designing with data models

Daniel Jackson

All rights reserved. Daniel Jackson, 2015
purposes

after this class you’ll understand
  why data models are useful
  what a data model is and what it means
  how to construct a data model

key ideas
separation of concerns: rep vs content
abstract structure with relations & sets
data models as invariants

handouts: cribsheet & quiz
what we covered

first class
why data models
syntax & semantics
reading the JS metamodel
comparing notations

second class
review of the JS metamodel (5)
reading the Gmail example (10)
the magic of relations (5)
writing a data model (10)
designing a data model (15)
representation strategies (15)
bad smells (15); automatic analysis (5)
why data models?
databases: so many choices!

**Showings**

<table>
<thead>
<tr>
<th>id</th>
<th>theater</th>
<th>screen</th>
<th>movie</th>
<th>time</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>5</td>
<td>2</td>
<td>7:00pm</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

**Theaters**

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
<th>location</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>3</td>
<td>&quot;Legacy Place&quot;</td>
<td>&quot;Dedham&quot;</td>
</tr>
</tbody>
</table>

**Movies**

<table>
<thead>
<tr>
<th>id</th>
<th>title</th>
<th>rating</th>
<th>genre</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>2</td>
<td>&quot;Fury&quot;</td>
<td>&quot;R&quot;</td>
<td>&quot;action&quot;</td>
</tr>
</tbody>
</table>

**Document**

<table>
<thead>
<tr>
<th>_id</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>title</td>
<td>&quot;Fury&quot;</td>
</tr>
<tr>
<td>time</td>
<td>&quot;7:00pm&quot;</td>
</tr>
<tr>
<td>rating</td>
<td>&quot;R&quot;</td>
</tr>
<tr>
<td>genre</td>
<td>&quot;action&quot;</td>
</tr>
<tr>
<td>theater name</td>
<td>&quot;Legacy Place&quot;</td>
</tr>
<tr>
<td>theater location</td>
<td>&quot;Dedham&quot;</td>
</tr>
</tbody>
</table>
in mongo: still choices!

**embedded**

```json
{
  title: "Fury",
  time: "7:00pm",
  theater: {
    name: "West Newton Cinema",
    location: "Newton"
  }
}
```

one document in the collection **Movies**

**relational**

```json
{
  title: "Fury",
  time: "7:00pm",
  theater: 1
}
```

one document in the collection **Movies**

```json
{
  _id: 1,
  name: "West Newton Cinema",
  location: "Newton"
}
```

one document in the collection **Theaters**
But nothing is gained —on the contrary! — by tackling these various aspects simultaneously. It is what I sometimes have called “the separation of concerns”, which, even if not perfectly possible, is yet the only available technique for effective ordering of one’s thoughts that I know of. This is what I mean by “focussing one’s attention upon some aspect”: it does not mean ignoring the other aspects, it is just doing justice to the fact that from this aspect’s point of view, the other is irrelevant. It is being one- and multiple-track minded simultaneously.

Dijkstra, On the role of scientific thought (EWD447)
separation of concerns

biggest idea in programming?

what the data is vs how to represent it

a tool for thinking
one concern at a time: a way to make progress
helps expand representation possibilities
surprise: focus exposes subtle issues in what data is

other advantages
can separate roles: designer vs. implementer
data is central

state characterizes behavior
invariant is like a planet’s orbit

not just databases
rich transient state too
wanted...

**a notation for data**
lightweight, minimal & easy to learn
more succinct than code
more precise than sketches

**representation independent**
leaves rep choices open
not tied to a particular database
an analogy: architectural plans

Stata
Student
Street

doors

column

stair
syntax of data models
example email app
IT WILL DO YOU GOOD.

Classify

Spread recycling!! To save limited natural resources for our children’s future.

