comments mechanism
problem analysis: reviews

Daniel Jackson
principle #1: adequacy of state

principle
› the designed state must be rich enough to support all functions

application
› consider each operation and check that object model supports it
› achieve succinctness by hiding detail

example
› do we need relation by: Comment -> User ?
› if commenting is anonymous, perhaps not
› if readers can’t navigate from author name to profile, perhaps not
› but any of these ops suggest relation is needed:
  edit_comment (user, comment)
  audit_comments (user)
  upvote (user, comment): requires user is not author of comment
**Devote**

*Requires:* user previously upvoted comment  

*Modifies:* upvotes table, upvote count for comment  

*Effects:* upvote is deleted from upvotes table, upvote count for comment is decremented
Object Model:
First Stage:

![Object Model Diagram]

Notes:
- A user can only give a comment one upvote, which is why those are a separate model and not just a counter on a comment.
principle #2: generalization

principle
› identify commonalities between sets for better uniformity in design and cleaner implementation

application
› when objects have shared properties, consider defining superset
› but don’t overdo it: note distinctions too

example
› users create posts, comments and responses
› these share author, dates, content
› but posts aren’t upvoted, may contain formatting, are editable
› but responses on comments and responses, and not on posts
› so might decide to generalize over comments and responses
one approach: responses are no different from comments
are differences between Reply and Post warranted?
› no voting on replies
› no reply on reply
a simple but effective generalization
nice to address moderation

- but what exactly does off_mod_queue mean? accepted or rejected?
- consider refining relations: comment on unmoderated comments?
- split moderates into: assigned, approved, rejected?

upvotes: nice use of relation!
(but won’t hold date of upvote, eg)
shared properties of users and admins?
good classification
› address relations to: split manages?
unifies comments and objects

“Objects are pieces of content that the user (client) designates as commentable (videos, images, blog posts, etc.)”

abstract sig Commentable {
    comments: set Comment
} {
    all c: comments | c.on = this
}
sig Comment extends Commentable {
    on: one Commentable,
    owner: lone User,
}
sig Object extends Commentable {
    site: one Site,
}
sig Site {
    objects: set Object,
    owner: one User,
} {
    all o: objects | o.site = this
}
sig User {
    comments: set Comment,
    sites: set Site,
}
principle #3: preserve in code

principle
› preserve generalization in code to simplify, reduce duplication and obtain correspondence between model and code

application
› for each generalization, choose an option
   option A: no generalization in code: two model classes
   option B: no generalization in code: one model class with tags
   option C: single table inheritance
   option D: polymorphic association
option A: two classes

everything

- Comment, Response in Post

approach

- create two distinct model classes, Comment and Response
- associations
  class Article < ActiveRecord::Base
    has_many :comments
  class Comment < ActiveRecord::Base
    belongs_to :article has_many :responses
  class Response < ActiveRecord::Base
    belongs_to :comment
option B: one class

example
› Vote, Downvote in Vote

approach
› create a Vote class with boolean attribute up
option C: single table inheritance

example
› Comment, Response in Post

approach
› rails generate scaffold Post type:string
› class Post < ActiveRecord::Base
› class Comment < Post
› class Response < Post

how it works
› creates one database table, but lets you use separate model classes

for more info
› http://www.martinfowler.com/eaaCatalog/singleTableInheritance.html
› http://ar.rubyonrails.org/classes/ActiveRecord/Base.html
option D: polymorphic association

example
› Staff, Customer in User
› comments: User -> Comment

approach

class Customer < ActiveRecord::Base
  has_many :comments, :as => :user

class Staff < ActiveRecord::Base
  has_many :comments, :as => :user

class Comment < ActiveRecord::Base
  belongs_to :user, :polymorphic => true

how it works
› Rails maintains id and type of user in comment table

for more info
› http://guides.rubyonrails.org/association_basics.html#polymorphic-associations