Globant

we are ready

Front End Engineer

Domain Drive Design

TOPICS

Introduction to DDD
Design a domain oriented architecture in JS

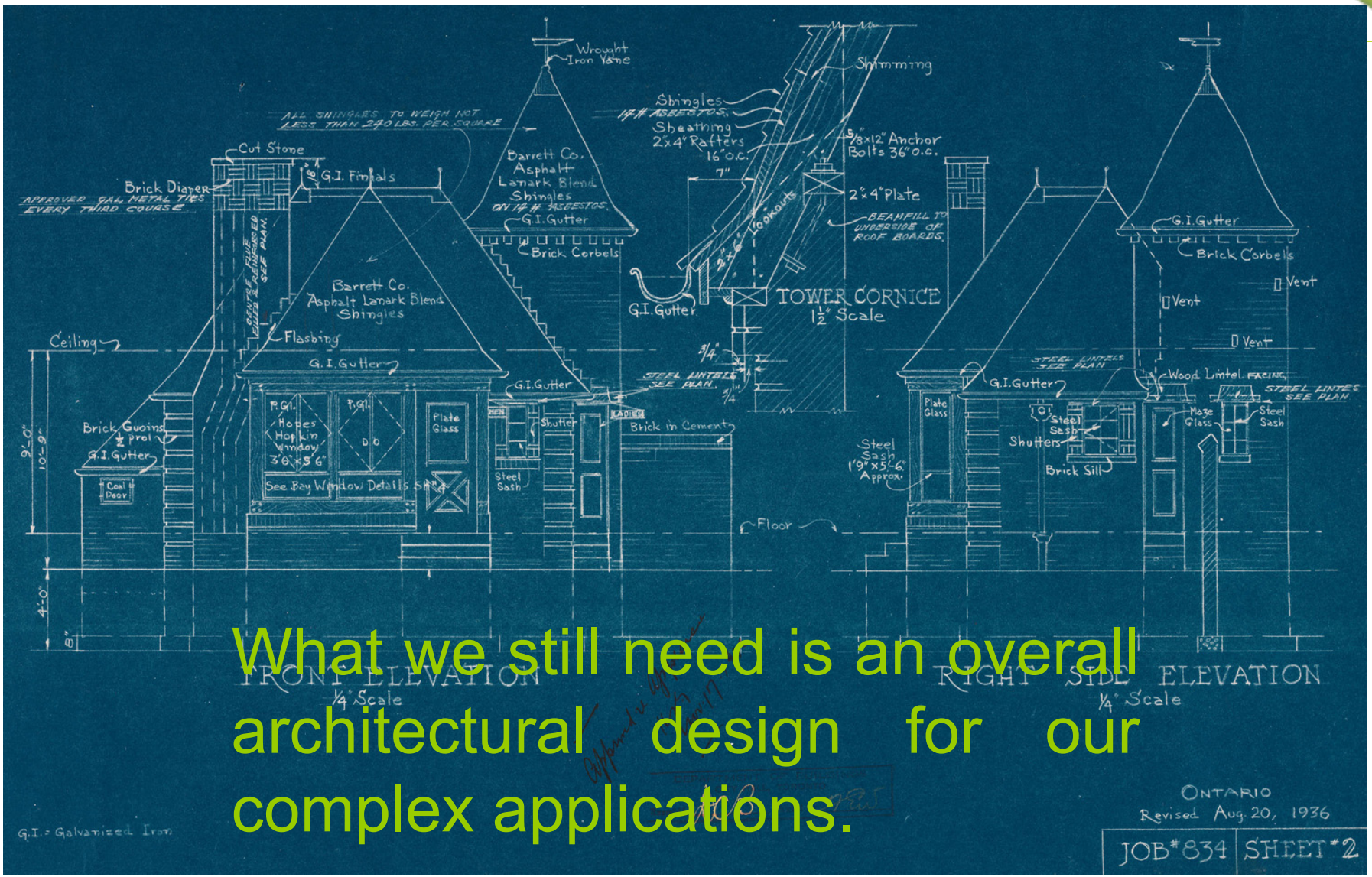
If you look off in the distance of the center of the photograph, you'll see a just-launched ship sailing away.

And here at the top, you'll notice a crane bringing in the first beam that will form the keel of a new ship to be constructed in the same drydock that just launched the previous ship.

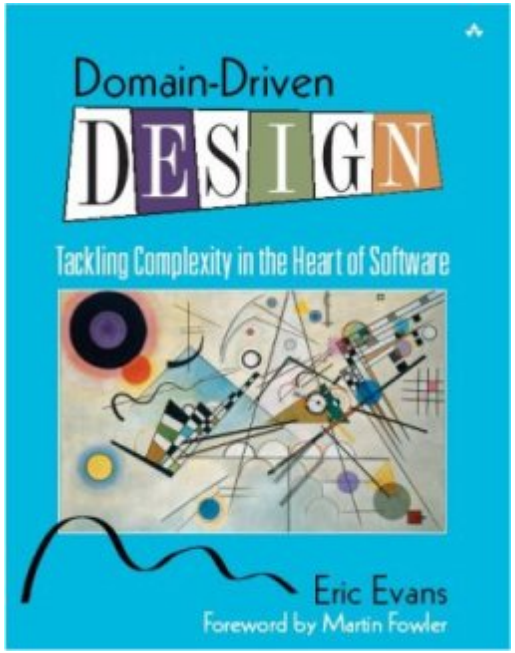
In the same way, projects never stop coming for us. We launch a project, and while it's still in sight, we start the next.

We need an architectural framework to manage the constant turnover of projects so that we don't build sinking ships.

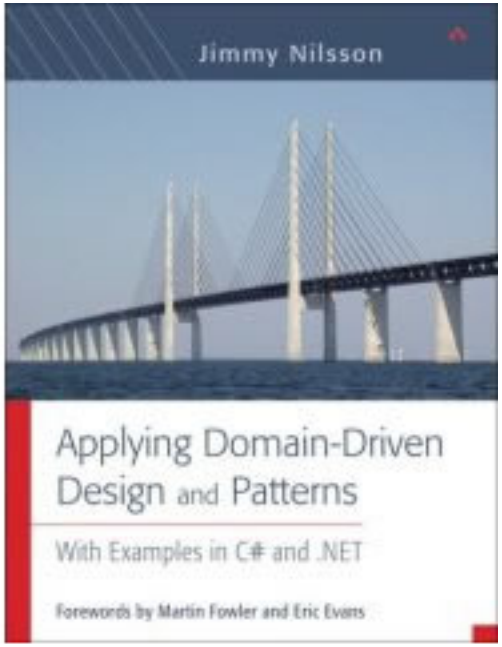

So why do so many applications still look like piles of coat hangers when we're done with them?
Well-modularized piles of coat hangers, but piles of coat hangers nonetheless.



