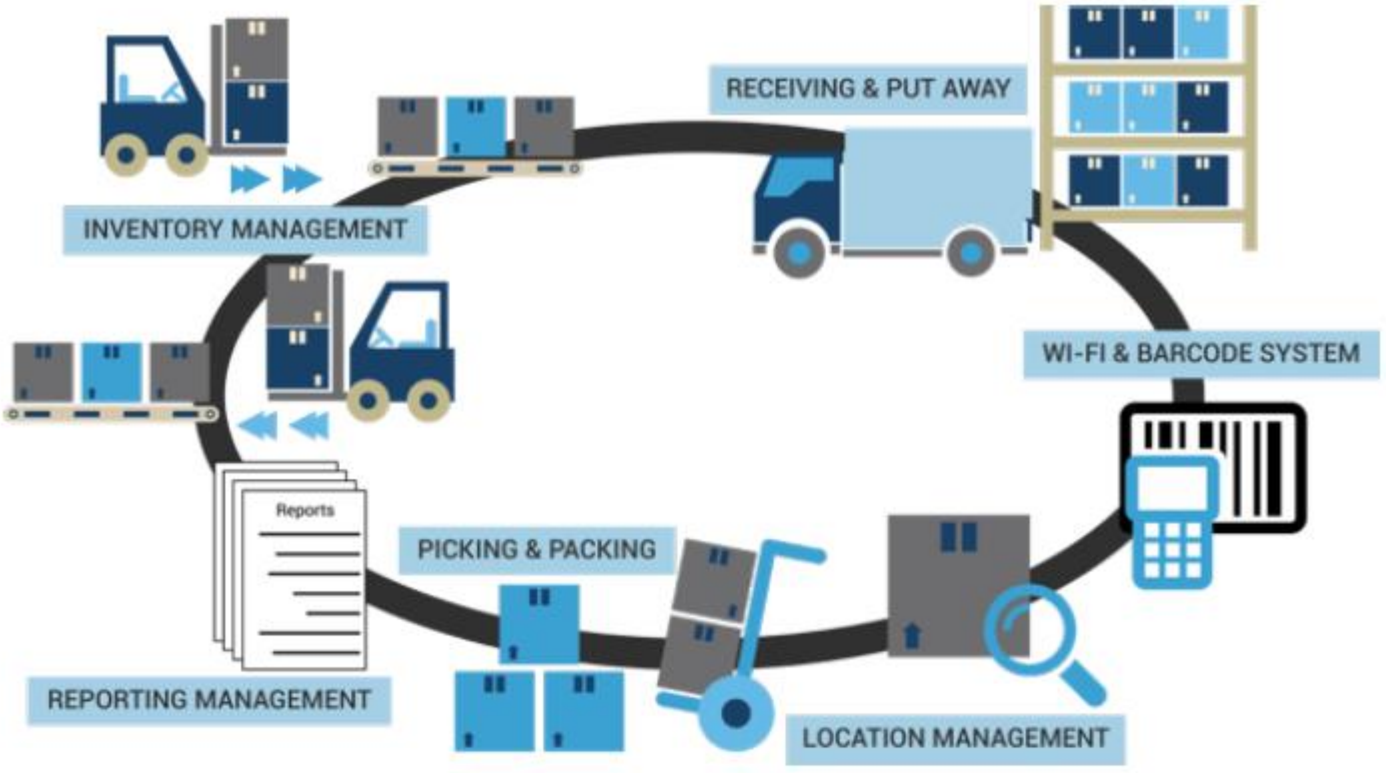
DDD Example (2)

Navigation System, e.g. Google Map

- Location: latitude, longitude, country, state, city, zip code, street
- name, street number
- Search for a path between departure and destination

DDD Example (3)

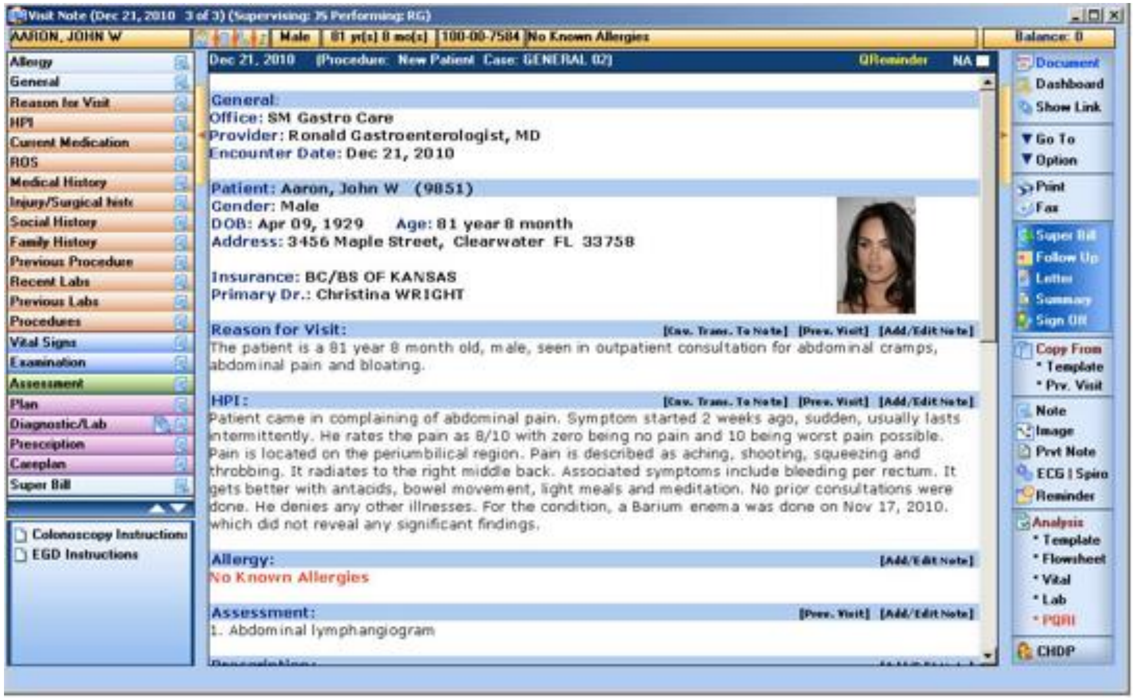
- Warehouse Management System, e.g. Amazon



DDD Example (4)

EMR (Electronic Medical Record) System

- User account information: name, age, insurance provider, etc.
- Medical History and Family Health History
- Visit Information



Visit Note (Dec 21, 2010 3 of 3) (Supervising: JS Performing: RG)

AARON, JOHN W | Male | 81 yr(s) 8 mo(s) | 100-00-7584 | No Known Allergies | Balance: 0

Dec 21, 2010 (Procedure: New Patient Case: GENERAL 02)

General:
Office: SM Gastro Care
Provider: Ronald Gastroenterologist, MD
Encounter Date: Dec 21, 2010

Patient: Aaron, John W (9851)
Gender: Male
DOB: Apr 09, 1929 Age: 81 year 8 month
Address: 3456 Maple Street, Clearwater FL 33758

Insurance: BC/BS OF KANSAS
Primary Dr.: Christina WRIGHT

Reason for Visit:
The patient is a 81 year 8 month old, male, seen in outpatient consultation for abdominal cramps, abdominal pain and bloating.

HPI:
Patient came in complaining of abdominal pain. Symptom started 2 weeks ago, sudden, usually lasts intermittently. He rates the pain as 8/10 with zero being no pain and 10 being worst pain possible. Pain is located on the periumbilical region. Pain is described as aching, shooting, squeezing and throbbing. It radiates to the right middle back. Associated symptoms include bleeding per rectum. It gets better with antacids, bowel movement, light meals and meditation. No prior consultations were done. He denies any other illnesses. For the condition, a Barium enema was done on Nov 17, 2010. which did not reveal any significant findings.

Allergy:
No Known Allergies

Assessment:
1. Abdominal lymphangiogram

DDD Core Concepts

- Model Driven Design
- Ubiquitous Language
- Layered Architecture
- Domain Elements: Entities, Value, Services, Modules
- Aggregates, Factories, Repositories

Model Driven Design

- A knowledge-rick domain model is a set of abstractions that depicts the objects, behaviors, and the enforced rules in a domain.
    - Distilling the model to drop inessential concepts
    - Binding the model and the implementation: crude prototype that forge the essential links early, and maintain it through iterations.
        - Modification to the code is a modification to the model.
        - Modification to model leads to immediate modification to the code.

Distilling Domain Model

- In a large system, there are so many contributing components, all complicated and all absolutely necessary, that the essence of the domain model can be obscured and neglected.
  - Not all parts of a design are going to be equally refined. **Priorities must be set**.
  - Boil the model down. **Find the CORE DOMAIN and make it SMALL.**

Binding Model and Implementation

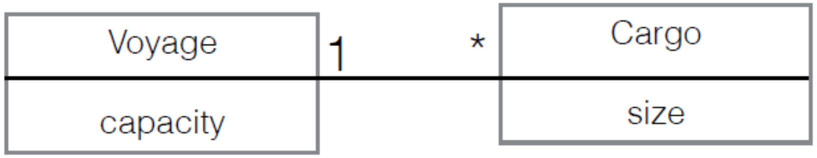
- Tightly relating the code to an underlying model gives the code meaning and makes the model relevant.
  - Code reflects the domain model, and becomes an expression of the model: change to code leads to a change to the model.
  - Revisit the model and modify it to be implemented more naturally in software/
- A modeling paradigm, *such OO programming*, helps the binding!

