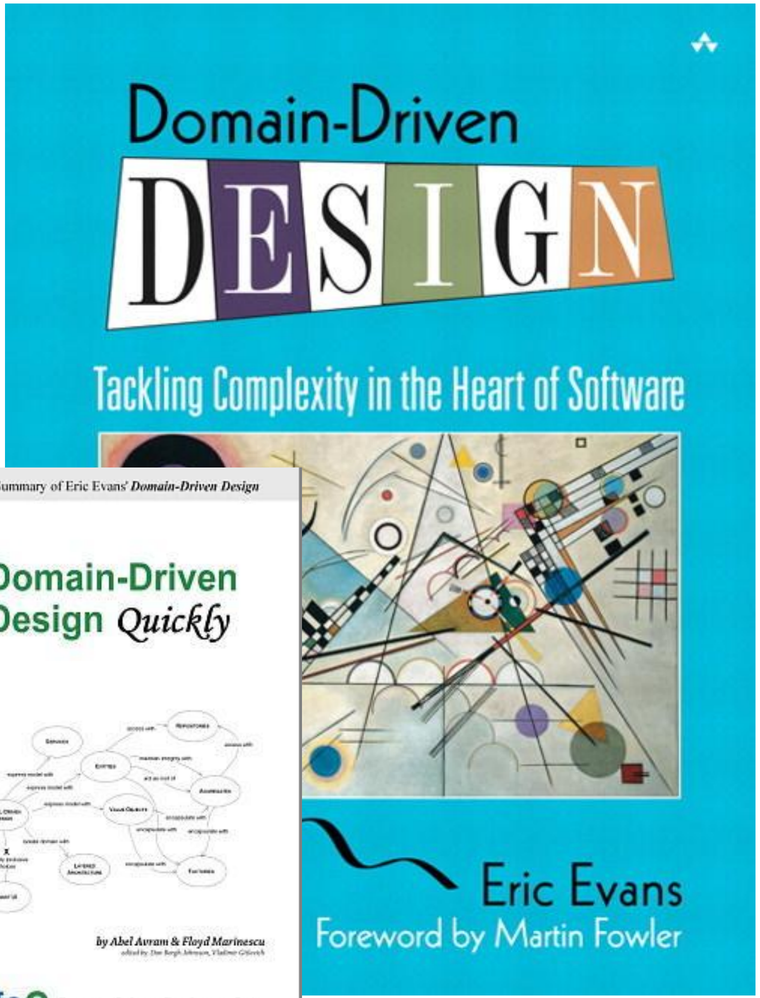
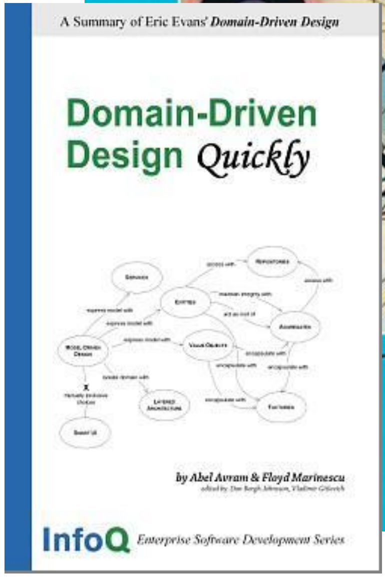
Domain-Driven Design

CS 618

Feb 21, 2012

Bill Kidwell


Domain-Driven
DESIGN
Tackling Complexity in the Heart of Software
Eric Evans
Foreword by Martin Fowler


A Summary of Eric Evans' Domain-Driven Design
Domain-Driven Design Quickly
by Abel Avram & Floyd Marinescu
InfoQ Enterprise Software Development Series

