Designing the M in MVC

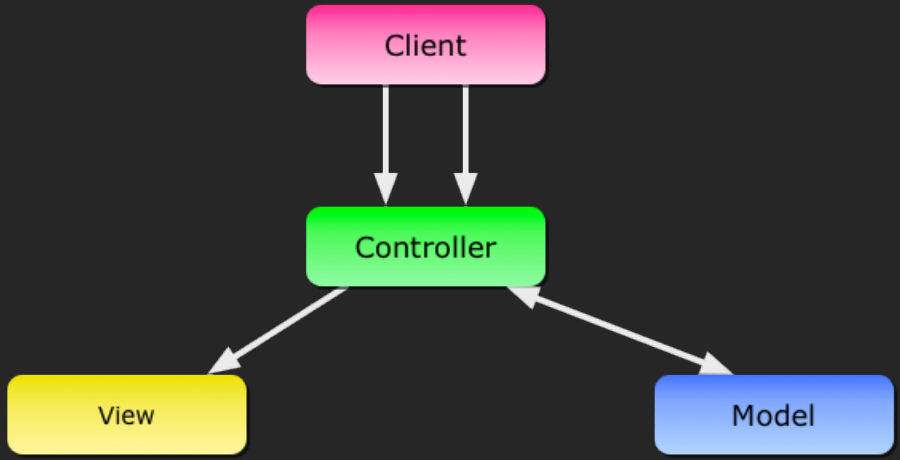
Rob Allen

@akrabat ~ May 2017 ~ http://akrabat.com

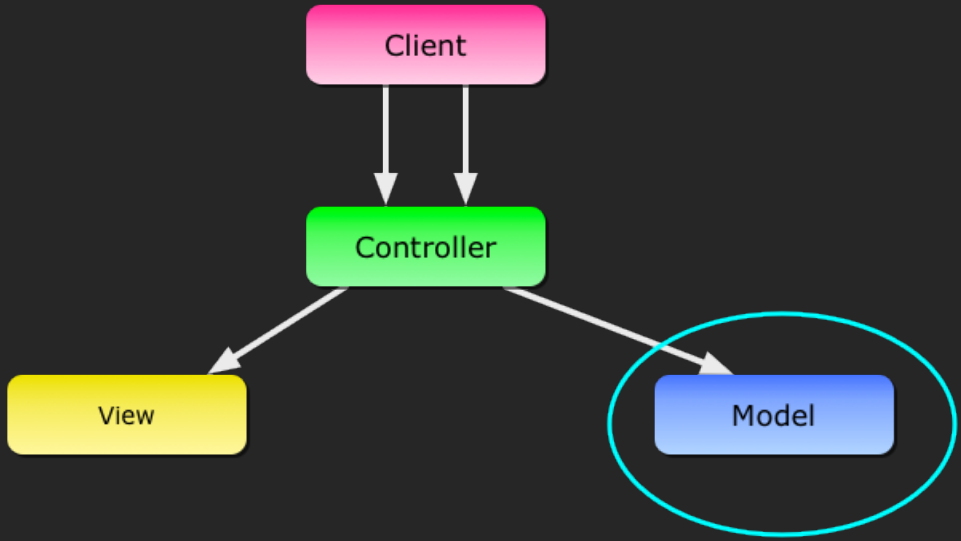
(slides at http://akrabat.com/talks)

The business logic
is the hard part

MVC



MVC

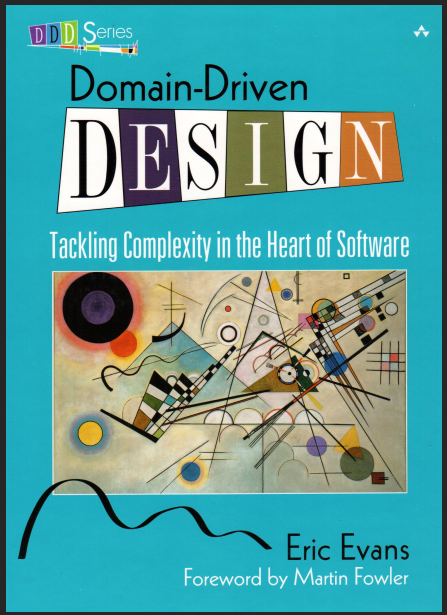


DDD is an approach to help you manage the model

Domain Driven Design

A set of patterns for the model:

- Language
- Strategy
- Tactical



Domain

The **domain** is the "real-world" subject of the project
The **model** is an abstraction of the domain

We communicate the model using:

- diagrams
- use-cases
- specifications
- etc.

