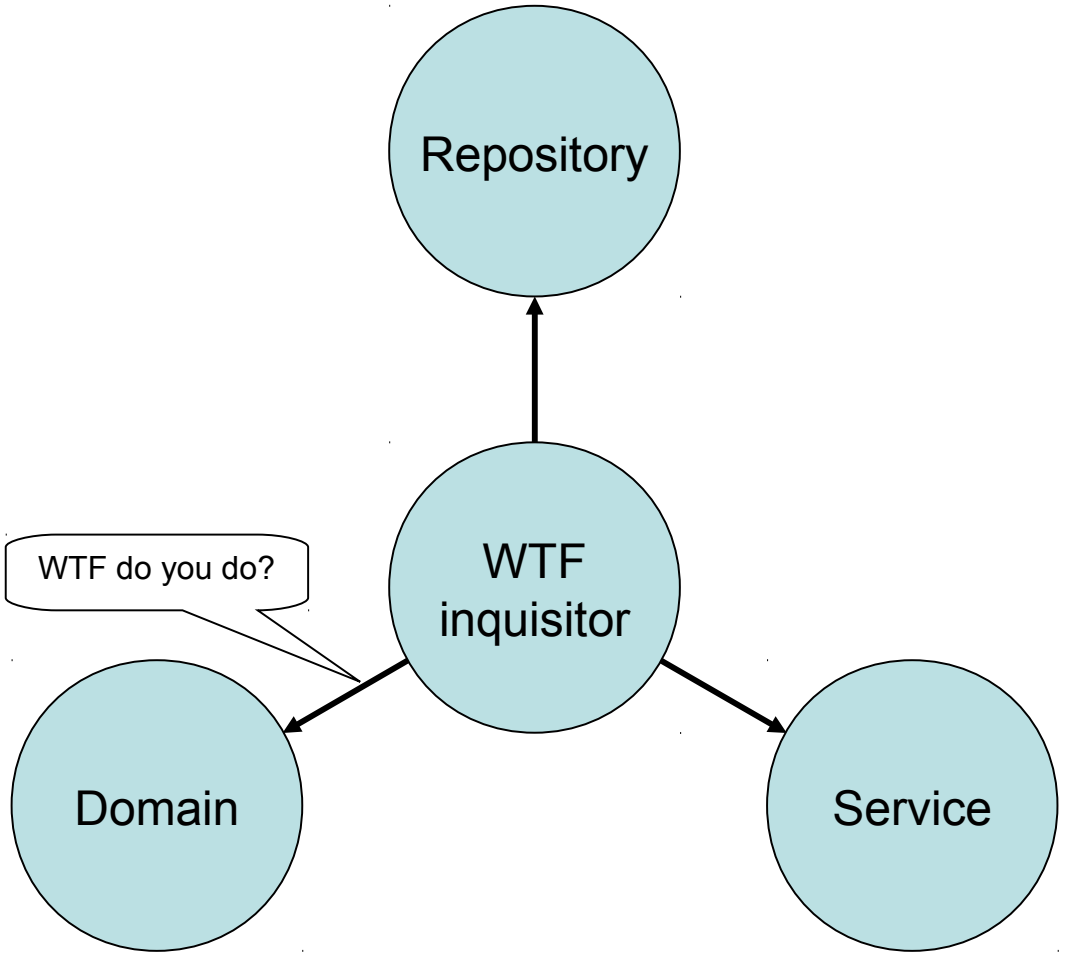
Domain Driven Design

Overview

- The Ubiquitous Language:
  - The model and the heart of the design shape each other
  - The model is the backbone of a language used by all team members
  - The model is distilled knowledge
- Benefit: it is easier to understand. Therefore, as the model changes it is easier to make changes to the code