Knowledge Crunching

- Developers and domain experts collaborate:
  - Draw in information and crunch it into a useful form.
  - Abstract and develop domain model.
- The developers learn important principles of the business, rather than to produce functions mechanically.
- Domain experts refine and distill what they know to essential, and they come to understand the conceptual rigor that software projects require.
- Continuous learning: ***when we set out to write software, we never know enough***

Extract Hidden Concepts

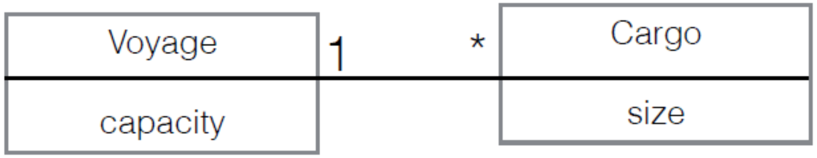
- A simple domain model could be the basis of an application, e.g. Booking cargos onto a voyage of a ship.



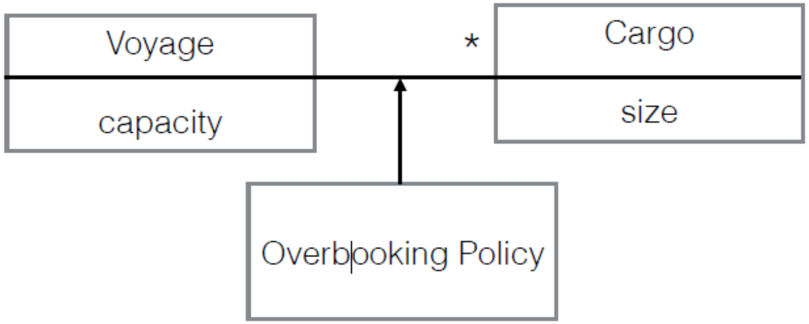
- A hidden concept: “overbooking”:
  - Last minute cancellations
  - Requirement: 10% overbooking

Extract Hidden Concepts

- A simple domain model could be the basis of an application, e.g. Booking cargos onto a voyage of a ship.



- A hidden concept: “overbooking”:
  - Last minute cancellations
  - Requirement: 10% overbooking



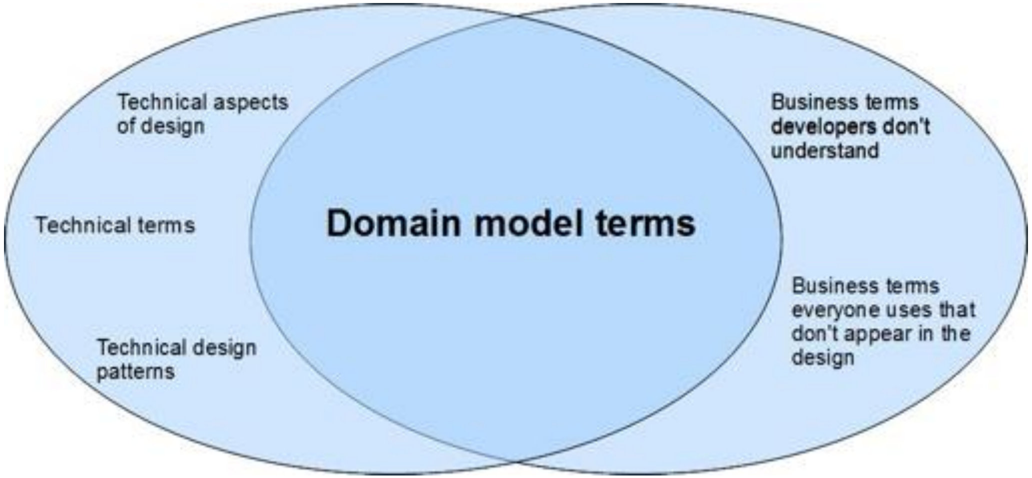
***Policy* is another name for the design pattern known as STRATEGY ---Gamma et al. 1995**

Communication and Use of Language

- Challenge:
  - Technical experts speaks technical terms
  - Domain experts use terminology in their field/domain.
- Solution:
  - Some standard language should be built among them!
- Goal: the shared vocabulary can be understood by both sides.

Ubiquitous Language

- Ubiquitous language refers to a common language structured around the domain model and shared between domain experts and technical developers.
  - Use the model as the backbone of a language (speech, diagrams, and code)
  - Change in the language is a change to the model (rename classes, methods, and modules)


Technical aspects of design
Technical terms
Technical design patterns
Domain model terms
Business terms developers don't understand
Business terms everyone uses that don't appear in the design

One Team, ? Language

- Question text, question prompt, question string, question title…
- Options, choice array, matching array, *answers*…
- Answer, correct answer, standard answer, user answer, response, answer sheet, solutions
- Survey, Test, Questionnaire, question set

One Team, One Language (Vocabulary)

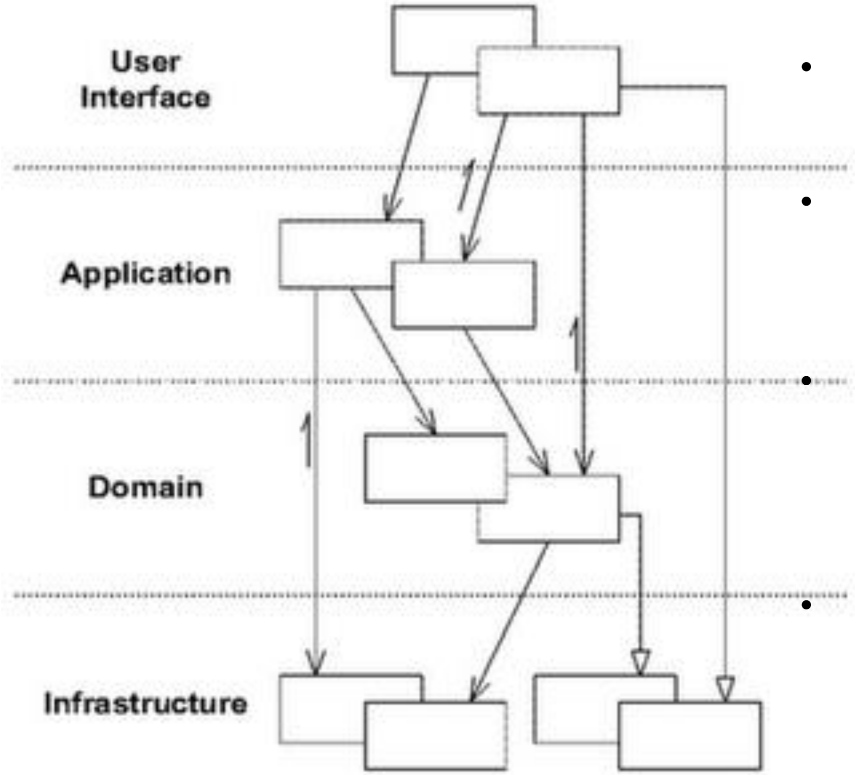
- **Question prompt**
- **Options**
- **Correct answer vs. user answer**
- **Answer sheet**

- Benefits of Ubiquitous Language:
  - Less risk of miscommunication
  - Faster and more efficient interaction
  - Knowledge of domain can reside in codebase
  - Source code easier to understand, maintain, and extend.

Layered Architecture

- DDD takes advantage of a layered architecture.
- Each layer address a different concern in the system.
- The basic principle of layered architecture:
  - Dependencies between layers is one-way, i.e. a lower layer never knows anything above. Within a layer, an object can use objects in this layer and in layers below it.
  - An indirect method is required when an object in a lower layer needs to have reference to an object in the layer above it.

A Typical Multi-layer Architecture



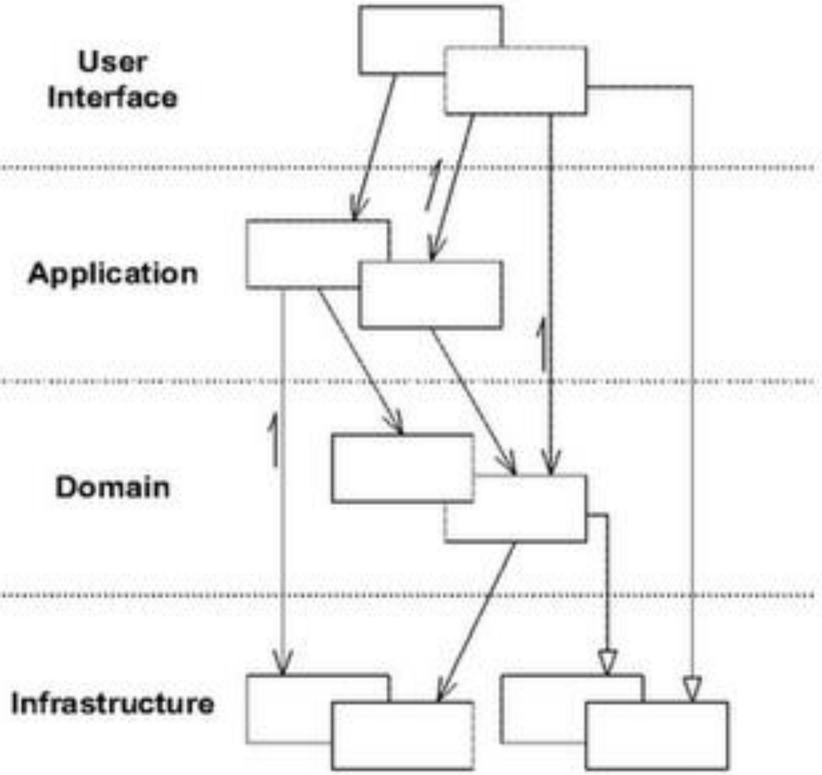
- Responsible for showing information to the user and interpreting the user's commands.