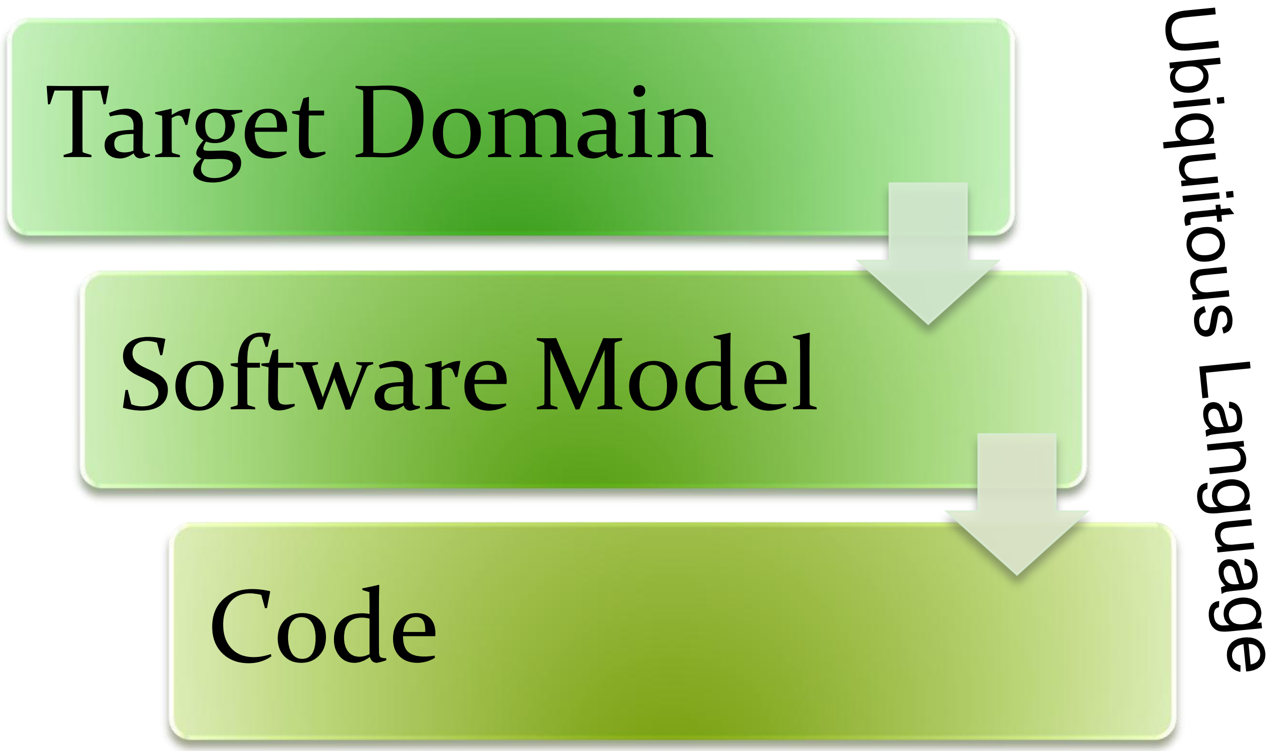
Domain-Driven Design

- Software models some aspect of the real world
- We build design models to understand what we are building, and how we will build it
- Symmetry between our software, design model, and the real world allow us to adjust to changes in the real world

Ubiquitous Language

- A common language between the domain experts and the developers
- The Domain model should be based heavily on the Ubiquitous Language
- *Discussion Point:*
  - *How does common language help with technical decisions? Examples?*

