CS312 Spring 2025 – Midterm Solution

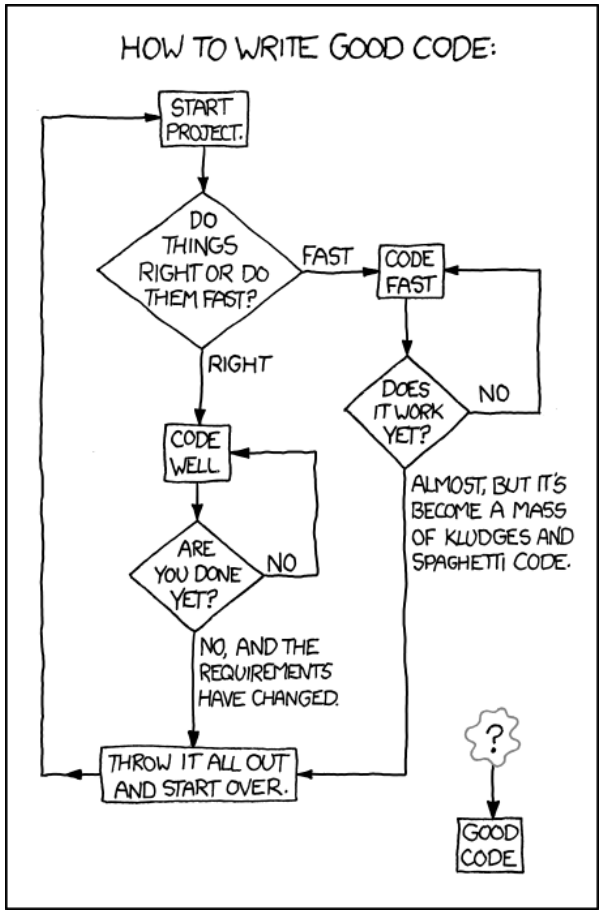
Name: _____

Date: _____ **Start time:** _____ **End time:** _____

Honor Code:

Signature: _____

This exam is closed computer, course web page, notes, texts, searching online, AI and consulting with others, i.e., closed everything except a notes page you prepare. You are only permitted *one* double-sided letter-sized sheet of notes of your own creation (may be typed). **You have 2 hours in a single sitting to complete the exam.** Read the problem descriptions carefully and write your answers clearly and *legibly* in the space provided. Circle or otherwise indicate your answer if it might not be easily identified. You may use extra sheets of paper, stapled to your exam, if you need more room, as long as the problem number is clearly labeled and your name is on the paper. If you attached extra sheets indicate on your main exam paper to look for the extra sheets for that problem.



Learning Target	Assessment
1	
2	
3	
4	
5	
6	
7	
8	

Question 1. User stories

You are developing a web application for organizing public events. When interviewing stakeholders, multiple respondents described wanting a calendar view that summarizes the number of events on each day of date range. Write two I.N.V.E.S.T. user stories for this feature, one from the perspective of an event planner, the other from the perspective of a potential event goer. Your user stories will be evaluated on format and quality.

(a) Event planner:

Solution:
As an event planner,
I want to view a calendar with a visual summary of scheduled events in a selected date range
So that I can quickly identify dates for my event that aren't to busy.

(b) Potential event goer:

Solution:
As a potential event goer,
I want to view a calendar with a visual summary of scheduled events in a selected date range
So that I can quickly identify when interesting events may be happening.

Question 2. Javascript

Assume the function `wait(sec)` returns a promise that resolves after `sec` seconds have elapsed. Assume that every other operation is instantaneous (e.g., takes 0 milliseconds). Remember that `Date.now()` returns the number milliseconds elapsed since 12:00AM January 1, 1970, UTC.

```
1  function do(time, since) {
2    return wait(time).then(() => {
3      console.log(Date.now() - since);
4    });
5  }
6
7  const start = Date.now();
8  do(2, start);
9  do(1, start).then(() => {
10   do(4, start);
11 });
12 do(3, start);
13 console.log("end");
```

```
1  async function do(time, since) {
2    await wait(time);
3    console.log(Date.now() - since);
4  }
5
6  const start = Date.now();
7  do(2, start);
8  do(1, start);
9  do(4, start);
10 await do(3, start);
11 console.log("end");
```

Consider the two code snippets above. Write the expected output for left-side code below on the left. If the right-side code produces the same result indicate below, otherwise provide the expected output below on the right.