- No knowledge but coordinate tasks and delegates work to collaborations of domain objects in the next layer (Kept thin).

- Responsible for representing concepts of the business, information about the business situation, and business rules. ***This layer is the heart of business software.***

- Provides generic technical capabilities that support the higher layers: message sending for the application, persistence for the domain, drawing widgets for the UI, and so on.

The Essential Principle

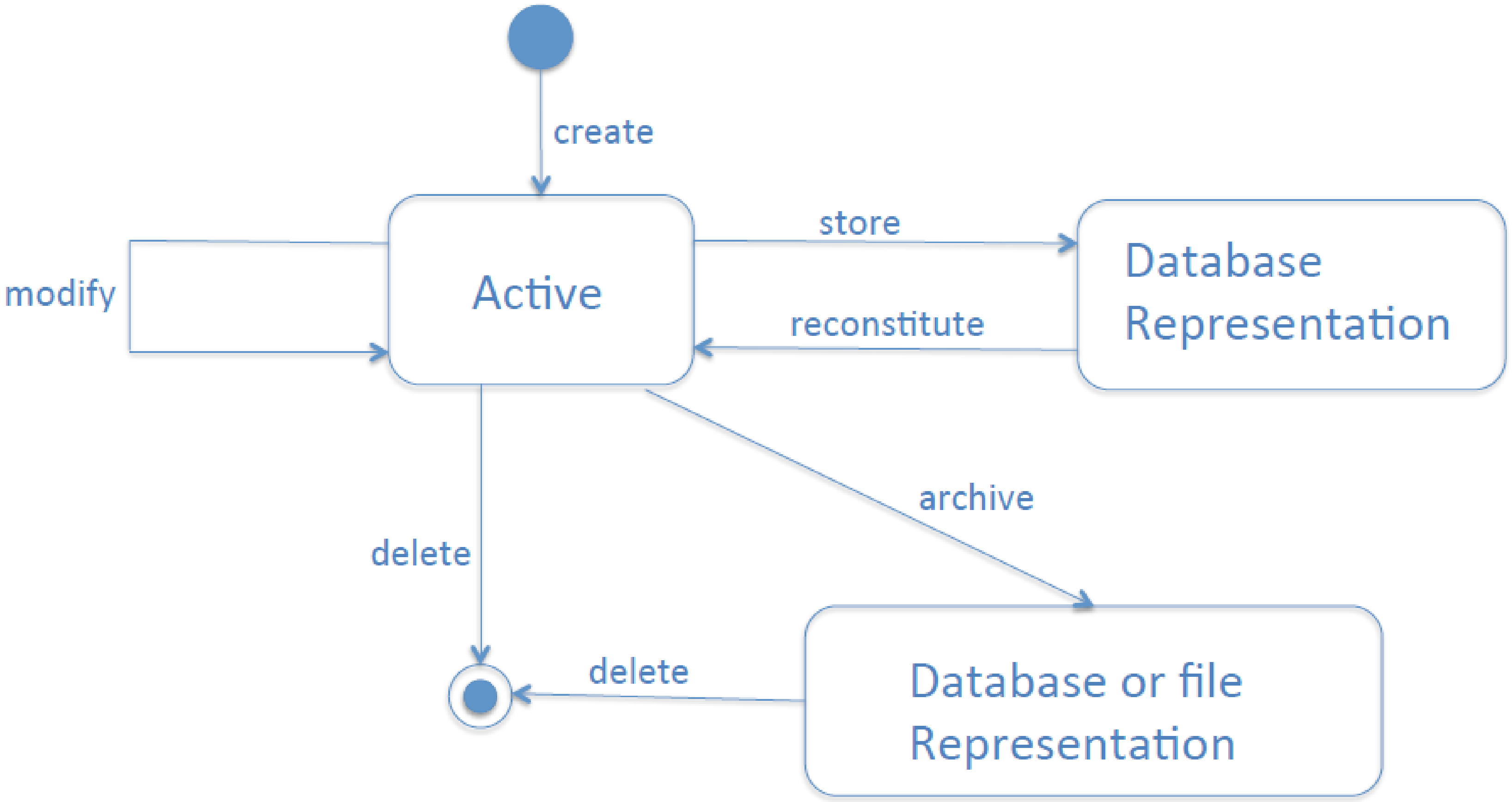


- Any element of a layer depends only on other elements in the *same layer* or elements of the *layers "beneath" it*.
- When *upward communicate* is needed, use mechanism such as callbacks or *OBSERVERS pattern* (Gamma et al. 1995)

Domain Elements (Building Blocks)

- Entities object: an object is distinguished by its identity.
  - For example, an application for booking seats in a stadium treat seats and attendees as ENTITIES.
- Value object: when you care only about the attributes of an element.
  - For example, a person may be modeled as an ENTITY with an identity, but that person’s name is a VALUE.
- Services: operations that do not conceptually belong to any object.
- Modules (e.g. packages) reflect concepts in the domain. Choose MODULES that tell the story of the system and contain a cohesive set of concepts.

Domain Object Life Cycle



On Page 123, Eric Evans shows a diagram for life of domain object.

Associations between Elements

- The interaction between modeling and implementation is particularly tricky with the associations between domain elements.
- A lot of associations may decrease maintainability: avoid unnecessary associations and use only a minimum number of them.
- Making association more controllable:
  - Prefer a traversal direction instead of bidirectional
  - Remove unnecessary associations

Aggregates

- A DDD aggregate is a cluster of domain objects that can be treated as a single unit.
- An aggregate will have one of its component objects be the aggregate root.
    - Any references from outside the aggregate should only go to the aggregate root.
    - The root can thus ensure the integrity of the aggregate as a whole.
- The term "aggregate" is a common one, and is used in various different contexts (e.g. UML), in which case it does not refer to the same concept as a DDD aggregate.

Factories

- Factories are a separate object (or interface) that in charge of creation for the instances of complex objects, and particularly aggregates.
- Three common design patterns related to factories:
  - Abstract factory
  - Factory method
  - Builder
- Creation in factories should be atomic.
  - If problems happens, exception message should be passed, instead of wrong results/
- Arguments are usually necessary

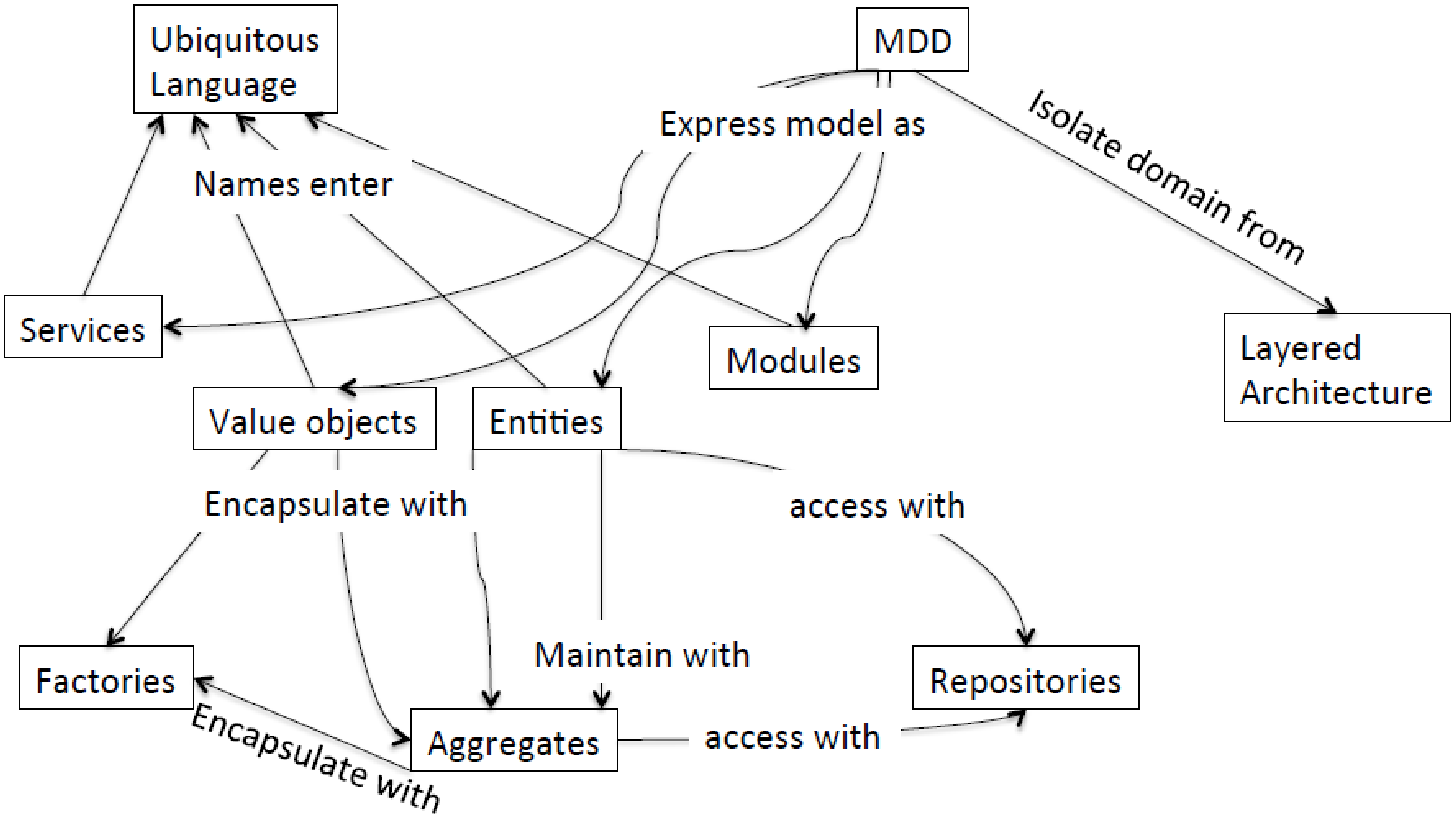
Repositories

- Developers need to get access to the domain objects. They can:
  - Use traversable associations
    - Complicates the model
  - Search the object from database
    - Domain logic degenerates into queries and searching
    - Entities becomes simple data container

Repositories

- Repositories represents all objects which meets certain requirement as a conceptual collections with advanced searching capability.
  - Addition/removal of certain objects are achieved by the algorithm behind the repository.
- Repositories provide the clients an easy interface to acquired an domain object and maintain its life cycle.
- Repositories decouple the client from the module from technical storage, or database storage.

