IBM Code of Conduct & Ethics Quiz

Introduction

Organisational Context

IBM is an American-founded technology company founded in 1911. It is a leading provider of software and services in hybrid cloud, artificial intelligence, and enterprise and strategic consulting services. IBM places a strong emphasis on ethical conduct, compliance, and professional responsibility. To support this, IBM maintains a comprehensive Code of Conduct and a set of ethical guidelines that govern employee behaviour, decision-making, and interactions with clients, partners, and colleagues.

Project Overview

This project develops an **IBM Code of Conduct & Ethics Quiz**, designed for internal use at IBM. The aim of the application is to provide IBMers with an accessible and interactive way to reinforce their understanding of IBM's ethical standards, including topics such as conflicts of interest, data confidentiality, responsible use of technology, and reporting unethical behaviour.

The application allows users to answer multiple-choice questions, receive immediate feedback on their performance, and have results securely stored for review or export. In terms of relevance, by testing knowledge through a structured quiz, the application supports ongoing compliance awareness and helps promote a consistent ethical culture across the organisation.

Technical and Non-Technical Perspective

From a non-technical standpoint, the application effectively highlights how software solutions can be used to support ethical considerations, governance, and professional standards within a large enterprise.

From a technical standpoint, the application follows a professional development lifecycle. It is designed, built iteratively, tested, deployed, and documented. The project is developed in **Python** and uses a graphical user interface created via the **Tkinter** library. Additionally, the application incorporates object-oriented principles, alongside automated testing and continuous integration.

Design Section

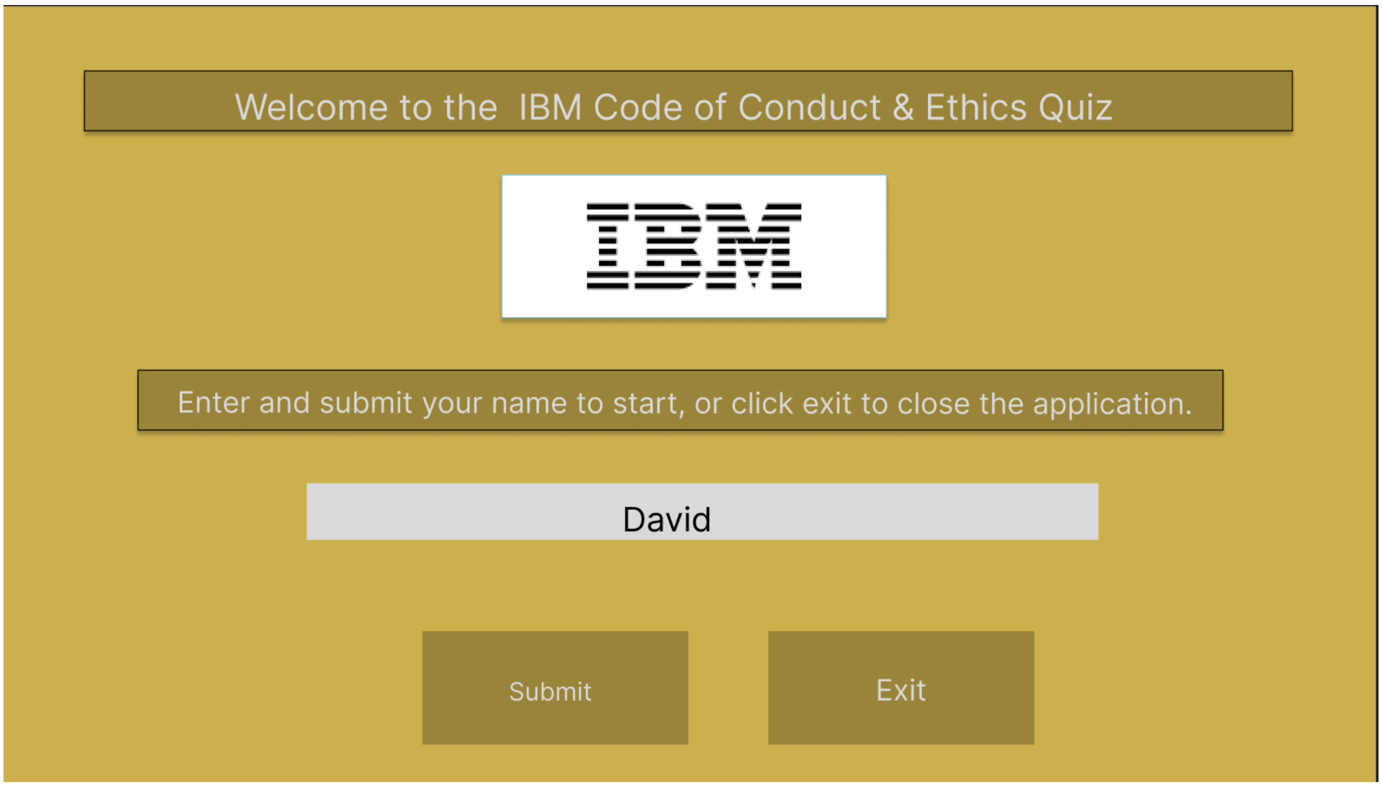
GUI Design

The graphical user interface (GUI) has been designed to provide a clear and accessible user experience for IBM employees. It prioritises simplicity, clarity, and ease of navigation while also maintaining a professional appearance appropriate for an enterprise environment.

