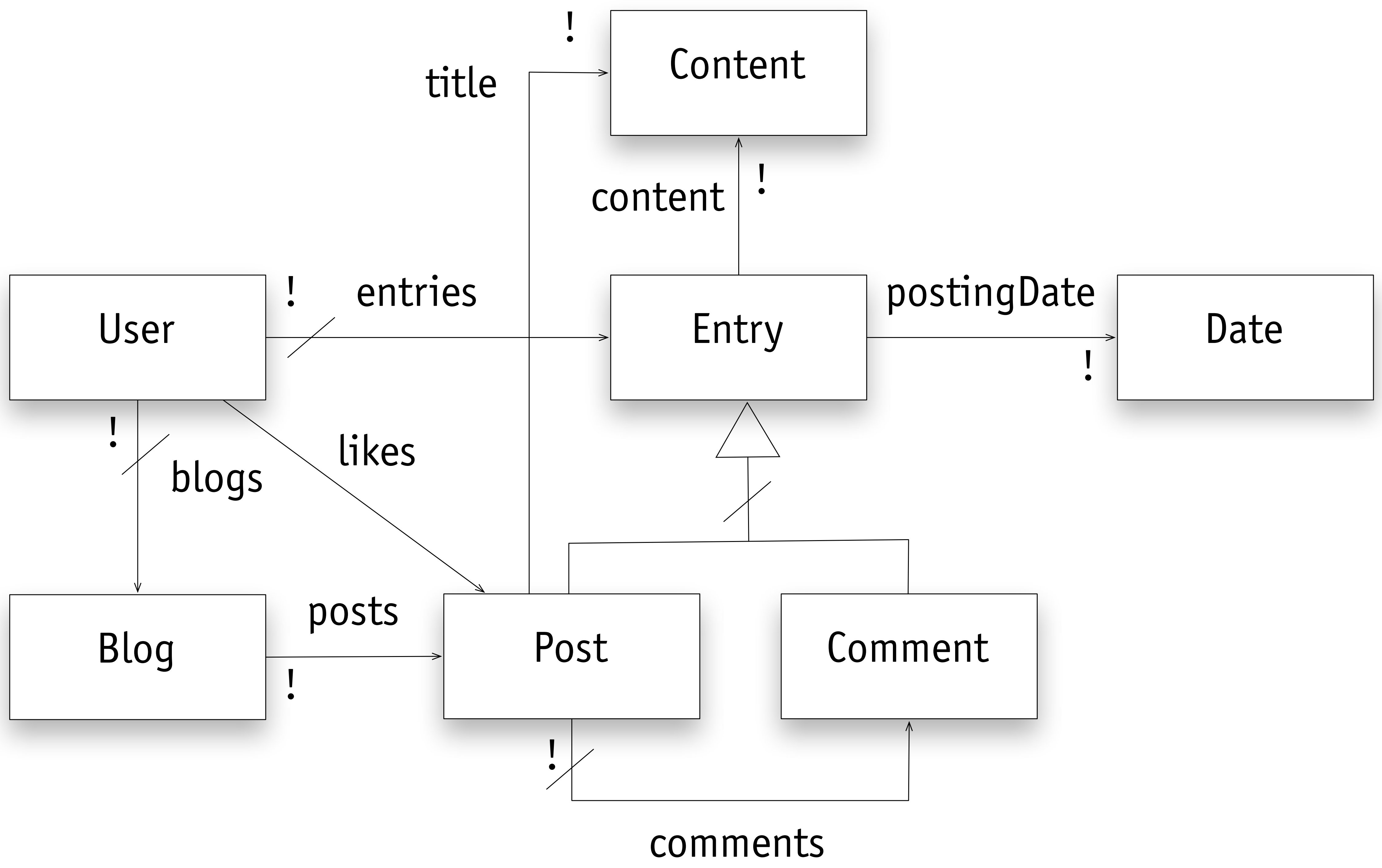
software
studio

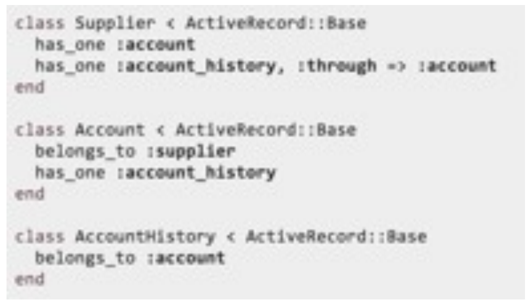
# implementing object models

Daniel Jackson

# an object model



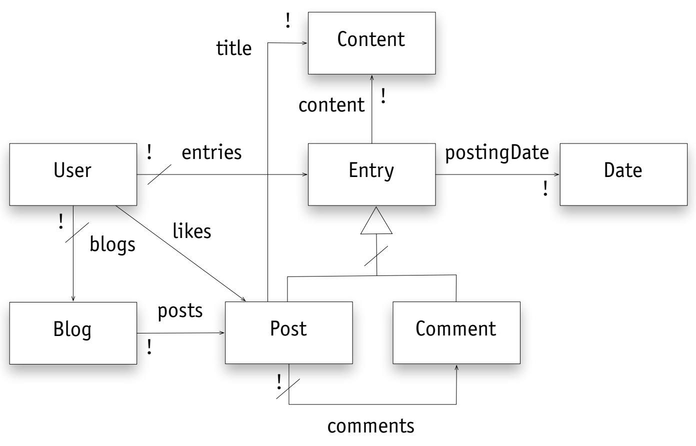
# one model, many implementations



```ruby
class Supplier < ActiveRecord::Base
  has_one :account
  has_one :account_history, :through => :account
end

class Account < ActiveRecord::Base
  belongs_to :supplier
  has_one :account_history
end

class AccountHistory < ActiveRecord::Base
  belongs_to :account
end
```



?

```ruby
class Supplier < ActiveRecord::Base
  has_one :account
  has_one :account_history, :through => :account
end

class Account < ActiveRecord::Base
  belongs_to :supplier
  has_one :account_history
end

class AccountHistory < ActiveRecord::Base
