software studio

design review: shopping cart

Daniel Jackson

puzzle

MIT tuition

› $20,885/term = $5221/course = $500/week

question you might ask

› what do I get for my money and effort?

› isn't this stuff I could learn by myself? or at a company?

question we ask

› what can we teach you that you can't learn elsewhere?

our answers

› big ideas: what's below the surface

› reflection: learning to design consciously (SOC, articulation)

› sensibilities: simplicity & clarity

› abstractions: rep-independent data model

reflective design questions

why?

- why am I doing this?
- what am I trying to achieve?

what?

- what am I trying to build?
- who or what will it interact with?

how?

- what state will I need to record?
- what will the key concepts be?

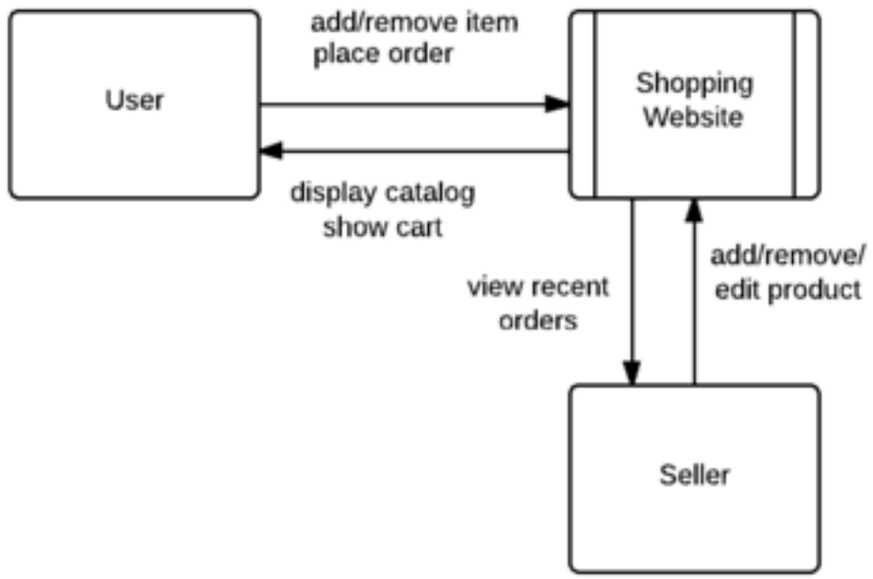
but..?

- what challenges arise?
- how might it fail or be broken?