Where does DDD come from?



Domain Driven Design
Eric Evans

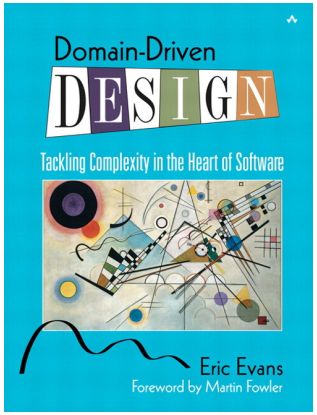


Applying DDD and Patterns
Jimmy Nilsson

What if my software isn't complex?

“Put all the business logic into the user interface. […] Use the most automated UI building and visual programming tools available.”

-Eric Evans



Manage complex software by placing primary focus on the domain.

—Domain-Driven Design

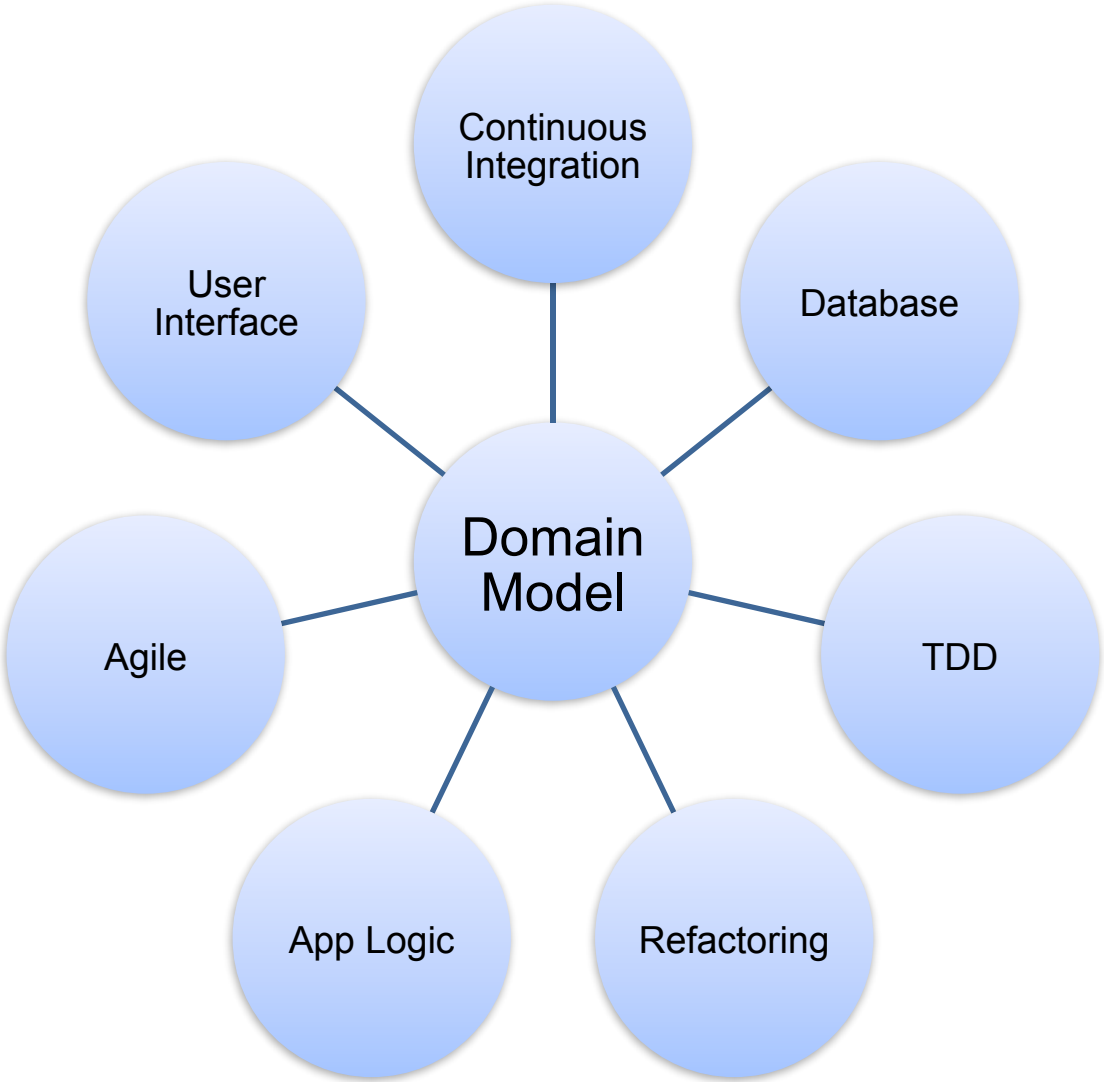
- Principles
- Techniques
- Practices

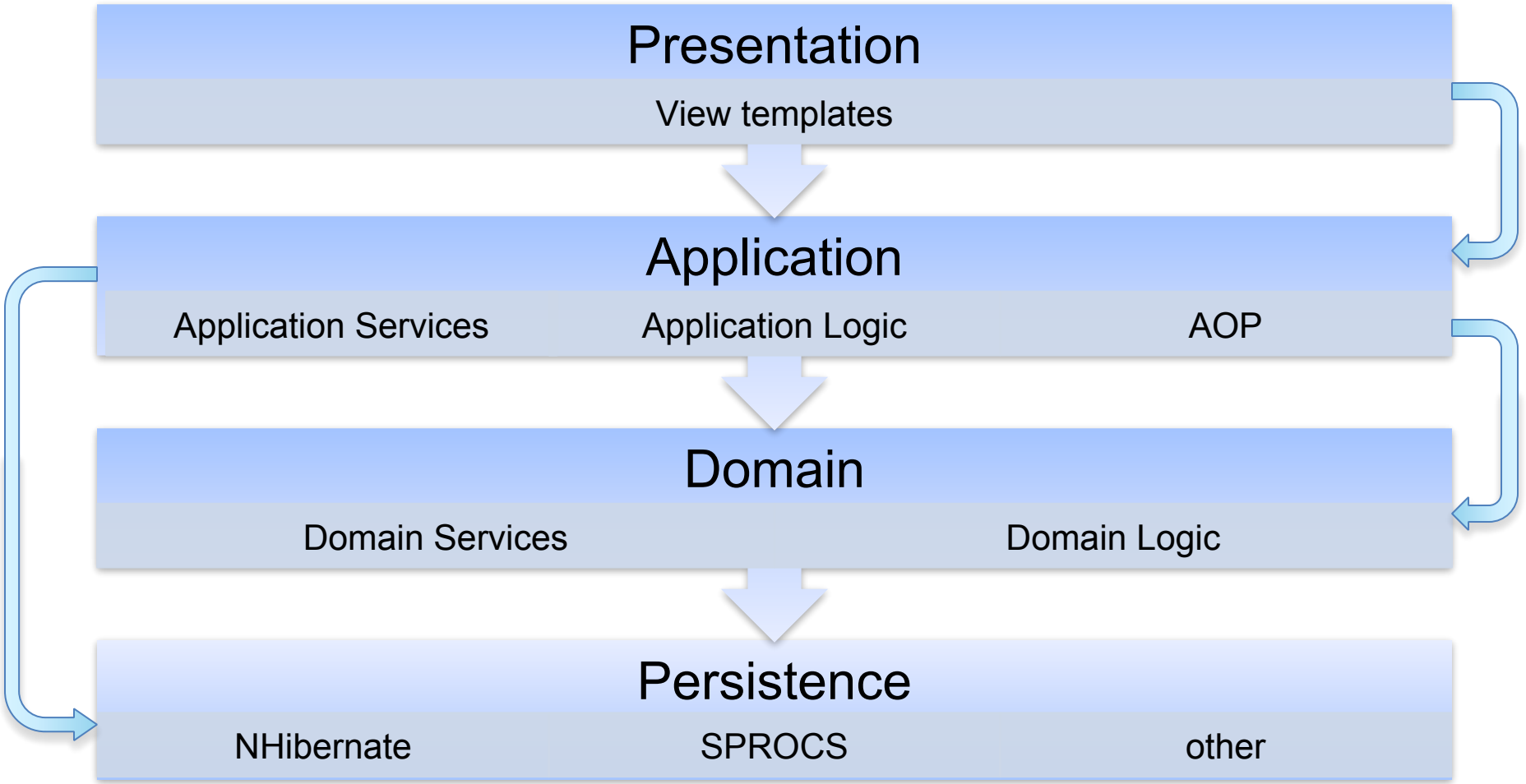
Abstraction of the knowledge in a domain expert's head

