why data modeling?
databases: so many choices!

**relational**

<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>

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

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

**object-oriented**

App
- showings
- theaters

Map
- time
- movie
- theater

Movie
- title
- rating
- genre

Theater
- showings
- name
- location

String
- "Fury"

Map
- "Legacy Place"
- "Dedham"

Showings
- _id: 3
- title: "Fury"
- time: "7:00pm"
- rating: "R"
- genre: "action"
- theater: name: "Legacy Place"
  location: "Dedham"
in mongo: still choices!

**embedded**

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

one document in the collection **Movies**

**relational**

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

one document in the collection **Movies**

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

one document in the collection **Theaters**
separation of concerns

a tool for thinking
- one concern at a time
- helps expand representation possibilities
- surprise: exposes subtle content questions

other advantages
- can separate roles: designer vs. implementer

biggest idea in programming?

what the data is vs how to represent it

seen this before?
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)
wanted...

a notation for data
› lightweight, minimal & easy to learn
› omits implementation decisions
› no bias to any particular database
› more expressive than databases
origins of our modeling notation

1700

1900

1940

1970

1980

1990

2000

ZF set theory

relational calculus (Tarski)

relational model (Codd)

Z notation

ER & other data models

model checking

Alloy Language

object model notations (OMT etc)

 Unified Modeling Language

Alloy Diagrams

logic diagrams (Euler, Venn, Peirce)

mathematical logic

object-oriented development

software verification

relational databases

1700

1900

1940

1970

1980

1990

2000

model checking
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.
data modeling notation
example email app
IT WILL DO YOU GOOD.  

Classify

Spread recycling!! To save limited natural resources for our children’s future.
identify sets of things

- Message
- Address
- Folder
specialize sets
identify dynamic subsets

Message

Unread
relate things (attributes)
relate things (fluents)
determine multiplicities

![Diagram with nodes: Message, Address, Object, Folder, Message, and directed edges labeled from, to, !, +, contents, and ?.]
graphical 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

- $A \subseteq B$
- $A \subseteq B$
- once an $A$, always an $A$
- $A_1 \subseteq B \land A_1 \subseteq B$
- $A_1 \cap A_2 = \emptyset$
- $A_1 \subseteq B \land A_1 \subseteq B$
- $A_1 \cap A_2 = \emptyset$
- $B \subseteq A_1 \cup A_2$
syntax summary relations

\[ R \subseteq A \times B \]

over time, each \( A \) is mapped by \( R \) to the same \( B \)s

over time, \( R \) maps the same \( A \)s to each \( B \)

\( R \) maps each \( A \) to \( n \) \( B \)s
\( R \) maps \( m \) \( A \)s to each \( B \)

+ one or more
* zero or more
! exactly one
? at most one
omitted = *
instance
example satisfying instance

one Inbox and one Trash and one Sent
no Predefined.contents & Folder
no Sent.contents & Unread
acyclic [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.\textit{contents} & Folder
no Sent.\textit{contents} & Unread
\textbf{acyclic} \{\textit{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

\[ \epsilon \]

\[
\begin{align*}
    \text{(Inbox)} & \xrightarrow{\text{contents}} \text{M0 (Unread)} \\
    \text{(Sent)} & \xrightarrow{\text{to}} \text{A0 (Address)} \\
    \text{(Trash)} & \xrightarrow{\text{contents}} \text{M1 (Message)} \\
    \text{(UserDef)} & \xrightarrow{\text{contents}} \text{A1 (Address)} \\
    & \xrightarrow{\text{to}} \text{A2 (Address)} \\
    \text{F0 (UserDef)} & \xrightarrow{\text{contents}} \text{F1 (UserDef)}
\end{align*}
\]

one Inbox and one Trash and one Sent
no Predefined.contents & Folder
no Sent.contents & Unread
acyclic [contents]
alloy in action
strange things

note that things are
 › distinguishable: have an identity
 › immutable: don’t change
 › indivisible: not structured

all structure and change is expressed in
 › relations: properties, containment, association, naming
 › subsets: membership in fluent set is like a boolean property
bad smells of data modeling
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 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 aren’t comparable
so atoms to be compared must have shared 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

Theater

Closest Theater

User

Location

currentLoc

closest

location

✔

✘
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
care with 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

Example 1 (incorrect):
- Theater
- Movie
- Time
- Rating
- String

Example 2 (correct):
- Theater
- Showing
- Time
- Movie
- Showing
- Showing
- Movie
- Rating
- String
model data that matters

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: modeling movies
are multiplicities and mutabilities right?
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)
design moves
making a nested structure

```json
{title: "Fury"}
{zip: "02139"}
{location: {zip: "02139"},
  name: "Kendall"}
{ 
  theater: {location: {zip: "02139"},
    name: "Kendall"},
  movie: {title: "Fury"}
  startTime: "7pm",
}
reversing a relation

{title: "Fury"}

{zip: "02139"}

{location: {zip: "02139"},
  name: "Kendall"}

{
  theater: {location: {zip: "02139"},
             name: "Kendall"},
  movie: {title: "Fury"}
}

{
  time: "7pm",
  showings: [
    theater: {location: {zip: "02139"},
              name: "Kendall"},
    movie: {title: "Fury"}]
}
relating structures

```
{title: "Fury"}

{zip: "02139"}

{location: {zip: "02139"},
  name: "Kendall"}

{
  startTime: "7pm",
  theater: 2,
  movie: 1
}

{__id: 1, title: "Fury"}

{__id: 2, location: {zip: "02139"},
  name: "Kendall"}
```
choosing a key

key must be a unique identifier
removing objects

```
{zip: "02139"}
{location: {zip: "02139"},
  name: "Kendall"}
{
  startTime: "7pm",
  movieTitle: "Fury",
  theater: {location: {zip: "02139"}}
}
```
adding objects (1)

why do this?
in anticipation of extending the model
eg, adding `Timing.mainAttractionStartTime`

```json
{startTime: "7pm"}
{timing: {startTime: "7pm"}, ...}
```
adding objects (2)

why do this?
to allow representation of many:many relation in relational DB, no other choice
repertoire of design moves

basic moves
› reverse relation
› add/remove object
› nest objects
› choose key

other moves
› add redundancy
› add index
subtypes

{name: "Kendall"}
{name: "Legacy Place", owner: 33}
subtypes

if subtype has no relations of its own
add type field

{title: "Fury", type: "new release"}
{title: "Psycho", type: "classic"}
justifying moves
what is data design?

large space of design moves
› many ways to nest, relate, restructure

but which is right?
› depends on how much data, how it’s used

also “technical debt”
› easy coding now or pay for it later?
factors that influence design choices

- storage space
- speed of queries
- speed of updates
- maintaining consistency
- complexity of queries
- complexity of updates
- flexibility for future elaboration

- choose to nest
- choose to relate
- add object
- remove object
- reverse relation
- relation is many:one
- relation end is immutable
evaluating this design
mutable location, size of theater record ⇒ theater not in showing
immutable title, search on movie ⇒ movie in showing
how to document design

data model
› diagram with textual constraints
› definitions of any obscure relations or sets

rep design
› contours drawn on diagram
› rationale for choices
› alternatives rejected

code must correspond to design!

if you must, fake it
› write the code first, then construct justification
› but eventually learn to think & express abstractly