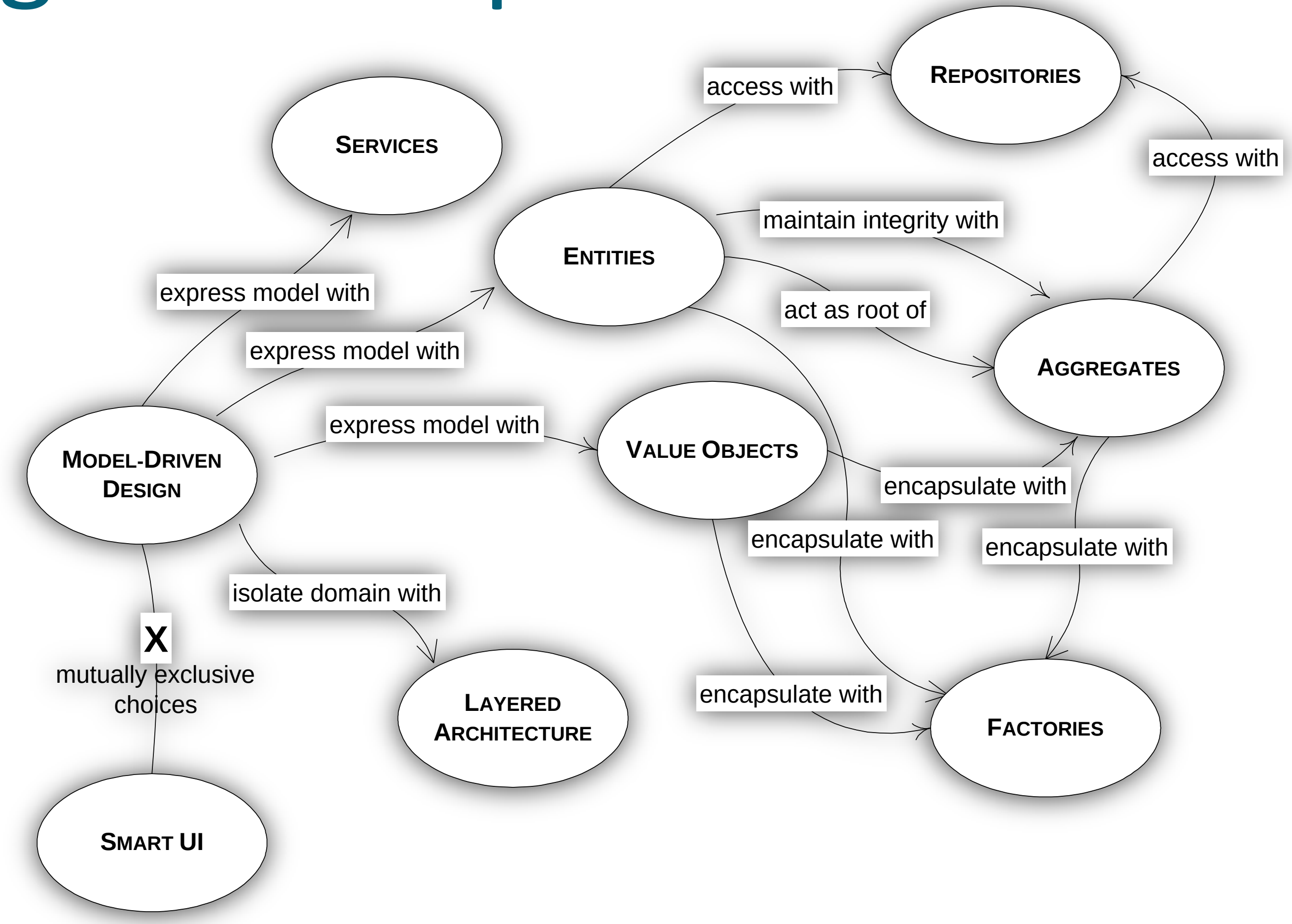
UL ties the models together



Model-Driven Design

- Tie the Implementation to the Model
- Provide tools that make this efficient
  - E.g. round trip reverse engineering tools
- Developers and Modelers are tightly coupled with this approach

Navigation Map



Evans' Layers (Isolating the Domain)

User Interface Layer	• A.k.a. Presentation Layer • Show Information • Interpret commands
Application Layer	• Thin layer, sirects UI commands to jobs in the Domain Layer • Should not contain Business Rules or Knowledge • No business "state", may have progress "state"
Domain Layer	• Business objects, their rules, and their state • The majority of the book focuses here
Infrastructure Layer	• Generic technical capabilities to support the higher layers • Message sending, persistence • Supports the interactions between topmost patterns

Entities

- Have an identity
  - Not the address of the object
  - What is the identity?
    - Consider two person objects, same name, same date of birth – separate identities
    - We often generate an identifier
      - Account Number

Value Objects

- Not all objects are entities!
  - We can't justify the overhead of creating and tracking identities for all objects
- It is recommended that value objects be immutable
- Examples of possible Value objects
  - Money/Currency class
  - Point class in a drawing application
  - Address class ?