© 1991 Produced by Super Planning Company Limited
identify sets of atoms

Message

Address

Folder

a SET of messages
specialize sets

*a folder is user-defined or predefined*

classification is IMMUTABLE: Inbox can’t become Trash

these happen to be singleton sets (but might not be)
identify dynamic subsets

Message

Unread

no immutability marking: a message can be unread now and not unread later

not italicized: may have messages that are not unread
relate atoms (attributes)

relation is target
IMMUTABLE: to address doesn’t change
relate atoms (fluents)

don’t want one relation represented by two arcs

a generalization, introduced to unify relations
determine multiplicities

Message from Address

Address to Message

exactly one

one or more

Object

Folder

Message

contents

at most one
adding textual constraints

one inbox, trash and sent folder
predefined folders not contained by or contain others
no unread messages in sent folder
contents is acyclic
syntax summary sets

A \subseteq B

A \subseteq B

A_1 \subseteq B \land A_2 \subseteq B

A_1 \subseteq B \land A_2 \subseteq B

A_1 \cap A_2 = \emptyset

A_1 \cap A_2 = \emptyset

B \subseteq A_1 \cup A_2

B \subseteq A_1 \cap A_2 = \emptyset

B \subseteq A_1 \cup A_2
syntax summary relations

\[ R \subseteq A \times B \]

- Over time, each A is mapped by R to the same Bs
- Over time, R maps the same As to each B
- R maps each A to n Bs
- R maps m As to each B
- \(+\) one or more
- \(*\) zero or more
- \(!\) exactly one
- \(?\) at most one
- omitted = *

used when R1 and R2 share same properties

short for:

\[ A \xrightarrow{R1, R2} B \]
semantics of data models
model = set of instances

\{x, y: \text{Nat} \mid x+y = 4\}

syntax: a constraint on a pair of numbers

\{(x:0, y:4), (x:1, y:3), (x:2, y:2), (x:3, y:1), (x:4, y:0)\}

semantics: a set of instances

syntax: model

semantics: set of instances
example satisfying instance

one Inbox and one Trash and one Sent
no Predefined.\text{contents} & Folder
no Sent.\text{contents} & Unread
acyclic [\text{contents}]
example relating wrong kinds

one Inbox and one Trash and one Sent
no Predefined.contents & Folder
no Sent.contents & Unread
acyclic [contents]
example multiplicity violation

one Inbox and one Trash and one Sent
no Predefined.contents & Folder
no Sent.contents & Unread
acyclic [contents]
example multiplicity violation

one Inbox and one Trash and one Sent
no Predefined.contents & Folder
no Sent.contents & Unread
acyclic [contents]
example textual constraint violation

one Inbox and one Trash and one Sent
no Predefined.contents & Folder
no Sent.contents & Unread
acyclic [contents]
monotonicity

adding diagram features adds constraints, so you can grow the diagram

\[ \text{Message} \xrightarrow{\text{from}} \text{Address} \]

\[ \text{from} \subseteq \text{Message} \times \text{Address} \]

\[ \text{Message} \xrightarrow{\text{from}!} \text{Address} \]

all \( m: \text{Message} \mid #m.\text{from} = 1 \)

\[ \text{from} \subseteq \text{Message} \times \text{Address} \]

\[ \text{Message} \xrightarrow{\text{from}!} \text{Address} \]

all \( m: \text{Message} \mid #m.\text{from} = 1 \)

all \( m: \text{Message} \mid m.\text{from} \text{unchanging} \)
exercise: reading data models
javascript objects

review constructors at Prototypes/Constructors in JavaScript Live slides

exercise
draw some interesting instances
add some textual constraints
solution: javascript

some constraints
slots of an object have distinct names
object can’t be its own prototype
function object has slot with name ‘prototype’

an interesting case
object and its prototype have slots with same name

a tricky case: is this true?
all o: Object | some s: o.constructor.slots | s.name = ‘prototype’ and s.value = o.prototype
why the ‘prototype rule’ isn’t true

not all objects have prototypes
the chain ends at the top with Object.prototype

you can change the value in the prototype slot
see JavaScript Live slides: Prototypes/Constructors
different from modifying the object in that slot!
exercise
what constraints hold?
in particular, how are clabels related to mlabels?
gmail labeling constraints

essential principle: clabel derived from mlabel

all c: Conversation | c.clabels = c.msgs.mlabels

some strange consequences
search for unlabelled messages gives conversations with labels
executing delete(label); add(label) modifies a conversation

a puzzle
can a conversation have a Sent label?
lesson

model forces precision & focus which results in tricky & subtle questions it’s never so simple as you first think
the magic of relations
strange atoms

note that atoms are
distinguishable: have an identity

but they’re also
immutable: don’t change
indivisible: not structured

so how to express structure & mutability?
relations!
(and subsets for fluents)
kinds of relationship

