Quiz · Object-Oriented Software Engineering · 2019-11-20

Name:

GitHub Identifier:

Directions

We don't allow collaboration. The only material you may consult is your cheat sheet.

Use pencil and eraser, not pen. Readability, organization, and clean presentation are part of what's evaluated.

Your answers must fit in the space provided. The amount of space is an indication of how detailed you're expected to be.

Explain your design decisions. If you think that some aspect of a question is left unspecified or ambiguous, just make a design decision, justify it, and continue answering the question.

Extra Points

For each question below, you get 5 extra points if you answer right, 0 points if you don't answer, and –5 points if you answer wrong.

1. In the TODOOSE video series we talked about a famous TV show. Which one? Seinfeld
2. My pet appeared in one of the videos of me working through an assignment. What species of animal is it? Cat
3. This semester we invited you to a talk in our department. The speaker was an important software engineer who wrote books on the topic and spoke in conferences around the world. Who's that person and what was the talk about?

   Sandi Metz : Grit & a bike ride & writing a book

Background

TODOOSE became the most popular to-do application on the market! We're expanding our team and you're in charge of training a new member, Patricia:

Collaboration Workflow

15 points

Patricia knows just enough about Git & GitHub to be dangerous: what's a repository and a commit, and how to push and pull. Explain to her step by step the workflow that we use to collaborate. This is the workflow that we recommended for the group projects and that we followed on the TODOOSE video series.

Your goal is for Patricia to *understand* the collaboration workflow, not necessarily for her to be able to *execute* it. Explain the reasons for the steps, for example, "we do X because it allows us to Y"; don't give recipes, for example, "click on button X in IntelliJ, then click on button Y on GitHub."

1. On the project board, create one note for every feature. This gives people a high-level of the development.
2. When ready to start working on a feature, convert it into an issue, break it apart into a list of tasks, and assign it to who'll be working on it. Again, the idea here is visibility.
3. Create a branch, and work on the feature on it. Using branches we can work on different features at the same time without conflicts
4. Push that branch and create a pull request associated with it. The pull request must be linked to the issue that it closes.
5. Iterate on the process of commenting on the pull request, reviewing the code, and pushing more commits.
6. When the feature is complete, merge the pull request, which will close the issue, and update the project board.

The project board shows the state of development of each feature: Note → Planned, Open Issue → In Progress, Closed Issue → Done.
The project board has one column per iteration.