Services

- Some aspects of the domain don’t map easily to objects
- A Service is some behavior, that is important to the domain, but does not “belong” to an Entity or Value object
- Example: Account Transfer
- Encapsulate an important domain concept
  - NOTE: Not just for technical infrastructure

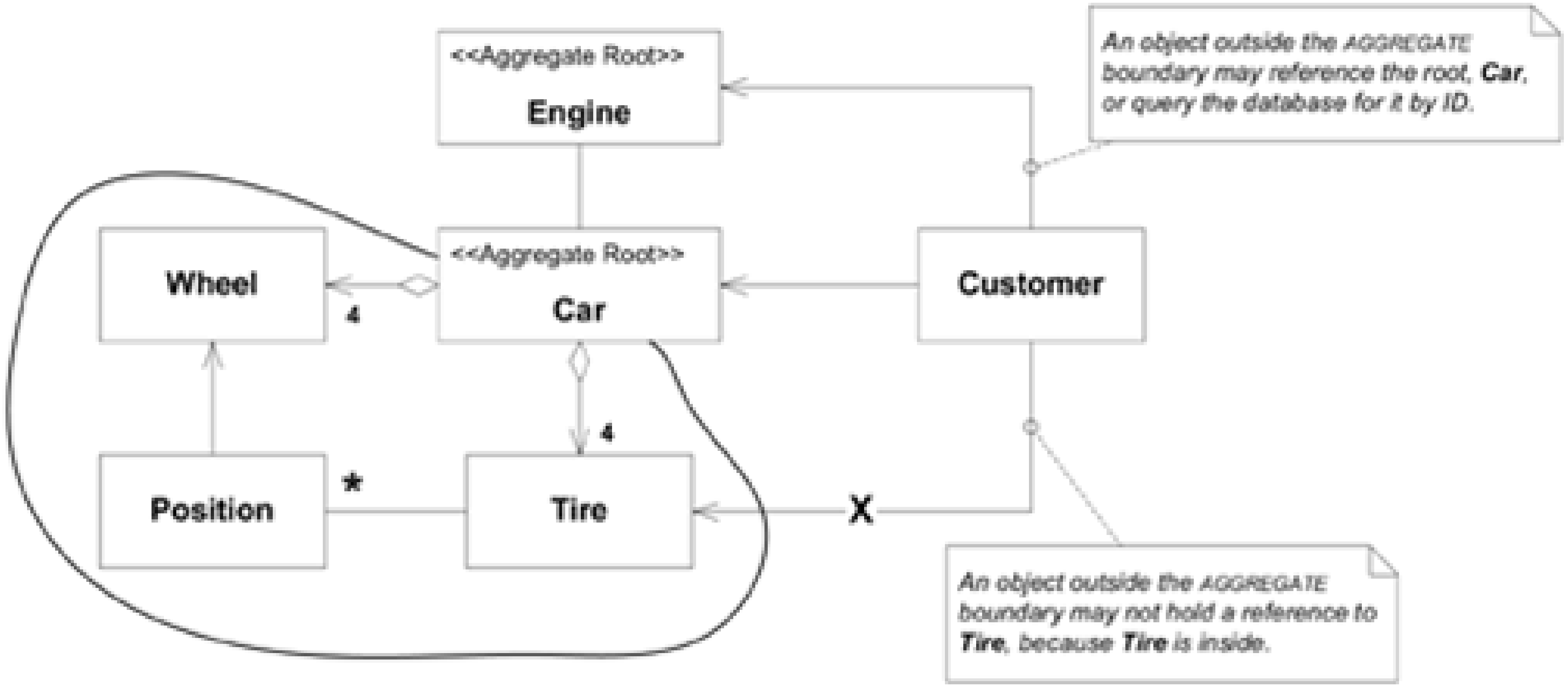
Characteristics of Services

1. The operation performed by the Service refers to a domain concept which does not naturally belong to an Entity or Value Object.
2. The operation performed refers to other objects in the domain.
3. The operation is stateless.