  belongs_to :account
end
```

```ruby
class Supplier < ActiveRecord::Base
  has_one :account
  has_one :account_history, :through => :account
end

class Account < ActiveRecord::Base
  belongs_to :supplier
  has_one :account_history
end

class AccountHistory < ActiveRecord::Base
  belongs_to :account
end
```

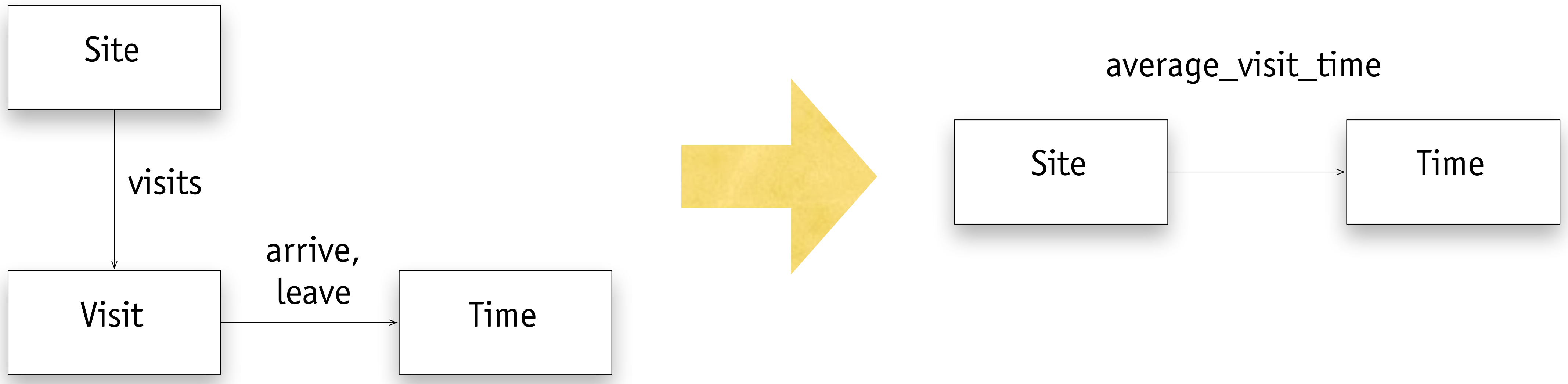
# question 1: what to represent?

**principle**

› don't actually want to store every object

**example: web analytics**

› may have set Visit
› but perhaps too many visits to save
› so instead store stats (eg, #visits)



# question 2: which sets are classes?

## principle

› some sets can be represented as primitive datatypes

## example: Content

› store as text attribute of Entry?

› or as model class in its own right?