Diagram of Building Blocks for DDD

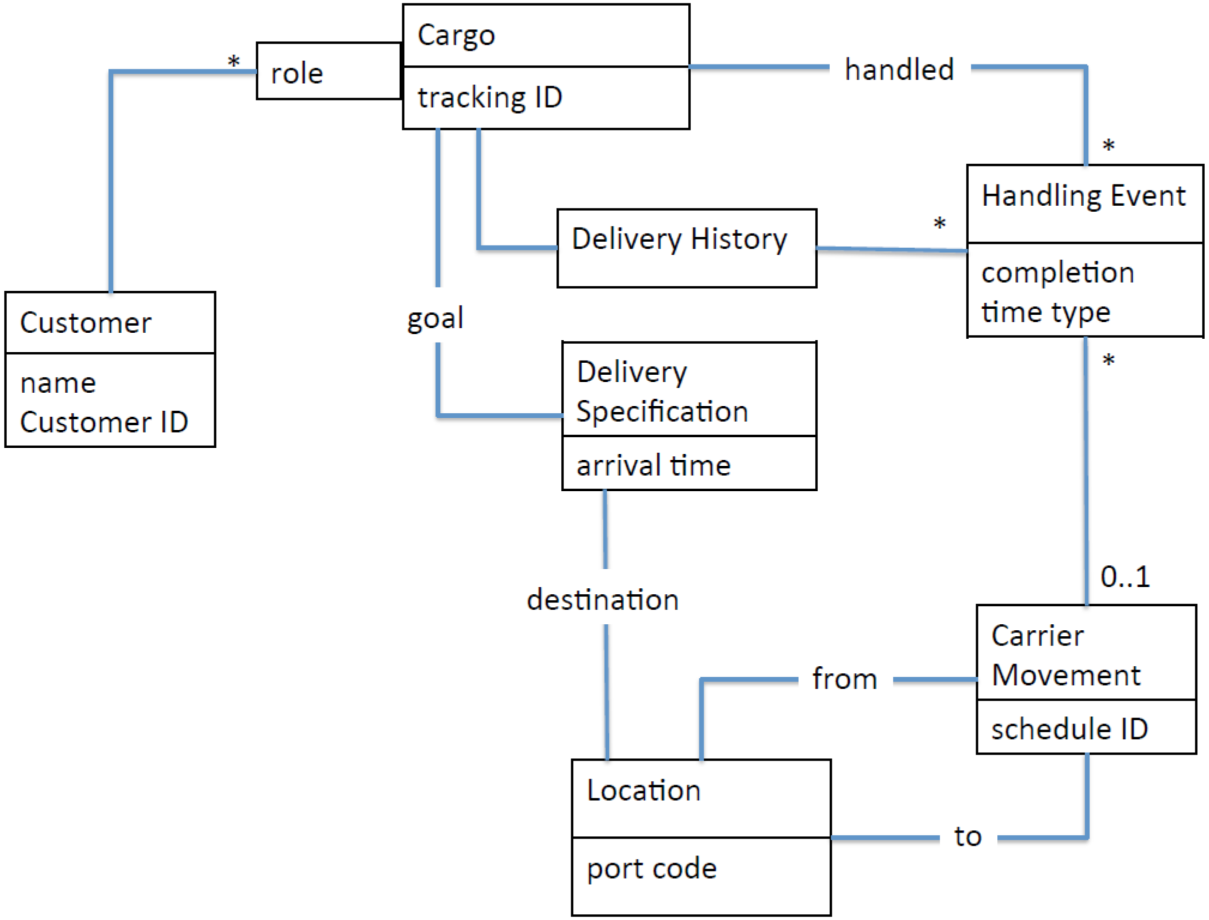


Cargo Example

Chapter 7 of Eric Evans' book gives an example of DDD.

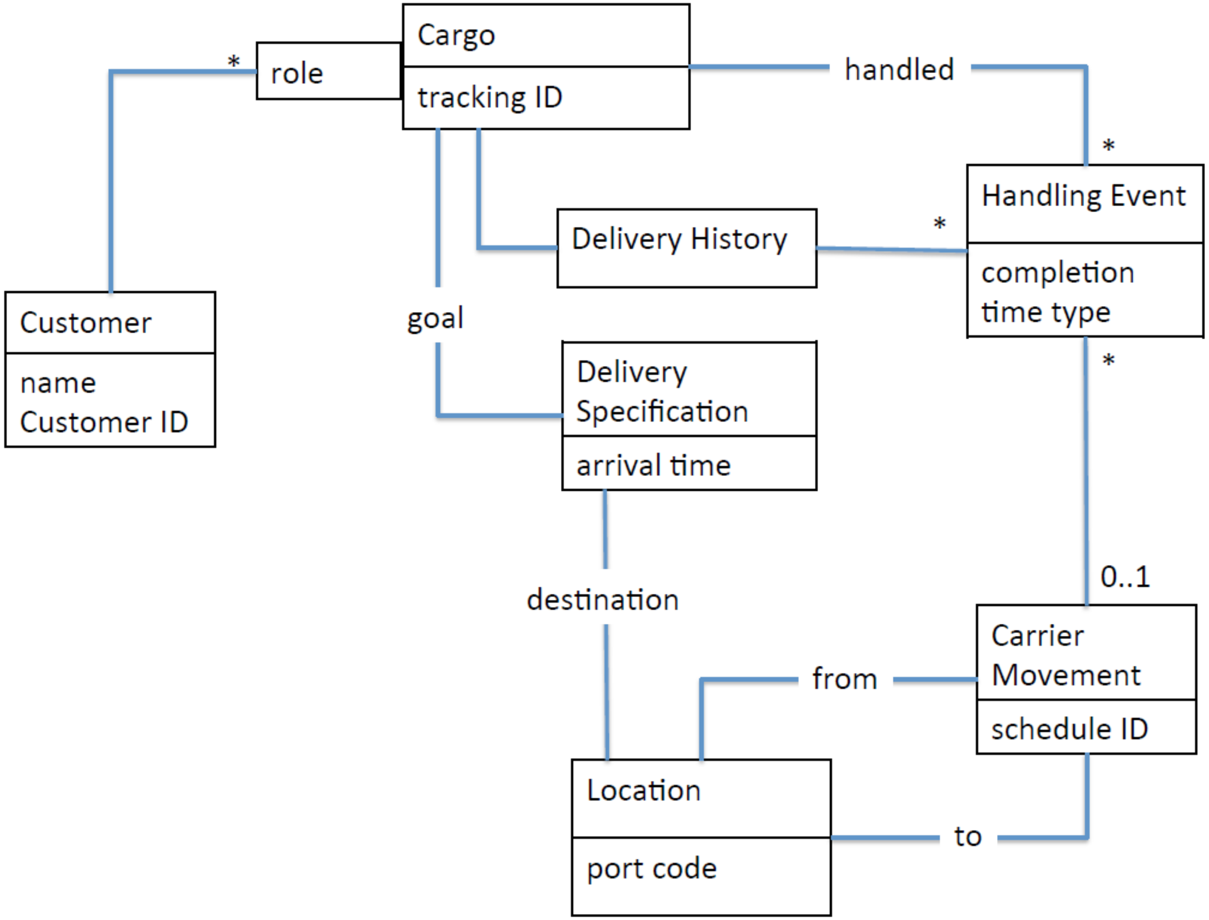
- Eric's team are developing new software for a cargo shipping company. The initial requirements are three basic functions:
  - Track key handling of customer cargo
  - Book cargo in advance
  - Send invoices to customers automatically when the cargo reaches some point in its handling