Aggregates

- A group of associated objects which are considered as a unit with regard to data changes
- An aggregate should have one root
- The root is an entity object
- Outside objects can reference root, but not the other members of the aggregate

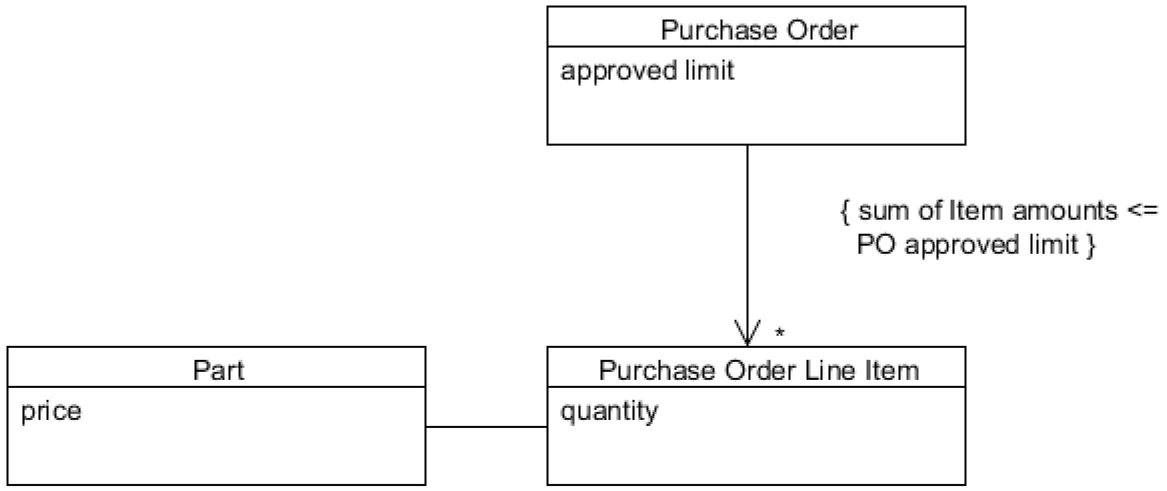
Aggregate Root Example



<<Aggregate Root>>
Engine
<<Aggregate Root>>
Car
Wheel
4
Customer
4
Position
*
Tire
X
An object outside the AGGREGATE boundary may reference the root, Car, or query the database for it by ID.
An object outside the AGGREGATE boundary may not hold a reference to Tire, because Tire is inside.

Domain-Driven Design: Tackling Complexity in the Heart of Software by Eric Evans

PO Example (from Evans)