Figam GUI Design Link - https://www.figma.com/design/jSgJQLkI1BMuzE6WGKZbGs/Quiz_App?node-id=0-1&p=f&t=8du06XqQwQSVRu5d-0

The application follows a linear user journey consisting of four primary screens:

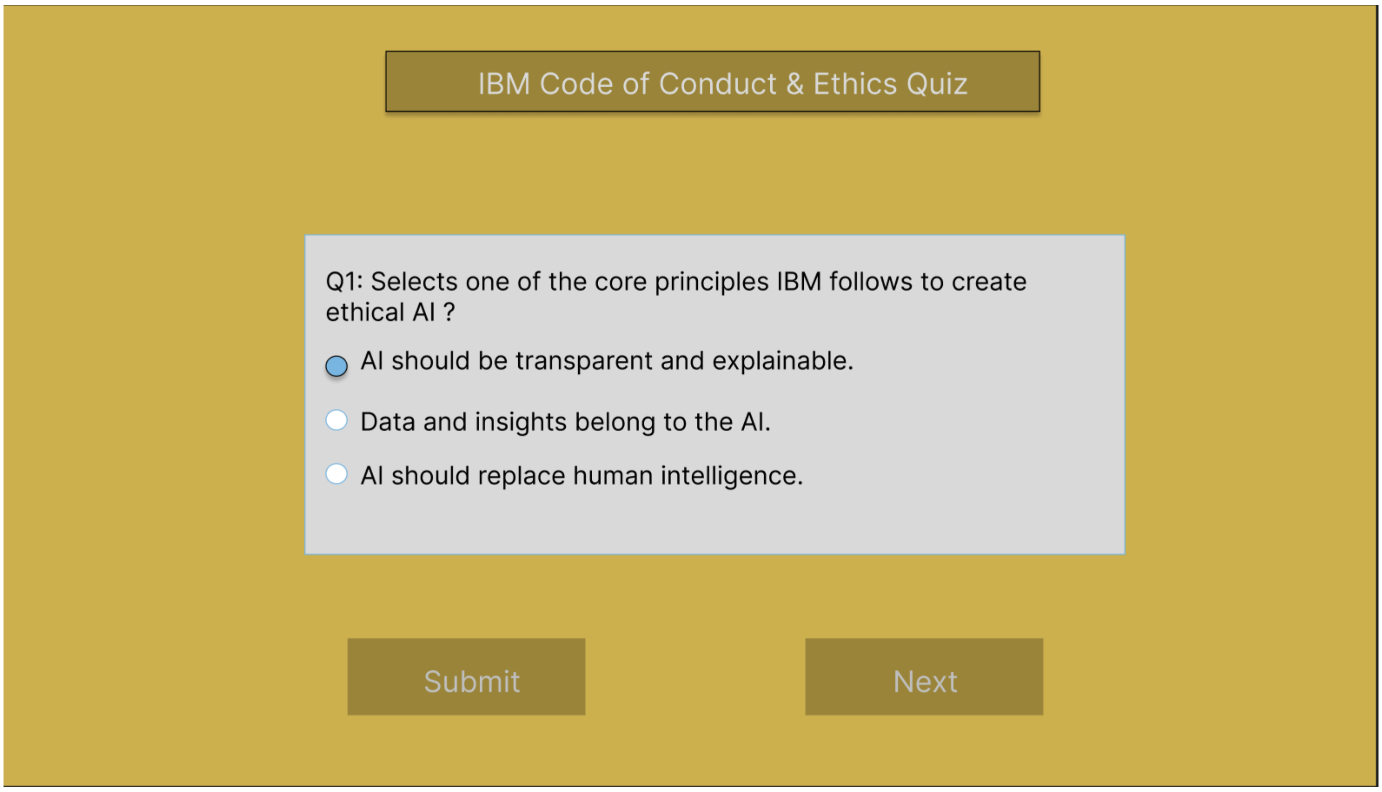
- **Welcome Screen**
  - Displays application title and welcome message.
  - Provides users with options to start the quiz by entering a username and submitting, or to exit the application via the exit button.


Welcome to the IBM Code of Conduct & Ethics Quiz

Enter and submit your name to start, or click exit to close the application.

David

- **Quiz Screen**
  - Displays a multiple-choice question follwed by the question number at a time.
  - Uses radio buttons for answer selection.
  - Prevents submission unless an answer is selected.
  - Provides a **Submit** button to confirm the selected answer.
  - Includes a **Next** button that changes the question. Otherwise, it remains inactive until submission.


IBM Code of Conduct & Ethics Quiz

Q1: Selects one of the core principles IBM follows to create ethical AI ?

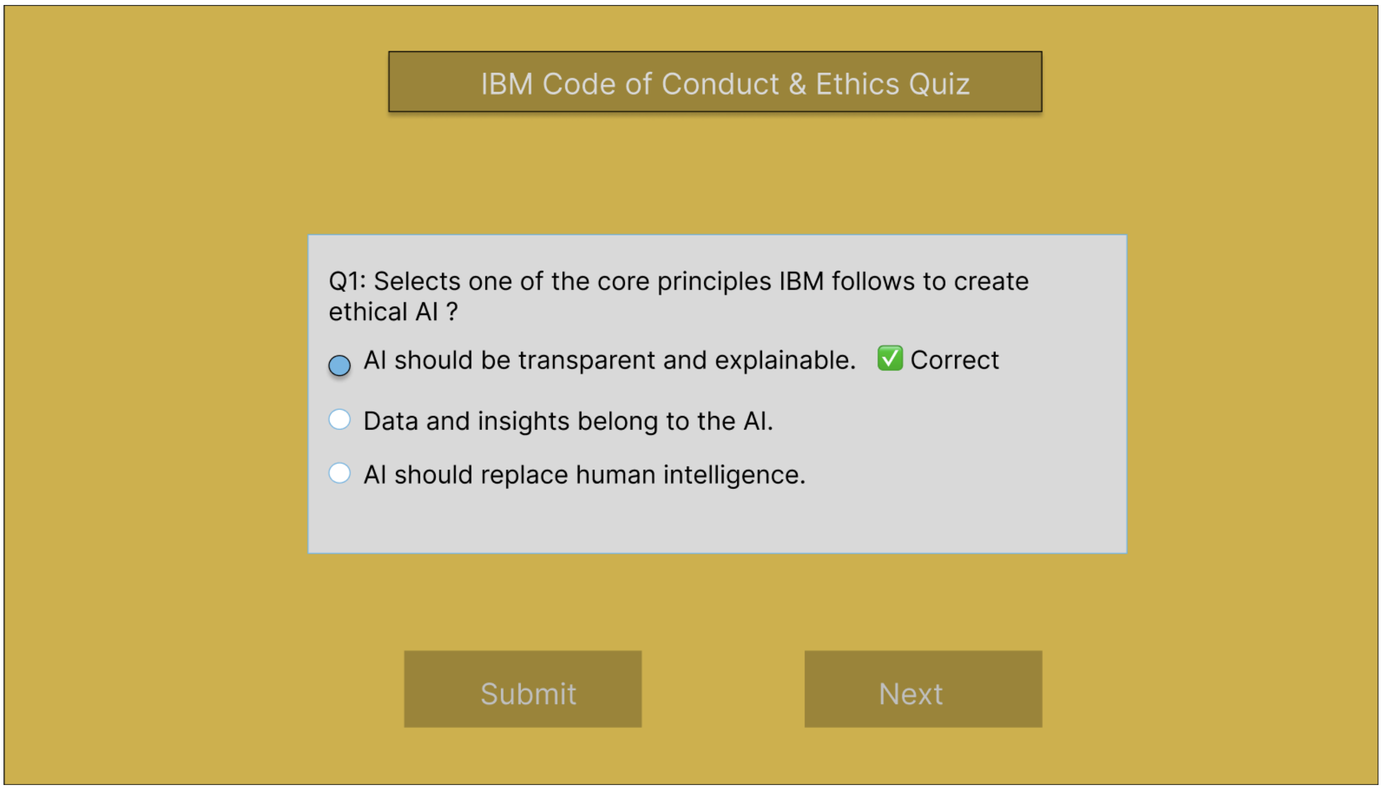
[✓ AI should be transparent and explainable.]