Intimately linked with the software implementation

If a domain model cannot be understood by sophisticated domain experts, then there is something wrong with the model.

10,000 ft. view




Presentation
View templates
Application
Application Services
Application Logic
AOP
Domain
Domain Services
Domain Logic
Persistence
NHibernate
SPROCS
other

Now, you can see:

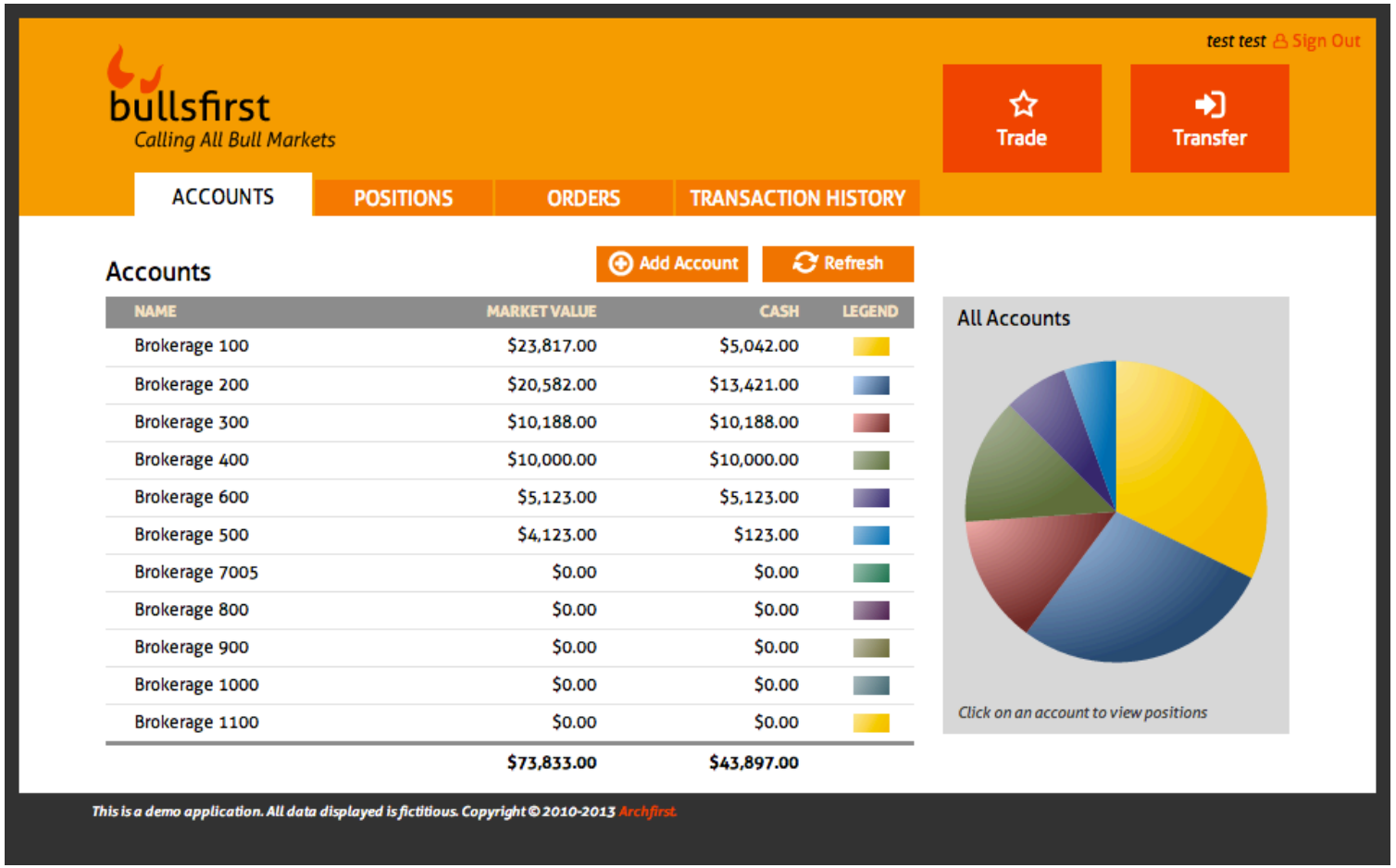
01 DDD Global Introduction by Eric Evans