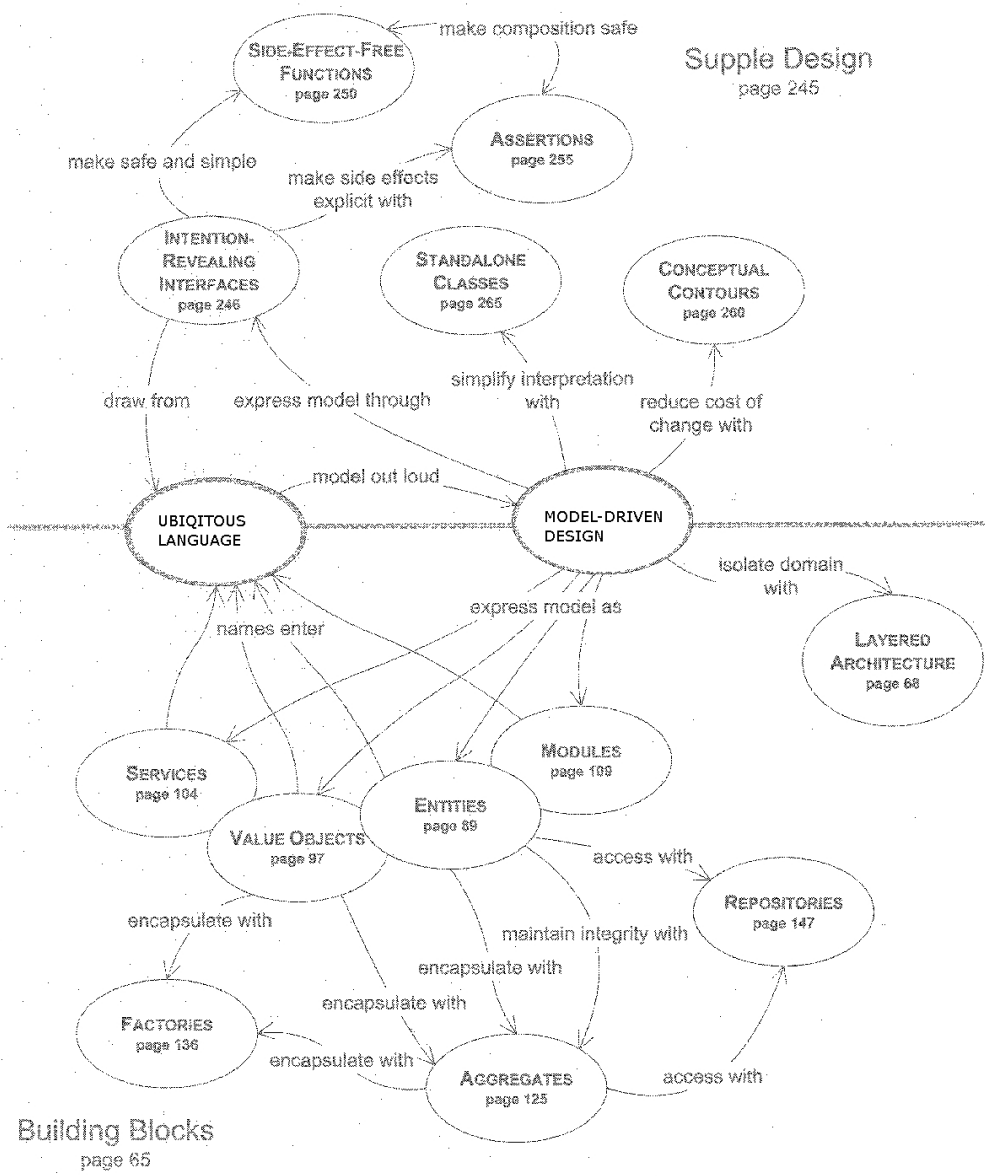
Ingredients

- Binding the model and the implementation
- Cultivating a language based on the model
- Developing a knowledge-rich model
- Distilling the model
- Brainstorming and experimenting


Repository
WTF do you do?
WTF inquisitor
Domain
Service

Components

- Entities
- Value Objects
- Aggregates
- Repositories
- Services
- Factories


Supple Design
page 245
SIDE-EFFECT-FREE FUNCTIONS
page 250
make composition safe
ASSERTIONS
page 255
make safe and simple
make side effects explicit with
INTENTION-REVEALING INTERFACES
page 246
STANDALONE CLASSES
page 265
CONCEPTUAL CONTOURS
page 260
draw from
express model through
simplify interpretation with
reduce cost of change with
model out loud
UBIQITOUS LANGUAGE
MODEL-DRIVEN DESIGN
isolate domain with
LAYERED ARCHITECTURE
page 68
express model as
names enter
SERVICES
page 104
MODULES
page 109
ENTITIES
page 89
VALUE OBJECTS
page 97
access with
REPOSITORIES
page 147
encapsulate with
maintain integrity with
encapsulate with
encapsulate with
FACTORIES
page 136
encapsulate with
AGGREGATES
page 125
access with
Building Blocks
page 65

Exercise Phase 1

- Create a simple trading application
- Scenario: Add a new trade
- Trade has a stock code and buy price

Exercise Phase 2

- Mark to market: calculate the current trade value based on a market value
- Print out all of the currently entered trades and their MTM value
- formula:

  mark to market value =

  current market value – trade buy value

Exercise Phase 3

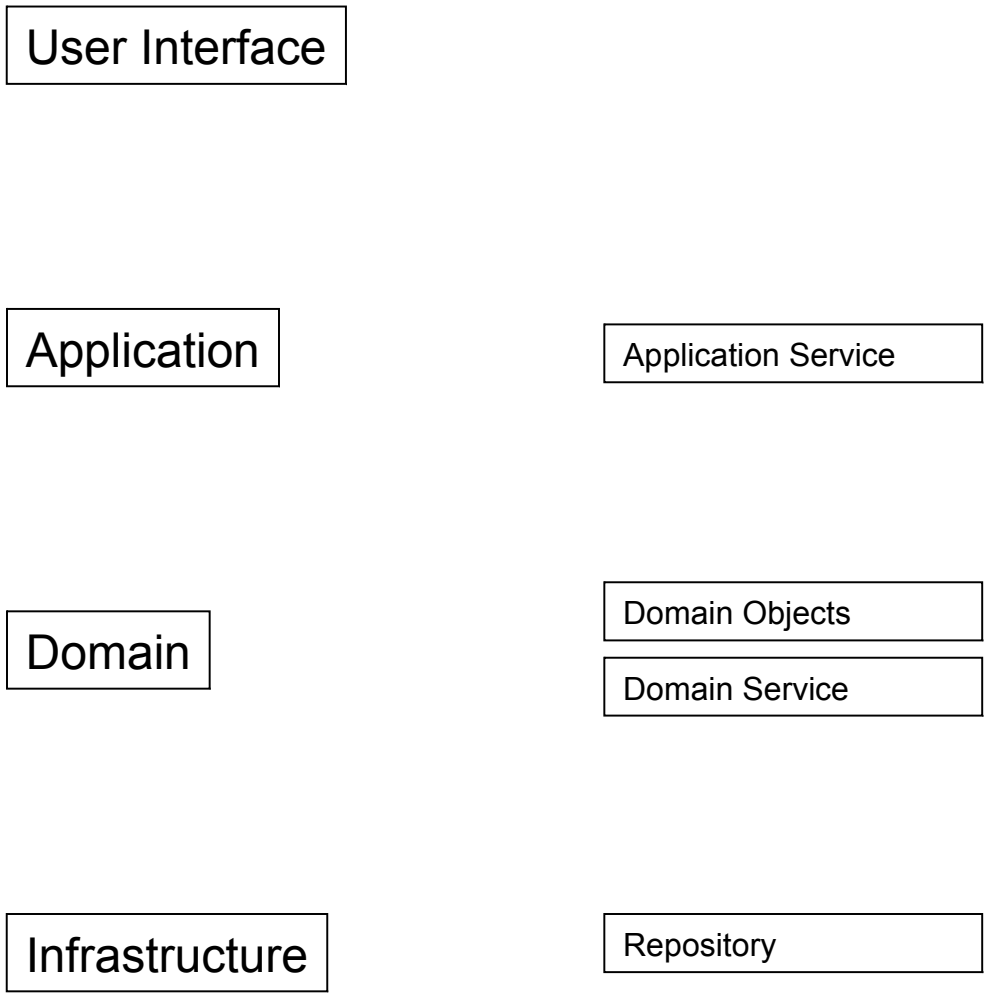
- Add ability to trade options (call only)
- An option has a strike price and a premium price

mark to market value =

current market value

- strike price
- premium

Architectural layers



Components overview

- Domain: core mapping of the model - business logic
- Entities: when an object is distinguished by its identity, rather than its attributes, make this primary to its definition in the model. eg: Customer, Product
- Value Object: If you only care about the attributes of an object, it is a Value Object.

  Don't give it any identity and avoid the design complexities necessary to maintain entities.

  eg: Address, Date
- Aggregates: Some objects are "bound together" because one object cannot change without its parent also being affected. Only the root (parent) object should be accessed.

  eg: OrderLine should only be accessed via Order

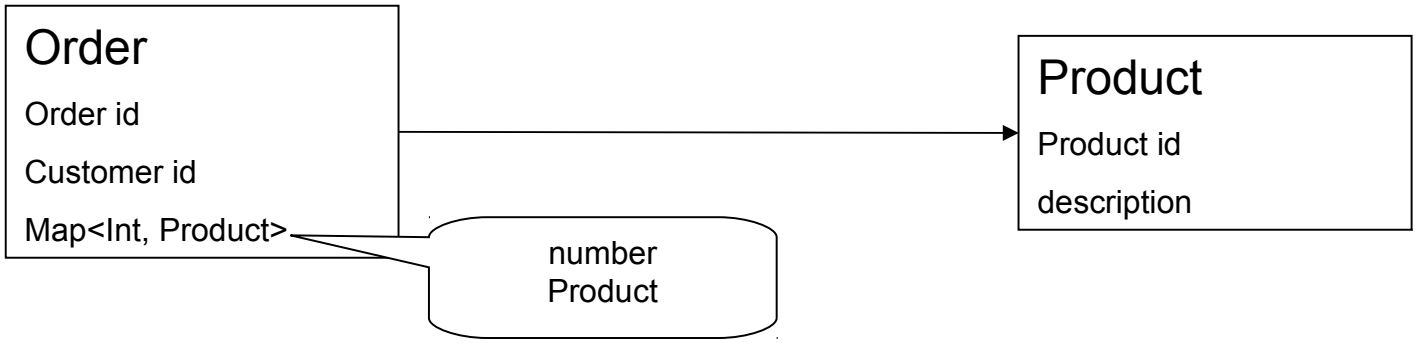
Components overview cont'd

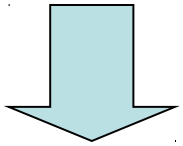
- Service: anything which doesn't naturally map to an object. Services can occur at many layers, such as the application and domain layers
- Repository: maps domain **objects as a collection**, remaining **ignorant of persistence**
- Factories: Create objects

Make implicit subjects explicit

- Relationships between objects will move from simple implicit relationships to fuller explicit relationships
- If you feel pain when modelling, stop and ask whether the model needs to be adjusted

Implicit => Excplicit example





Relationship between Order and Product has become “fuller”, moving from simple map to OrderLine object