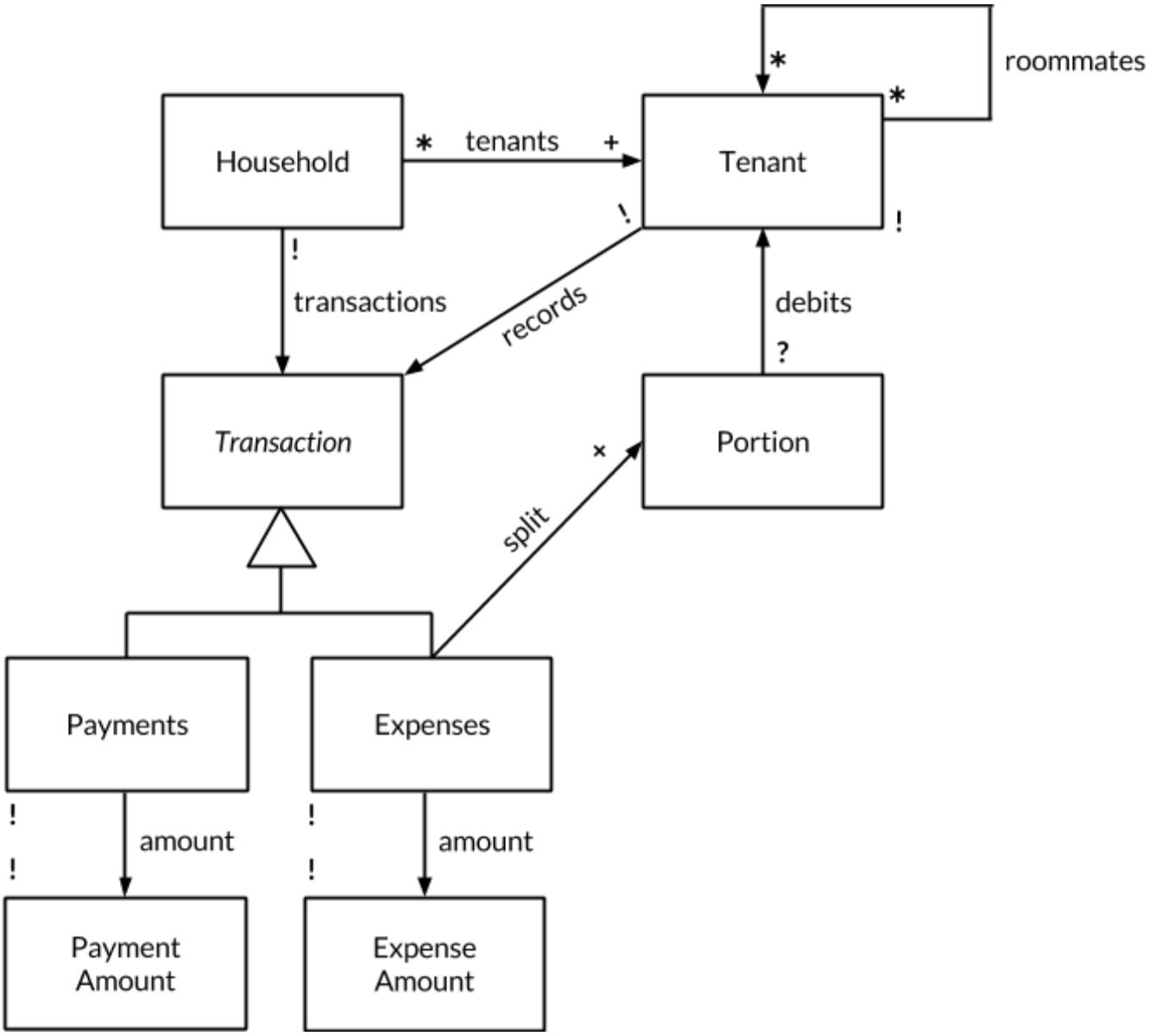
Connie Huang (connieh) | **Stephen Suen** (ssuen) | **Sashko Stubailo** (sashko)

6.170 P3.2 | RUMI - MINIMUM VIABLE PRODUCT

FEATURE SUBSET

- **Households**
  - **Household management.** Users can create a new household and add other Rumi users to start tracking group expenses. A household is a collection of users, each of which can log expenses within a group and also log private payments to another roommate. All of these transactions are logged and balanced so users can keep a tally of how much is owed to them by the household or how much they owe to the household.
- **Bookkeeping**
  - **Expense logging.** Users can track expenses by adding a cost, description, and expense breakdown among roommates.
  - **Payment logging.** Users can also log when they have privately paid another roommate, which factors into the overall balance that is displayed at the top of each household.
  - **Total balance display**. Users can see who owes the group how much money after all of the expenses and payments.