Is DDD the right choice?

DDD requires time, effort and collaboration with the business experts.

Benefits:

- Useful model of the problem domain
- Collaboration between the business & the software teams
- Better user experiences with the software
- Better architecture through better understanding

"We have really everything in common with America nowadays except, of course, language"

Oscar Wilde

Ubiquitous Language

- A common language between the developers and the business
- Limited to how the domain problem
- Describes how business itself thinks and operates
- Ensures that the team are all on the same page

The UL is the agreed concepts, meanings and terms for the project.

Creating the Ubiquitous Language

Conversations in the team exploring how the business operates.

- Identify the business processes
- Find the inputs and outputs
- Document it all! pictures, use-cases, glossary, workflow diagrams, etc.

Bounded Contexts & Context Maps

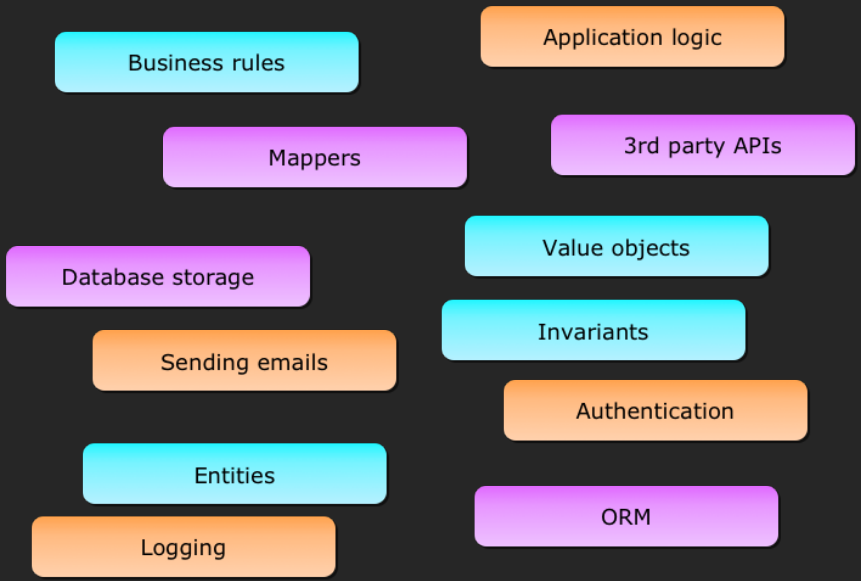
The context where our UL is valid is called the *Bounded Context*

Insurance example:

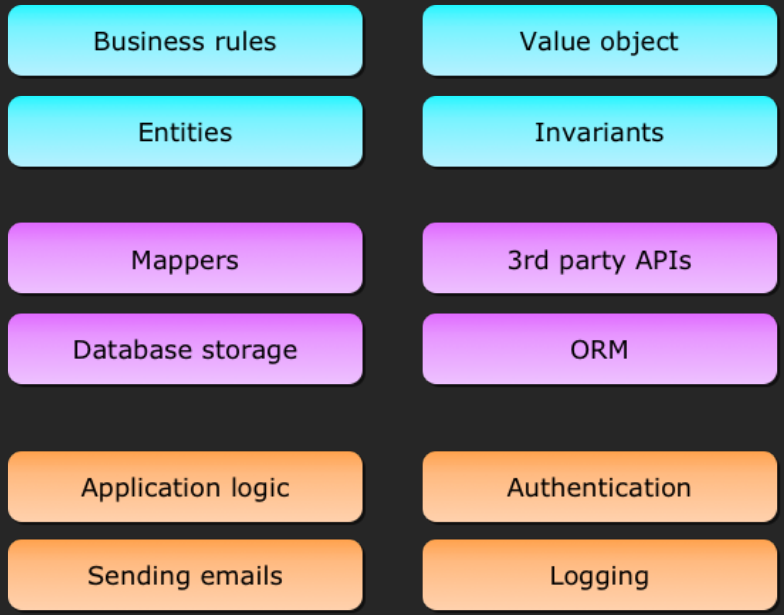
- Quotation
- Policies
- Underwriting
- Billing
- Customer management

Putting it into practice

What's in the model?