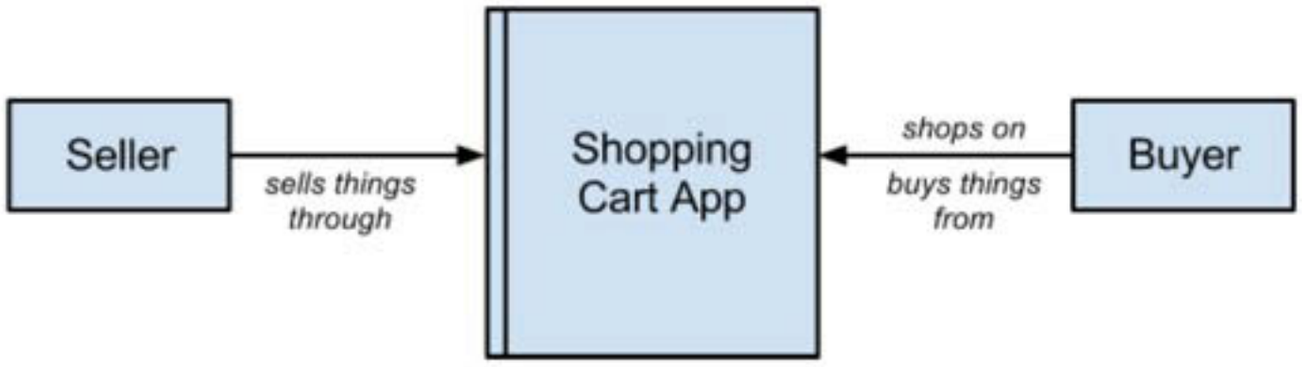
context diagrams

context diagrams

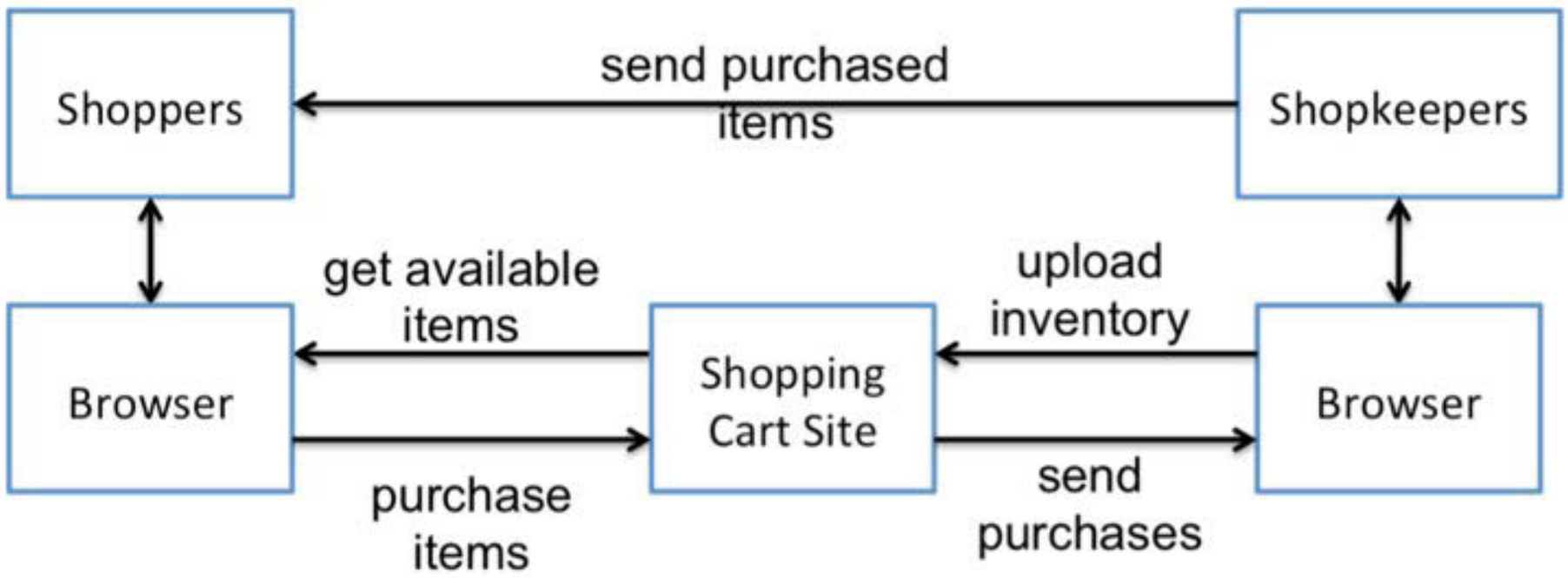
doesn't identify system



good

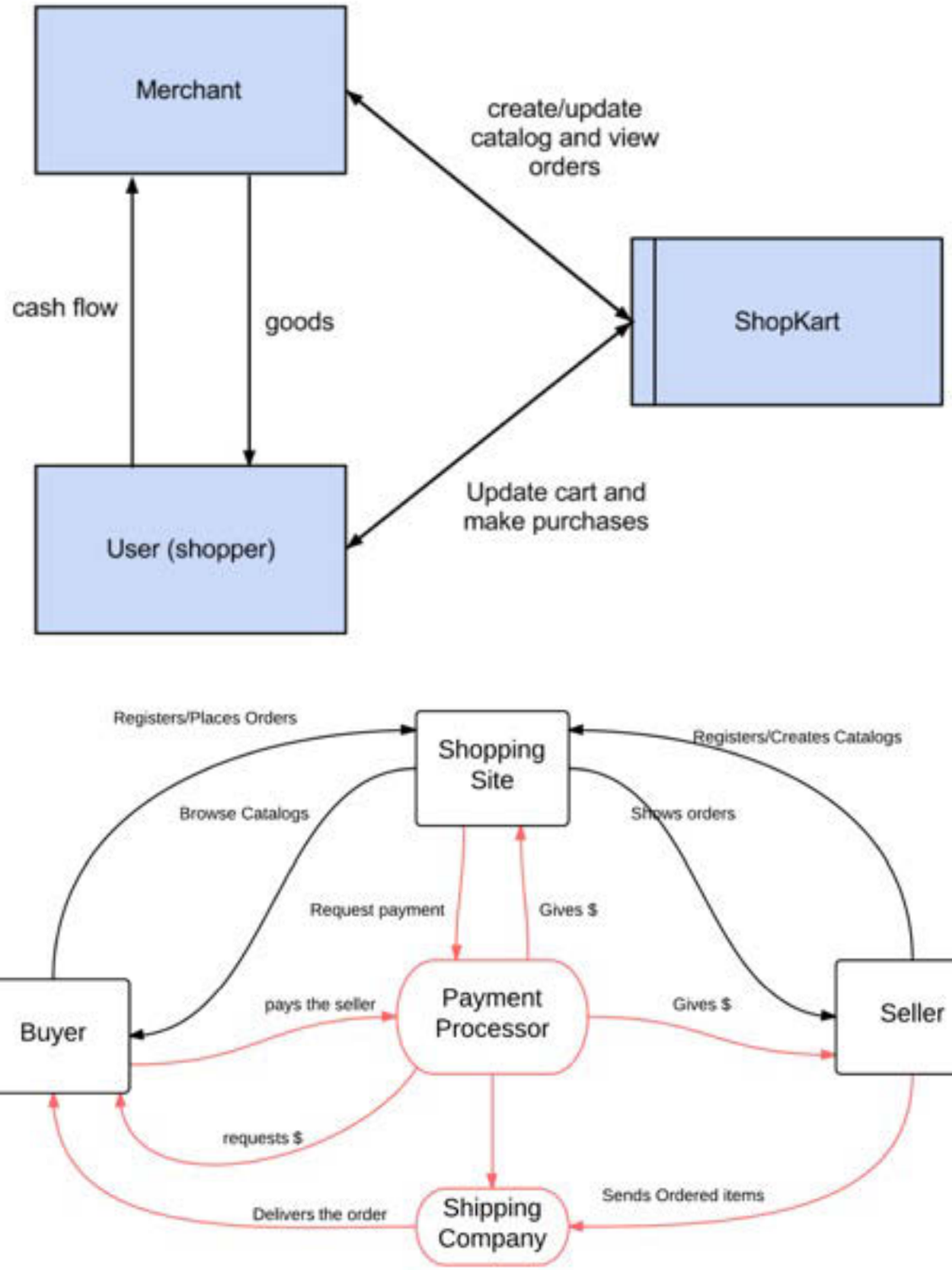


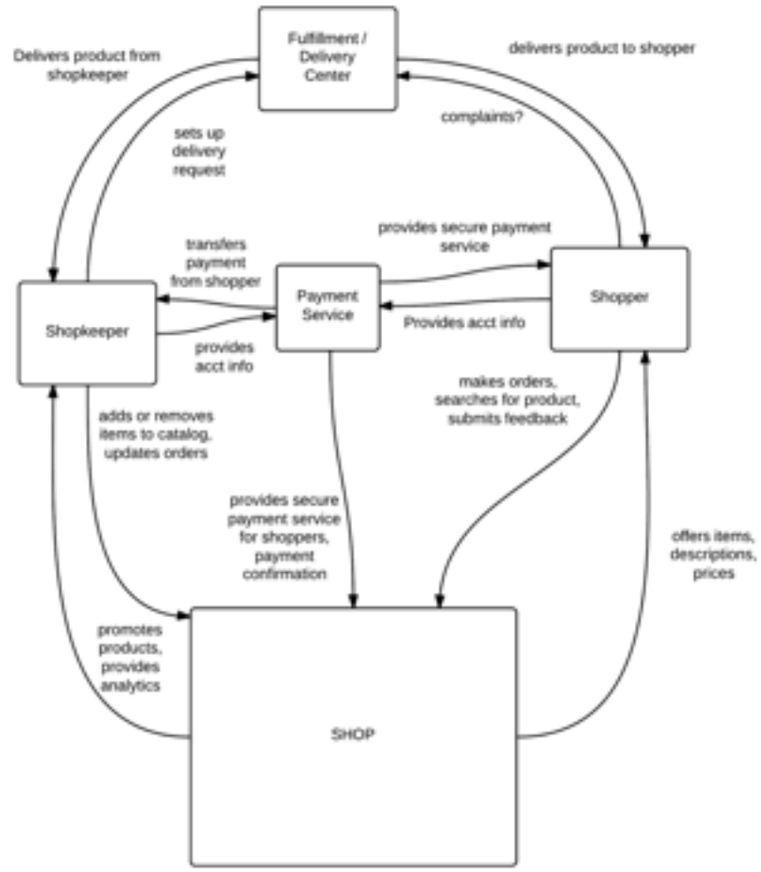
vague actions & flows

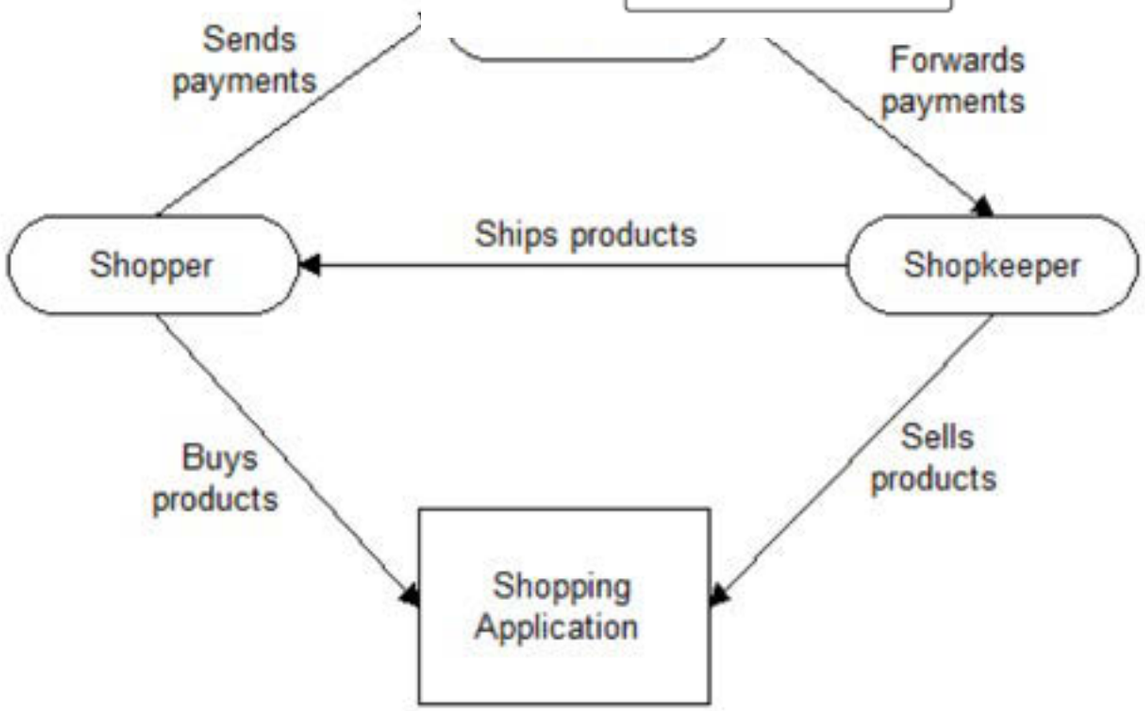


nice, but why browsers?

thinking about context: yes!







summary

what is context diagram?

› defines boundaries of system
› who/what does it talk to?

what you're trying to do

› identify external interfaces, gross dataflows

common mistakes

› splitting system into multiple boxes
› using arrow for direction of action, not dataflow
› missing important features (eg, maintaining catalog)
› missing important actions (eg, viewing items)

concepts

concepts: don't repeat yourself

Key Concepts:

Item: An item for sale.
relations: has_many cart_items, has_many order_item, has_many carts through cart_items, has_many orders through order_items
attributes: price, name, shop_id

DRY: code details

AZCart models the following concepts:

- **User**: Represents a user on the site, could be either a Shopper or a Shopkeeper
- **Catalog**: Represents a catalog of items. Each Shopkeeper owns a Catalog and can add/remove/edit items in the Catalog.
- **Cart**: Represents a cart of items. Each Shopper owns a Cart and can add/remove/edit quantities of items in the cart.
- **Order**: Once the Shopper hits "Checkout," an Order is made from the Cart, which is available for the shopkeeper to approve or deny.
- **Item**: Represents an individual item in the Catalog. Each item can also belong to multiple Carts and Orders.

DRY: OM details

Key Concepts
Shopkeepers are able to use the site to portray the goods and services they offer and other details such as price and inventory. Customers in turn can aggregate items they wish to purchase and once they have selected everything they wish to purchase, they can complete one transaction to purchase these items.

DRY: features

concepts

Key concepts

The key concepts in ShopKart are **carts, catalogs,** and **products.** A catalog consists of all the products the merchant chooses to make available for people to buy, and a user places products in his cart for purchase. The notion of a cart also provides a method to view past orders: when a cart is purchased, its internal attributes are frozen, and it is now considered a past order.

nice, but explain products

concepts

The key concepts in the design of ShoppingCart are users, carts, line_items, and items.

- User: Can be a signed-in (recognized) or not signed-in (unrecognized) visitor to the site. A user has only one active cart at a time. A user can also be a seller, and modify the catalog.
- Cart: A cart is a collection of line_items. An active cart is what can be modified by the user, while an inactive cart is an order: a historical view at what customers have ordered. A cart belongs to a single users, who in turn can only have one active cart at a time.
- Line_item: A line_item is an item in a cart,and has the same attributes as items plus a quantity. It gets deleted when the product is represents gets deleted.
- Item: An item is an item for sale. It has a certain price, name, and description.

nice, but reduce overlap with OM; also explain item

summary

what are concepts?

› key ideas that characterize the design

what you're trying to do

› separate the obvious from the non-obvious
› define novel or confusing concepts precisely

common mistakes

› repetition: overlap with features and OM
› not focusing on novelties (eg, cart vs order)
› missing key issues (eg, one of a kind item vs descriptor)
› watering down with minor details
› circular definitions (item is an item)

features

Following are seminal features offered by Flexion:

nice categorization

1. Maintaining a Cart

   Flexion allows users to maintain items they wish to purchase in a cart temporarily.

2. Buying Items

   Users can decide when they wish to buy items by checking out and paying for items existing in their cart.

3. Maintaining an Inventory

   Flexion allows users to maintain items they wish to sell in an inventory. The items in a user's inventory are visible for purchase to all users of Flexion.

4. Selling Items

   Users sell an item successfully when they add the item to their inventory and another user purchases the item; Flexion provides the transaction protocol required for this to occur, giving users the capability to sell items.

5. Browsing Items without a User Account

   Users can browse items being sold by other users of Flexion without having to create an account or login first.

nice division into roles

3.1. Feature descriptions

myShop will simulate the real life shopping experience by providing to the shopper and seller the virtual analogs of what they would use and see in real life.
In particular, for the shopper, ability to:

- **window shop:** shoppers can view items sold in a seller's shop.
- **maintain a cart:** shopper can add and remove items from seller's shop to a cart. No shopper registration is required.
- **place an order:** shopper can purchase the contents of their cart.

None of these actions will require shipper registration.

And for the seller, ability to:

- **own shop:** seller can create and maintain more than one shop, each with their own set of items.
- **maintain shop/inventory:** seller can add and remove items from shops.
- **maintain items:** seller can change the price and number of a certain type of item.
- **maintain orders:** seller can view orders across his shops and mark as closed when they are fulfilled.

summary

what are features?