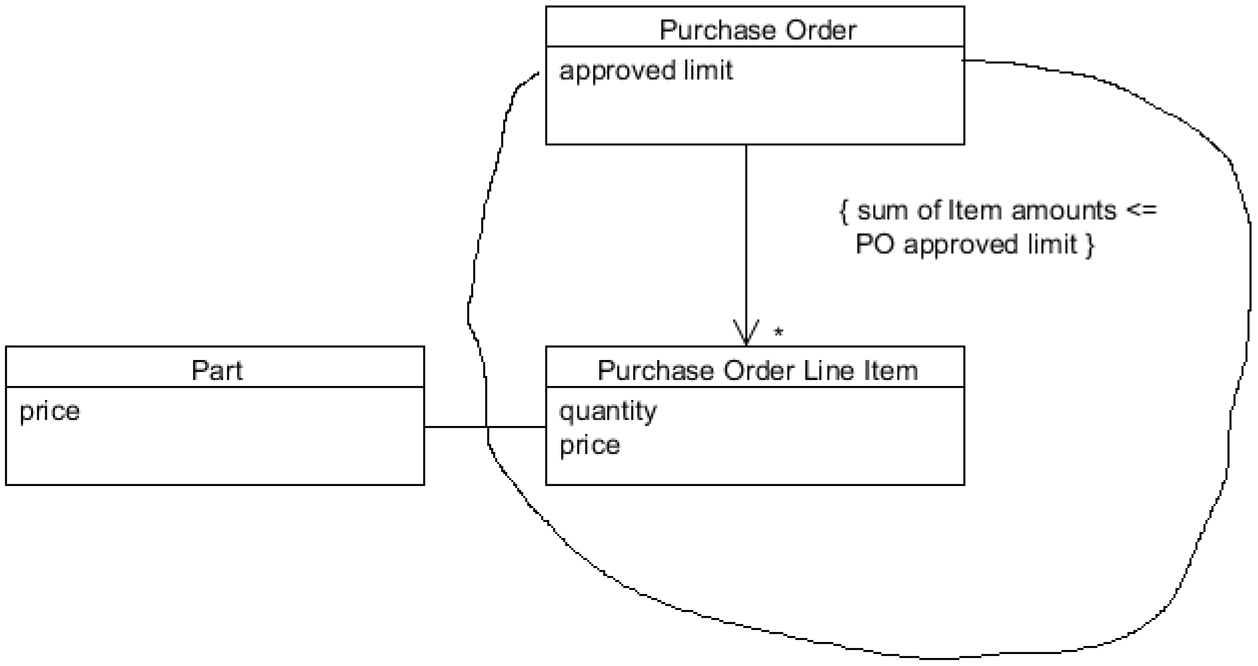


Domain-Driven Design: Tackling Complexity in the Heart of Software by Eric Evans

PO Example (cont’d)

- Parts are used in many Pos (high contention)
- Fewer changes to parts than Pos
- Changes to part prices do not necessarily propagate to existing POs

PO Example (cont’d)



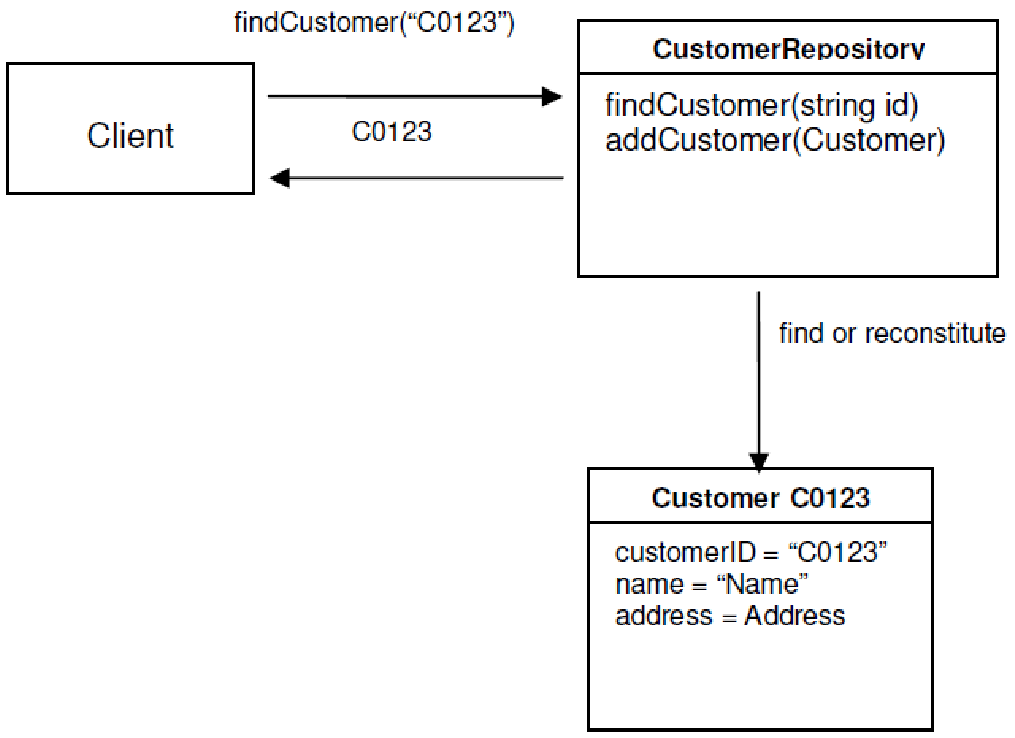
Factories

- Encapsulate the information necessary for object creation
  - Includes logic for all creating all the members of an aggregate
  - Allows us to enforce invariants during creation
  - Related GoF Design Patterns
    - Factory Method, Abstract Factory
  - Designing the Factory Interface
    - Each operation must be atomic
    - The Factory will be coupled to its arguments