Data and insights belong to the AI.

AI should replace human intelligence.

Answer / Feedback

- Provides immediate visual feedback indicating whether the selected answer is correct or incorrect.
- Ensures feedback is shown before users can continue, reinforcing learning outcomes.



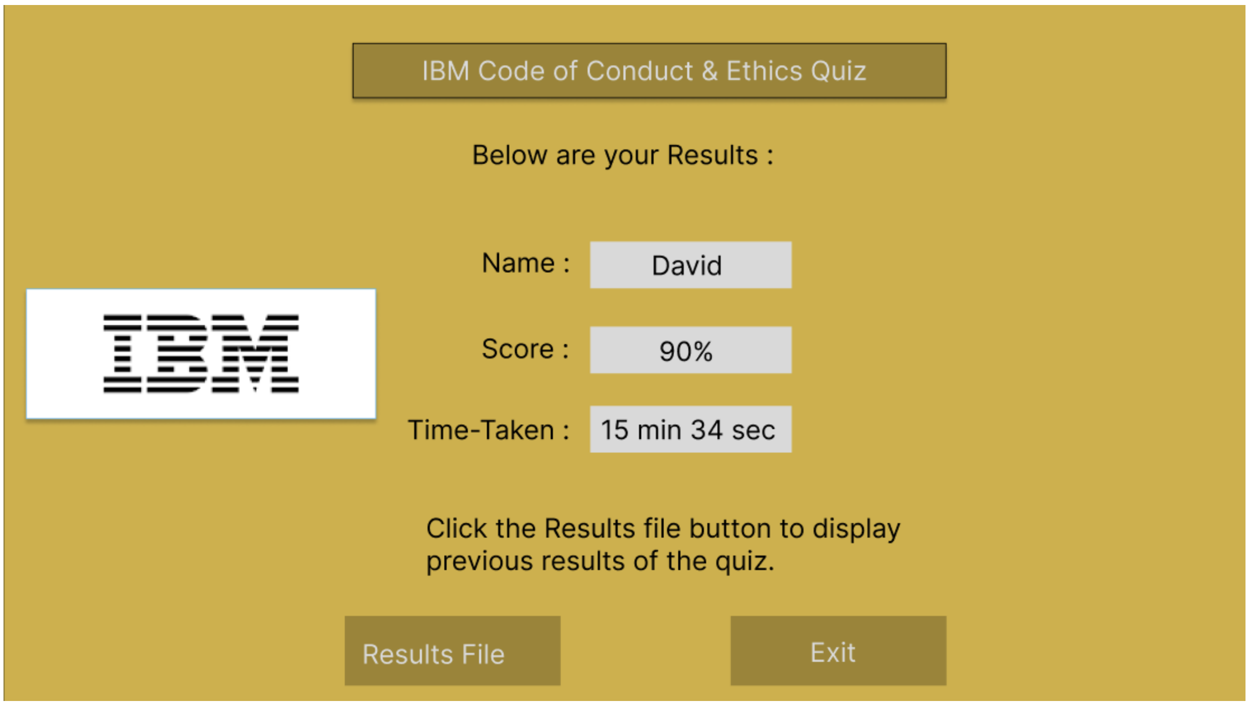
Q1: Selects one of the core principles IBM follows to create ethical AI ?

[✓ AI should be transparent and explainable.] ✅ Correct

Data and insights belong to the AI.

AI should replace human intelligence.

- **Results Screen**
  - Displays the outcome of the completed quiz.
  - Acts as the final stage of the quiz user journey.
  - **Current Results View**
    - Displays the user’s name entered at the start of the quiz.
    - Shows the user’s final score in the format : number of correct answers/ total number of questions. E.g 8/10
    - Displays the total time taken to complete the quiz.
    - Ensures results are presented in a clear and readable format.
    - Includes a button to access stored quiz results or to exit the application.