Shipping Domain Model



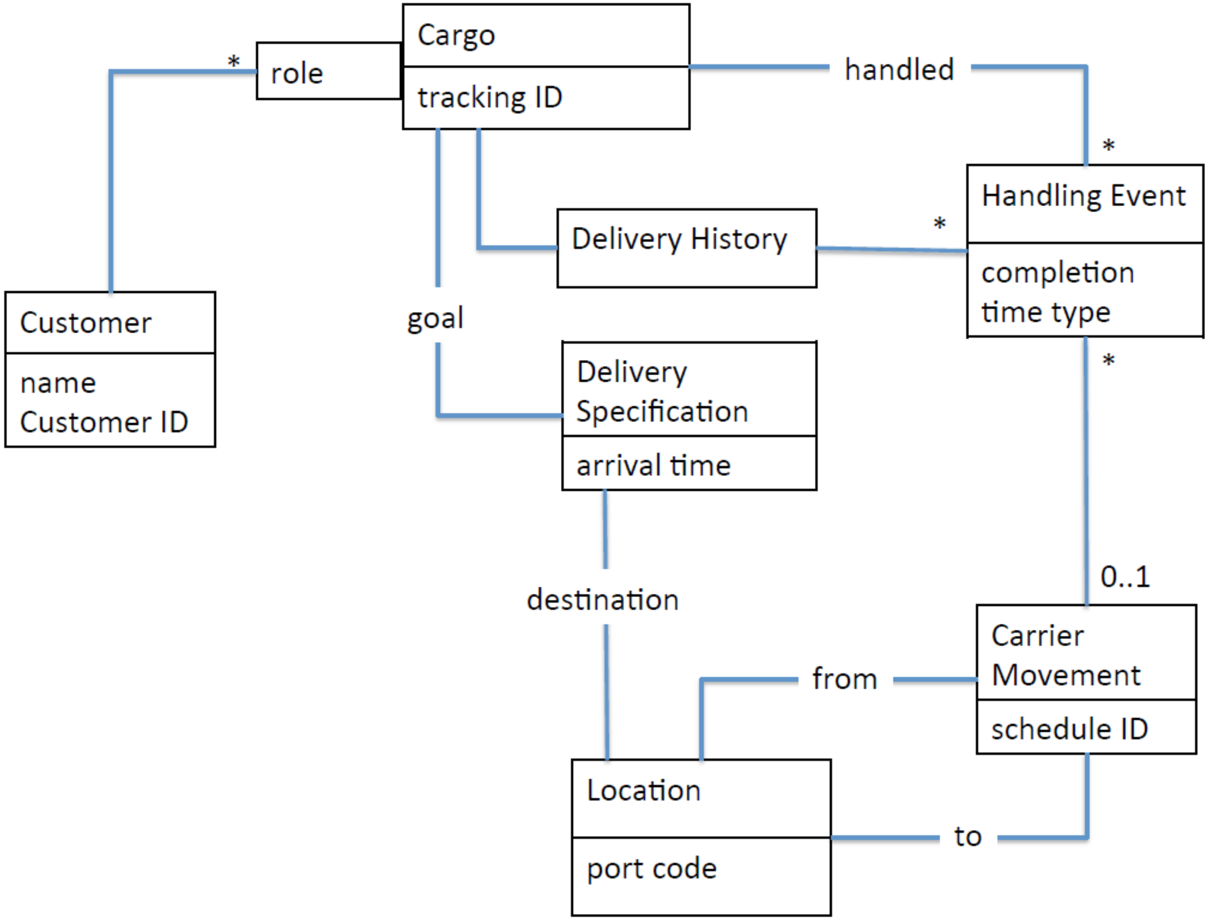
- All the concepts needed to work through the requirements are present in this model, assuming appropriate mechanisms to persist the objects, find the relevant objects, and do on.

Shipping Domain Model



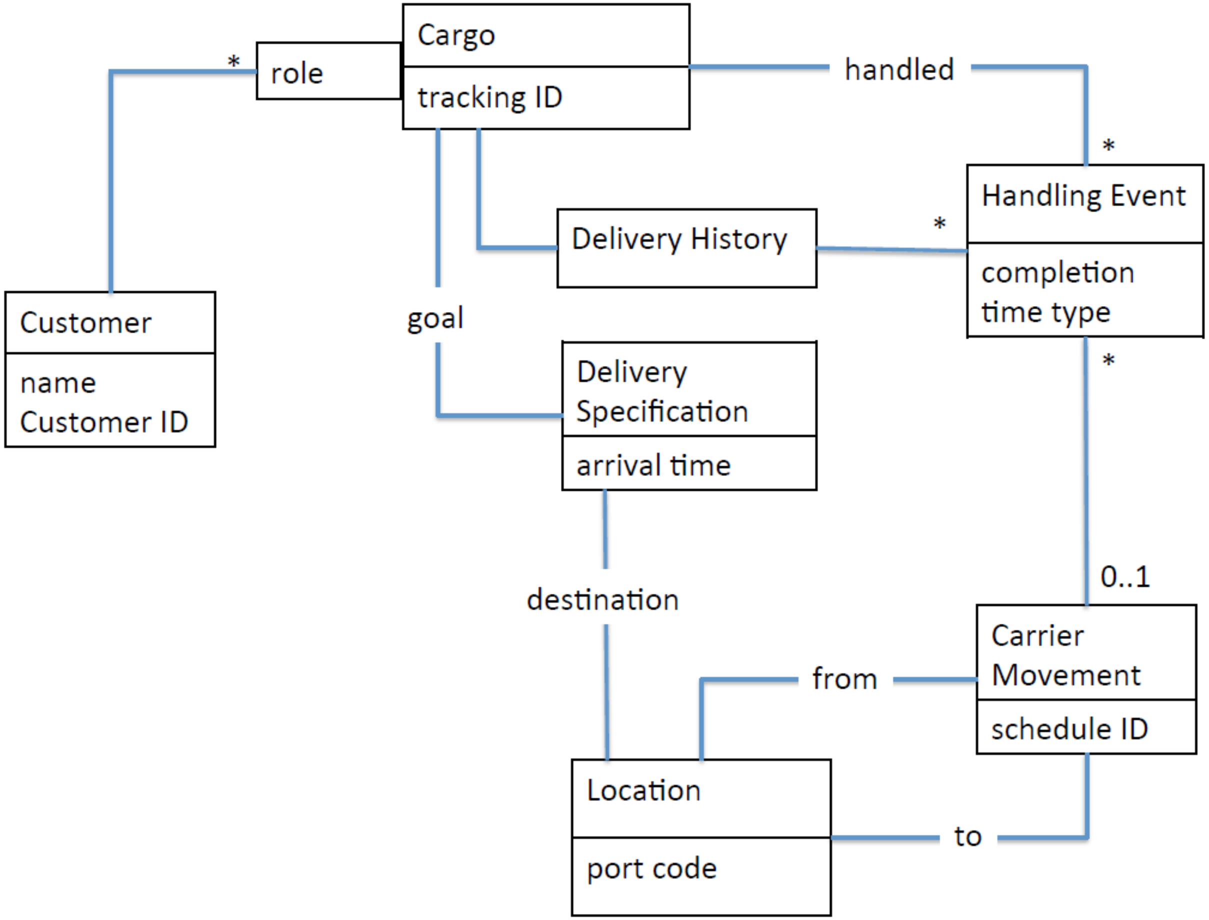
- Multiple customers are involved with a Cargo, each playing a different role
- The Cargo delivery goal is specified
- A series of Carrier Movements satisfying the Specification will fulfill the delivery goal

Shipping Domain Model



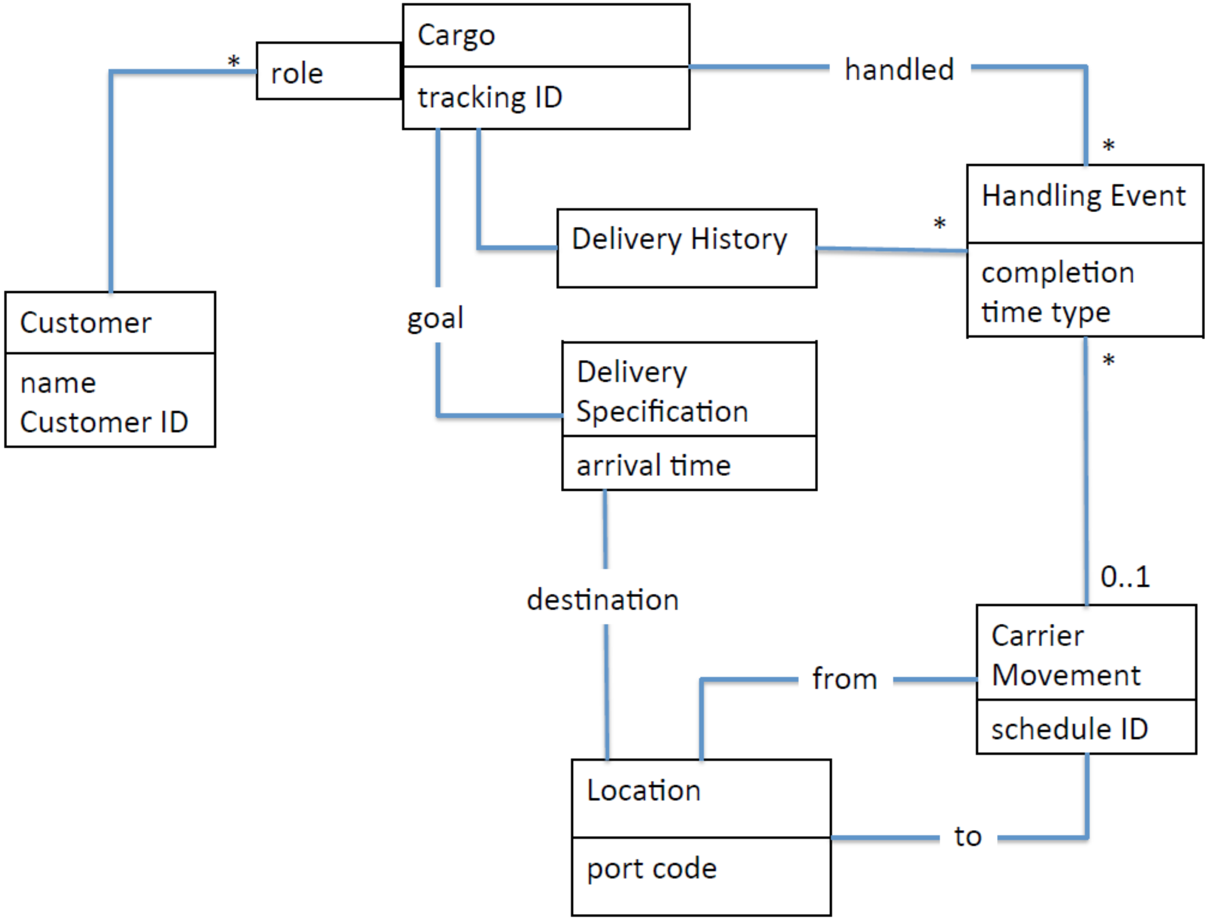
- A role distinguishes the different parts played by Customers in a shipment
  - Shipper
  - Payer
  - Receiver