Organising the model



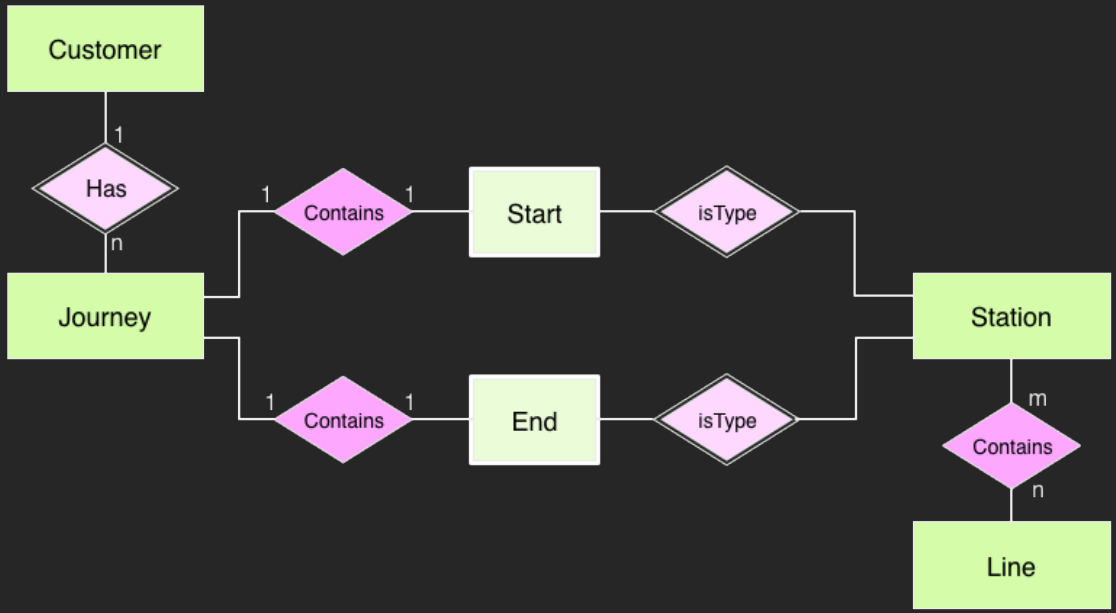
Plan a journey between stations

How do we model this?

Identify the key objects

- Customer
- Journey
- Station
- Line

Identify the relationships



Entities represent things in the problem-space

Entity

- Means something to the customer
- An object defined primarily by its identity
- Mutable
- Persisted
- Has a life cycle

Identity

- The identity of an object is what it means to the customer
- Unique within the scope of the domain

For a tube station, this is as much its *name*, as its database ID.

My customer is never going to talk to me about station 43, they are going to say "Euston Square".

Value objects

- Defined primarily by its attributes
- Immutable
- Simple!
- Do not exist without an entity

Entity vs Value Object

“When people exchange dollar bills, they generally do not distinguish between each unique bill; they only are concerned about the face value of the dollar bill. In this context, dollar bills are ***value objects****.*

However, the Federal Reserve may be concerned about each unique bill; in this context each bill would be an ***entity****.”*

Wikipedia

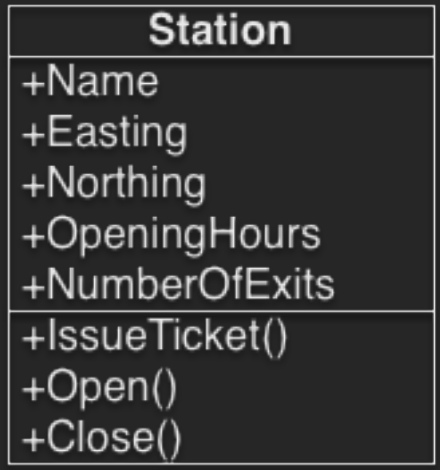
Aggregates

A group of Entities and Value objects that need to be consistent when persisted.