Below are your Results :

Name : David

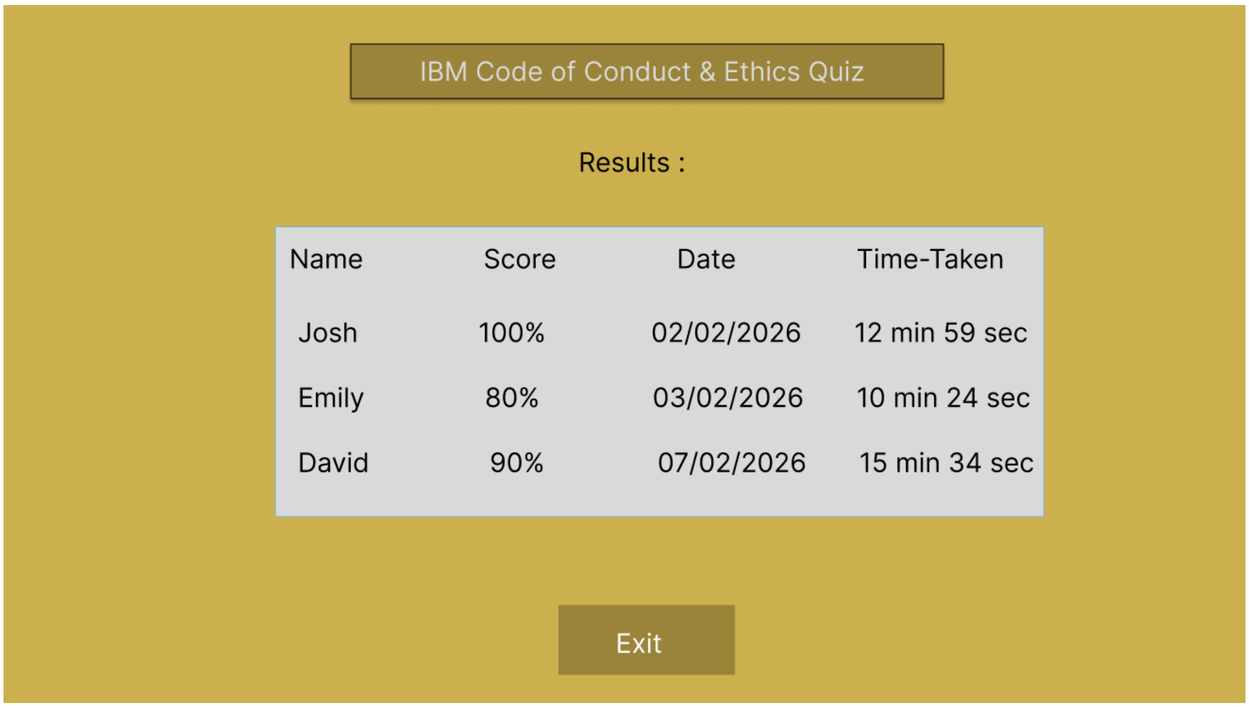
Score : 90%

Time-Taken : 15 min 34 sec

Click the Results file button to display previous results of the quiz.

  - **Stored Results View**

- Displays when the user selects the **Results_File** button.
- Loads and presents previously stored quiz results from CSV file.
- Shows historical data such as user names, scores, completion dates and times.
- Allows users or administrators to review past quiz attempts.



IBM Code of Conduct & Ethics Quiz

Results :

Name	Score	Date	Time-Taken
Josh	100%	02/02/2026	12 min 59 sec
Emily	80%	03/02/2026	10 min 24 sec
David	90%	07/02/2026	15 min 34 sec

Functional Requirements

- The application must provide a welcome interface that allows users to start the quiz or exit the application.
- The application must allow users to enter their name prior to starting the quiz.
- The application must present ethics-based multiple-choice questions related to IBM's Code of Conduct.
- The application must display one question at a time during the quiz.
- **Question Handling**
  - The system must present a dedicated **Question Screen** where users can select a single answer using radio buttons.
  - The system must prevent users from submitting a question unless an answer has been selected.
  - The system must record the user's selected answer upon submission.
- **Answer/Feedback**
  - The system must indicate whether the selected answer is correct or incorrect, after submission.
  - The system must display feedback before allowing the user to proceed to the next question.
  - The system must allow users to navigate sequentially through the quiz using a *Next* button.
- **Results Handling**
  - The system must display a **Results Screen** upon quiz completion.
  - The Results Screen must show the user's name, final score, and total time taken to complete the quiz.
  - The system must store quiz results in persistent storage.
  - The system must provide a *Results File* button to allow users to view previously stored quiz results.
  - The system must load and display past quiz results data when the *Results File* button is selected.
- **Data Management**
  - The system must read quiz questions from a persistent data source, CSV file.
  - The system must handle missing or invalid data without crashing.

Non-Functional Requirements

- **Performance**
  - The application must load the first quiz question within **1.5 seconds** after the user selects “Start Quiz”.
  - Each screen transition (e.g., Question → Feedback → Next Question) must complete within **500 milliseconds** on a standard workstation (8GB RAM, 2GHz CPU).
- **Usability**
  - All interactive elements (buttons, radio buttons, input fields) must have a minimum size of **40×40 px** to support accessibility and ease of use.
- **Accessibility**
  - The user interface must use a minimum font size of **12pt** for readability.
  - Buttons and radio buttons must have a minimum interactive size of **32×32 pixels**.
  - Text and background colour contrast must be at least **3:1**.
  - All error or validation messages must be clearly displayed for a minimum of **3 seconds** before navigating away.
  - All controls must be fully keyboard-operable
- **Reliability**
  - The system must not lose quiz results during a normal session; stored results must persist across restarts.
  - CSV write operations must complete successfully **100% of the time** under normal operating conditions.
- **Data Integrity**
  - Quiz results must be written to the CSV file in **under 200 milliseconds** after quiz completion.
  - Partial or corrupted rows in the results CSV must be skipped gracefully without crashing the application.
- **Security**
  - The application must not store any sensitive personal data beyond the user’s optional display name.
  - The CSV file must contain only: name, score, time taken, and timestamp — no additional fields.
- **Portability**
  - The application must run without modification on Windows, macOS, and Linux systems using **Python 3.10+** and their built-in Tkinter libraries.
- **Maintainability**
  - Core quiz logic must be separated from the GUI so that at least **80% of logic functions** can be unit-tested without opening the interface.
  - All classes and functions must contain docstrings following **PEP 257** conventions.
- **Testability**
  - The system must provide at least **10 automated unit tests** with a minimum **75% code coverage** for logic-related modules.
  - All input validation functions must produce consistent outputs. (testable under pytest).