Shipping Domain Model



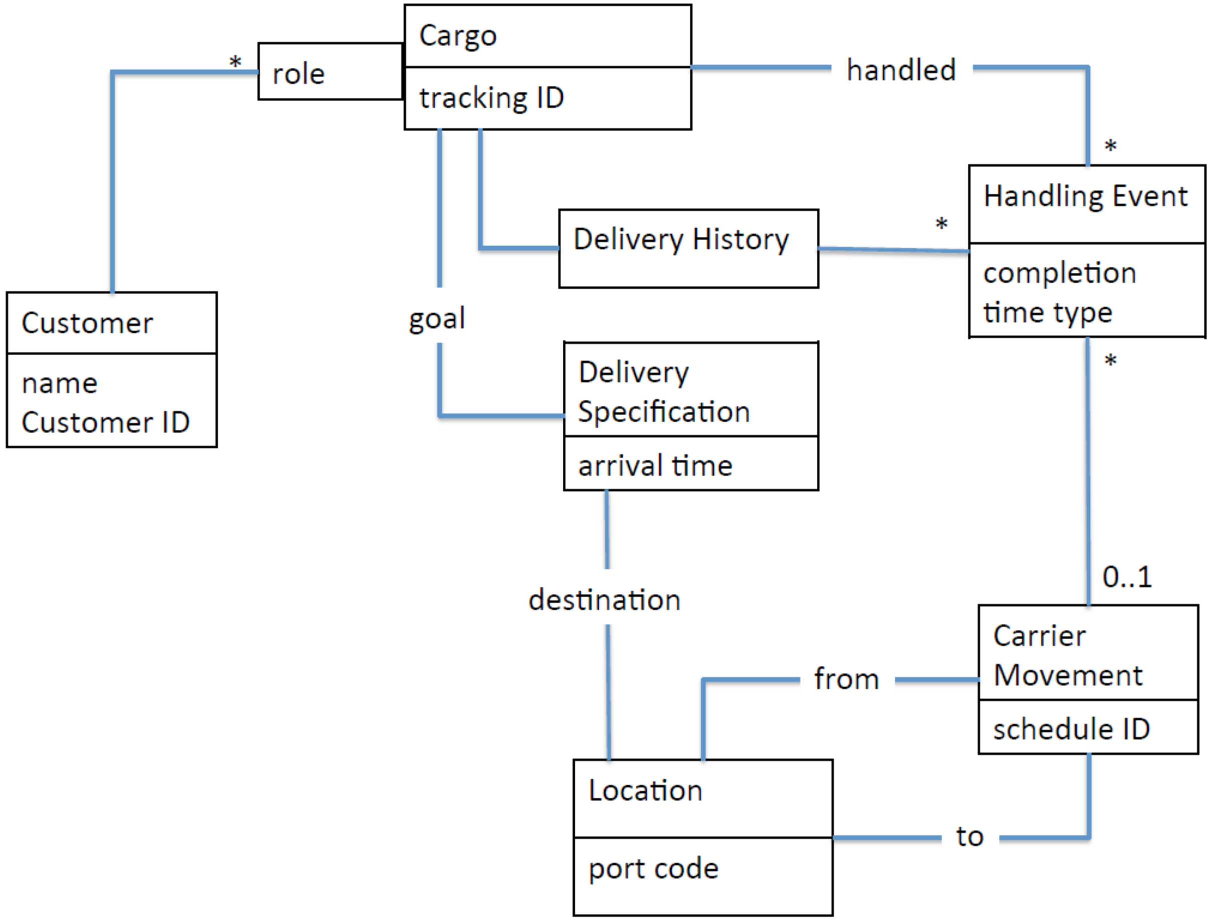
- A Handling Event is a discrete action taken with the Cargo
  - Loading
  - Unloading
  - Claimed by receiver

Shipping Domain Model



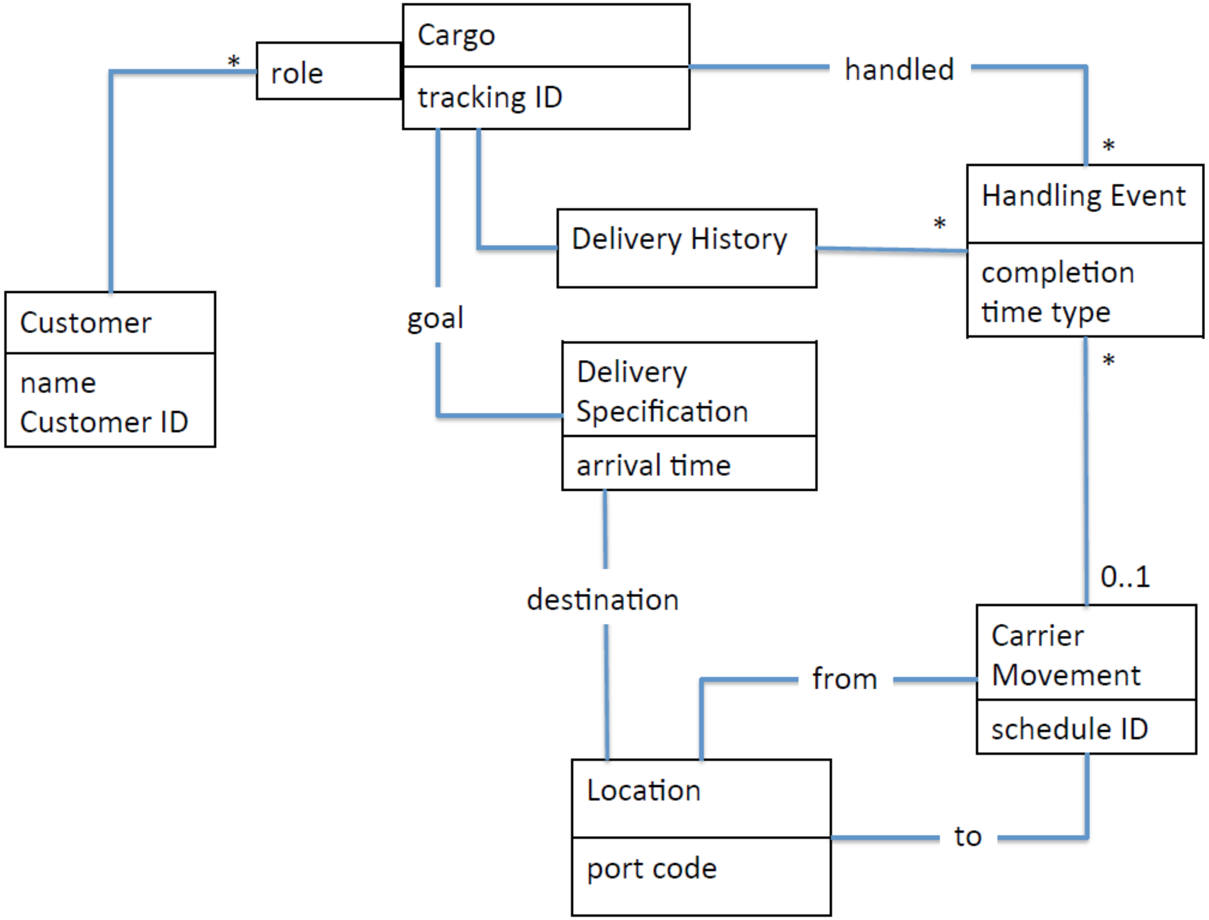
- A Handling Event is a discrete action taken with the Cargo
  - Loading
  - Unloading
  - Claimed by receiver
- *How this could be incorporated in the design?*