## when to use a class?

› no suitable primitive

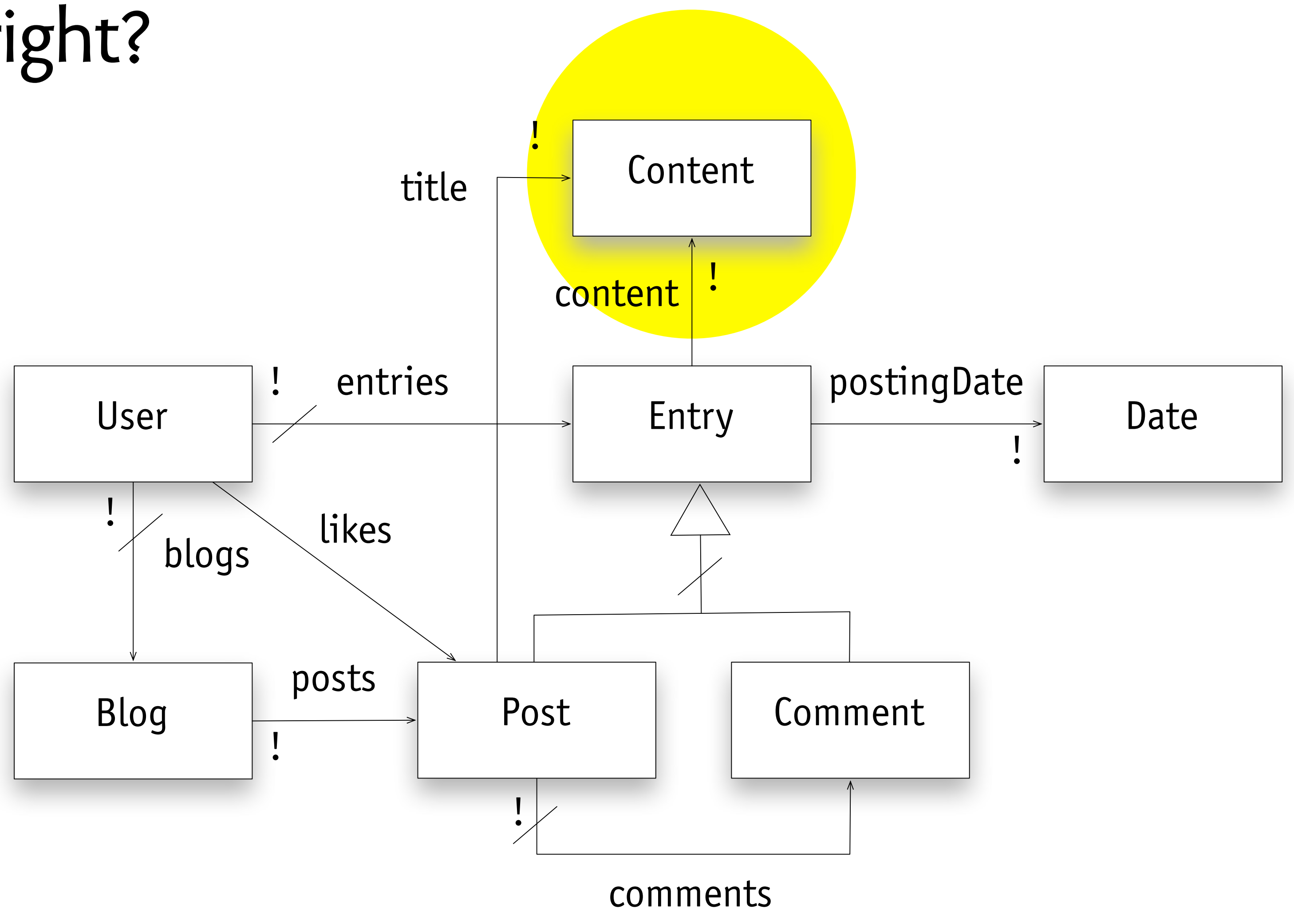
› want methods on type

› will have its own attributes

## another option

› Rails aggregation

› lets you map multiple columns to single object