http://youtu.be/-6s3TV7ngfo

Building a Domain-Driven Application

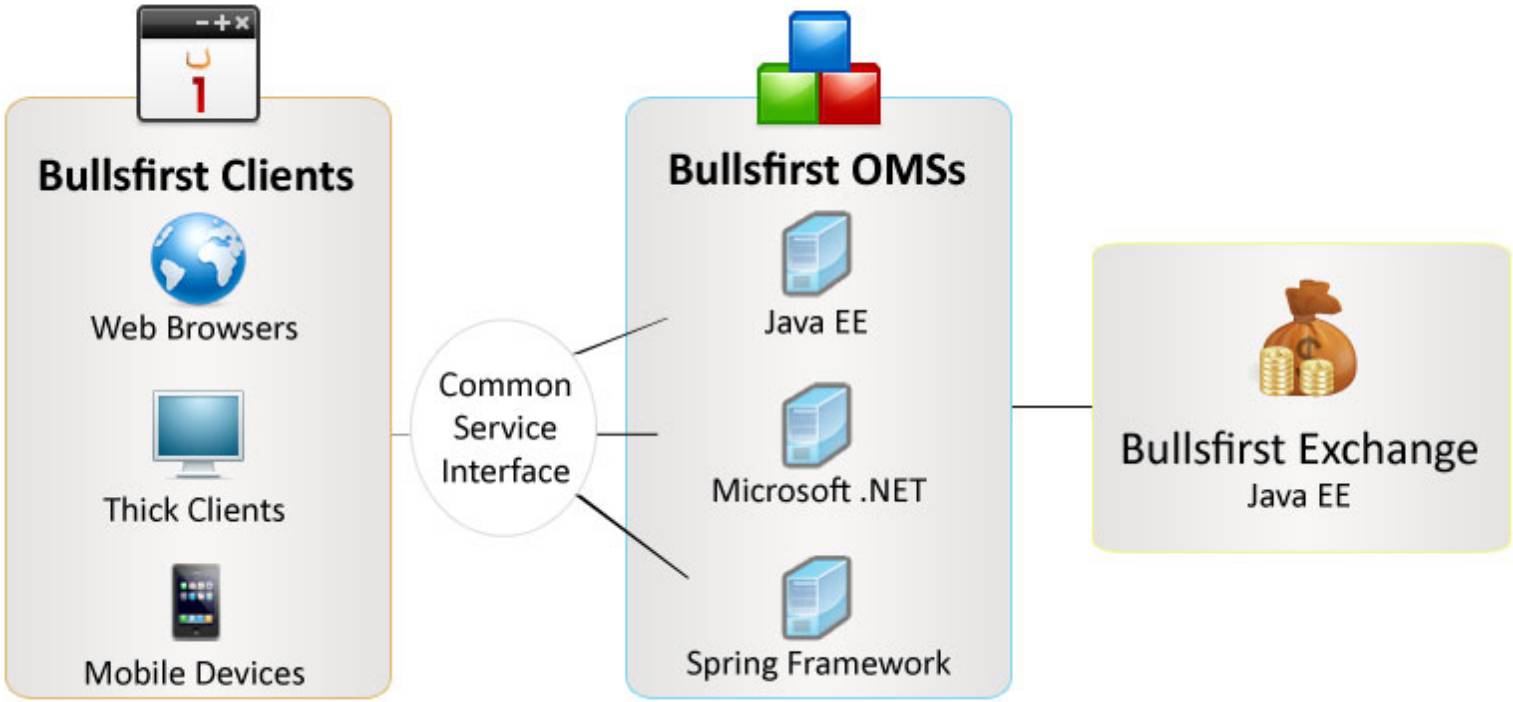
This is a screen shot of an example trading application we want to build. Bullsfirst is an open-source proof-of-concept project that we use to test out a variety of application technologies.

https://github.com/archfirst/bullsfirst-jquery-backbone



Building a Domain-Driven Application

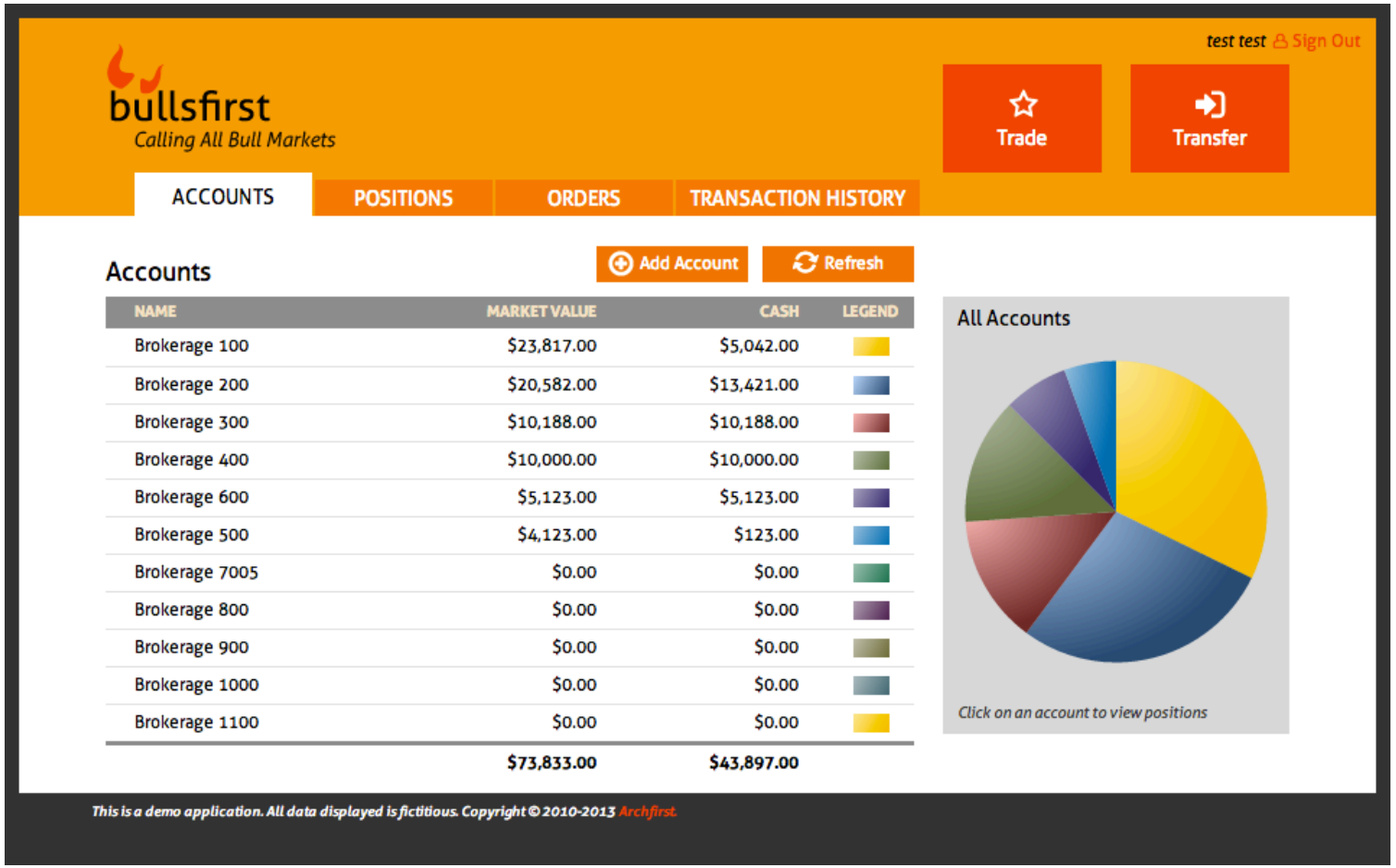
We use it to test both front-end client technologies and middle-tier solutions.
We find it to be a bit more robust and grueling test than your typical todo list.
On the front-end, we've built sample implementations in iOS, Silverlight, ExtJS, and now Backbone.