› areas of functionality
› actions that go together
› for some larger purpose

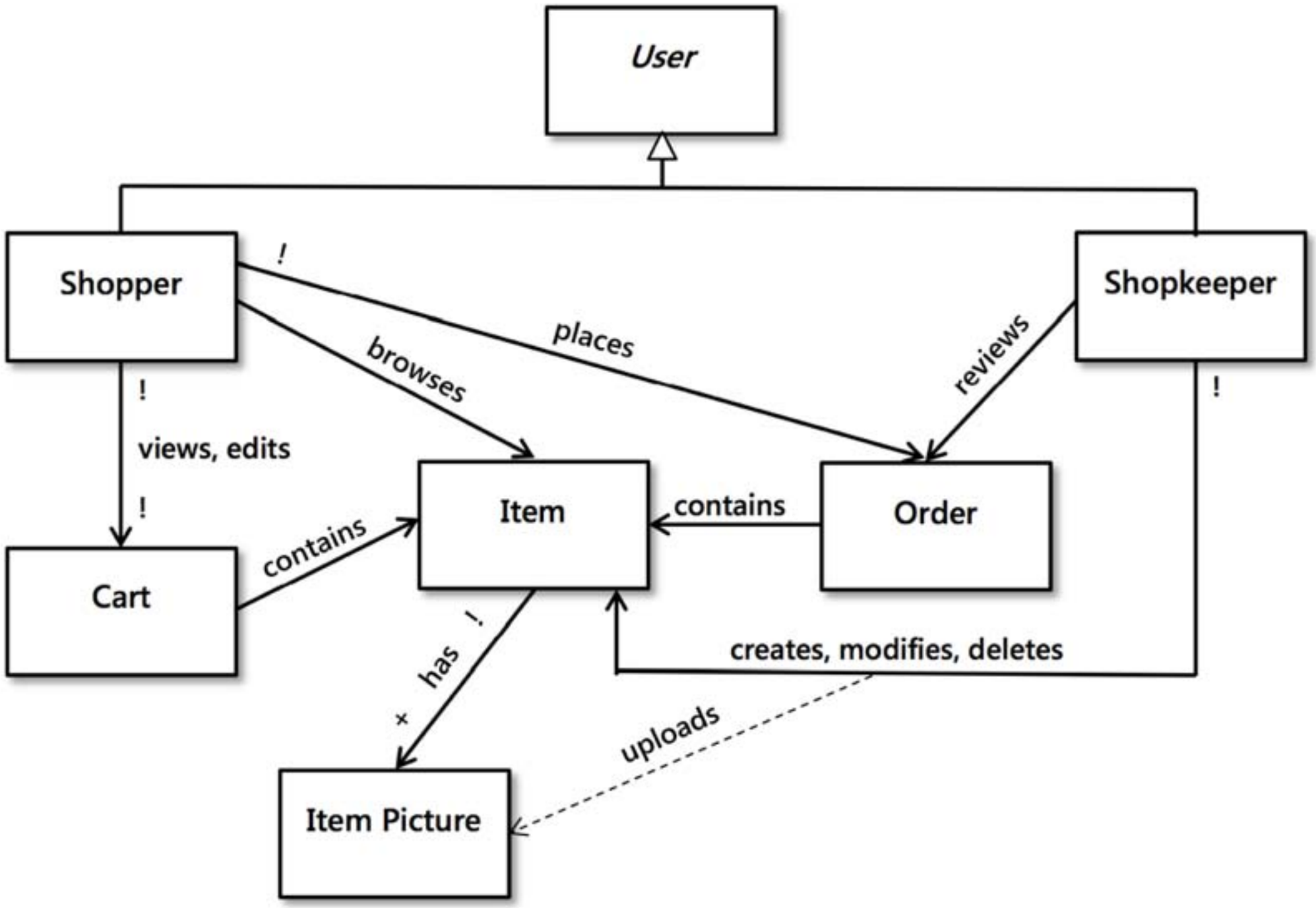
what you're trying to do

› provide a high level outline
› identify groupings of lower-level functions
› tie back to the user's purpose

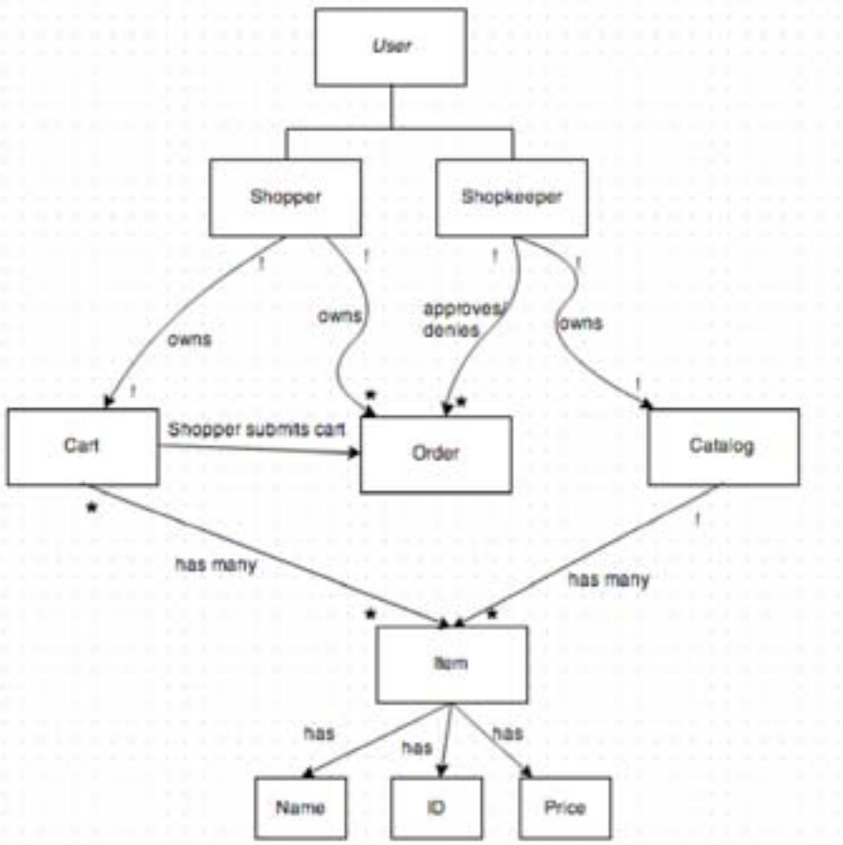
common mistakes

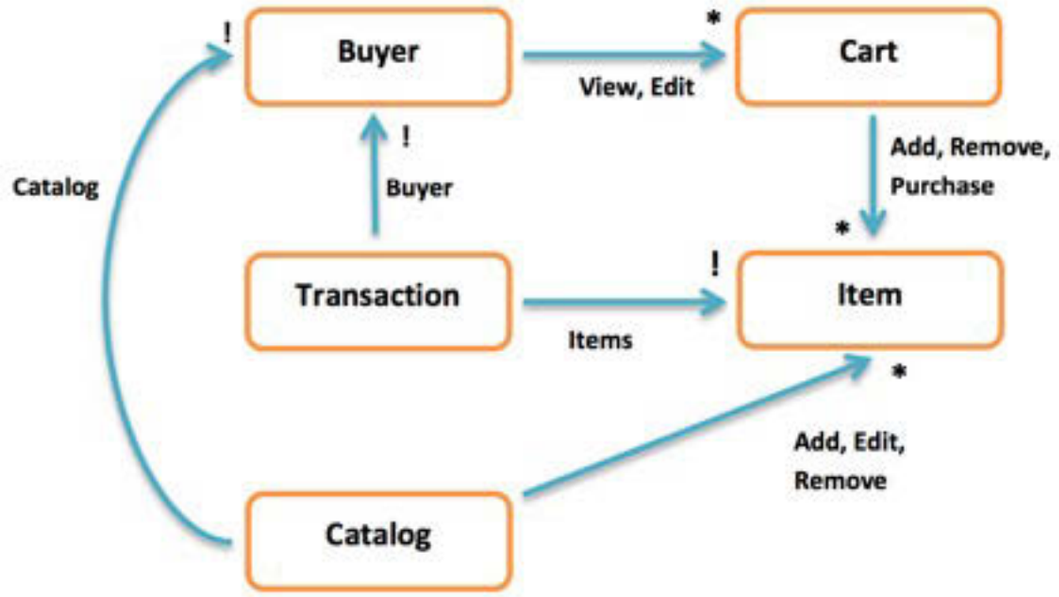
› just listing actions and not grouping into features
› no organization, eg by role or end-goal