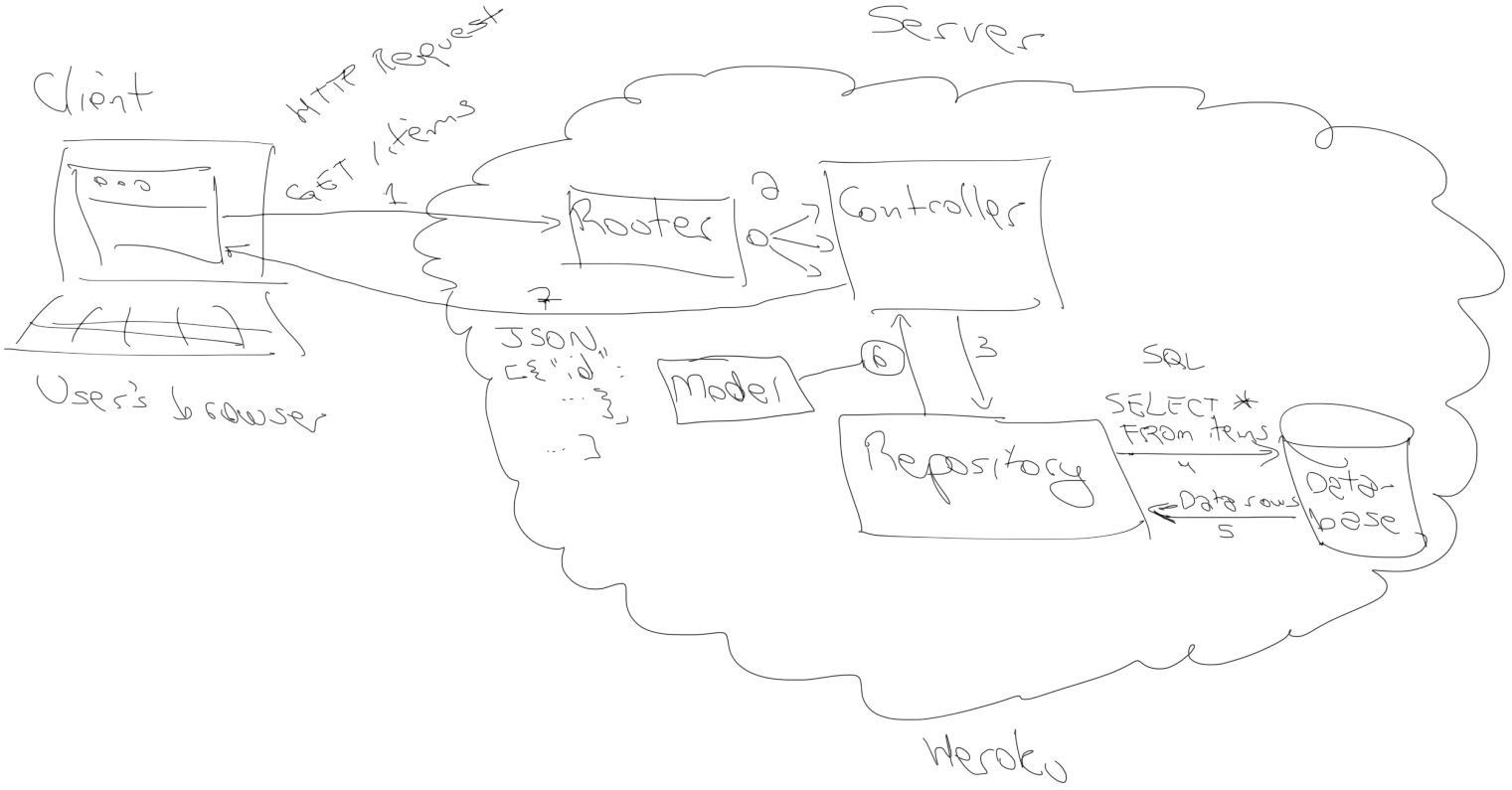
Architecture of a Web Application

Patricia is new to web development. You're going over the TODOOSE codebase with her and she asked you the following questions:

1. **1 point** In the deployed version of TODOOSE, where does the *server* part of the application run? On Heroku
2. **1 point** And where does the *client* part of the application run? On the user's browser.
3. **1 point** What part of the application is responsible for *originating* HTTP requests: server or client? Client
4. **1 point** And what part of the application is responsible for *responding* to HTTP requests: server or client? Server
5. **7 points** The server is composed of Models, Controllers, Router, Repositories, and Database. What are each of these for?

   Models: Business logic & source of truth for data
   Controllers: Coordinate the work of other components
   Router: Find which action of which controller responds to a certain HTTP request.
   Repositories: Map between models & the database
   Database: Persist data across server runs & guarantee data integrity

6. **1 point** What language does the Java part of the server use to communicate to the Database? SQL
7. **1 point** And in what format is the data communicated from the server to the client? JSON
8. **7 points** Let's bring together the answers from the questions above. Draw a diagram tracing through the request–response cycle to get the list of items (`GET http://localhost:7000/items`). Indicate where the different parts of the application are running; show where the request originates; show the components of the server that are activated, and the order in which they're activated; and include examples of the data as it's transmitted. We drew a similar diagram in **Lecture 1: Design Rudiments**.


Client
HTTP Request
GET /items
1
Server
Router
2
Controller
7
JSON
[{"id": ...}, ...]
Model
6
3
SQL
SELECT * FROM items
4
Repository
Data rows
5
Data-base
User's browser
Heroku