Shipping Domain Model



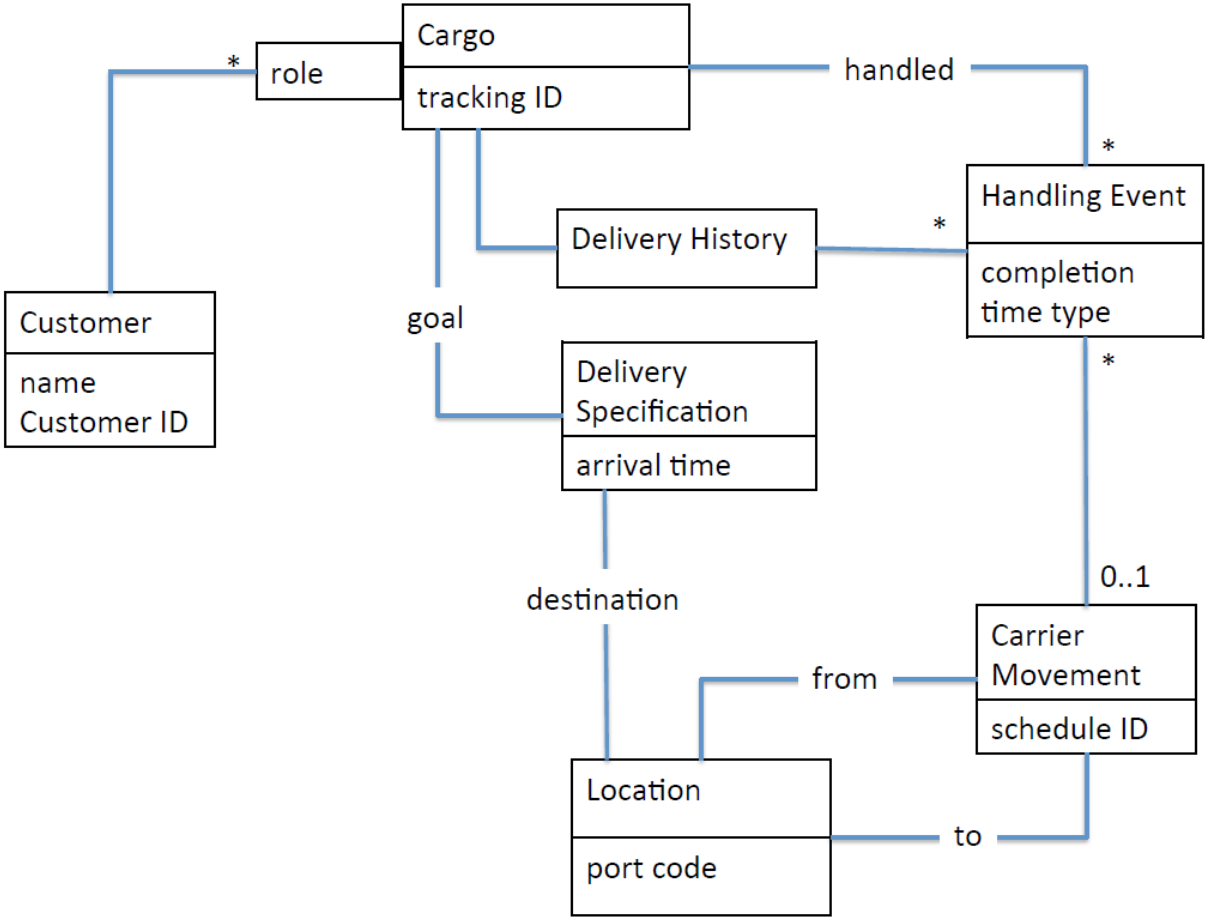
- Delivery Specification defines a delivery goal, which at minimum would include a destination and arrival date.
  - Separation of concern
  - Encapsulation
  - Expressive

Shipping Domain Model



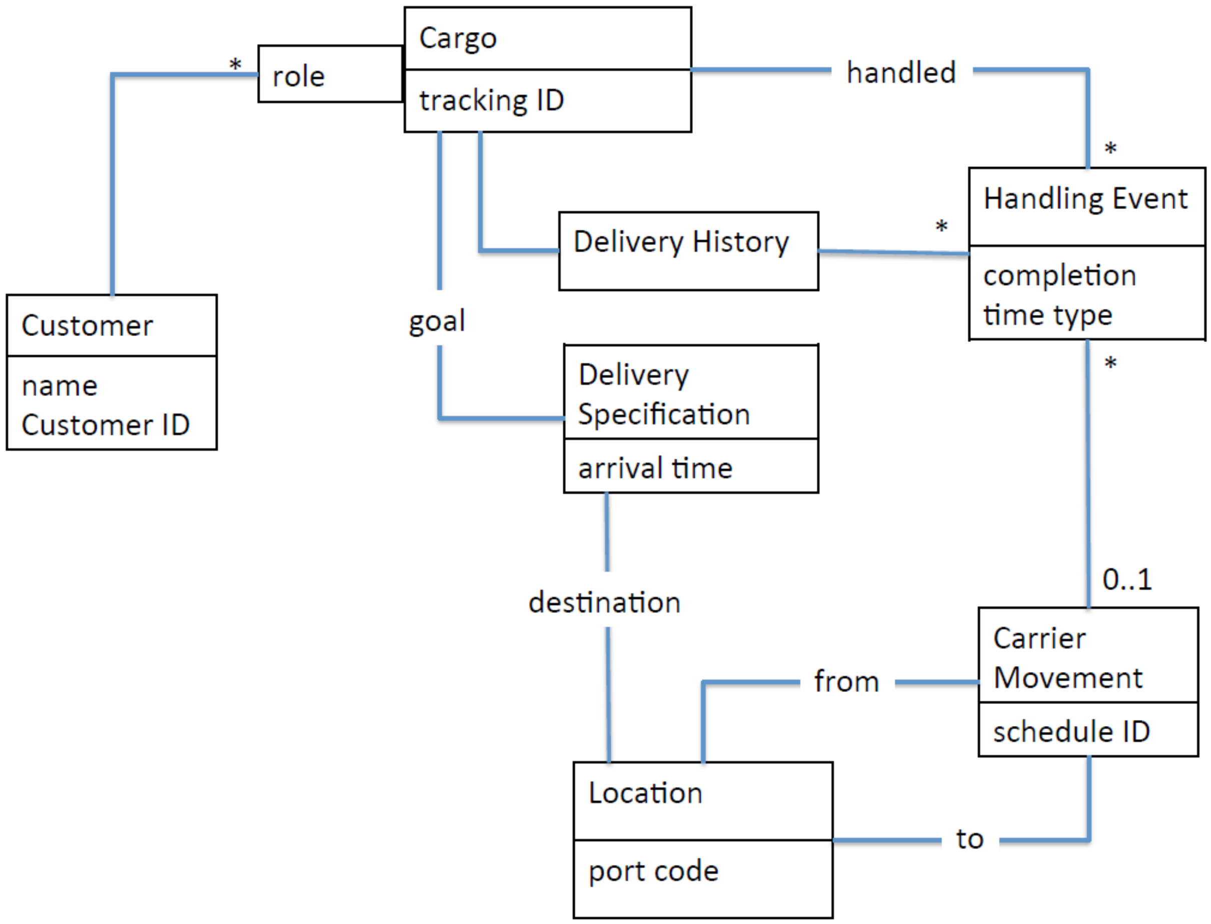
- Carrier Movement represents one particular trip by a particular Carrier (truck or ship) from one location to another
- Cargoes can ride can go from one place to another for the duration of one or more Carrier Movements.

Shipping Domain Model



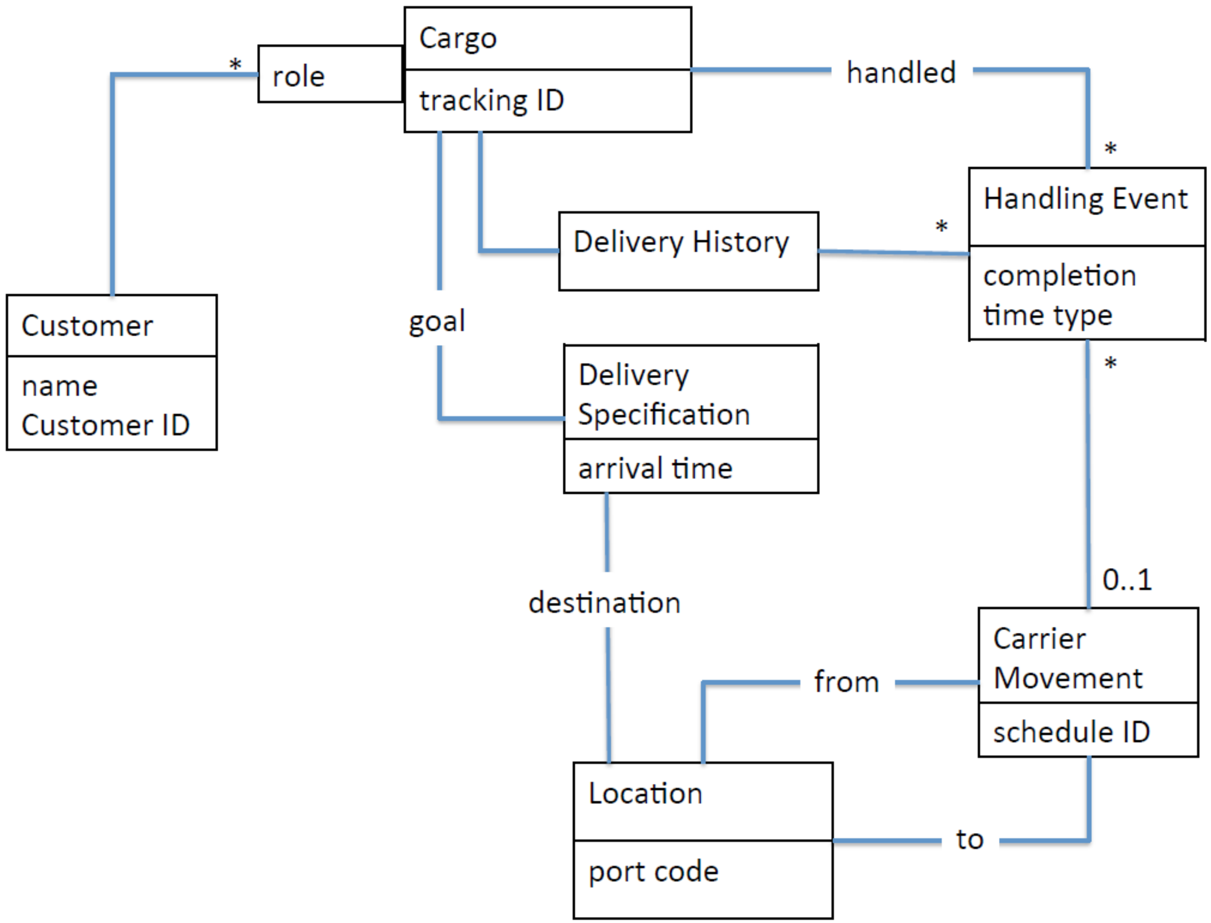
- Delivery History reflects what has actually happened to a Cargo, as opposed to the Delivery Specification, which describes goals.