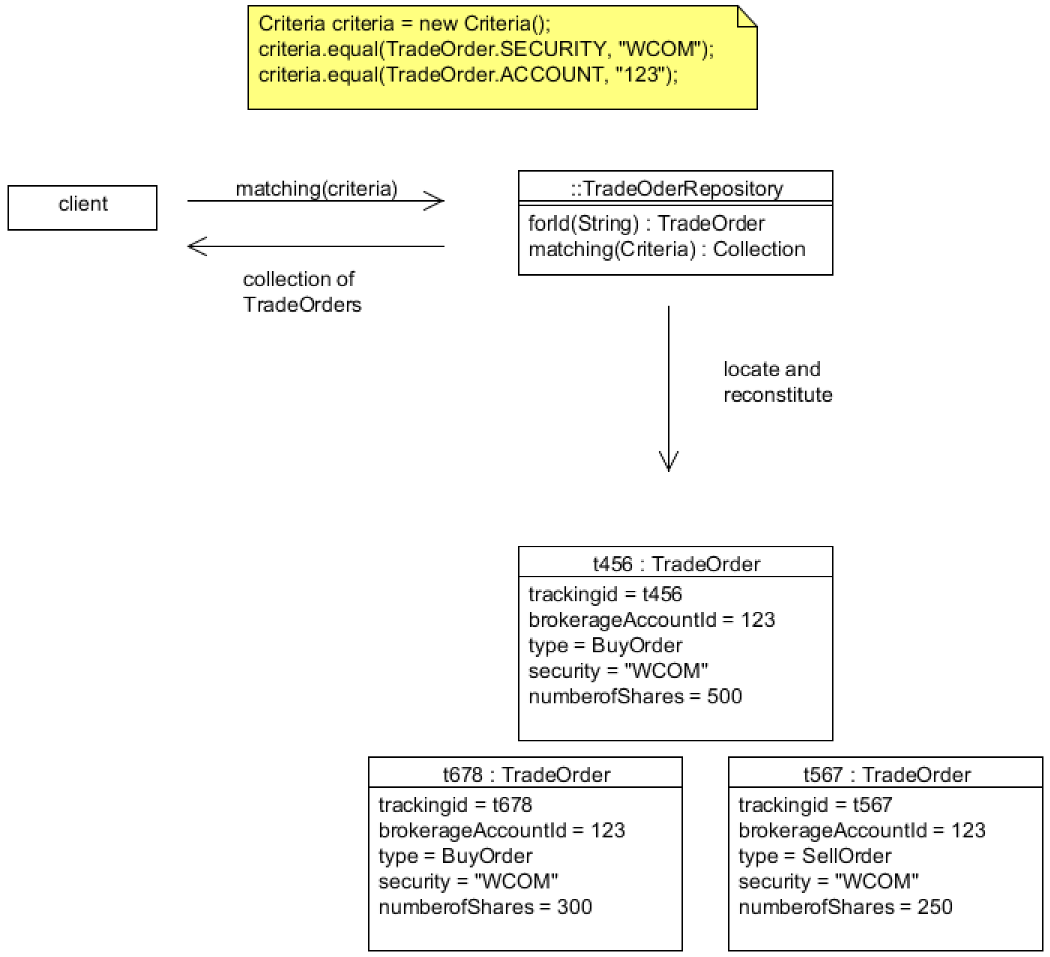
Repositories

- Encapsulates logic to obtain object references
- Provides a mechanism to persist/retrieve an object
  - Keeps persistence code out of the domain layer
- Repository interface should be driven by the domain model
- Repository implementation will be closely linked to the infrastructure