## question 3: which associations?

**principle**

› create associations to allow navigation

**example: entries**

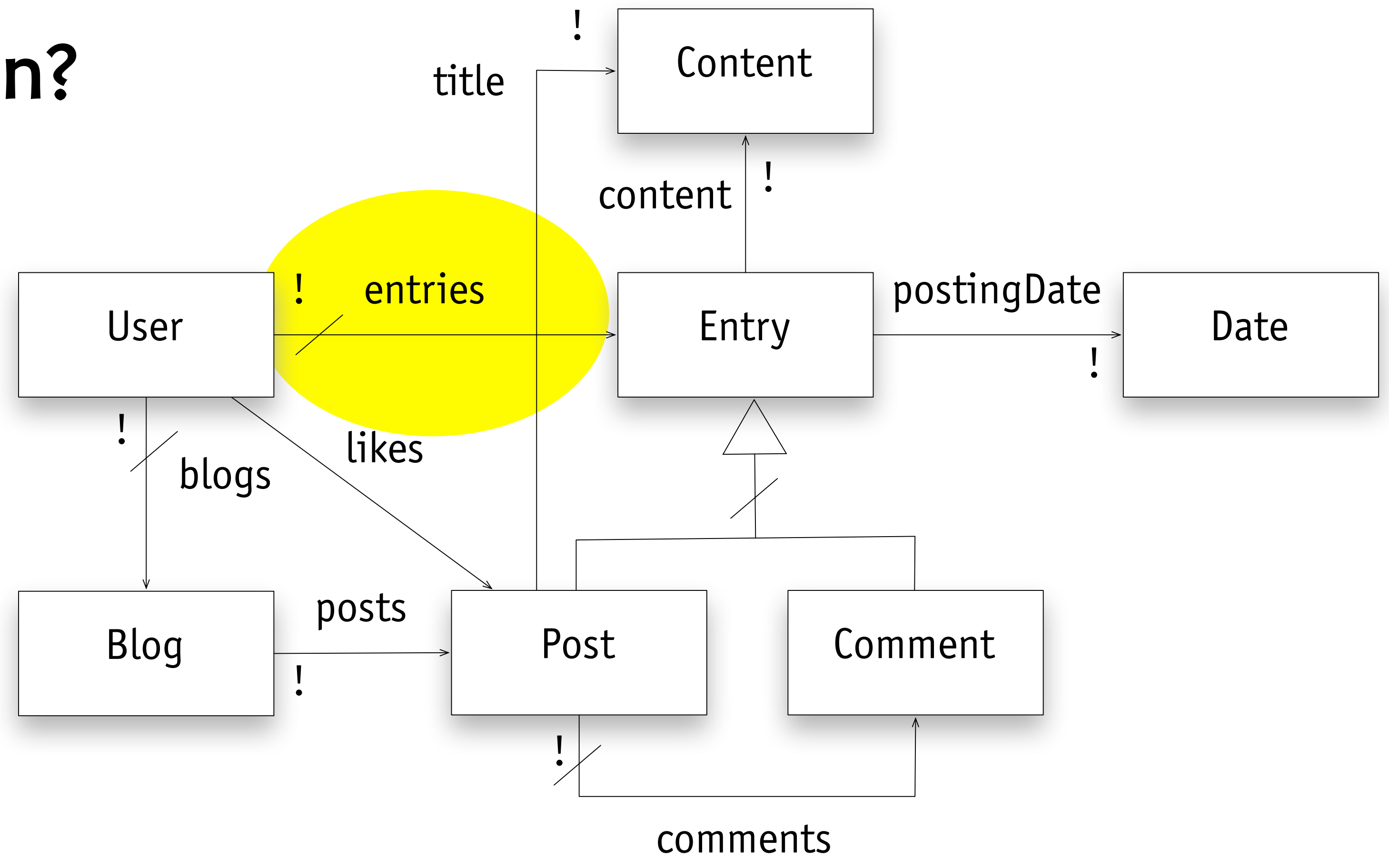
› has_many in User?

› belongs_to in Entry?