object model

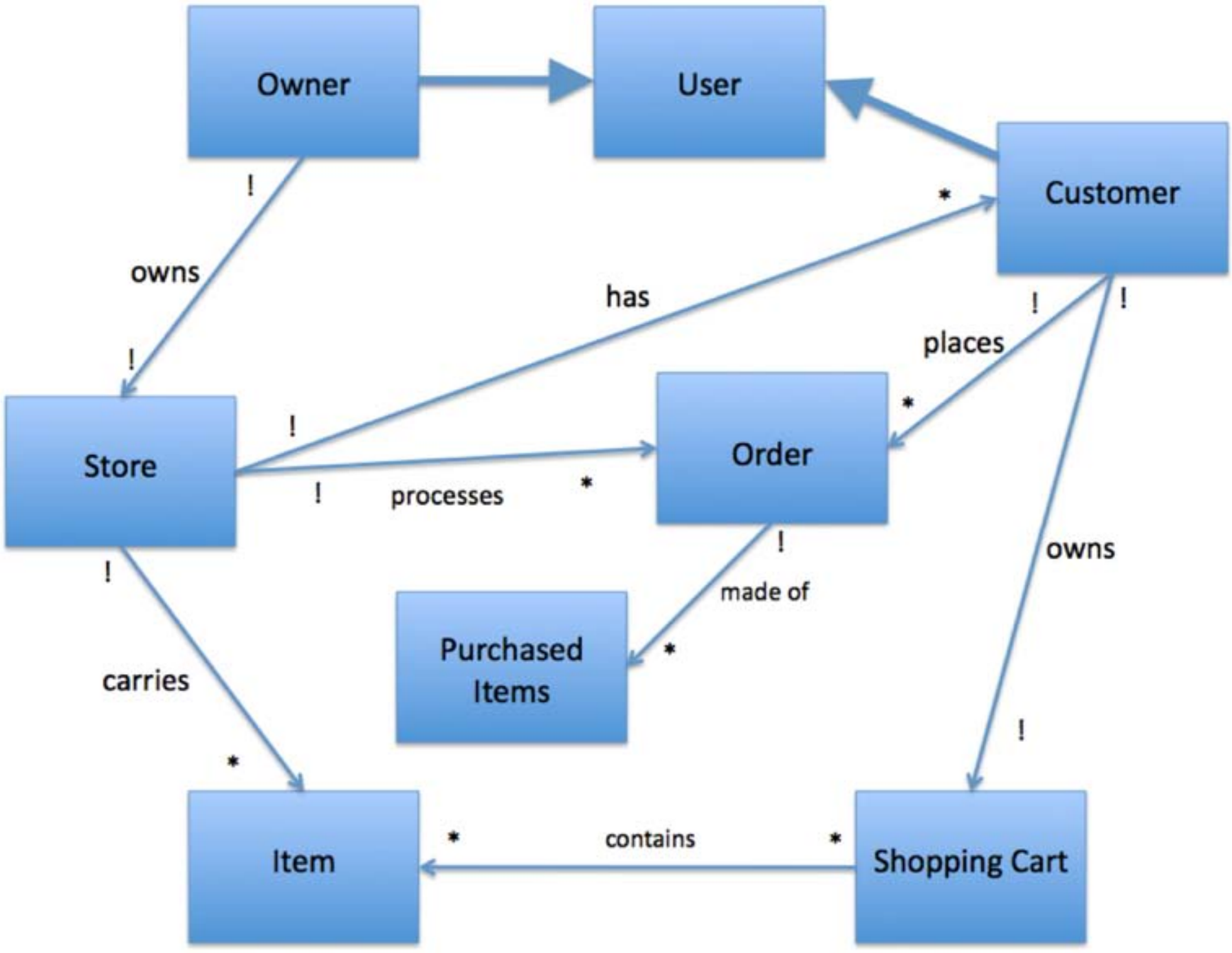


not an OM: edges should represent state, not actions

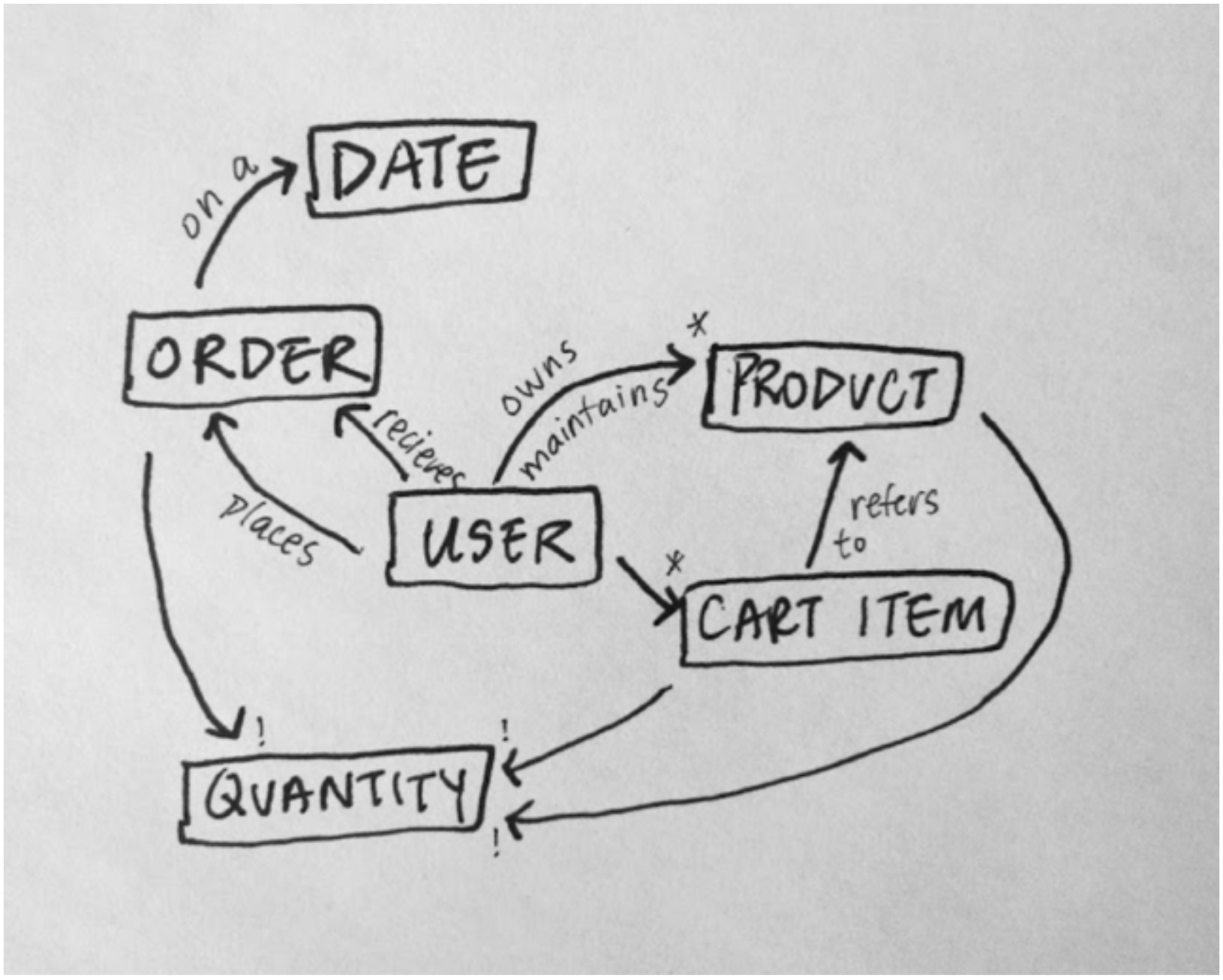




not OMs: edges should represent state, not actions

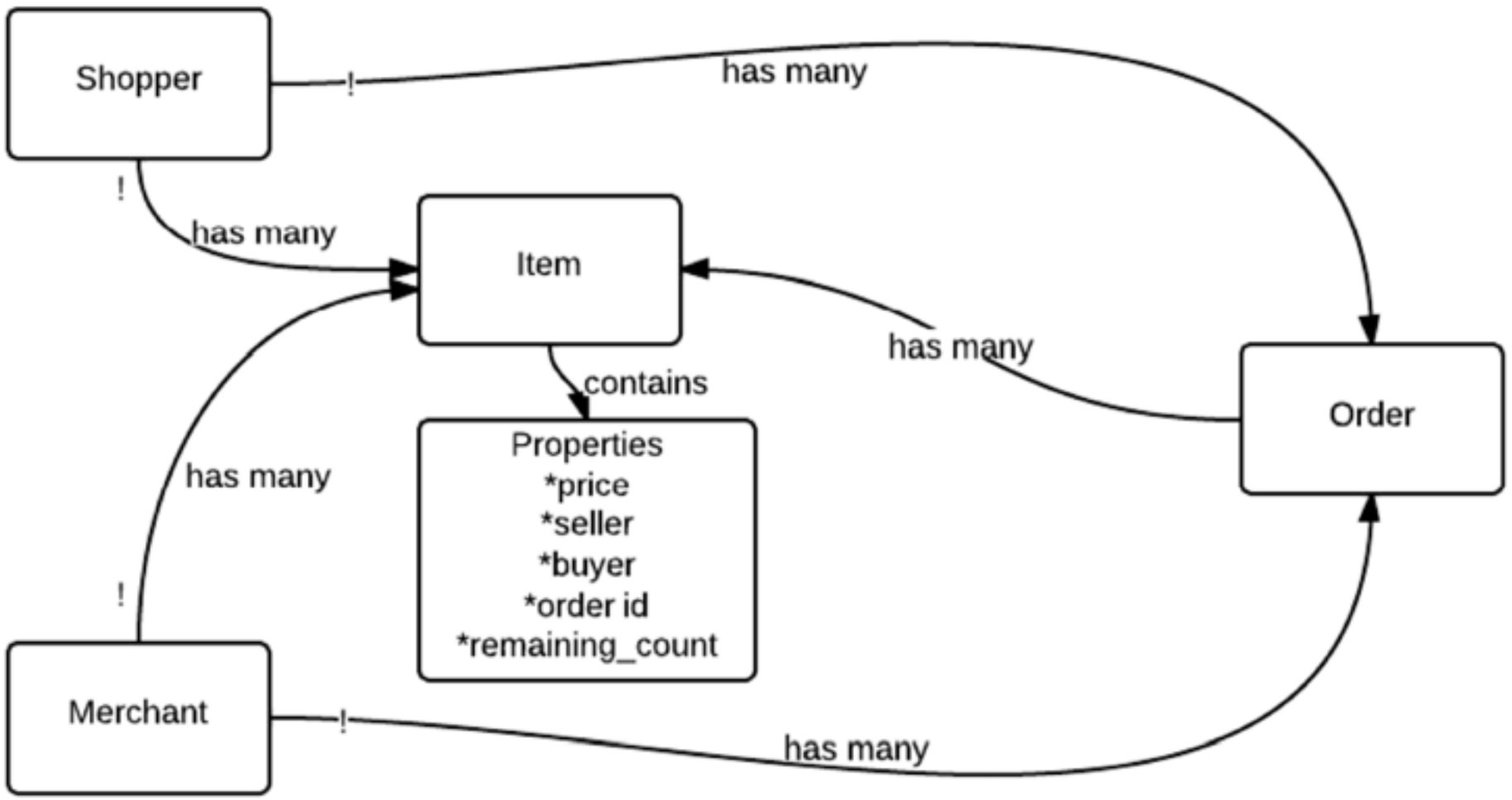


nice, but: shared properties of users?
customer has exactly one cart

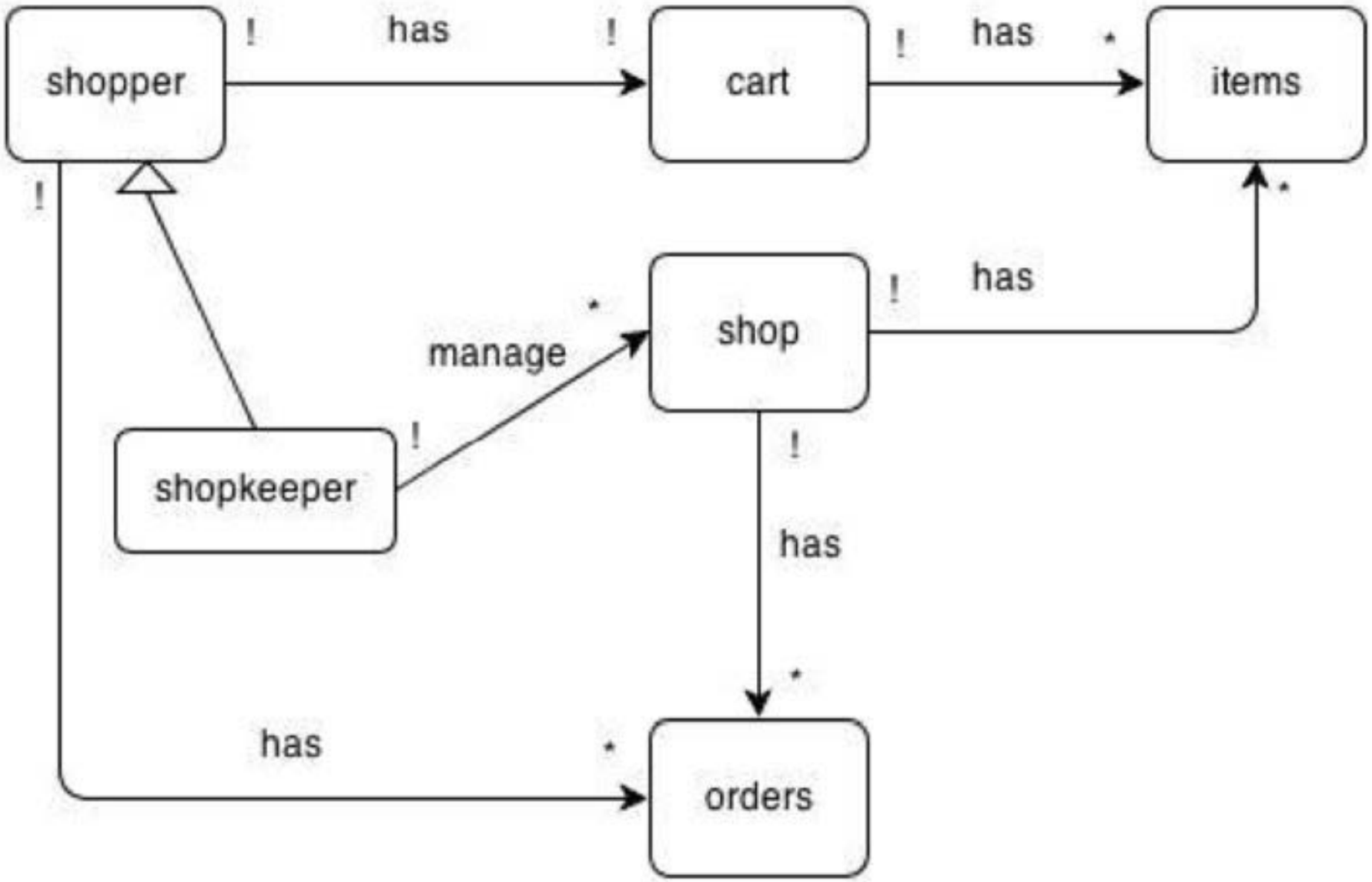


what is *places*?