Building a Domain-Driven Application

But you can see this is the type of application we're trying to build. It's got a good bit of interaction, and there's a very sophisticated domain model behind all of this data. So how do we go about building this?

.



The Framework

The first thing we want to do is figure out our framework.

We want a framework that will support our Domain-Driven ambition as much as possible.

This includes staying out of our way when we need it to.

Building a Domain-Driven Application

backbone.js

In an ecosystem where overarching, decides-everything-for-you frameworks are commonplace, and many libraries require your site to be restructured to suit their look, feel, and default behavior — Backbone should continue to be a tool that gives you the freedom to design the full experience of your web application.

- Jeremy Ashkenas

Building a Domain-Driven Application

Pattern

Component

- View
- Template
- ViewModel

Building a Domain-Driven Application

Because all of our views are built off of our BaseView with its child management methods, we can nest views within views within views ad infinitum.

The Process

So now that we have our framework squared away, let's talk about the process of starting to think about and build a domain-driven application

Building a Domain-Driven Application

Discovery

- Begin constructing the domain model
- Develop an UBIQUITOUS LANGUAGE

Building a Domain-Driven Application

Ubiquitous Language

The first and most important step in this process is the discovery phase.

In this phase, we consult with the business experts to start to piece together the domain model, which again, is just the idea in the experts' heads.

In this, we start to compile what is known as an ubiquitous language

Building a Domain-Driven Application

Ubiquitous Language

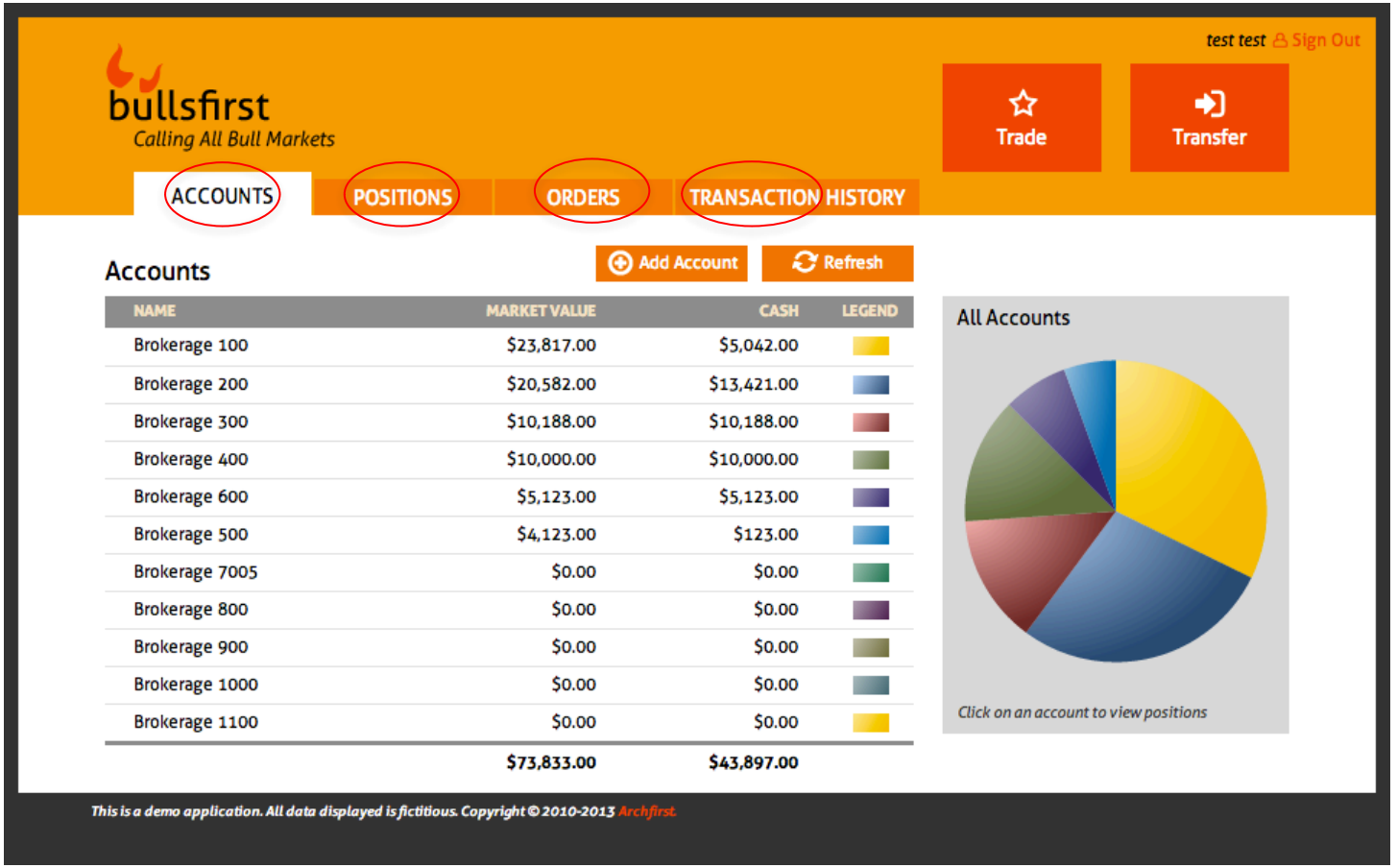
We want every person who touches this project, from the client to the business analyst to the project manager to all of the developers to speak the same language.

Instead of trying to explain things in our lingo having to do with design patterns and databases and such, we instead rigorously define terms as the client understands them.

We then don't have to translate anything between the client and the developers, we're all speaking the same language.