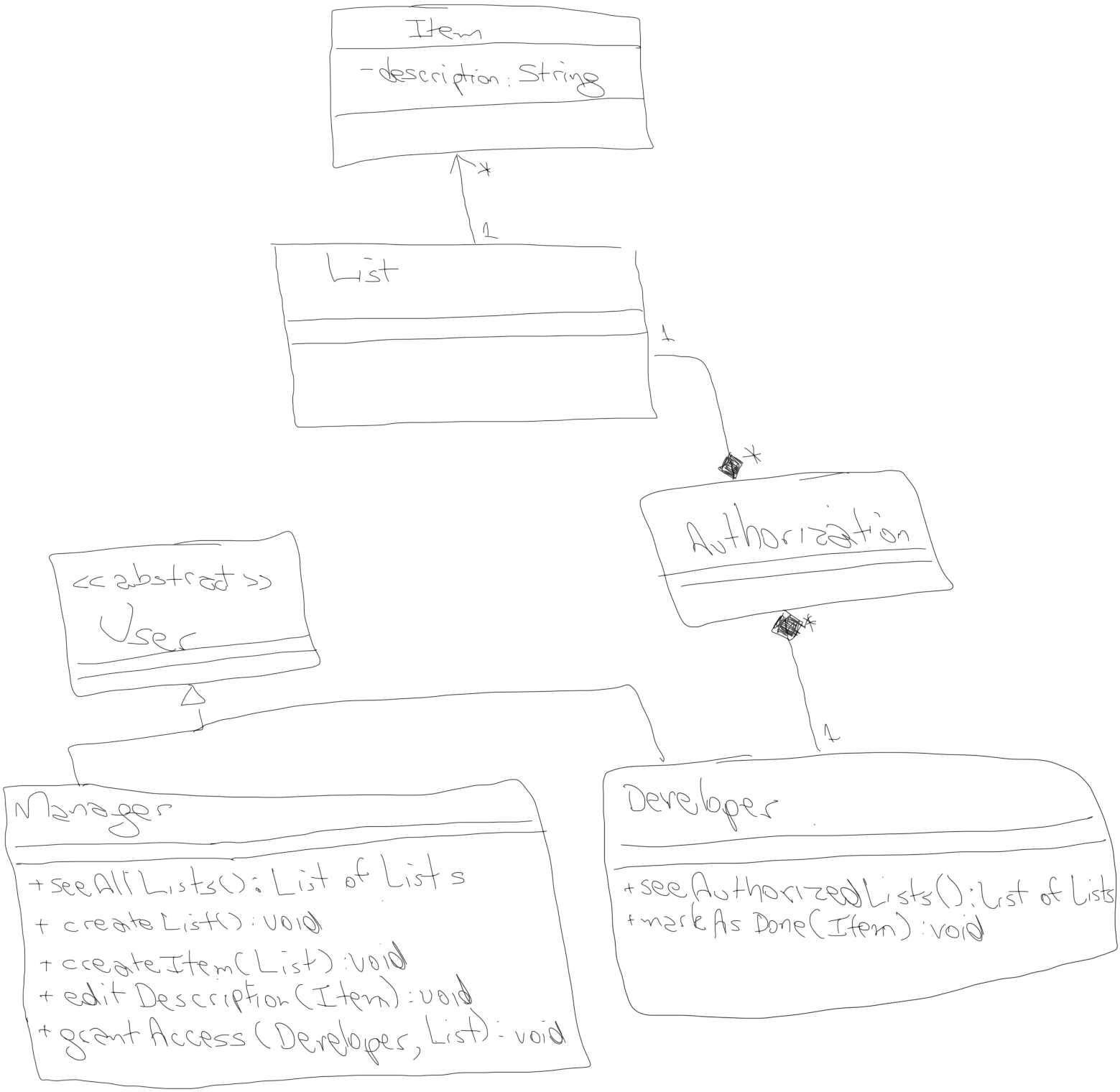
Class Diagram

15 points

Thanks to your help Patricia now understands the existing codebase and she's ready to start working on new features for TODOOSE. The idea behind these features is to make TODOOSE more suitable for managing a team of software engineers. Here's a breakdown:

- Introduce the notion of multiple to-do lists. For example, if we used TODOOSE to manage the development of TODOOSE itself, then we could have different to-do lists for the teams working on TODOOSE Community Edition and TODOOSE Enterprise Edition.
- Introduce the notion of users, who may be either project managers or developers.
- Project managers may see all to-do lists and create new ones. They may create new to-do items in these lists and edit the items descriptions, but not mark them as done.
- Developers may only see the to-do lists that a manager allowed them to see. They may mark to-do items as done, but not create new ones or edit their descriptions.