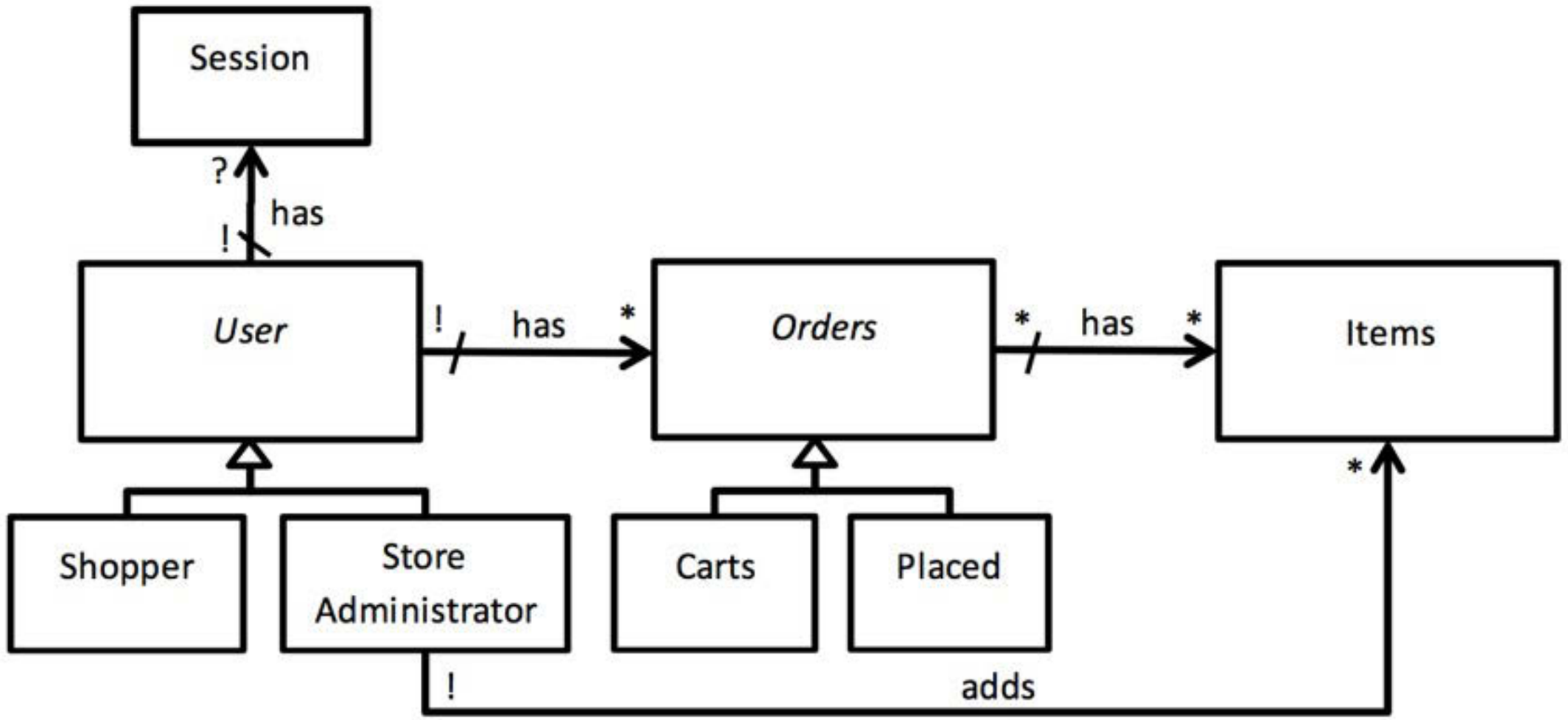
has many edges with same label



merchant has many orders: placed? fulfilled?