- Message
  - Body
  - File
    - attachments

  "containment"

- User
  - User Name

  "naming"

- File
  - Permission
    - perms
    - grantedTo
      - User

  "properties"

- User
  - groups
    - Group

  "association"
exercise:
writing
a data model
movie database schema

```javascript
var Showing = mongoose.model('Showing', mongoose.Schema({
  startTime: String,
  theater: {type: Number, ref: 'Theater'},
  movie: {type: Number, ref: 'Movie'}}));

var Movie = mongoose.model('Movie', mongoose.Schema({
  _id: Number,
  title: String}));

var Theater = mongoose.model('Theater', mongoose.Schema({
  _id: Number,
  name: String,
  location: {type: Number, ref: 'Address'}}));

var Address = mongoose.model('Address', mongoose.Schema({_id: Number,
  zip: String}));
```

exercise

draw an object model for this schema

guess suitable mutability markings
extending the model

can you add these features?
theater chains (eg, Cinema De Lux, Regal, Showcase)
film festivals (program of movies shown around town)
double features (two movies in one sitting)
automatic analysis of data models
about alloy

Alloy is a language for describing structures and a tool for exploring them. It has been used in a wide range of applications from finding holes in security mechanisms to designing telephone switching networks.

An Alloy model is a collection of constraints that describes (implicitly) a set of structures, for example: all the possible security configurations of a web application, or all the possible topologies of a switching network. Alloy's tool, the Alloy Analyzer, is a solver that takes the constraints of a model and finds structures that satisfy them. It can be used both to explore the model by generating sample structures, and to check properties of the model by generating counterexamples. Structures are displayed graphically, and their appearance can be customized for the domain at hand.

At its core, the Alloy language is a simple but expressive logic based on the notion of relations, and was inspired by the Z specification language and Tarski's relational calculus. Alloy's syntax is designed to make it easy to build models incrementally, and was influenced by modeling languages (such as the object models of OMT and UML). Novel features of Alloy include a rich subtype facility for factoring out common features and a uniform and powerful syntax for navigation expressions.

The Alloy Analyzer works by reduction to SAT. Version 4 was a complete rewrite that included Kodkod, a new model finding engine that optimizes the reduction, and a new front end.
alloy in action: the model

```alloy
open util/relation
sig Address {}
abstract sig Object {}
sig Message extends Object {
  from, to: Address
}
sig Unread in Message {}
abstract sig Folder extends Object {
  contents: set Object
}
sig UserDefined, Predefined extends Folder {}
one sig Inbox, Sent, Trash extends Predefined {}

fact {
  -- contents in Folder lone -> Object
  no Predefined.contents & Folder
  -- no contents.Predefined & Folder

  no Sent.contents & Unread
  acyclic [contents, Object]
}

run {
  some contents.contents
  -- some Unread
} for 6
```
alloy: showing graphical model

graphical model, generated from text
alloy: an instance

generated automatically

UserDefined

Inbox

Message

Address

Sent

Trash

contents: 2
from: 1
to: 1

contents

to

contents

from

oops! we need more constraints
our graphical notation
has a textual version
is formal, so analyzable

in 6170
you’ll write textual constraints
as addenda to diagrams
can use Alloy if you like
or just English...
bad smells of data modeling
don’t be lazy with names

why?
it’s never as obvious as it seems
choosing a good name helps designer get clarity
beware primitive types

why?
- type has syntactic or semantic properties
- so may want to store in special way,
  and/or use special validators
don’t duplicate

why?
recognize shared properties & generalize
leads to cleaner user interface & cleaner code
don’t mention low level ids

why?

