Concept Modularity

6.1040 Recitation 3

Today's recitation is going to be a hands on exercise! In groups of 2–3, we'll come up with a modular concept design for a

Grocery Shopping App

Setup

Imagine you wanted to build an app that maintains **shopping lists** for users.

1. Each shopping list should contain a set of **items** and a desired **quantity for each item**, such as 2 lbs of chicken thighs, 3 tomatoes, 1 pack of napkins.
2. Users should be able to maintain **multiple shopping lists** (e.g. “Weekly Grocery Run”, “Mapo Tofu Recipe”, “Friday Family Dinner”).
3. Users can create a **new shopping list** that **merges items from existing shopping lists**. Duplicate items have quantity equal to the sum of their individual quantities in each shopping list (e.g., if one shopping list has 3 tomatoes, and another has 4 tomatoes, the merge of these two will have 7 tomatoes)
4. Users can **mark which items have been bought.**

Guiding Questions

- **How would you break down this app idea into modular, reusable concepts?**
  - Each concept should have exactly 1 purpose. A concept might need to be broken down into smaller concepts if you find yourself trying to convey multiple purposes.
  - At the same time, a concept might be too limited in scope if you can't seem to find a convincing operational principle, and maybe it should be part of a larger concept.
- **What are some synchronizations you can use to compose your concepts?**
  - Syncs take actions from multiple concepts and either constrain or automate the order of their executions (e.g. concept X can't execute action A until concept Y executes action B, or concept X executes action A whenever concept Y executes action B)
- **What are some generic types that your concepts might use?**
  - Generic types are not defined within the concept, but are passed in as identifiers
- **What is the minimum state that your concept must store to meet its purpose?**
  - States are either sets (of unique identifiers) or binary relations.
- **What are some invariants that you need to maintain?**
  - These are integrity constraints that are necessary for your concepts to be used correctly.

State Representation

concept ShoppingList [User]

state

- a set of ShoppingLists with
  - a User
  - a name String
  - a set of Items
- a set of Items with
  - a name String
  - a quantity Number
  - a unit String

Invariant 1: An item cannot belong to multiple shopping lists. Why?

Invariant 2: The quantity of each item must be greater than 0.

Actions

concept ShoppingList [User]

actions

- **createList**(user: User, name: String) : ShoppingList
- **addItem**(list: ShoppingList, name: String, quantity: Number, unit: String)
- **removeItem**(list: ShoppingList, name: String, quantity: Number, unit: String)
- **mergeLists**(user: User, lists: set of ShoppingList) : ShoppingList

Which actions will help preserve the invariants?

createList (user: User, name: String): ShoppingList

- **effects:** adds a new shopping list

Can I add two items with the same name but different units?

addItem (list: ShoppingList, name: String, quantity: Number, unit: String)

- **requires:** list exists, quantity is greater than 0
- **effects:**
  - if an item with the same name and unit already exists in the list, increase its quantity by the specified quantity.
  - else, create a new item and add it to the list.

removeItem (list: ShoppingList, name: String, quantity: Number, unit: String)

- **requires:** list exists, an item with this name and unit exists in the list, the item's quantity is at least this specified quantity
- **effects:**
  - if specified quantity is less than this item's existing quantity, decrement this item's quantity by the specified quantity
  - else, remove this item from the list

mergeLists(user: User, lists: set of ShoppingList): ShoppingList

- **requires:** all lists exist
- **effects:**
  - creates a new list. For all the items in the given list, create a copy and add it to the new list.
  - If two or more lists have items with the same name and unit, only create one copy with quantity equal to the sum of the individual quantities

Additional features!

What if you want to modify your concepts to account for some additional features?

Deadlines and Reminders

Users can specify deadlines or reminders for when items need to be bought.

Non-shopping items

Other than keeping track of items to shop for, users can also add to-do tasks, e.g. “Call Grandma to ask if she’s coming to dinner”

Collaborative shopping

Users can assign items / tasks to other users to do.

Separation of Concerns

concept ShoppingList [User]

purpose plan out what items (and how much of each) you need to shop for

principle
Users make shopping lists, add the things they need, and can combine lists so quantities add up.

concept ToDoList [User, Task]

purpose track which tasks need to be done and which are complete

principle
Users can create to-do lists, add tasks, and mark tasks done or undone.

Why?

State Representation

concept ToDoList [User, Task]

state

- a set of ToDoLists with
  - a User
  - a name String
  - a set of Tasks
- a set of Tasks with
  - a description String
  - a status Flag

Invariant: A task can only belong to one todo list.

Actions

concept ToDoList [User, Task]

actions

- **createToDoList**(user: User, name: String): ToDoList
- **addTask**(list: ToDoList, taskDescription: String)
- **markTaskDone**(task: Task)
- **markTaskUndone**(task: Task)

Which actions will help preserve the invariants?

createToDoList(user: User, name: String): ToDoList

- **effects**: adds a new empty to-do list

addTask(list: ToDoList, taskDescription: String)

- **requires:** list exists
- **effects:** creates a new task with the given description and status = undone, and adds it to the list

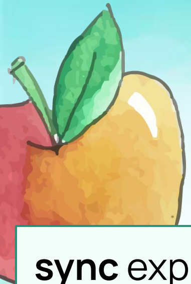
removeTask(list: ToDoList, task: Task)

- **requires:** list exists and task belongs to the list.
- **effects:** removes the task from the list.

markTaskDone (task: Task)

- **requires:** task exists.
- **effects:** sets the task's status to done.

Limitation: if we think about shopping items as tasks, the quantities will be all or nothing; you'd have to check off "buy 4 tomatoes" as a singular task



Sync

```
sync exportToDo
  ● when Request.exportShoppingList (shoppingList)
  ● then
      ○ ToDoList.createToDoList (user: shoppingList.User, name: shoppingList.name) : (todoList)
      ○ for each item in shoppingList.items:
          ■ ToDoList.addTask (list: todoList, taskDescription: item.name + item.quantity + item.unit)
```

What are the pros and cons of doing it this way?

Pros: easy to synchronize
Cons: if you make any modifications to one, they will not be reflected in the other

What other ways could you imagine doing this?

What other features can you think of for this app?

Here's an example of the full concept specification!