UPDATED DATA MODEL



ISSUES POSTPONED

- **Even out button.** It should be easy to click a button and see who you should pay and how much to even out everyone's balance to 0. We are still working on a way to help users understand that choosing to "even out" will set the balance to 0 (i.e. nobody will owe the group any money, and the group will not owe anyone any money) through a series of transactions between users.
- **Venmo integration.** Although Venmo integration is the feature that makes it easy for users to split the money, we wanted to first test the user experience of the cost tracking and balancing. We've decided to postpone Venmo integration until we know that users are comfortable with our key concepts of portioning expenses within our expense log. For now, we will settle with the black box of our payment system.
- **Collaboration features and warnings.** It should be easy to see if another user is currently editing an expense or payment so there are no conflicts. We also want to include a log of edits made to a single transaction, so no one roommate can maliciously reapportion any costs.
- **Consistent styling between rows and editing forms.** When transactions are added or modified, the expense log shifts around because there are so many elements dynamically updating. We plan to fix this small interface bug to create a smoother experience.
- **Readable form errors and validation.** There's some errors that are hard to read, and some are not present - for example, you can currently create a household with an empty name.
- **Security.** We need to make sure that nobody can edit something that isn't theirs, particularly because there is money involved. We also want to implement a change log that can't be deleted or modified so that users can see everything that's going on.

USER TESTING + FEEDBACK

- **Group feedback.** We believe the current state of Rumi is still valuable to users even though it is not complete without Venmo integration. Users can track their expenses and see how much everyone owes to the group. When users want to pay each other, Rumi helps the group make sure that all transactions are even and nobody comes out with a negative balance. From this iteration of the application, we are ready to move forward to test the ideas of expense logging and tracking payments within the application before any money is actually transferred with Venmo (or with cash in person).
- **User testing feedback.** Friends who have examined Rumi like that an expense can be split among roommates in a single form (one row of the expense log) as opposed to a form with multiple steps. However, the idea of positive and negative numbers in relation to the household is a concept that we would like to expand on in the next iteration, where we hope more labels or tooltips can help a user understand what a specific number means (a +40.38 means that a user owes $40.38 to the other roommates).

DESIGN CHALLENGES

Modeling data in MongoDB

- *Problem*
  - All of the team members are accustomed to using relational databases such as those that Rails uses by default. In a relational database, one usually creates many tables and relates them to each other using other tables. Then, when you need to query for those relations you can use a join query which is usually done for you by an ORM like ActiveRecord. In MongoDB, everything is modeled with documents and doing join queries is not efficient. This required us to rethink how we would store our data.
- *Options available*
  - Three collections: Users, Households, Transactions
    - These are three pretty separate concepts in the application. A user can own households and expenses, and create and edit both of them. Expenses would belong to households through a household_id. This would be pretty natural to implement coming from the Rails relational model.
  - Two collections: Users and Households
    - In this option, expenses and payments would be in an array in Households. This makes sense because expenses would only be displayed when looking at a household page (they wouldn't be shown on their own), but it makes editing expenses a bit more complex because it requires doing array operations.
- *Choice*
  - We decided to have only *two collections* because otherwise each page would have to do many queries and lookups to get all of the right expenses for the household. The way Meteor templating works makes it natural to use one object. One downside of this, however, is that the whole page tends to re-render when an expense is deleted. This issue would still exist with the three collections model, but perhaps there is a third option that would be better than both of the ones we considered. However, the re-rendering problem will be fixed in the next version of the Meteor templating engine, so maybe it won't be a problem.

Displaying Expenses and Payments

- *Problem*
  - There is a lot of information to display in one table, especially if there are more than 4 users. The display must be compact yet readable and easy to get information from at a glance.
- *Options Available*
  - Separate table of payments from table of expenses
    - This makes sense because payments and expenses are separate concepts with different structures. On the other hand, it's hard to see what happened when chronologically. This means that the user would have to rely more on the final Balance calculations to check how much money is owed by other roommates.
  - Rolling log of expenses and payments
    - This display of the expense log reads more like a chronological "log" than a spreadsheet, which is easier for a user to understand and navigate when looking for a specific transaction. The user would likely be more comfortable looking through the log and seeing how payments even out over time, instead of having to rely on the final Balance calculation.
- *Choice*
  - We decided to separate the table of payments from the table of expenses, because we thought the display of a payment and expense form in the same table was too cluttered. Unlike Venmo where pay/charge are linked very closely as the final step because there is a single transaction between two people, Rumi logs expenses between multiple people. Thus, our pay/charge model is not only a matter of direction of payment, but also how many people are involved in a transaction.
  - By separating the two actions of expense logging vs. making payments, we hope to make it clear to users that expense logging is between multiple people, whereas making payments (without choosing to "even out", which is an automatic calculation) is between two people.