Help Patricia get started by drawing a class diagram of TODOOSE including these new features. Include classes, attributes, methods, parameters types, return types, associations, multiplicities, whole-part diamonds, inheritance, annotations, and so forth. Don't include implementation-specific details, for example, getters and setters, identifiers, controllers, and so forth.


Item
-description: String
*
1
List
1
*
Authorization
*
1
<<abstract>>
User
Manager
+seeAllLists(): List of Lists
+createList(): void
+createItem(List): void
+editDescription(Item): void
+grantAccess(Developers, List): void
Developer
+seeAuthorizedLists(): List of Lists
+markAsDone(Item): void

Design Principles & Design Patterns

You're teaching Patricia about Javalin and you mentioned that it includes an example of a design pattern called *Fluent Interface*. Here's an excerpt from `Server.java` to show it:

```
Javalin.create(/* ... */)
       .events(/* ... */)
       .routes(/* ... */)
       .exception(/* ... */)
       .start(/* ... */);
```

Patricia wanted to learn more about Fluent Interfaces, but she couldn't find it in the classic books & catalogs of design patterns. Help her out:

1. **5 points** Consider the Design Principles of *Keep It Simple, Stupid* (KISS), *Don't Repeat Yourself* (DRY), and *Interface Segregation* (the **I** in **SOLID**). Is the Fluent Interface in Javalin following or breaking these principles? Why?

KISS: From the user's perspective, the principle is followed, because it's just one class to learn. From the developer's perspective, it's more complex, because they must maintain the fluent interface.

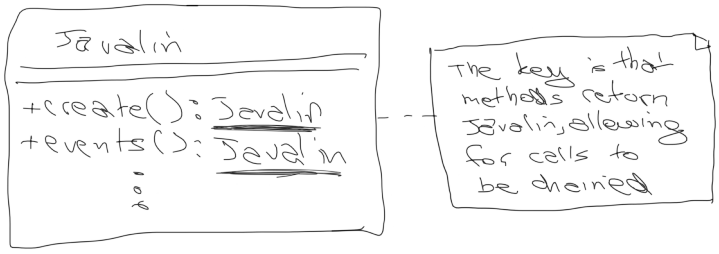
DRY: The principle is followed, because it allows us to avoid code like:

```
var app = Javalin.create(...)
app.events(...); app.routes(...); ...
```

↳ Repetition

I: Not following, because the interface of the Javalin class is big & includes some methods that are not closely related.

2. **5 points** Draw a class diagram to capture the essence of the Fluent Interface as implemented in the `Javalin` class.



Implementation

Help Patricia understand some parts of the codebase that are using newer language features:

1. **5 points** In `Server.java`, Patricia found the following line:

   ```
   var itemsController = new ItemsController();
   ```

   Explain to Patricia the use of `var` in Java: What is it doing? Why is it a good idea to use it? How would we have written a line like this before the introduction of `var`?

   Local type inference. Allow us to declare variables without stating their type, when the type is obvious from the initial value. In older versions of Java you'd write:
   ItemsController itemsController = new ItemsController();

2. **5 points** Patricia went to Javalin's website to learn more about it, and right on the homepage she found the following line:

   ```
   app.get("/", ctx -> ctx.result("Hello World"));
   ```

   Patricia has never seen code like

   ```
   ctx -> ctx.result("Hello World")
   ```

   before. Explain it to her: What is this code doing? Why did the designers of Javalin want you to write code like this when calling `app.get()`?

   Lambdas. Blocks of code with arguments. Or functions that don't have names (anonymous functions). In this lambda, ctx is the argument, and ctx.result(...) is the body. Javalin uses this because we don't want to run this body right away, but only in response to a request. The app.get(...) is only installing the lambda as the handler for the request.

3. **5 points** In the JavaScript part of the application, Patricia found a function that looks like the following:

```
async function getDataFromServer() {
  /* ... */ await fetch("/items") /* ... */
}
```

Explain `async/await` to Patricia: What is this doing? Why do we want to use `await` with `fetch()`?