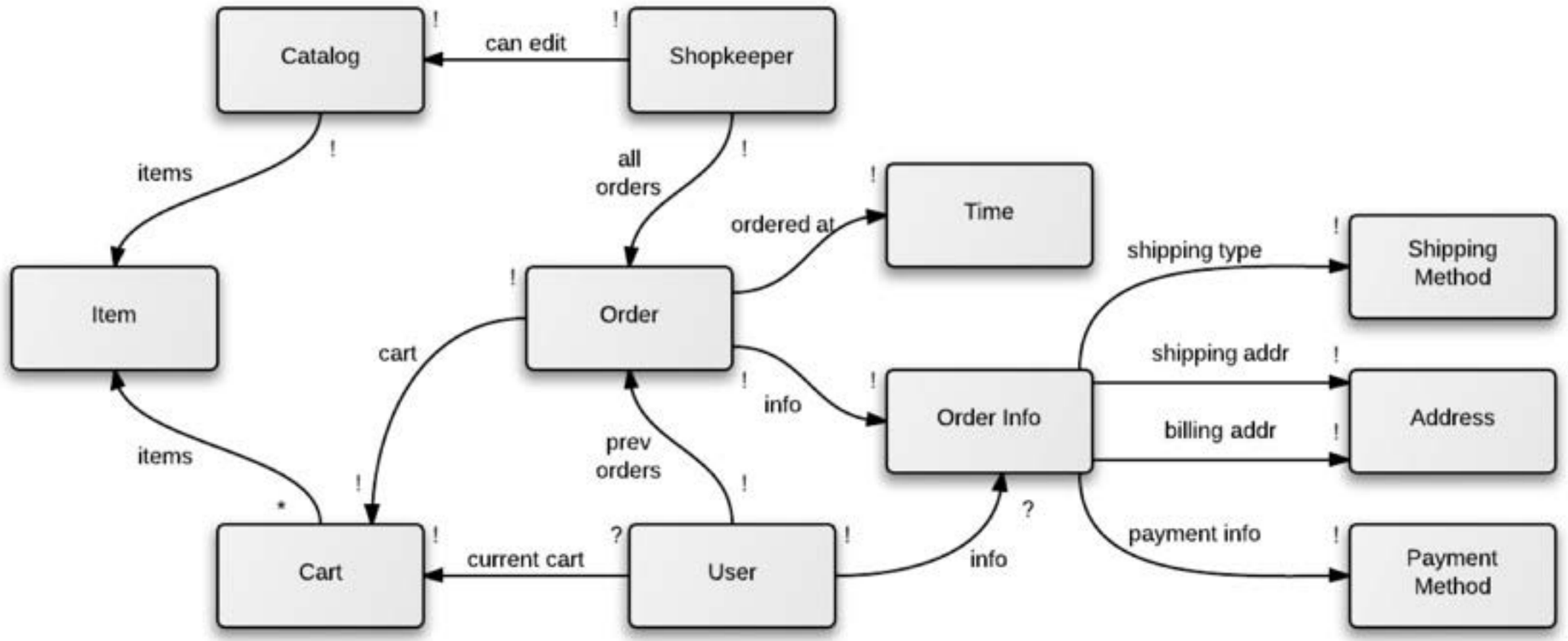


has not specific enough

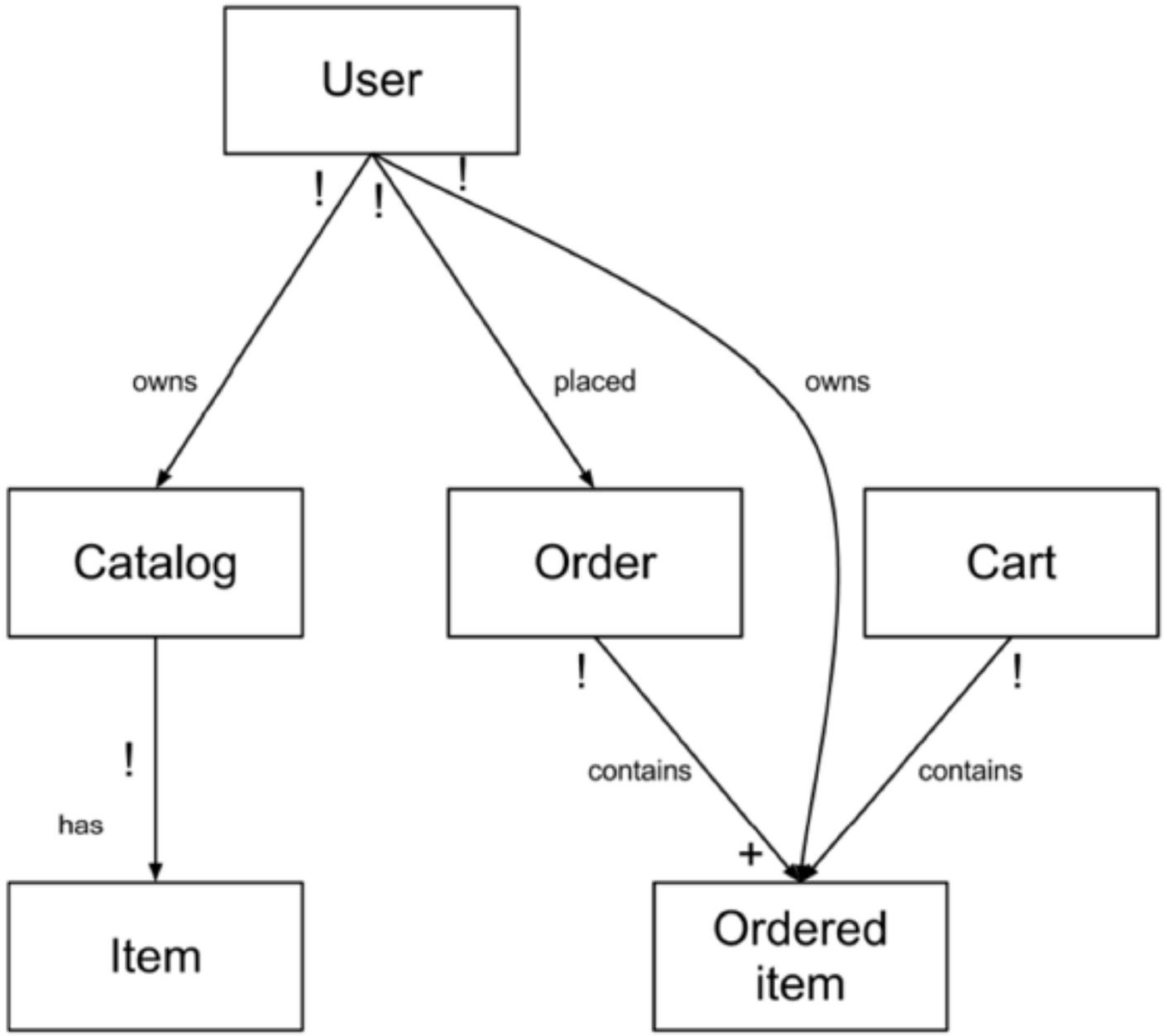


more unspecified has relationships

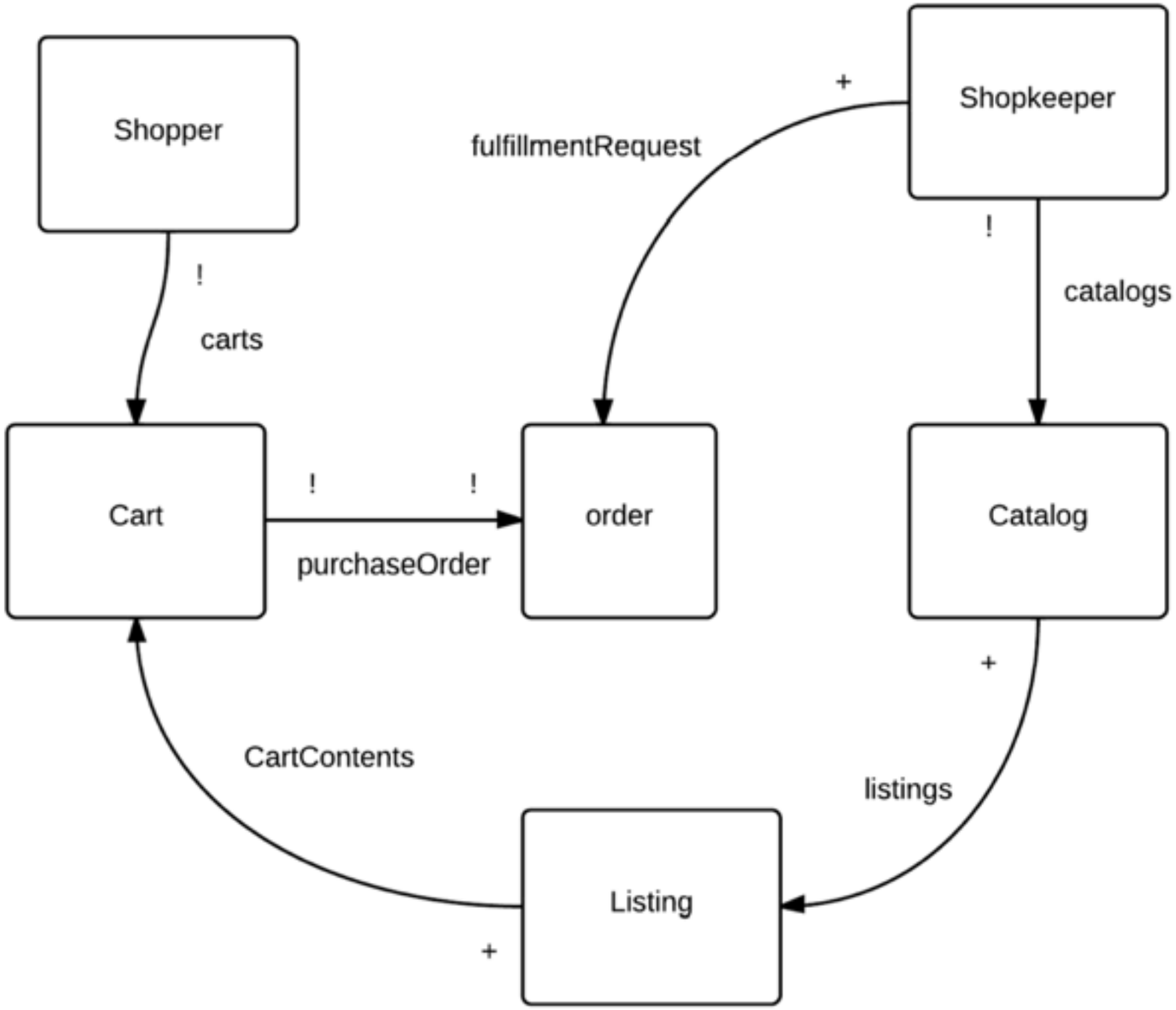
good attempt



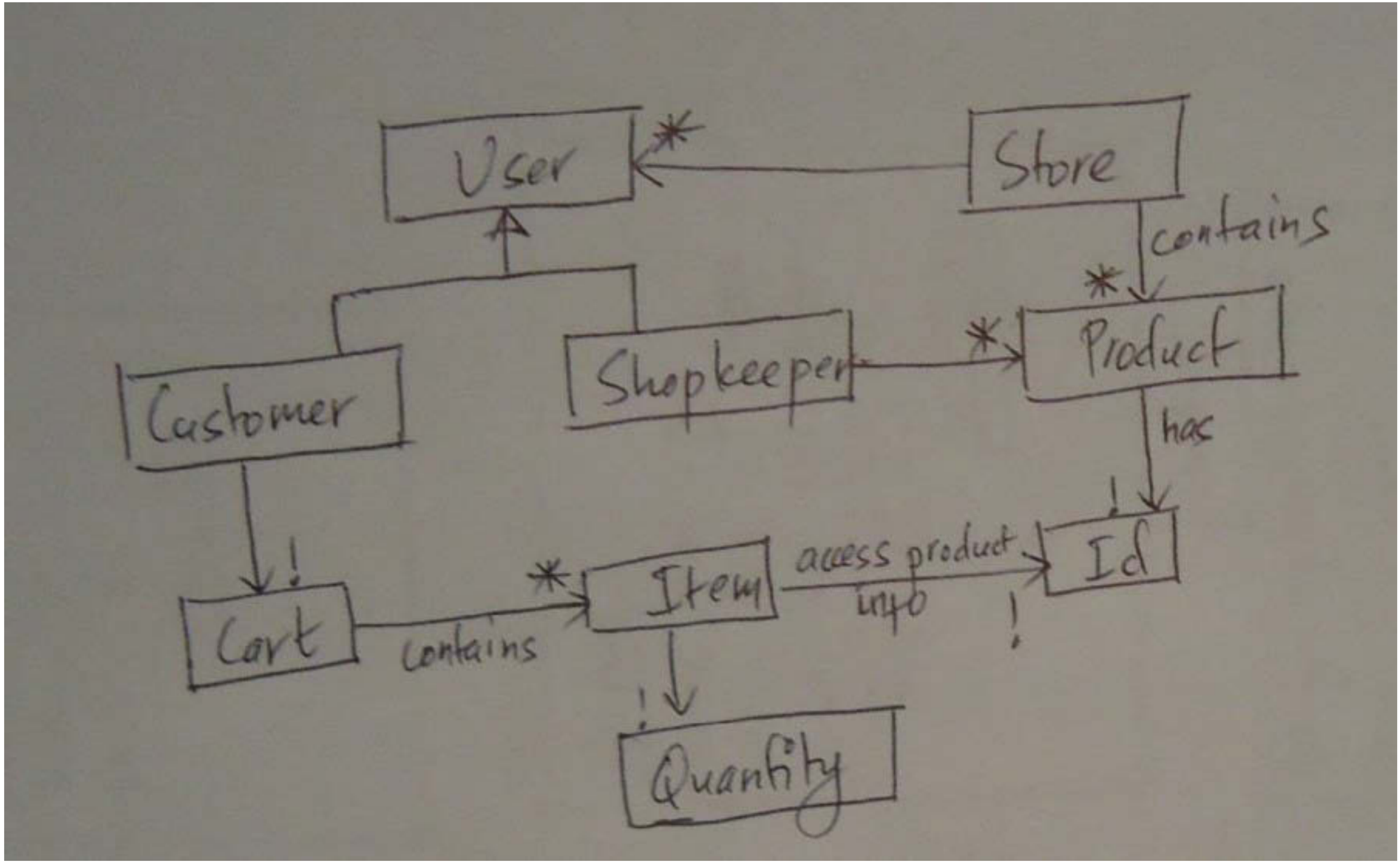
what's *User.info*? *can_edit*?