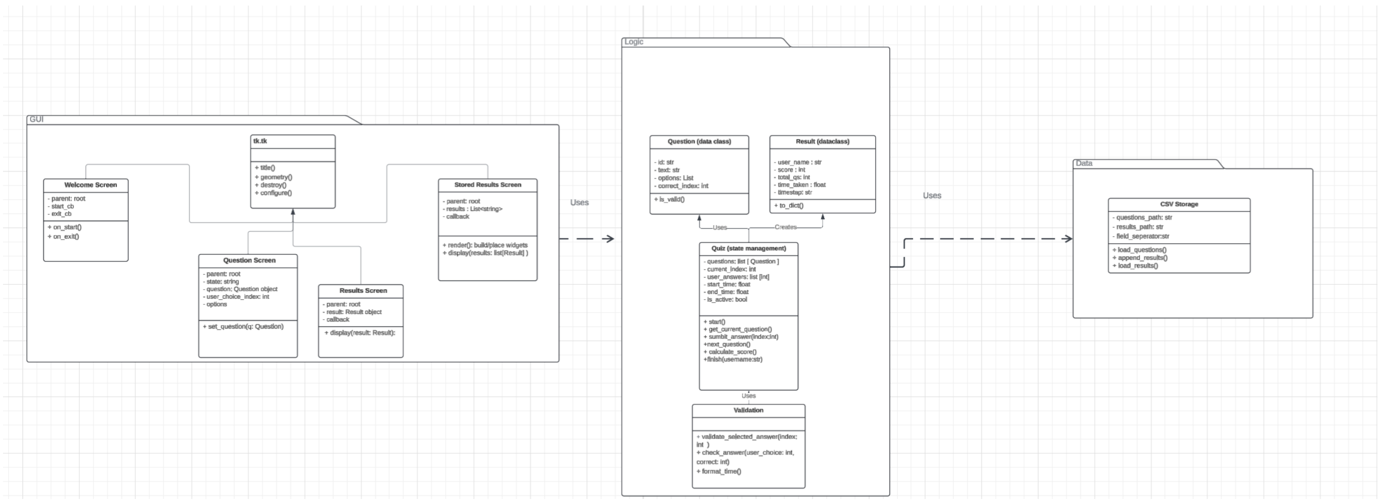
Tech Stack Outline

- **Programming Language: Python**
  - The core language used to develop the quiz application.
- **GUI Framework: Tkinter**
  - Built-in Python library used to create the graphical interface.
  - Provides buttons, labels, radio buttons, and layout managers needed for the quiz screens.

- **Data Storage: CSV Files**
  - Allows persistent storage that is easy to read or edit.
  - Used to store quiz results and quiz questions.
  - Supports exporting results.
- **Version Control: Git & GitHub**
  - Git is used for tracking changes and maintaining a clear development history.
- **Testing Framework: pytest**
  - Enables automated testing of core quiz logic and input validation functions.
  - Allows the core quiz logic to be tested independently from the GUI, ensuring that functions like scoring and validation can be tested reliably without opening the Tkinter interface.
- **Continuous Integration: GitHub Actions**
  - Runs automated tests on every commit or pull request.
  - Helps maintain code quality and ensures new updates do not break existing functionality.

Class Diagram

- Below is the class diagram :



Development Section

GUI Layer

Welcome Screen

- WelcomeScreen is the initial GUI screen displayed when the application starts.

```python
class WelcomeScreen(tk.Frame):
def __init__(self, parent, start_callback, exit_callback):
    super().__init__(parent, bg='#D3AF37')
    self.start_callback = start_callback
    self.exit_callback = exit_callback
```

- Class Inheritance & Initialisation - Inheriting from tk.Frame makes the screen a self-contained, reusable component. Storing callbacks as instance variables is crucial for communication between the UI and application logic, allowing the screen to respond to user actions.

```
    self.grid_columnconfigure(0, weight=1)
```

- This configures the layout of the screen.

```
    ttk.Button(btns, text="Start Quiz", command=self.on_start).grid(row=0, column=0, padx=6)
```

- The command=self.on_start parameter connects the button click to an event handler. Without this, clicking the button does nothing and no communication occurs with the rest of the application.

```
    def on_start(self):
  name = self.name_var.get().strip()
  self.start_callback(name)
```

- This method retrieves user input and passes it to the main controller via the callback. It's the critical "exit point" where the welcome screen transitions to the quiz.

```
try:
  logo_image = PhotoImage(file=logo_path)
except tk.TclError as e:
  print(f"Error loading logo: {e}")
```

- This error handling prevents the application from crashing if the image file is missing or corrupted, it demonstrates proffesional programming and makes the program robust.

Question Screen

- Displays a quiz question with multiple choice options and supports two distinct modes: question mode where users select answers, and feedback mode where answers are revealed.

```
class QuestionScreen(tk.Frame):
    def __init__(self, parent, submit_callback, next_callback):
        super().__init__(parent)
        self.submit_callback = submit_callback
        self.next_callback = next_callback
        self.state: Literal["question", "feedback"] = "question"
        self.selected = tk.IntVar(value=-1)
```

- Class Inheritance & Initialisation - Inheriting from tk.Frame makes the screen a self-contained, reusable component that manages both the question display and answer collection. Storing callbacks as instance variables (submit_callback and next_callback) allows for communication between the UI and the main application controller. The state variable tracks which mode the screen is in, and selected tracks which option the user picked.

```
def set_question(self, q: Question):
    self.question = q
    self.q_text.config(text=q.text)
    self.selected.set(-1)
    self._render_options(q.options)
```