**when to declare an association?**

› declare association in A to B if you want to navigate from A to B

› eg, user.entries, entry.user



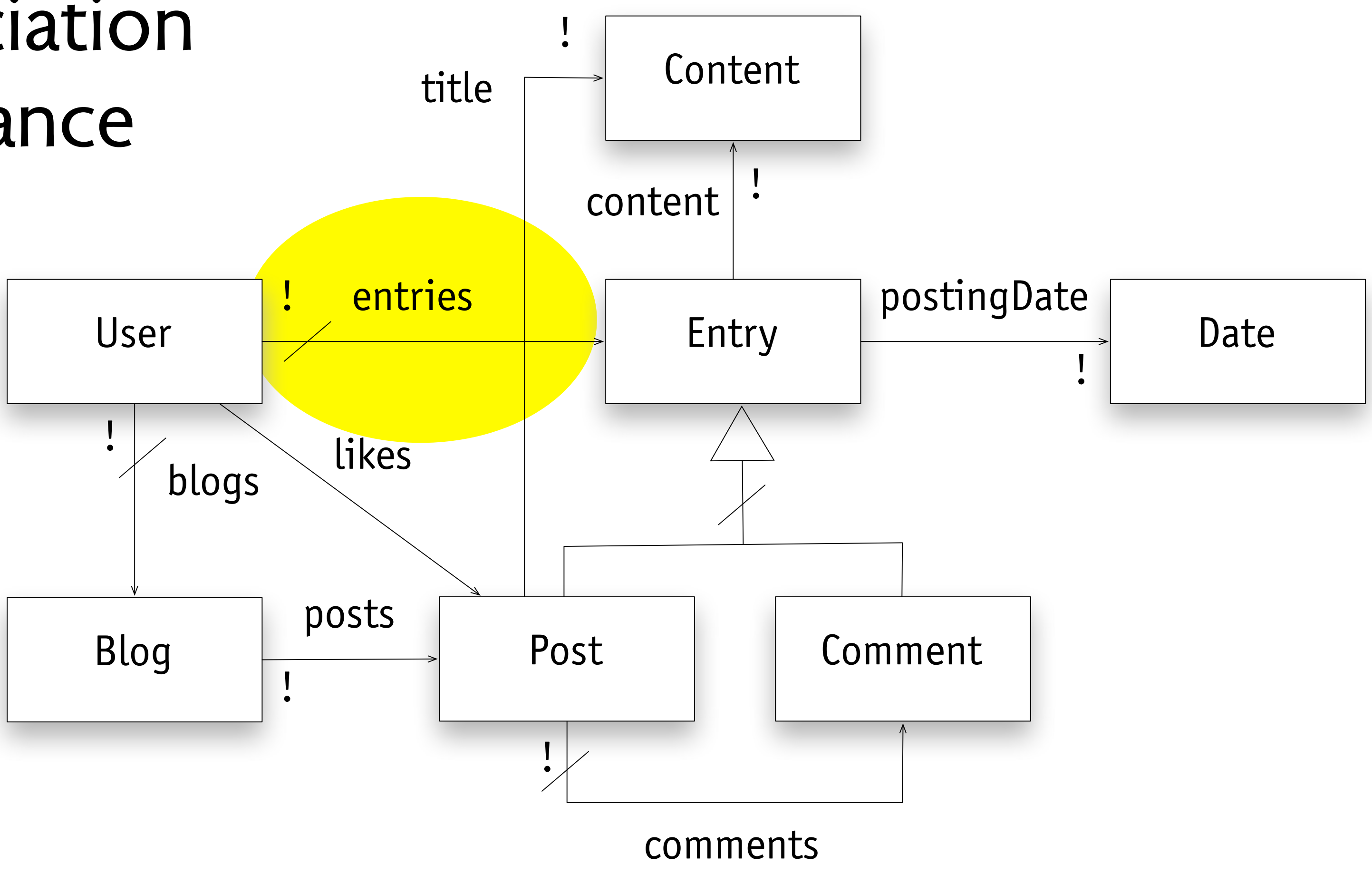
## question 4: generalizations?

**principle**

- databases don't have inheritance, so need special hacks

**options**

- two classes, no generalization
- one class, no generalization
- 2 classes, polymorphic association
- 3 classes, single table inheritance



# advanced considerations

**redundancy**

- avoid costly joins for frequent navigations
- add redundant data
- example: blog.user.profile.contact.email -> blog.email