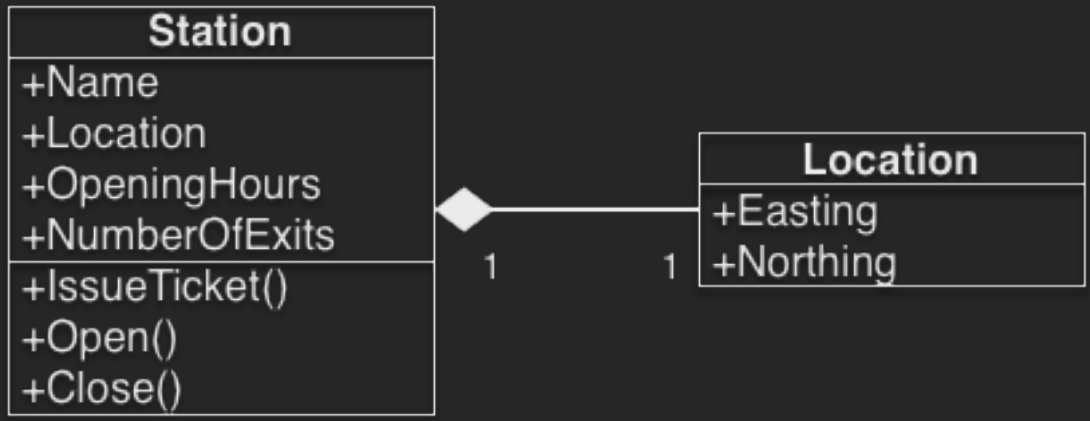
Example:

- Order
- Line item

A station has a location



A station has a location



Domain services

If a SERVICE were devised to make appropriate debits and credits for a funds transfer, that capability would belong in the domain layer.

Eric Evans

Domain services

- We map the business processes to services
- Represents behaviour in the domain
- A service does not have internal state
- Usually a point of connection for many objects

Domain services

Conditions for using a domain service:

- Calculations requiring input from multiple domain objects
- Performing a significant business process
- Transforming a domain object from one composition to another

Let's look at some code

An entity

```
 1 class Journey {
 2     function getStart() {}
 3     function setStart(Station $start) {}
 4
 5     function getStop() {}
 6     function setStop(Station $stop) {}
 7
 8     function getRoute() {}
 9     function setRoute(Route $route) {}
10 }
```

A service

```
1 class RoutingService {
2   pubic function calculateRoute(Station $from, Station $to) : Route {}
3 }
```

Anaemic domain model

When you look at the behavior, and you realize that there is hardly any behavior on these objects, making them little more than bags of getters and setters.

Instead there are a set of service objects which capture all the domain logic.

Martin Fowler

Entity with behaviour

```
1 class Journey {
2     public function __construct (Station $from, Station $to) {}
3
4     public function calculateRoute() : Route {}
5 }
```

What happens if calculateRoute() is complex?

Implement as a helper

The entity calls the helper, passing a reference to itself.

```
 1 // Helper class
 2 class JourneyRouter {
 3     public function calculateRoute(Journey $journey) : Route {}
 4 }
 5
 6 // Journey class
 7 class Journey {
 8     function calculateRoute() : Route {
 9         $route = $this->journeyRouter->calculateRoute($this);
10     }
11 }
```

Persistence

Persistence options

A simple domain model can use TDG or Data Mapper;
a complex one will require Data Mapper or an ORM.
Aggregates need an ORM

Pick the right one!

But don't use ActiveRecord pattern!

But don't use ActiveRecord pattern!

It integrates the database code into your domain model

Table Data Gateway

- Operates on a single database table
- Contains all the SQL for accessing the table
- Doesn't know anything about the entity
- Simple to implement

Table Data Gateway

```
1 class JourneyGateway {
2   function __construct($dbAdapter) {}
3
4   function find($id) {}
5   function findForStartingStation($stationId) {}
6
7   function insert(array $data) {}
8   function update($id, array $data) {}
9 }
```

Data Mapper

- Class to transfer data from objects to the database and back
- Entity aware
- Isolates the domain model from the database
- Not limited to a single table

Data Mapper

```
1 class JourneyMapper {
2   function __construct($dsn, $username, $password) {}
3
4   function find($id) : Journey {}
5   function findForStartingStation($stationId) : [Journey] {}
6
7   public function save(Journey $journey) {}
8 }
```

Increased scope: ORM

Data mapping that works with the object graphs is known as an *Object Relational Mapping*

Must-have for aggregates, but use a pre-written ORM library!