- This method loads a new question onto the screen by updating the question text and rendering all answer options. It resets the selection to -1 (nothing selected) to ensure users start fresh with each new question.

```
def show_question_mode(self):
    self.state = "question"
    self.lock_inputs(False)
    self.status_lbl.config(text="")
    self.submit_btn.grid(row=0, column=0, padx=4)
    self.next_btn.grid_forget()
```

- This method switches the screen to question mode by unlocking radio buttons, clearing status messages, and showing the Submit button while hiding the Next button. This mode allows users to freely select and change their answer before submitting.

```
def show_feedback_mode(self, correct_index: int, selected_index: int):
    self.state = "feedback"
    self.lock_inputs(True)
    for i, lbl in enumerate(self._option_labels):
        if i == selected_index == correct_index:
            lbl.config(text="✓", foreground="green")
        elif i == selected_index and selected_index != correct_index:
            lbl.config(text="✗", foreground="red")
        elif i == correct_index:
            lbl.config(text="✓", foreground="green")
```

- This method switches to feedback mode by locking the radio buttons and displaying visual feedback symbols. A green checkmark (✓) appears next to the correct answer, and a red X (✗) appears next to the user's answer if it was wrong.

```
def lock_inputs(self, locked: bool):
    for rb in self._radio_widgets:
        rb.config(state="disabled" if locked else "normal")
```

- This method enables or disables all radio buttons based on the locked parameter.

```
# Input validation
def on_submit(self):
    sel = self.get_selection()
    if sel is None:
        self.status_lbl.config(text="Please select an answer before submitting.")
        if self._radio_widgets:
            self._radio_widgets[0].focus_set()
        return
    self.submit_callback(sel)
```

- This method validates that the user has actually selected an answer before allowing submission. If no answer is selected, it displays an error message and focuses the first option to guide the user. If valid, it calls the submit callback to pass the selected index to the main controller, demonstrating proper error handling and user guidance.

```
def _render_options(self, options: list[str]):
    for w in self.options_frame.winfo_children():
        w.destroy()
    self._radio_widgets.clear()
    self._option_labels.clear()

    for i, opt in enumerate(options):
        row = ttk.Frame(self.options_frame)
        row.pack(anchor="w", pady=2)
        rb = ttk.Radiobutton(row, text=opt, value=i, variable=self.selected)
        rb.pack(side="left")
        fb = ttk.Label(row, width=2)
        fb.pack(side="left", padx=(8, 0))
        self._radio_widgets.append(rb)
        self._option_labels.append(fb)
```

- This method creates radio buttons for each answer option. It first clears any previously rendered options, then creates a row for each option containing both a radio button and a placeholder label for feedback symbols.

Results Screen

- ResultsScreen displays the final quiz results to the user after completing the quiz. It shows the user's name, final score, and time taken, along with a button to view all previous quiz attempts.

```
class ResultsScreen(ttk.Frame):
    def __init__(self, parent, view_results_callback):
        super().__init__(parent)
        self.view_results_callback = view_results_callback

        self.name_lbl = ttk.Label(self, text="")
        self.score_lbl = ttk.Label(self, text="")
        self.time_lbl = ttk.Label(self, text="")
```

- Inheriting from ttk.Frame makes the results screen a self-contained and reusable component. Storing view_results_callback as an instance variable allows the screen to communicate with the main controller when the user wants to view past results. Three empty labels (name_lbl, score_lbl, time_lbl) are created to hold the result information, which will be populated after the quiz completes.

```
def display(self, result):
    self.name_lbl.config(text=f"Name: {result.user_name or '—'}")
    self.score_lbl.config(text=f"Score: {result.score}/{result.total_questions}")
    self.time_lbl.config(text=f"Time taken: {result.time_taken}s")
```

- This method takes a Result object and populates the three labels with the actual quiz results. It displays the user's name, the final score and the time taken in seconds. This separation of data and display allows the screen to be reusable for different quiz results without needing to recreate the UI.

```
actions = ttk.Frame(self)
actions.grid(row=4, column=0, pady=(12, 0), sticky="w")
ttk.Button(actions, text="Results File (View Previous Results)",
command=self.on_view_results).grid(row=0, column=0, padx=4)
```

- Creates a container frame (actions) to hold View Previous Results button. The command=self.on_view_results parameter connects the button to an event handler. Using a container frame makes the code more reusable, and easier to add new features to the system at a later date.

```
def on_view_results(self):
    self.view_results_callback()
```

- This event handler calls the view_results_callback when the user clicks the "View Previous Results" button, allowing the screen to trigger navigation to the stored results.

---

stored Results Screen

- StoredResultsScreen displays all past quiz attempts in a table format, allowing users to review their past performance with key metrics including name, score, date/time, and time taken to complete the quiz.

```
self.table = ttk.Treeview(
    self,
    columns=("name", "score", "timestamp", "time_taken"),
    show="headings",
    height=12
)
```

- I used Tkinter's Treeview widget with four columns to display results in a clean, organized table format. The Treeview is ideal for displaying structured data because it supports multiple columns and sorting capabilities. The height=12 parameter allows 12 rows to be visible at once, this provides an efficient way to display a results without overwhelming the user interface.

```
for col, text in (("name", "Name"), ("score", "Score"), ("timestamp", "Date & Time"),
                 ("time_taken", "Time Taken")):
    self.table.heading(col, text=text)
    self.table.column(col, width=150, anchor="w")
```

- This loop creates four columns for the table. For each column, it does two things: (1) sets the heading text that appears at the top (like "Name", "Score", "Date & Time", "Time Taken"), and (2) configures the column width to 150 pixels and aligns the text to the left.

```

def display(self, results):
    for r in results:
        # Convert seconds to minutes and seconds
        minutes = r.time_taken // 60
        seconds = r.time_taken % 60
        time_formatted = f"{minutes}m {seconds}s"

        self.table.insert(
            "",
            "end",
            values=(
                r.user_name or "—",
                f"{r.score}/{r.total_questions}",
                r.timestamp,
                time_formatted
            )
        )
```