◯ Both snippets produce the same output

Solution:

```
end
1000
2000
3000
5000
```

Solution:

```
1000
2000
3000
end
4000
```

Question 3. Testing

You are developing a React component named `WordGame` for playing a word guessing game. The component takes the hidden word as a prop. The component provides a text input where the user can enter their guess and button "Guess" to check their guess. Incorrect guesses are displayed in a list below the input with an × appended. A correct guess is displayed at the end of the list with a ✓ appended. When the user guesses correctly the component displays the time elapsed since the game started (the component was mounted), e.g.,"You guessed in 5.2s!". Using the skeleton below, implement **pseudo-code** for a F.I.R.S.T. unit test to verify that game correctly handles a correct guess after one incorrect guess. You do **not** need to provide executable Javascript, instead describe the steps of your test as pseudo-code. For example, one of the steps in your pseudo-code might be:

```
Assert mock function was not called
```

You may or may not need all of the functions below. You only need to include pseudo-code in bodies of the functions relevant to your answer.

```
describe("Word game", () => {
  beforeEach(() => {
```

Solution:

```
Set mock system time to 1000
```

```
  });
  afterEach(() => {
```

Solution:

```
Clear the mocks
```

```
  });
  test("Shows correct end of game after one incorrect guess", () => {
```

Solution:

```
Render the WordGame component with prop "HiddenWord"

Find the guesses list and assert it is empty

Find text input and enter "input1"
Find and click the "Guess" button
Find the guesses list and assert it is ["input1×"]

Advance mock time 3s

Find text input and enter "HiddenWord"
Find and click the "Guess" button
Find the guesses list and assert it is ["input1×", "HiddenWord✓"]

Find and assert the presence of "You guessed in 3s!" message
```

A satisfactory test will mock or otherwise meaningfully control or measure time, perform one incorrect guess (entering guess text and clicking the “Guess” button), asserting on the content of the “guess” list after the guess, and then perform a correct guess, asserting on the contents of the “guess” list and presence of the correct “You guessed message ...” with the time. Tests that include mocks need to clear the mocks. Tests that don’t control time need a way to consistently measure and assert the elapsed time.

```
  });
});
```

Question 4. Scenarios

On your application's login page the user can toggle the password visibility, i.e. switch between masked characters and plain text, by clicking on an icon in the password field. When the password is hidden, a "Show password" icon appears, when it is in plain text a "Hide password" icon appears. Write a Gherkin-style test scenario for changing the password visibility. You do not need to provide the implementation details of the tests, just describe the scenario for the test.

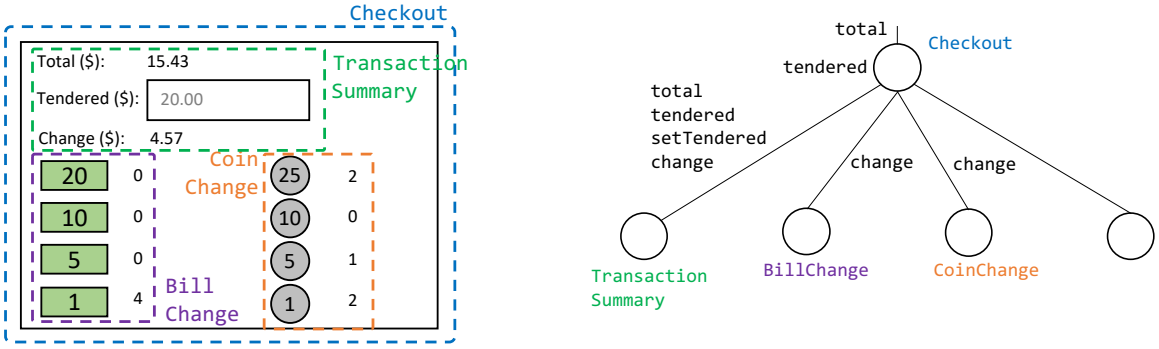
Solution:

```
Given the user is on the login page
And has entered the username and password "user1" and "password1"
And the password is shown as len("password1") masked characters
And the "Show password icon is displayed
When the user clicks the "Show password" icon
Then the password should be shown as the plain text "password1"
And the "Hide password" icon is displayed
When the user clicks the "Hide password" icon
Then the password should be shown as len("password1") masked characters
```

Question 5. React

You are implementing the `Checkout` component shown below with React. It receives a numeric prop representing the total purchase. Entering an amount paid (tendered) by the customer in cash computes the necessary amount of change and how the change should be provided, i.e., the number of bills and coins of the denominations shown. Outline and label the wireframe (below, top) with a possible set of components. Label the tree (below, bottom) with components to show the hierarchy. Label the tree nodes with state implemented in that component and label the tree edges with props passed to each component (similar to the figure in programming assignment 2). Repeated components can be labeled once in the tree. The top-level component and its prop(s) are labeled for you. Any implementation reflecting good React practices will be accepted. You may not need all the nodes in the tree or may need to add nodes depending on your design; cross out any unused nodes. Your component, state and prop names should be sufficiently descriptive that their role is clear.

Solution:



An explanation is not required for full credit, but is provided here for clarity. Any solution consistent with React best practices was accepted, here is one approach. There is a single piece of state, `tendered` stored in the parent component `Checkout`. The amount of change and the number of each denomination can be derived from `total` prop and the `tendered` state and so should not be represented as their own state (to maintain a single source of truth). That tendered amount is set from within `TransactionSummary` which needs the total, amount tendered, compute change and a callback to update the tendered amount as props. We use two components for bill change and coin change responsible for computing the number of each denomination from change. Although not necessary and not shown here for simplicity, the change entries (denomination and number) are good candidates for another component to encapsulate the repetitive elements.