ORM

Persistence layer is more complicated:

- Storage of entire object graphs to the database
- Identity map to hold loaded objects
- Unit of Work to track changed objects for saving

Repositories

- 1-1 relationship between repository and aggregate
- Manage the loading and storing of aggregates
- Often oriented around collections of aggregates

Web services

- The persistence storage could be a web service
- Data mappers work really well

Integrating our model into the application

The application layer*

> *It does not contain business rules or knowledge, but only coordinates tasks and delegates work to collaborations of domain objects in the next layer down.*
>
> Eric Evans

* (Known as *service layer* in PoEAA)

Application layer

We can sub-divide:

- Application services
- Infrastructural services

Application services

If the banking application can convert and export our transactions into a spreadsheet file for us to analyze, this is an application SERVICE.

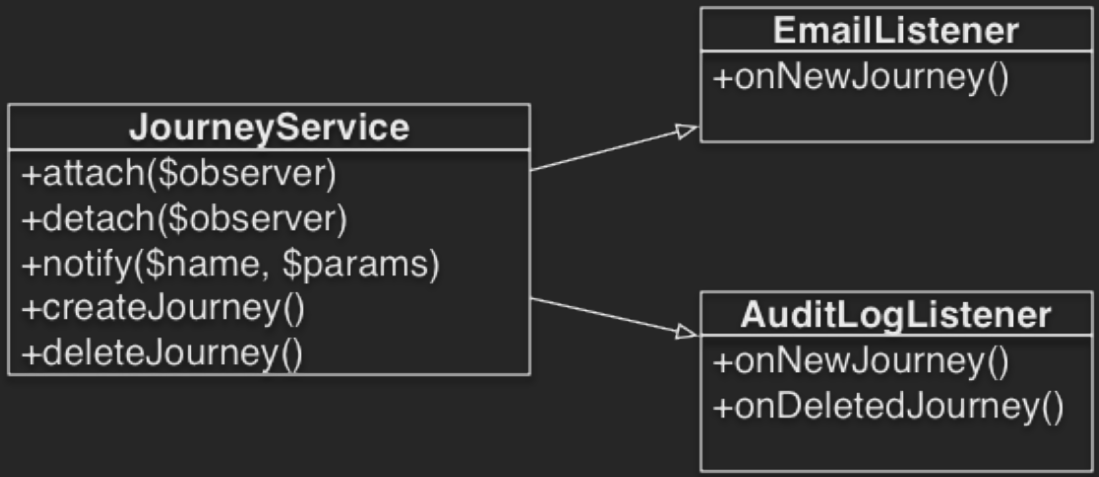
Eric Evans

Application service

```
 1 class JourneyCreator {
 2     public function createJourney(Customer $customer, Station $from, Station $
 3     {
 4         $journey = $customer->createJourney($from, $to);
 5         $journey->calculateRoute();
 6
 7         $this->notifier->send($journey);
 8         $this->auditor->log("Journey created", $journey);
 9     }
10 }
```

Beware the Fat service

Decouple using Observer pattern



Infrastructural services

A bank might have an application that sends out an email to a customer when an account balance falls below a specific threshold. The interface that encapsulates the email system, and perhaps alternate means of notification, is a SERVICE in the infrastructure layer.

Eric Evans

Infrastructural service

- Standard: there's no business or application logic
- Reusable across applications
- Do not write your own - this is what the ecosystem is for

To sum up

To sum up

Ubiquitous Language:

Agreed vocabulary with the business

Bounded Context:

Area of the problem space where the Ubiquitous Language is valid

To sum up

Entity:

Object with identity that do stuff

Value object:

Immutable with no identity

Domain service:

Behaviours that don't belong to an entity

To sum up

Mappers / Repository:

Transfer your model to and from persistence

Application services:

Isolate your domain model from your controllers

Infrastructure services:

Support services for your application

Final point

The success of a design is not necessarily marked by its longevity. It is the nature of software to change.

Eric Evans

Thank you!

Please rate this talk on joind.in

Rob Allen ~ @akrabat ~ akrabat.com