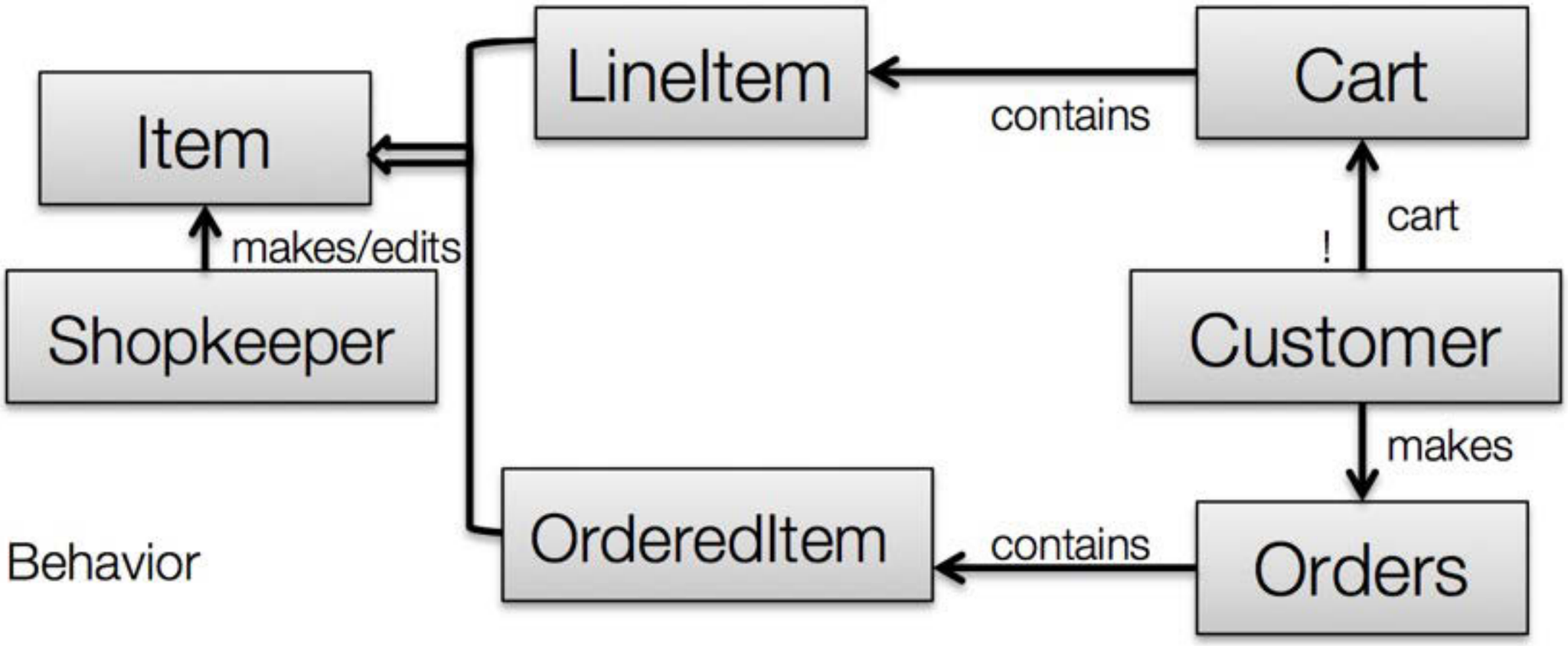
user-cart? items and ordered items disjoint?



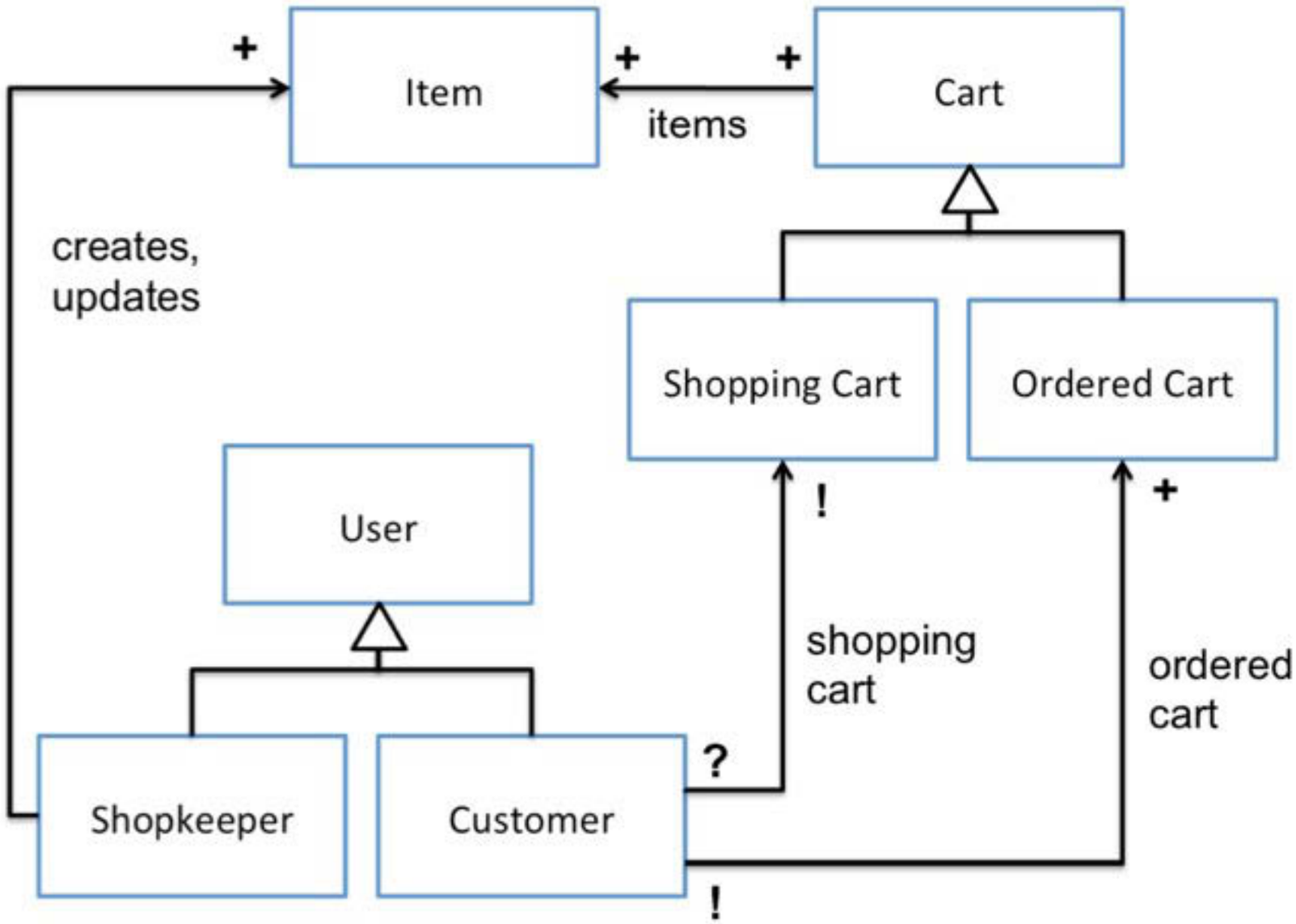
why is order needed if 1-1 with cart?



what's user-store relation?



why two kinds of item? why no generalization of cart/order?



good, except for creates/updates; should be "offered"?

summary

what is the object model?

› abstract representation of state

what you're trying to do

› capture what the system must store
› but avoid implementation details

common mistakes

› confusing state and action
› using "has" labels, making actual relation unclear
› omitting state (eg, order placed vs order fulfilled)

a rule of thumb

› beware of active verbs as edge labels (adds, removes)
› stative verbs are ok (offers, likes, posts)
› past participles are ok (posted, added, selected)

security concerns

Security requirements:

1. Non-seller cannot edit/create products, and cannot access order listing
2. Cookies should be cleared after exiting window to minimize 'sharing' sessions between one customer to another
3. Seller can only access order listing using a username and a password
4. If the seller forgets to log out, the site should automatically log the seller out, so that the next user at the computer will not have access to orders list
5. Only admin user can create more users

mixes security requirements and particular threats

BlackMarket has the following security requirements: it must ensure a reasonable level of protection for its users' usernames, passwords, and private information such as purchase habits. BlackMarket may be vulnerable to the following potential security threats:

- A hacker can mass create usernames and passwords. I can mitigate this by implementing a human verification method such as math captcha. I can also require user to verify the account creation through email.

- A hacker can create an account and generate mass orders or advertise multiple fake products. This will flood the app's database and also affect other users' experience. I will need some methods to put a cap on the number of orders placed and allow users to report spam.

- The fact that cookies are used to track user's log-in state and shopping cart makes the system vulnerable to packet sniffing whereby someone intercepts traffic between a computer and the Internet. Once the value of a user log-in cookie is stolen, it can be used to simulate the same session elsewhere by manually setting the cookie - session hijacking.

- Since the app requires users to interact directly to complete transactions, I will also need to implement identity verification methods and advise users not to meet with their counter-parties if suspect.

very good enumeration of risks

security concerns: example

summary of key security requirements

› no tampering with merchant catalog (eg, prices, stock)
› no bogus orders (for existent or non-existent shoppers)
› no stealing of personal data (past orders, address, etc)
› no stealing of credit card numbers
› no viewing of competitors' orders by merchants
› no viewing of shoppers' orders
› no illegal sales (pharma, illicit drugs, porn)
› no denial of service attacks

how standard attacks are mitigated

› XSS: sanitization; CSRF: form tokens; cookies: encrypt, expire

threat model: assumptions about attackers

› attackers have physical access to clients but not servers (so expire)
› attackers may snoop (so use SSL for purchases)

security strategies

to prevent bad access

› authentication + access control on all sensitive actions
› expiring cookies and deleting on logout

to protect financial data

› SSL encryption for financial data submission
› no storage of credit card numbers

to lower risk of illicit merchants

› charge merchant fees
› provide user reporting of abuse
› vigilant moderator team

summary

what you're trying to do

› identify the key security risks
› think about strategies to address them
› understand what makes this problem special

common mistakes

› ignoring domain-specific security requirements
› not describing the problem, just the solution
› mixing the problem and the solution

why this matters

› need a clear sense of what the major risks are
› … what options the designer has to address them
› … what risks will remain

design challenges

4.4 Managing User Activity via Sessions

Challenge: A user is allowed to browse Flexion and add items to his cart without having signed in to his account - how should this be managed?

Options:

1. Use the provided Rails cache. actually not relevant to this
2. Store session information in the user's cookie, which resides in the user's (client) side.
3. Store session information in a database model on the server side and some information on the user's (client) side in a cookie.

Solution: I chose to store session information primarily in a database model on the server side, and use the user's cookie briefly for ID storage (which the session's did automatically). The rails cache stories session information temporarily which is lost upon close of the application, and thereby this option was not chosen. Extensive use of the user's cookie would incur security concerns as the cookie resides on the client's side, and thereby this option was not considered either.

what happens on login?
sharing between sessions?

nice problem/solution separation

4.7 Adding an item to a user's cart when it already exists in another user's cart

Challenge: When a user adds an item that is present in another user's cart, should the user be allowed to add the item to the cart, and thereby should the item be allowed to exist in multiple carts?

Options:

1. Allow item to exist in multiple carts.
2. Don't allow item to exist in multiple carts.

Solution: Given competitive nature of the market provided by Flexion, this problem may be prevalent given that a large number of users are subscribed to Flexion. The solution decided upon was to allow items to exist in multiple carts. When a user proceeds to check out and buys the item, the item is tagged accordingly (discussed in 4.1). This allows Flexion to maximize its opportunity to make a transaction occur; it exacts pressure on users to check out which can potentially lead to higher number of transactions; lastly, it is fair in its protocol as it promotes a first come first serve approach.

generally good, but could be clearer about what the problems are and what the solutions involve

Design of carts: The user wants to be able to add items to cart(s) across the different stores.

1. **Maintain different carts:** Each store will have a different cart associated with it, so users have to navigate to different stores within our site to access the relevant carts
2. **Maintain only one cart:** Users can add items to a cart from any store.