This is a way to run slow operations without causing the browser to hang while waiting. Under the hood, async/await is just a more convenient way of using promises. Async marks a function as containing slow operations, and await marks the slow operation. Await is only valid within async functions. Examples of slow operations are: making HTTP requests (that's what fetch is doing), parsing JSON coming from the server, and so forth.

Security & User Management

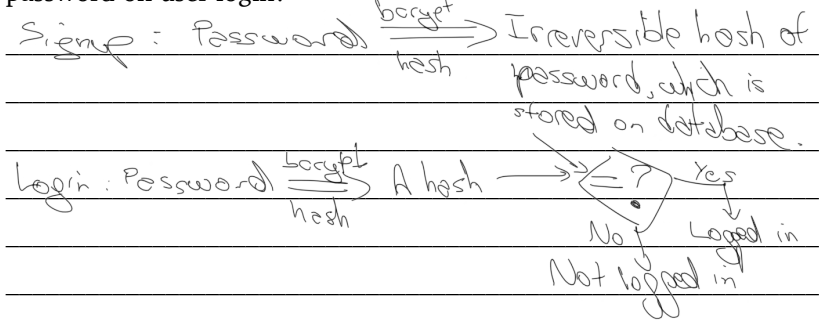
1. **2 points** Following the design you proposed in **§ Class Diagram**, Patricia started adding the notion of users to TODOOSE. She proposed to simply store passwords in plan text in the database. Why is this a bad idea?

Because attackers may be able to see the database, for example using SQL Injection. And people tend to reuse passwords.

2. **2 points** Your arguments convinced Patricia to protect the passwords, and she invented her own algorithms for doing it. Why is this a bad idea?

Because software engineers don't know all the details that go into a secure algorithm. It's better to reuse generally accepted solutions created by security experts.

3. **2 points** Your arguments convinced Patricia to use a standard algorithm for password protection, bcrypt. Now she's trying to understand a little bit about how it works. On a high level, what does bcrypt do to the password on user signup, and how does it verify a password on user login?



2 points Patricia was reading about bcrypt and the documentation mentioned a *salt*. What's that? Why is it necessary?

A salt is a random string that is appended to the password before the hashing to prevent Rainbow Table attacks, in which attackers store a cache of the hashes of many common passwords.

4. **2 points** Why must we protect passwords with Key Derivation Functions like bcrypt, instead of regular hashing algorithms like MD5 and SHA?

Because KDFs are designed to be slow and counter a brute-force attack, while MD5 and SHA are designed to be fast.

5. **5 points** Recall from **§ Class Diagram** that project managers may not mark items as done—that's the job of the developers. Patricia is wondering where she should implement the logic to check whether a user is a project manager and prevent them from marking items as done. Should she do it on the client, or on the server? Why?

Server, because the client code runs on the user's machine, which can't be trusted. We could forge requests using Postman for example, to bypass any security validations on the browser.
For usability reasons, we may want to implement the feature on the client as well, for example, to hide buttons for features that aren't allowed.

Programming Paradigms

Patricia saw us playing with the codebase for the Rock–Paper–Scissors game. We have two different versions of code that accomplishes the same task:

Version 1

```
class Rock {}
class Paper {}
class Scissors {}

function toString(playerChoice) {
  if (playerChoice instanceof Rock) return "💎";
  if (playerChoice instanceof Paper) return "📄";
  if (playerChoice instanceof Scissors) return "✂️";
}
```

Version 2

```
class Rock {
  toString() {
    return "💎";
  }
}

class Paper {
  toString() {
    return "📄";
  }
}

class Scissors {
  toString() {
    return "✂️";
  }
}
```

1. **2 points** Which version is written in an object-oriented style, and which is written in a functional style?

   Version 1 -> Functional
   Version 2 -> Object-Oriented

2. **8 points** In general, when would you advise Patricia to write code in an object-oriented style? And when would you advise her to write code in a functional style?

   Object-oriented: You expect to add more types of data that have the same behavior, for example, Spock & Lizard. Or the method is very closely related to the identity of the object for example its string representation.
   Functional: You expect to add more functions on the same data, for example, beats(), and so forth.