Building a Domain-Driven Application

Ubiquitous Language

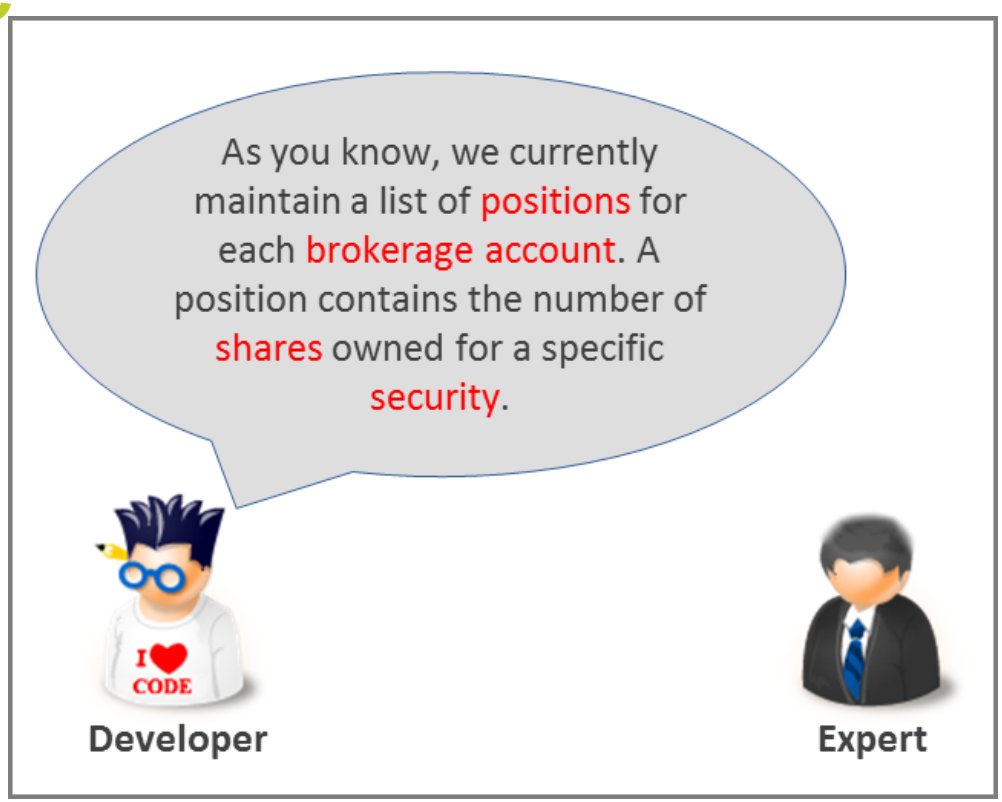


Building a Domain-Driven Application

Ubiquitous Language

The developer has a good basic understanding of the domain, but we continue to iterate on both the domain model and the ubiquitous language.

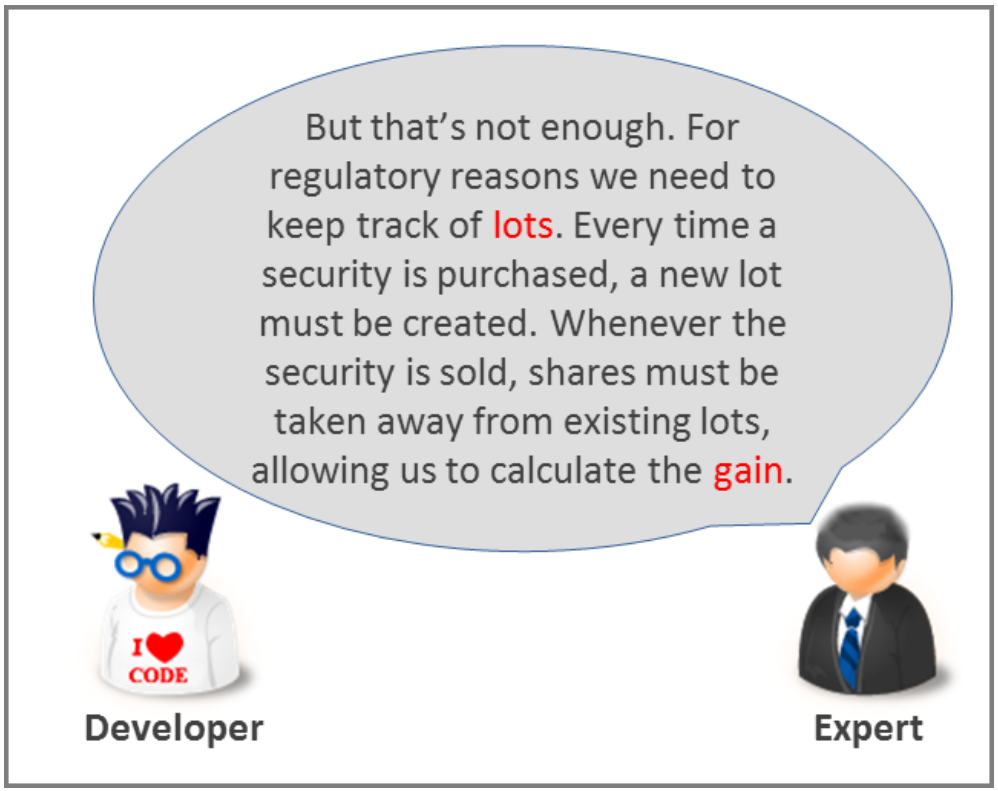
Here, the developer is recapping their understanding of the domain model.



Building a Domain-Driven Application

Ubiquitous Language

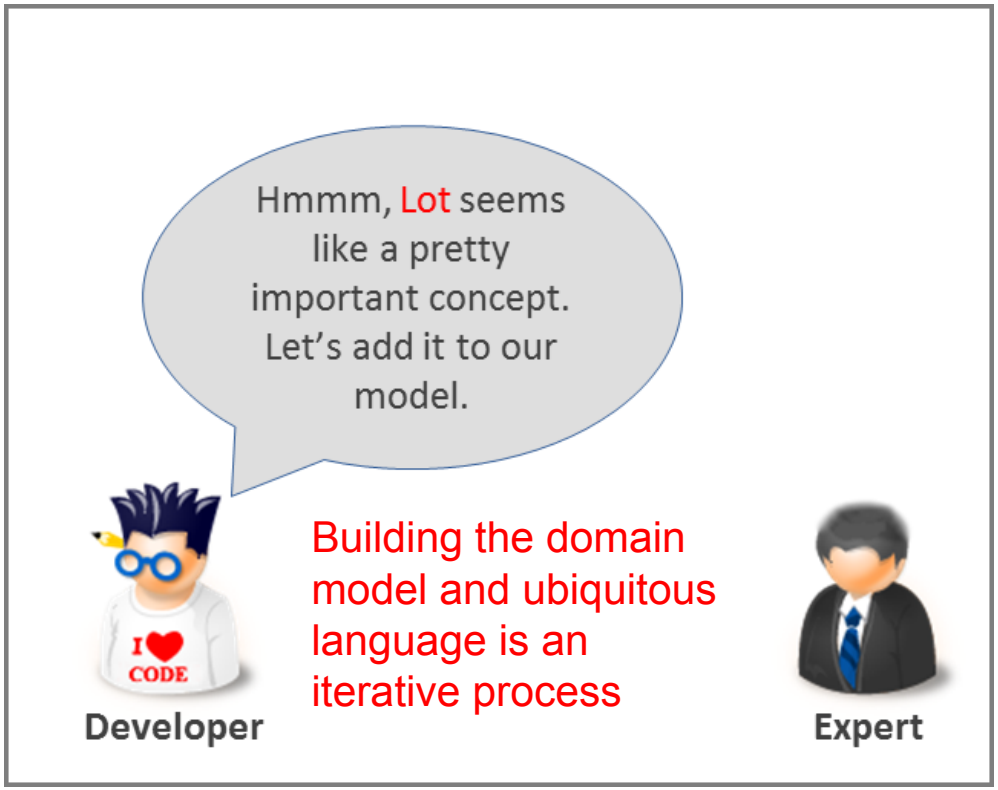
But it turns out that the domain expert actually thinks of this a bit differently.