We picked option 2 since we wanted to provide a user experience like that of Amazon, where the notion of multiple sellers is abstracted away from the notion of a cart. It's easier and more intuitive for the user to just add items to their cart instead of remembering which items where in which carts.

nicely explained, but what about orders?

Shoply has the following key concepts:

- **User** - a user can either be a buyer or a seller and has certain properties like name, password, etc.
- **Store:** a listing of **products,** along with store name, description etc
- **Cart –** a list of **cartitems** a buyer has added to his cart
- **CartItem –** contains an **item** a user has added to the cart along with the quantity
- **Item –** contains item information like name, price, description, etc.
- **Order –** contains a list of snapshots of **items**, as well as the user who placed the order
- **OrderItem –** a snapshot of all the items in an order along with the quantity for each

all items in order from one store?

Can a shopper be a shopkeeper and a shopkeeper be a shopper?

One way to manage users is to have a group of users who choose to be shopkeepers when they registered their accounts. The rest of the users choose to be shoppers. Another way is to allow a user to be both. I choose the second way because although there is a clear boundary on user's types on the server side for the first way, web users may not be happy with it. For example, for the first approach, what will happen if a shopkeeper wants to shop? Does he have to register for a new account, with a different e-mail? To get rid of these questions, I decide to make a registered user do whatever they want.

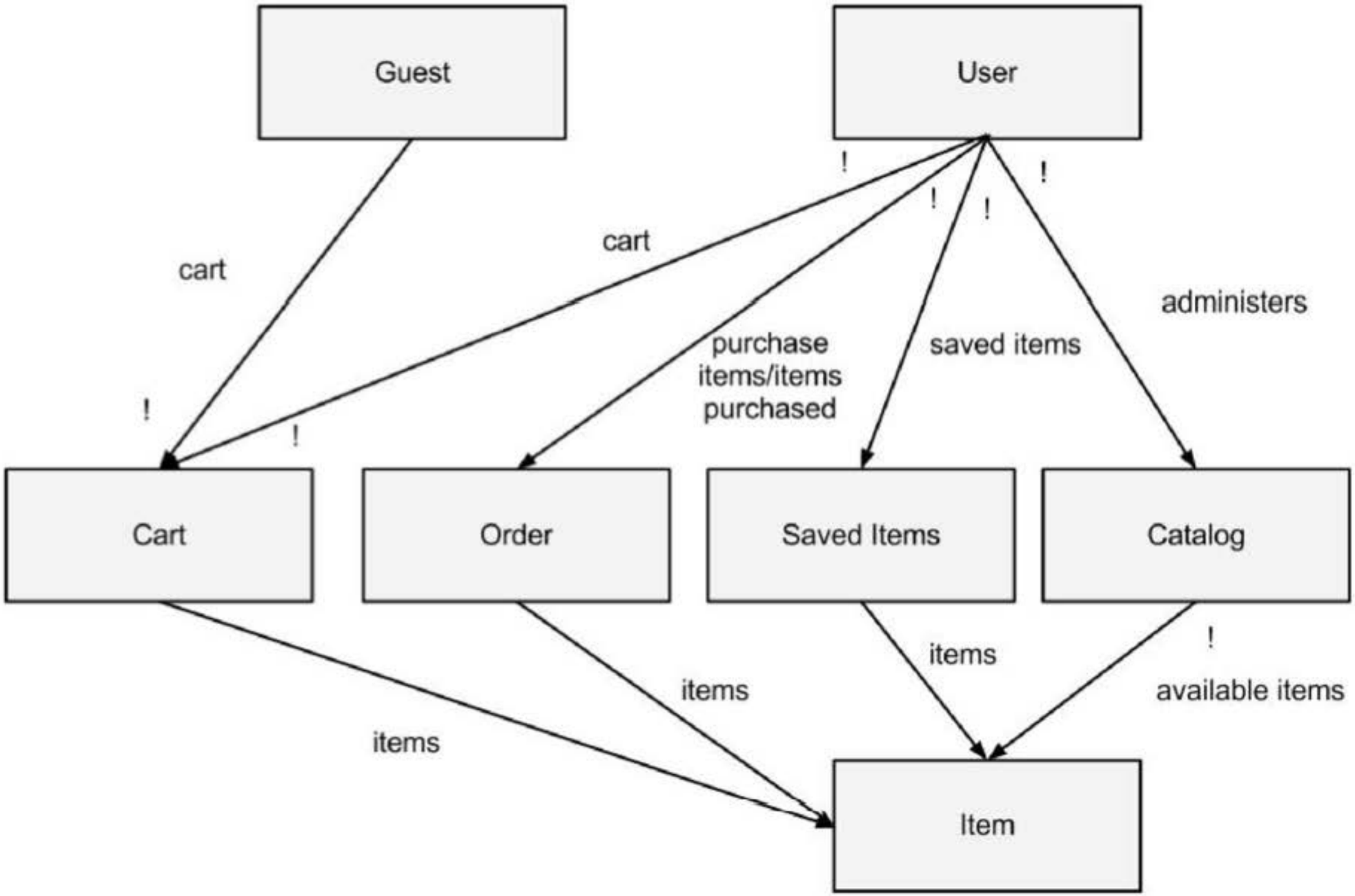
good observations! how will you achieve this?
some functions only appear if you're a merchant?

How to keep past orders?

There are several ways to store past orders in the database because an order can be specified by shopper's id, shopkeeper's id, item' id etc. One approach is to store everything by its id. Storing a user's id with an order (making an order belongs_to a user) simplifies the method for displaying past orders for a user (either a shopper or a shopkeeper). Storing an item id makes it easier to query all orders on this particular item. These can be automatically done easily by using has_many and belongs_to in rails.

Another approach is to store everything as its value, i.e., store shopper's e-mail, shopkeeper's e-mail, item name, catalog name, price at the current point, quantity, etc. This approach makes sure the recent orders that a user view stay the same even if a shopkeeper edit an item name or item price. The data stored in the table is the information that a user, as a shopper or a shopkeeper, cares about. This also means that in the object model, order does not belong to any other object.

good, but explain the whole problem first



OM for last example:
items: Order -> Item ???

1. **Access control mechanism**

   There is a subtle trade off between a simple user interface and providing strong security mechanisms to mitigate some of the concerns raised in Section 3.

   (a) **Password protection:** Only allow the user to add items to cart or to the inventory after the user has either created or entered a password.

   (b) **Sessions with authentication later:** Allow the user to add items to cart or to the inventory, but require authentication before checkout or additing a new item to the inventory.

   I chose option 2 in part because it was enforced on us during tutorial and lecture. Apart from this, there are some other reasons why option 2 might be preferred. Option 2 makes SimpleCart lightweight. When the user arrives at the homepage, the user can directly start selecting items to add to the cart. This also makes SimpleCart's user interface simple. Since authentication is performed before the items in the cart can be checked out or before new items can be added to the inventory, security is not compromised.

many good points made here

be careful about terms: password, authentication

is cart also persisted? if so, does login do merge?

4. **Resolving concurrency between two purchases of the same item**

Another design challenge faced was to make a decision on how to handle the concurrency case when two users placed the same item in the cart and tried to checkout simultaneously. SimpleCart could do the following:

(a) Allow only the first user to purchase the item.

(b) Place the item on hold and notify the seller to make the decision, possibly through a bidding process.

In order to keep SimpleCart simple, option 1 was chosen. When a user purchases a particular item, that item is removed from catalog of all items as well as from the inventory of the seller. Therefore, in its current design each item can only be bought by one user. Hence, the user who first makes a succesful payment for the item gets the item, and the second user receives an error message saying that he or she cannot purchase the item. SimpleCart's current design can be easily extended to handle the notion of quantity of each item by maintain a quantity field in the item model.

excellent!

8. **Deleting carts that do not belong to any registered (signed up) user** Following on from the discussion in the previous point. Suppose that a user adds items to a cart but never logs in or signs in. How should the cart created be handled, and more importantly when should it be destroyed? Note that the sessions will expire at the expiration time, however this point deals with when the cart should be removed from the database.

   (a) Write a chronical job on the server that runs periodically (session time expiration) and deletes carts that are not associated with a user.

   (b) Maintain one special cart for an unlogged in user and transfer over items when the user logs in

   (c) Dynamically resize the carts table so that such carts can be deleted

   (d) Delete the cart when the user logs in

Option 1 would clearly be the most ideal. However due to limited resources and expertise, I was not able to effectively implement such a solution (however it was great to read about how one could do so !). Moreover, I do not think we have the required permissions to implement such a chronical task function in our heroku server. Option 2 is a convenient and performance-effective strategy to resolve this problem. However there are a lot of concurrency issues to deal with. Suppose two unlogged in users visit the site, both adding items to the cart. All of these items would be stores in this special cart and transferred to the user when he or she logs in. However, it would be extremely challenging to distinguish items belong to each particular users since they are stored in the same cart. Also, this notion of a shared cart for unlogged in users goes against the proposed object model where theoretical each user must have at most one cart. Option 3 of dynamically resizing the table provides a reasonable solution, but does not provide good performance. If there are a large number of ca
all the carts before deciding wh
website, an undesirable outcome. I chose option 4 because of its simplicity. It provides the functionality required, gives good performance, however is not very resourceful. I would still end up creating a lot of carts for users who place items in carts but never end up logging in. This is a limitation of my design.

thoughtful analysis; note clear admission of limitation

Transaction mechanism: we can allow users to make payment throught two methods:

- Credit card payment
- Settle in person

Credit card payment will solve many problems such as mass fake orders because payment is required before orders can be sent to sellers, However, it requires much more rigorous measures for validation, security, and routing with bank accounts. For the scale of this project, I opt for the second option and allow users to settle payment by themselves. BlackMarket will simply notify sellers of new orders and include the buyers' contacts rather than other shipping addresses in the order. Note that buyers can only include their provided email addresses as their payment contacts so as to avoid scams.

Another concern is the possibility of price or quantity changes when an order is in a shopping cart. I will fix this by asking the server to check before actually sending the order.

good discussion, but second issue is really different

summary

what you're trying to do

› identify the key design issues before coding
› explore alternatives & justify your choices

common mistakes

› not explaining the problem
› not structuring clearly (problem, options, choice)
› not following through (eg, what happens when item disappears?)
› vague description of problem ("how to represent X")

a general issue

› too much focus on code issues, not enough on behavior design

sample behavioral design problems

can shopper add items without logging in?

› if so, what happens when they login?
› what if they do this in multiple browsers at once?

multiple merchants

› can cart mix items from different merchants?
› what happens to order in this case?

is inventory limited?

› contention between shoppers?

handling failures

› what if merchant can't provide item? undoing orders?

merchant-shopper contention?

› change or price, item description?
› during shopping, after order, after fulfillment?

6.170 Software Studio
Spring 2013