Question 6. REST

For each of the following pages in a NextJS-based event management application, provide an appropriate RESTful front-end (browser) URL for that page and, where relevant, an appropriate RESTful server API endpoint (HTTP verb and URL) that component would interact with. An example is provided below.

Page	Page URL	API HTTP verb and URL
View all articles on Simplepedia	**/articles**	**GET /api/articles**
Search for events near a zipcode	**/events/search?zip=05753**	**GET /api/events?zip=05753**
Update an event details	**/events/5/edit**	**PUT /api/events/5**
Add a comment to an event	**/events/5/comments/new**	**POST /api/events/5/comments**

Question 7. Data modeling

You are developing a web application for organizing public events through which people can find and register to attend events. You will be using a relational database to store the data for this application.

(a) Identify the *minimum* set of models you would define in your server backend to implement the following user story:

As a past attendee, I want to review events that I have registered for and attended, so that I can provide feedback to the event organizers and information to future potential attendees.

Solution: The minimum models would be `User` and `Event` connected via two different join tables and models: `Registration` to store registration/attendance and `Review` to store the reviews. Answers that did not specify the join tables as separate models, but indicated the role for the join tables and the data they stored were accepted.

(b) Which of the following best describe the relations between the following pairs of entities. Select one answer for each pair, then briefly explain your answers.

User and Event

- ◯ One-to-One
- ◯ One-to-Many
- √ **Many-to-Many**
- ◯ No relation

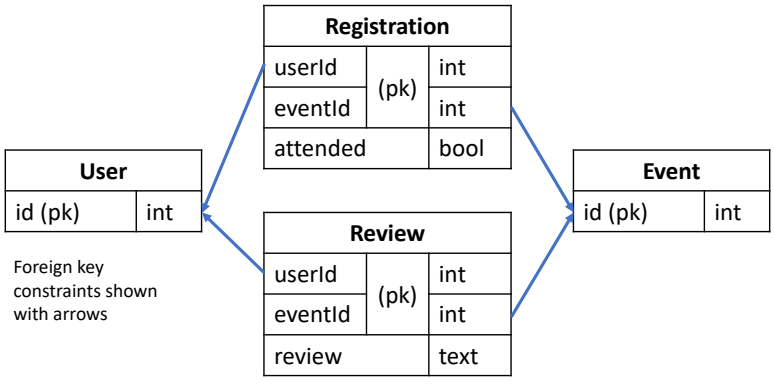
Event and Review

- ◯ One-to-One
- √ **One-to-Many**
- ◯ Many-to-Many
- ◯ No relation

Solution: A user can register for many events and event can have many registered users, so the relationship between User and Event is many-to-many. User and Event also a have a many-to-many relationship between via Review. Since an event can have many reviews, but each review is associated with a single event, the relationship between Event and Review is one-to-many.

(c) In a normalized schema designed for a relational database (RDBMS), what schema would be needed to implement the user story in part (a)? You do not need to provide SQL, just the attributes, their types, the primary key, and any foreign key constraints needed to implement the user story. You only need to provide the schema relevant to the user story, not the entire schema for the application.

Solution: A minimal solution is shown below:



Since a user can only register for an event or review an event once, we can use a composite of the `userId` and `eventId` as the primary key for the `Registration` and `Review` tables. Solutions

that also used separate incrementing primary keys for those tables were accepted.

Question 8. Development processes

For each of the following, indicate whether the action would be consistent with the best practices for software development as described in class or not consistent. Here "consistent" is defined as consistent with good development practices generally, not that it was required as part of our class. Briefly explain each answer.

(a) Keep a technically complex feature as a single user story to ensure continuity during its development, especially across sprints.

◯ Consistent √ **Not consistent**

> **Solution:** Complex features should be decomposed into smaller, valuable increments (i.e., the "S" and "V" in INVEST) that can be completed within a single sprint, even if the full feature requires multiple stories. We can use "epics" to link related user stories.

(b) Regularly rebase the `main` branch to simplify and linearize the project's commit history.

◯ Consistent √ **Not consistent**

> **Solution:** Rebase or other similar techniques rewrite history, which can cause problems for other developers working from shared history. Once branches are shared, their commit history should not be modified. Rebasing to simplify and linearize the commit history *is* appropriate for private branches, but not for shared branches like `main`.

(c) Configure your Continuous Integration (CI) test pipeline to treat any warnings as errors, i.e., the CI pipeline fails if any warnings are generated.

√ **Consistent** ◯ Not consistent

> **Solution:** We want to immediately address any actual warnings or disable false-positive warnings. Ignoring warnings can lead to missed problems in the future. Treating all warnings as errors is a good practice to ensure that we address all warnings.