every object has an implicit identity anyway
how it’s represented is an implementation detail
but note: user-visible ids (such as usernames) are relevant
don’t split types

why?
distinct types are disjoint, so couldn’t ask whether movie and book have same title
so atoms to be compared must have the same type
don’t use set when order matters

why?
tuples of a relation have no order
implementer can choose an unordered collection
use subsets, not boolean flags

why?
flag is low level way to represent
obscures dynamic classification
prevents recording multiplicity graphically
don’t use attributes

why?
attributes are an ill-defined idea, and just complicate the notation
better to factor out the relations that matter
don’t confuse state with actions

why?

data model describes what is *remembered*
that is, what’s stored in the state
don’t use subsets for relational state

why?
subset is with respect to a context (a user)
without this, data structure won’t work
another example of bad subset
collections aren’t domain objects

why?

collection objects are implementation details unless they have properties beyond their contents
don’t split a relation

why?
splitting obscures generalization
leads to duplication in code
watch out for 3-way relations

why?
student-class-grade is a 3-way relation
need a “tuple” type such as Enrollment
another example of 3-way relation
model data that matters

- don’t ask ‘what do I know about students?’
- but: ‘what must my app know about students?’

why?
- no point modeling the easy stuff
- focus on the hard parts
- in this case how a student is identified matters
- home town probably does not (except maybe for MIT Giving)
exercise: designing a data model
party play app

the idea
host of a party plays music
wants to let her guests suggest songs to play

how it works
guests suggest songs on youtube
host adds the suggestions she likes

your task
device a data model for this app
a team assignment app

**problem**
designing an app to make team assignments
students enter their preferences
an algorithm or superuser makes teams

**exercise**
create a data model for the app
representation strategies
movie database model

Diagram:
- Showing
  - startTime
  - theater
  - movie
- Time
- Theater
  - location
  - zip
- Movie
  - name
  - title
- String
- Address
  - zip
- Zipcode
objects become classes (Java), documents (Mongo), tables (SQL)

principle
fields map to ? or !

identify objects
**choose nesting**

- **principle** achieve encapsulation guided by mutability

  - **in Mongo:** might nest Address in Theater; other relations become refs
  - **in Java and SQL:** no nesting

  - in Java: create abstract types; in Mongo: avoid copying mutables; in SQL: irrelevant

```
SHOWING
  ┌───────────┐
  │ startTime │
  │          │
  │          │
  └───────────┘
  │ theater │
  │        │
  │        │
  └─────────┘
  │ movie │
  │      │
  │      │
  └-------┘
  │ location │
  │         │
  │         │
  └---------┘
  │ name │
  │     │
  │     │
  └------┘
  │ title │
  │      │
  │      │
  └-------┘
```

- **Zipcode**
  - zip

- **Address**
  - location

- **Movie**
  - name
  - title

- **String**
  - name

- **Theater**
  - location
  - name

- **Time**
  - startTime
introduce collections

principle
only when needed

in Mongo:
Showing,
Theater,
Movie

in Java:
Theater,
Movie

in Java: root object with collections as fields; in Mongo: collections; in SQL: not necessary
add redundant components

principle
base on expected navigations

in Mongo: index Showing by Movie title

in Java: Movie.getShowings()

in Java: extra fields; in Mongo: indexes; in SQL: indexes, extra tables
define keys

principle
use multiplicities to identify keys

in Mongo or SQL: multikey index for Theater

in Java: Theater.find(name, zip)

in Java: extra fields; in Mongo: indexes; in SQL: indexes, extra tables
comparing notations
origins of our modeling notation

- logic diagrams (Euler, Venn, Peirce)
- ZF set theory
- relational calculus (Tarski)
- relational model (Codd)
- Z notation
- ER & other data models
- object model notations (OMT etc)
- model checking
- Alloy Language
- Alloy Diagrams
- Unified Modeling Language

Mathematical logic
- object-oriented development
- software verification
- relational databases
the entity-relationship model

Fig. 11. An entity-relationship diagram for analysis of information in a manufacturing firm

alloy diagram notation

- all relations binary
- for 3-way relationship, introduce tuple
- can express generalization
- mutability
- for 3-way relationship, introduce tuple
why Alloy diagrams?

simple & small syntax
  comparable to ER
  much less complex than UML

easy to draw
  requires fewer special widgets

crucial features for modern programming
  generalization (subsets/supersets)
  mutability markings

backed by textual notation
  but don’t expect you to use it