- This method populates a table with all previous quiz attempts by iterating through a list of Result objects. Each result is inserted as a row showing the user's name score in "correct/total" format, the full date and time of the quiz, and the time taken converted from seconds to a readable "Xm Ys" format. The time conversion uses integer division (//) to get minutes and the modulo operator (%) to get remaining seconds.

```
def on_exit(self):
    self.exit_callback()
```

- This method allows users to close the application and complete their session after reviewing past results.

```
def focus_default(self):
    self.table.focus_set()
```

- Allows users to navigate through rows using arrow keys without requiring a mouse click

---

Logic Layer

Models

This module contains the core data classes used throughout the application:

- Question: Represents a single quiz question with options and correct answer
- Result: Represents the outcome of a completed quiz attempt

Question Class - Data Structure for Quiz Questions

```
@dataclass(frozen=True)
class Question:
    id: str
    text: str
    options: List[str]
    correct_index: int
```

- **@dataclass(frozen=True)** decorator automatically creates a class for storing data while making it immutable (cannot be changed after creation). This ensures questions remain exactly as they are once loaded from the CSV file. The frozen property prevents accidental modifications and protects data integrity.

```
# Validation
def is_valid(self) -> bool:
    return bool(self.text.strip()) and len(self.options) >= 2 and 0 <= self.correct_index <
len(self.options)
```

- **The is_valid()** method checks three conditions: (1) the question text is not empty, (2) there are at least 2 answer options, and (3) the correct answer index is within bounds. This validation catches data errors before the quiz starts, preventing crashes or undefined behavior during gameplay.

Result Class - Data Structure for Quiz Results

```
@dataclass(frozen=True)
class Result:
    user_name: Optional[str]
    score: int
    total_questions: int
    time_taken: float
    timestamp: str
```

- **Immutable Result Storage** - Similar to Question, Result uses `@dataclass(frozen=True)` to create an immutable record of how a user performed on the quiz. This ensures results cannot be accidentally modified after being created, maintaining historical accuracy and preventing data corruption.

```
def to_dict(self) -> dict:
    return {
        "user_name": self.user_name or "",
        "score": self.score,
        "total_questions": self.total_questions,
        "time_taken": self.time_taken,
        "timestamp": self.timestamp,
    }
```

- **Converting to Dictionary Format** - The `to_dict()` method converts the Result object into a dictionary, making it easy to save to CSV files or JSON. The `self.user_name or ""` handles the case where a user didn't enter their name by converting None to an empty string, ensuring consistent data when persisting to files.

Quiz Logic - The Quiz class manages the entire quiz flow, tracking questions, answers, timing, and score calculation.

Starting the Quiz

```
def start(self):
    self.current_index = 0
    self.user_answers = [None] * len(self.questions)
    self.start_time = time.time()
    self.end_time = None
    self.is_active = True
```

- **Initialize Quiz State** - Resets the quiz to the beginning by clearing previous answers, setting the current question to 0, and recording the exact start time using `time.time()`. The `is_active = True` flag indicates the quiz is running. This method is called when the user clicks "Start Quiz" on the welcome screen.

Getting and Submitting Answers

```
def get_current_question(self) :
    return self.questions[self.current_index]

def submit_answer(self, selected_index: int):
    self.user_answers[self.current_index] = selected_index
```

- **Current Question & Answer Recording** - `get_current_question()` returns the question the user is currently viewing by using `current_index` as an array index. `submit_answer()` stores which option the user selected in the `user_answers` array at the same index position. This parallel array structure keeps answers aligned with their questions.

Navigation Through Questions

```
def next_question(self) -> bool:
    if self.current_index + 1 < len(self.questions):
        self.current_index += 1
        return True
    return False
```

- **Question Navigation** - Checks if there are more questions remaining before advancing. Returns `True` if successful (move to next question), or `False` if at the last question (quiz should end). The conditional `self.current_index + 1 < len(self.questions)` prevents moving past the final question.

Score Calculation

```
def calculate_score(self) -> int:
    score = 0
    for ans, q in zip(self.user_answers, self.questions):
        if ans is not None and ans == q.correct_index:
            score += 1
    return score
```

- **Comparing Answers with Correct Answers** - Loops through user answers paired with their corresponding questions using `zip()`. For each answer, checks if it matches the question's `correct_index`. The `ans is not None` condition handles skipped questions. Returns the total count of correct answers.

Finishing the Quiz and Creating Results

```
def finish(self, user_name: str | None):
    self.end_time = time.time()
    self.is_active = False
    time_taken = round(self.end_time - (self.start_time or self.end_time), 2)
    ts = datetime.now(timezone.utc).strftime("%Y-%m-%dT%H:%M:%SZ")
    return Result(
        user_name=user_name,
        score=self.calculate_score(),
        total_questions=len(self.questions),
        time_taken=time_taken,
        timestamp=ts
    )
```

- **Quiz Completion and Result Generation** - Records the end time and calculates elapsed time by subtracting `start_time` from `end_time` (in seconds). Creates a UTC timestamp for when the quiz was completed. Packages all data into a Result object: user's name, calculated score, total questions, time taken (in seconds, raw format), and timestamp. This Result object is then passed to the results display screens.

The Quiz Flow

```
1. start() → Record time, reset answers, set is_active = True
2. Loop:
   - get_current_question() → Display question
   - User selects answer
   - submit_answer() → Record selection
   - next_question() → Move to next (returns True) or end (returns False)
3. finish() → Calculate score, record end time, return Result
```

Validation logic

Validation Module - Pure utility functions for validating user input and formatting time data.

Validating User Answer Selection

```
def validate_selected_answer(idx: int | None, num_options: int) -> bool:
    return idx is not None and 0 <= idx < num_options
```

- **Checking Valid Selection** - Ensures the user actually selected an answer and that the selection is within valid bounds. The function checks two conditions: (1) `idx is not None` confirms a selection was made (not empty), and (2) `0 <= idx < num_options` ensures the index points to an actual option. This prevents crashes from out-of-range indices.