Shipping Domain Model



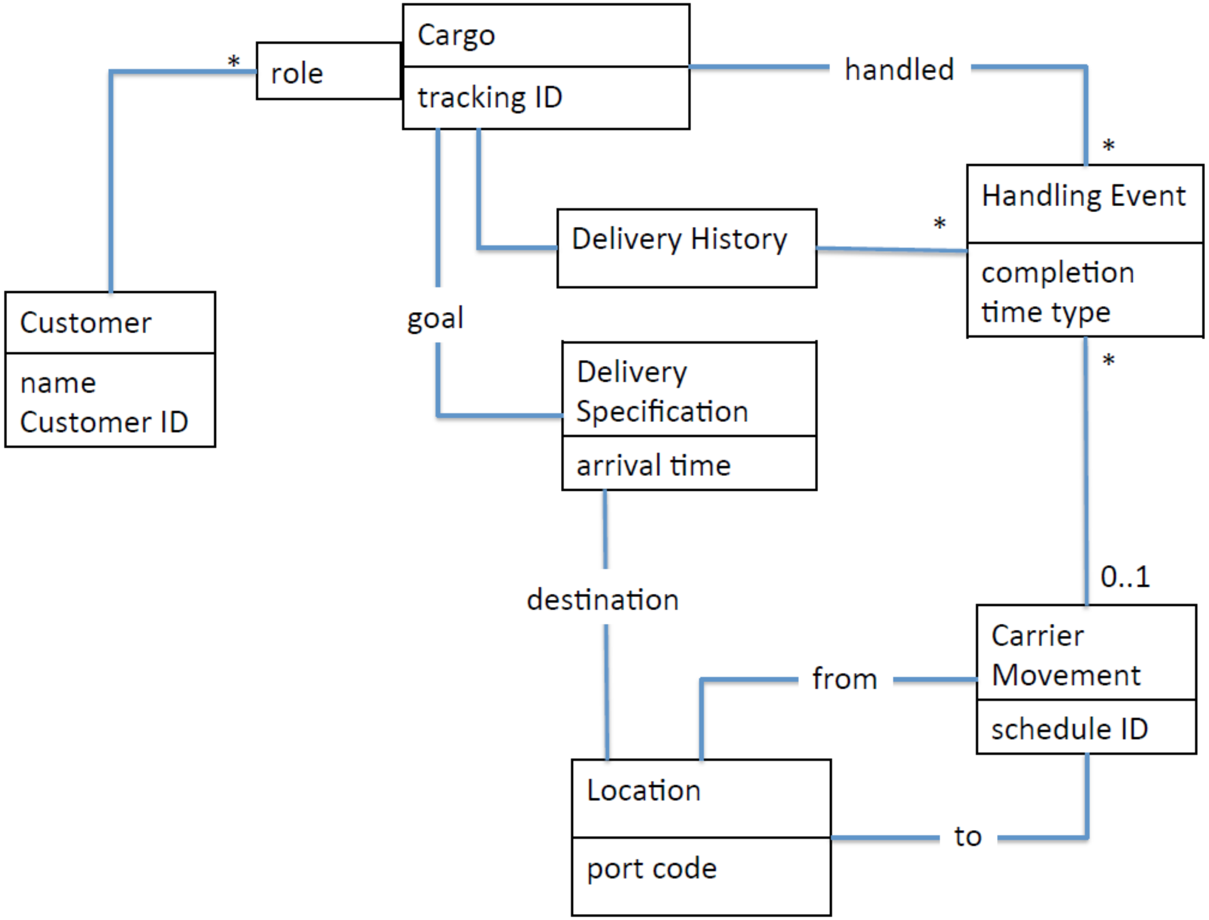
- Delivery History reflects what has actually happened to a Cargo, as opposed to the Delivery Specification, which describes goals.
- *What defines a successful delivery?*

Identify Entities and Value Objects



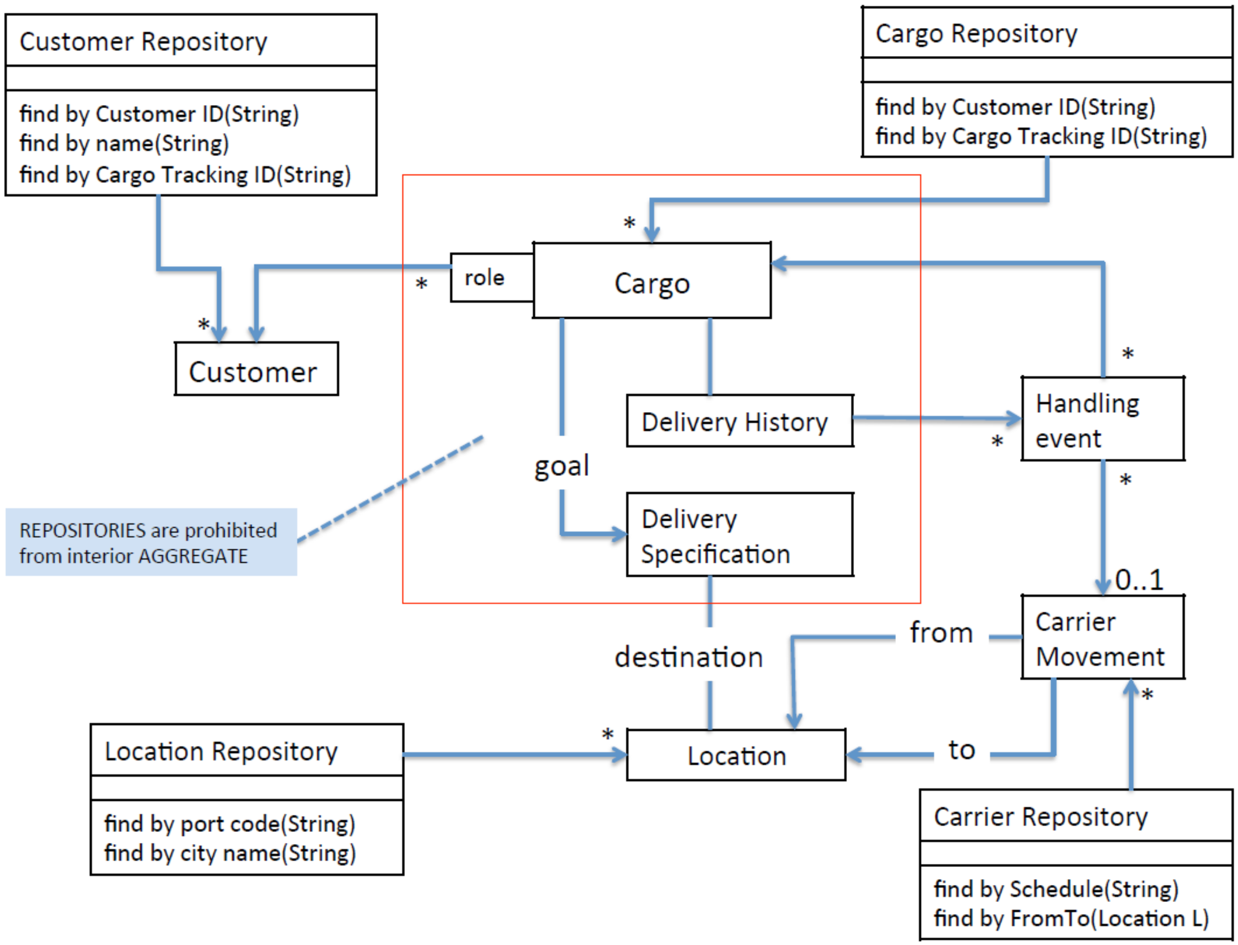
1. What’s the key different between entities and value objects?
2. Which objects are entities and which are value objects, why?

Aggregate Boundaries



Where aggregations may happen and which are the aggregation roots?

Selecting Repositories



Customer Repository

find by Customer ID(String)
find by name(String)
find by Cargo Tracking ID(String)

Cargo Repository

find by Customer ID(String)
find by Cargo Tracking ID(String)

Location Repository

find by port code(String)
find by city name(String)

Carrier Repository

find by Schedule(String)
find by FromTo(Location L)

REPOSITORIES are prohibited from interior AGGREGATE

Application Layer

- A Tracking Query that can access past and present handling of a particular Cargo
- A Booking Application that allows a new Cargo to be registered and present the system for it
- An Incident Logging Application that can record each handling of the Cargo (providing the information that is found by the Tracking Query)