Building a Domain-Driven Application

Ubiquitous Language

So instead of horseshoeing this new information into our existing implementation, we're going to take time to incorporate this new concept properly into both our mental model AND our software model.



Building a Domain-Driven Application

Ubiquitous Language

The domain model (and therefore the software implementation of that model) will continue to change throughout the project.

Better abstractions, epiphanies, and clarifications will cause us to continuously perfect our understanding and implementation of the model.

How do we iteratively change this model, which is central to the rest of our application, without breaking everything else? That brings us to the next step in our process.

Building a Domain-Driven Application

Isolate the Domain

User Interface Layer	Accepts user commands and presents information back to the user
Application Layer	Manages transactions, translates DTOs, coordinates application activities, creates and accesses domain objects
Domain Layer	Contains the state and behavior of the domain
Infrastructure Layer	Supports all other layers, includes repositories, adapters, frameworks etc.

Building a Domain-Driven Application

Express the Model in Software

Distinguish Entities, Value Objects, and Services

Set Aggregate Boundaries

Implement Repositories and Factories

Building a Domain-Driven Application

Entities

Entities are objects that contain a specific identity in your domain

Long lasting and non-transient, thus need to be persisted to a database for later retrieval

.Equals overload will compare unique properties

Contains all logic pertaining to state and all relevant domain logic

Beware anemic domain anti-pattern

No persistence logic!

Value Object

Value objects are defined by their attributes instead of their identity

Should be immutable, and no identity within domain

Make value objects whole classes that describe simpler Entity classes

Entities vs. Value Object

Examples would be seats on a Southwest Airlines flight. On most airlines, a seat is an entity – a particular seat is in a particular spot in an aircraft and seats a particular person.

But on Southwest, there is no assigned seating, so the seat is just an object with the attribute of being able to hold a person. There is no need to distinguish between separate seats.

Building a Domain-Driven Application

Services

Services are the parts of the domain model that aren't really objects – they are more actions.

For example, in Bullsfirst, we have an Instrument Service that goes out and gets the current prices of all of the stocks in our system.

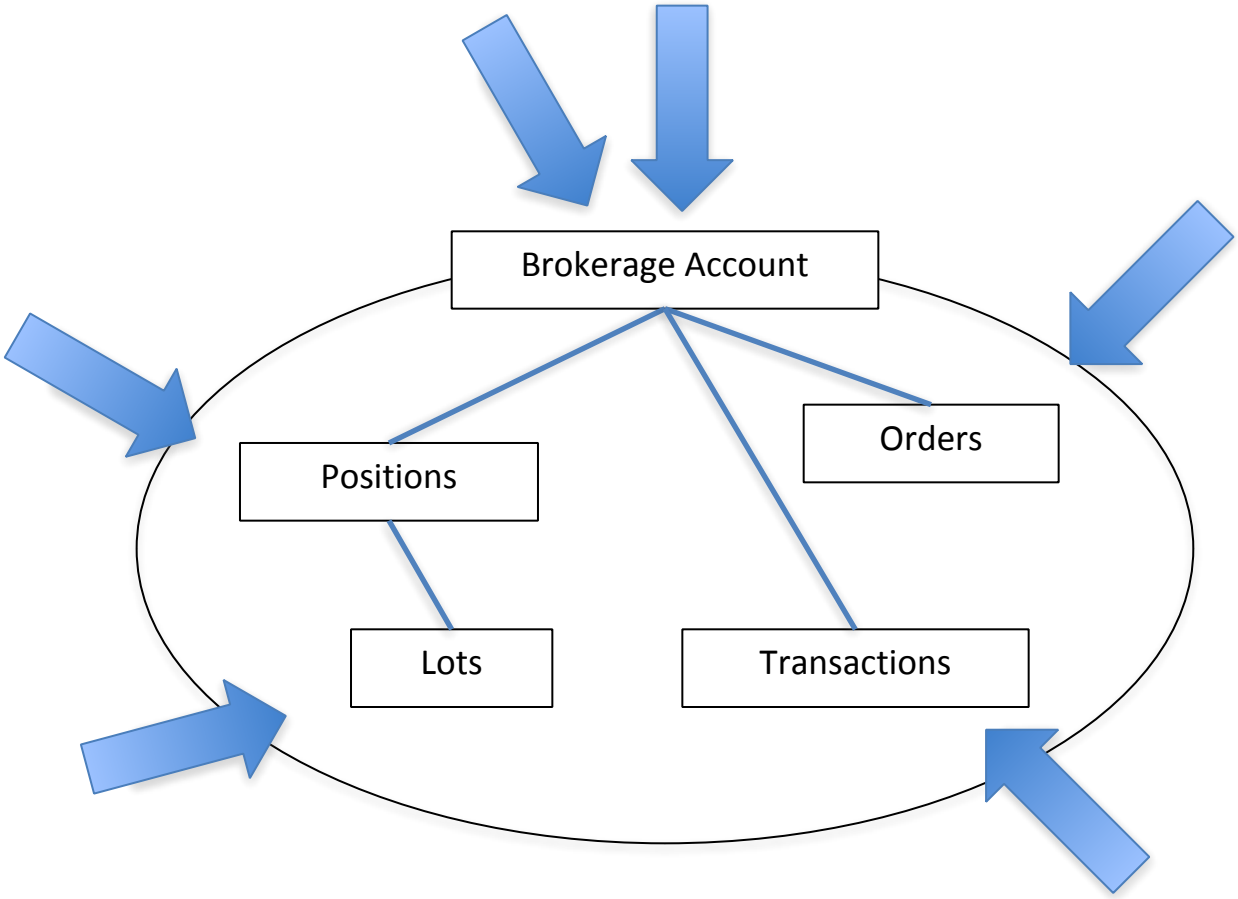
Once we've distinguished the component parts of our domain model, we need to set aggregate boundaries and implement resources and factories.

Set Aggregate Boundaries

When external objects in our UI or application layer can modify all parts of our domain model willy-nilly, it becomes impossible to maintain data integrity.

Instead, we define aggregates based on closely-related objects in our domain model. External objects can then only reference the aggregate root, which must be an entity (it has identity). This root controls access, and enforces business rules within the aggregate, which can contain either entities or value objects.