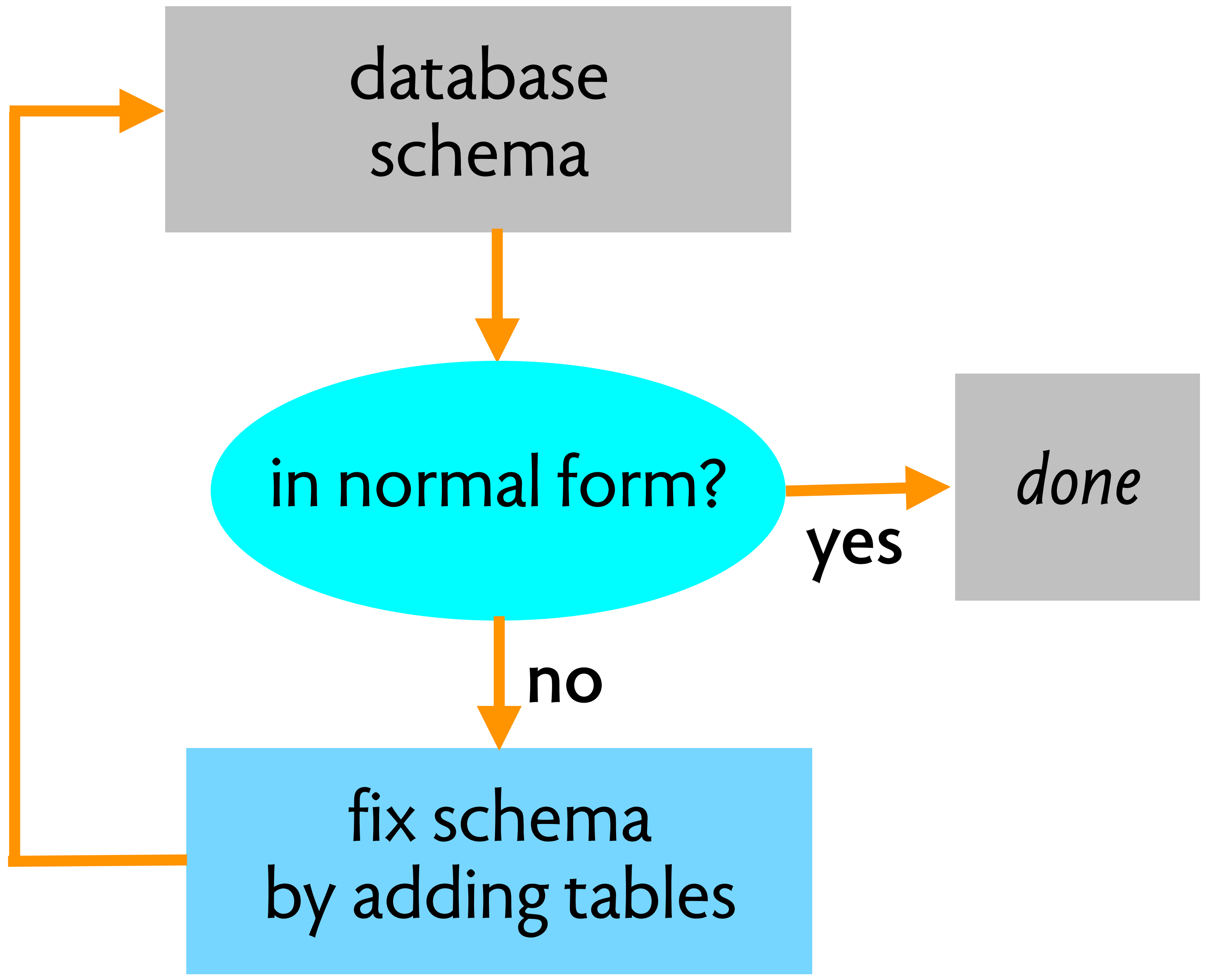
**security**

- separate critical data into separate table(s)
- fine-grained access control supported by databases
- example: credit card number for customers

**avoiding contention**

- beyond this course (transactions, locking, etc)
- but good to separate high/low frequency
- example: don't store user profile and user tracking in same table

# traditional database design



# example of normal form violations

*reviews*

| *reviewer* | *subject* | *rating* | *email* | *ratingstars* |
|---|---|---|---|---|
| Chloe Closure | Lucid | 3 | cc@mit | *** |
| Chloe Closure | Clover | 5 | cc@mit | ***** |
| Ann Alert | Clover | 5 | aa@mit | ***** |
| Ben Bitdiddle | Cosi | 3 | ben@mit | *** |
| Ben Bitdiddle | Lucid | 4 | ben@mit | **** |

**example**

› a reviewing database in one table

**second normal form**

› no field depends on just part of a key

› key is (<u>reviewer</u>, <u>subject</u>), <u>email</u> depends on <u>reviewer</u> alone

**third normal form**

› no field depends another field but not on the key

› eg, <u>ratingstars</u> depends on <u>rating</u> alone

## so?

**these problems don't arise**

- if you started with an object model

**but may arise if you**

- started with tables
- or with a bad object model