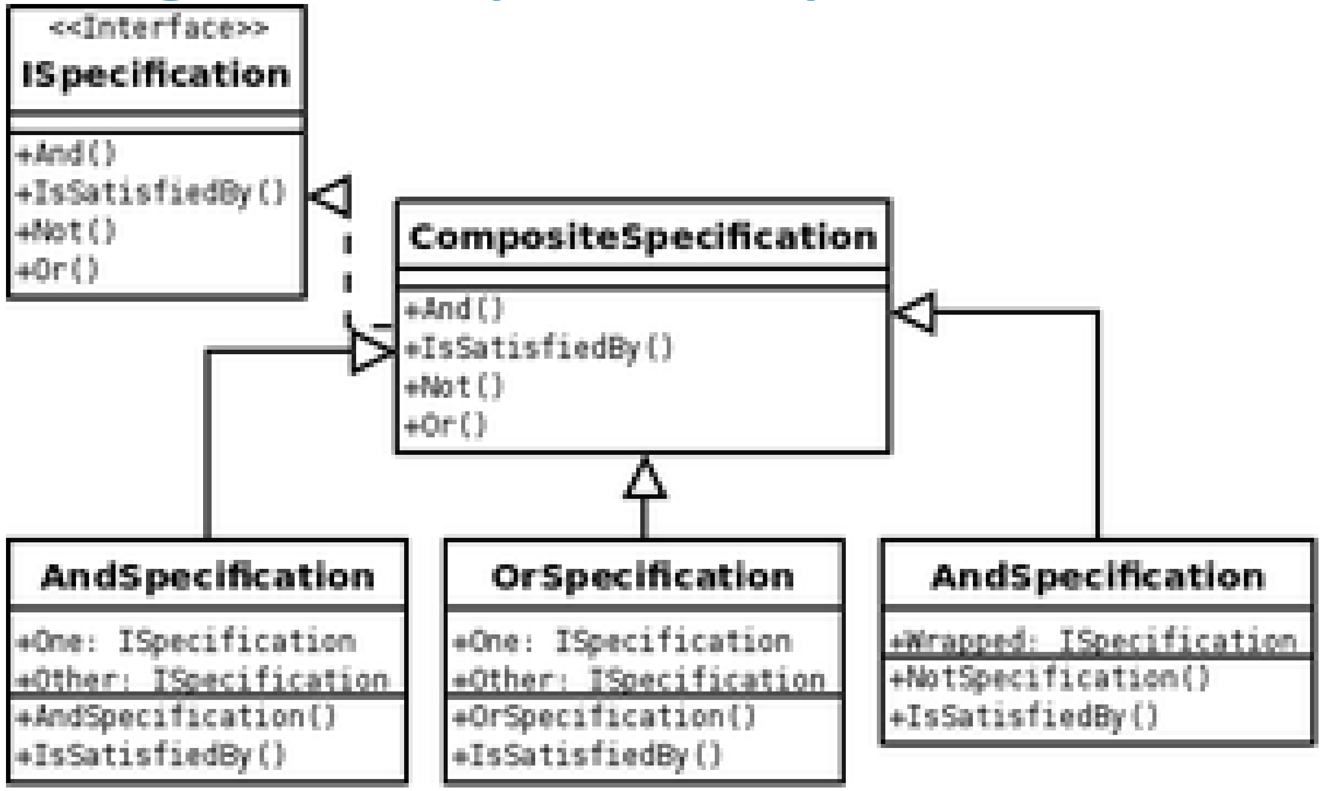
Repositories



Repository (Specification based query)



Building Complex Specifications



<<Interface>>
ISpecification
+And()
+IsSatisfiedBy()
+Not()
+Or()
CompositeSpecification
+And()
+IsSatisfiedBy()
+Not()
+Or()
AndSpecification
+One: ISpecification
+Other: ISpecification
+AndSpecification()
+IsSatisfiedBy()
OrSpecification
+One: ISpecification
+Other: ISpecification
+OrSpecification()
+IsSatisfiedBy()
AndSpecification
+Wrapped: ISpecification
+NotSpecification()
+IsSatisfiedBy()