Building a Domain-Driven Application



Building a Domain-Driven Application

Repositories

Global interface

Provides illusion of in-memory collection of all objects

Provide add/remove methods for managing persistent objects

For each class/aggregate root setup a Repository for in-memory access

Building a Domain-Driven Application

Repositories

Manages the middle and end of an objects lifecycle

Deals with the data store

Caches as necessary

Repositories can be DB, XML, Flat file etc.

Building a Domain-Driven Application

Repositories

In Backbone, dealing with the data store involves invoking the Model or Collection's own fetch() method.

In Bullsfirst, in theory, our data could quickly become stale, so we aren't currently caching most (if any) of our data requests.

Building a Domain-Driven Application

Factories

Take on the responsibility of creating new objects.

Tying to domain or application can be ungainly

The Repository returns data, but the factory creates new objects out of whole cloth.

Tying the responsibility of creating and validating objects to either the domain or the application can be really sludgy, so we generally want to avoid that.

However, Backbone has Collection.add and Model.validate methods that can take care of this for us

Now, you can see:

02 DDD

http://youtu.be/kosVHvkjvgUo

Intention Revealing Interfaces

Student.IsRegisteredIn(course)

Registration.PaymentMethodUsed(creditPayment)

Side Effect Free Functions

Command functions

Modification and return functions

Assertions, AKA Design By Contract

Stand alone classes

Intention-Revealing interfaces

The interface of a class should reveal how that class is to be used

If developers don't understand the interface (and have access to source), they will look at the implementation to understand the class

At that point, the value of encapsulation is lost

So, name classes and methods to describe their effect and purpose, without reference to their implementation

These names should be drawn from the Ubiquitous Language

Supple Design

Side-Effect-Free Functions

Operations (methods) can be broadly divided into two categories: commands and queries

Commands are operations that affect the state of the system

Side effects are changes to the state of the system that are not obvious from the name of the operation. They can occur when a command calls other commands which call other commands etc.; the developer invoked one command, but ends up changing multiple aspects of the system

Queries are “read-only” operations that obtain information from the system but do not change its state

Supple Design

Side-Effect-Free Functions

Operations that return results without producing side effects are called functions

A function can call other functions without worrying about the depth of nesting; this makes it easier to test than operations with side effects

To increase your use of side-effect-free functions, you can separate all query operations from all command operations commands should not return domain information and be kept simple queries and calculations should not modify system state

Supple Design

Assertions

After you have performed work on creating as many side-effect free functions and value objects as possible, you are still going to have command operations on Entity objects

To help developers understand the effects of these commands, use intention-revealing interfaces AND assertions

Assertions typically state three things the pre-conditions that must be true before an operation the post-conditions that will be true after the operation invariants that must always be true of a particular object

Supple Design

Assertions

Many languages provide an assert mechanism

If not, you can move assertions for particular operations to test cases

Here's an example of pre-conditions and post-conditions

```
public void removeAttribute(String name) {
        assert (name != null);
        _atts.remove(name);
        assert (!_atts.keySet().contains(name))
}
```

Supple Design

Standalone Classes

Interdependencies make models and designs hard to understand

They also make them hard to test

We should do as much as possible to minimize dependencies in our models

Modules and Aggregates are two techniques already discussed for doing this; They don't eliminate dependencies but tend to reduce and/or limit them in some way

Supple Design

Standalone Classes

Another technique is to identify opportunities for creating stand-alone classes for domain concepts

Such classes do not make use of any other domain concept

The Pigment Color class of the Paint example in the book is one such instance; it has clearly defined responsibilities and stands alone

Initiation

Refactoring

Timing

Makes the model more flexible

DDD Framework Tools

Caste MonoRail/ASP.NET MVC
NHibernate, Entity Framework, O/R Mappers
Castle Windsor/Spring.NET for DI/IoC
Log4Net/NLog
TeamCity for CI
Nant/MsBuild

Strategy Pattern
Encapsulate domain processes

Composite Pattern
Make whole/part domain concepts interchangable

Specification Pattern
Encapsulate specifications within objects for algorithms

Now, you can see:

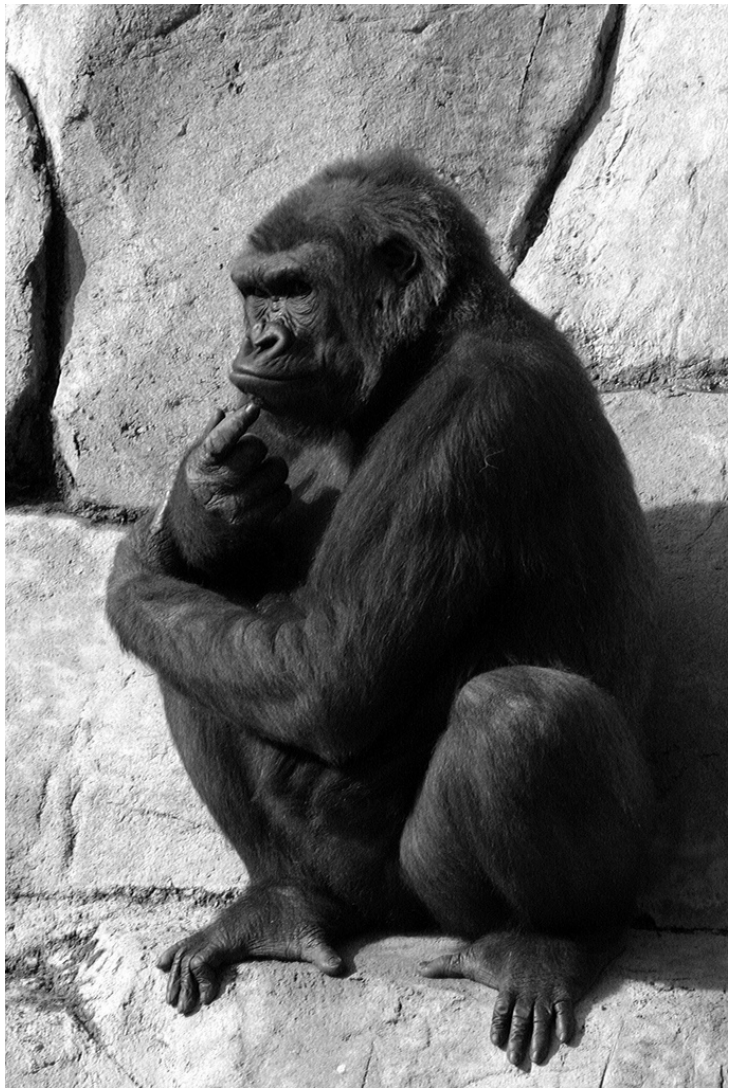
03 The State of the Art in DDD by Eric Evans

http://youtu.be/7eqJeeJyJmY

When you come across what seems to be a fairly complex software problem, DON'T START CODING.

Take a cue from this kid and think about it for a while.

Sit on your hands if you have to.



Thank you!

Globant
we are ready