Checking Answer Correctness

```
def check_answer(selected: int, correct: int) -> bool:
    return selected == correct
```

- **Answer Comparison** - Compares the user's selected answer index directly with the correct answer index.

Formatting Time for Display

```
def format_time(seconds: float) -> str:
    seconds = int(seconds)
    m, s = divmod(seconds, 60)
    h, m = divmod(m, 60)
    return f"{h:02d}:{m:02d}:{s:02d}" if h else f"{m:02d}:{s:02d}"
```

- **Converting Seconds to Readable Format** - Transforms raw seconds into a human-readable time string (MM:SS if under 1 hour, or HH:MM:SS if 1+ hours). The `divmod()` function efficiently splits time into components: `divmod(seconds, 60)` extracts minutes and remaining seconds, then `divmod(minutes, 60)` extracts hours and remaining minutes. The format specifier `{h:02d}` adds leading zeros (e.g., "04:05" instead of "4:5"). The conditional `if h` omits hours from display when they're zero, keeping short quiz times compact (e.g., "02:15" instead of "00:02:15").

Summary

These validation functions are **stateless utilities** used throughout the application: answer validation before submission prevents errors, answer checking determines correctness for feedback, and time formatting displays quiz duration in a user-friendly format.

CSV Repository

CSV Repository - Handles all file I/O operations for loading quiz questions and saving/loading quiz results from CSV files.

Loading Questions from CSV

```
def load_questions(self):
    choices = row["choices"].split(self.field_sep) if row.get("choices") else []
    q = Question(
        id=row.get("id", ""),
        text=row.get("text", ""),
        options=[c for c in choices if c],
        correct_index=int(row.get("correct_index", -1)),
        category=row.get("category") or None,
        difficulty=row.get("difficulty") or None,
    )
    if q.is_valid():
        questions.append(q)
```

- **Reading and Validating Questions** - Opens the questions CSV file and reads each row as a dictionary using `csv.DictReader`. For each row, it splits the choices field using the custom separator (default "||") to extract individual answer options. Creates a Question object and validates it using `is_valid()` before adding to the list. Invalid or malformed rows are skipped silently. Returns an empty list if the file doesn't exist, allowing the app to start even without questions.

Saving Quiz Results

```
def append_result(self, r: Result):
    try:
        with open(self.results_path, "x", newline="", encoding="utf-8") as f:
            writer = csv.DictWriter(f, fieldnames=[...])
            writer.writeheader()
    except FileExistsError:
        pass
    with open(self.results_path, "a", newline="", encoding="utf-8") as f:
        writer = csv.DictWriter(f, fieldnames=[...])
        writer.writerow(r.to_dict())
```

- **Smart File Creation and Appending** - Uses two-step approach: first tries to create the file with headers using mode "x" (exclusive creation), and if the file already exists, catches the exception and skips header creation. Then appends the result using mode "a" (append). This ensures headers are written exactly once while allowing multiple results to be added over time. Converts the Result object to a dictionary using `to_dict()` for CSV writing.

Loading Historical Results

```
def load_results(self):
    results.append(
        Result(
            user_name=row.get("user_name") or None,
            score=int(row.get("score", 0)),
            total_questions=int(row.get("total_questions", 0)),
            time_taken=float(row.get("time_taken", 0.0)),
            timestamp=row.get("timestamp") or "",
        )
    )
```

```
def load_results(self):
    results.append(
        Result(
            user_name=row.get("user_name") or None,
            score=int(row.get("score", 0)),
            total_questions=int(row.get("total_questions", 0)),
            time_taken=float(row.get("time_taken", 0.0)),
            timestamp=row.get("timestamp") or "",
        )
    )
```

- **Reconstructing Results from CSV** - Reads the results CSV file and converts each row back into a Result object. Handles type conversions: converts score and total_questions to integers, time_taken to float, and treats empty user_name as None. Malformed rows are skipped to prevent crashes. Returns an empty list if the file doesn't exist, allowing the stored results screen to display gracefully even with no history.

Testing Strategy & Methodology

Testing Approach Overview

My testing strategy for the IBM Code of Conduct & Ethics Quiz application uses a two-tiered approach combining **manual testing** and **unit testing**. This combination ensures comprehensive coverage of both user-facing functionality and underlying business logic.

Why This Approach?

Manual Testing forms the foundation of the strategy because it allows me to validate the complete user journey and end-to-end workflows. By systematically testing each screen (Welcome, Question, Results, Stored Results), I can verify that the application behaves as intended from a user's perspective. Manual testing is essential for evaluating user experience, checking responsive design, validating error messages, and confirming data persistence across quiz attempts. This approach catches integration issues that might not be apparent when testing individual components in isolation.

Unit Testing complements manual testing by focusing on the accuracy of individual functions and methods in isolation. By writing automated tests for core business logic (validation functions, quiz state management, CSV data handling, model validation), I can quickly verify that these components work correctly and continue to work correctly as the code evolves. Unit tests allow me to test fundamental operations such as : answer validation, score calculation, time formatting, and data conversion—produce correct results under various conditions. Additionally, unit tests serve as executable documentation, demonstrating how each function should behave.

Testing Plan Structure

Phase 1: Manual Testing (20 test cases across all screens)

- Tests the complete user flow: Welcome Screen → Question Navigation → Results Display → Historical Results Viewing
- Validates UI/UX: button clicks, input fields, error messages, visual feedback
- Confirms data integrity: scores calculated correctly, results saved to CSV, persistence works across sessions
- Tests edge cases: empty names, single character inputs, multiple quiz attempts
- Error handling: verifies graceful failure when CSV files missing

Phase 2: Unit Testing (Automated tests for core modules)

- Validation functions: `validate_selected_answer()`, `check_answer()`, `format_time()`
- Quiz logic: initialization, answer submission, score calculation, result creation
- Models: `Question.is_valid()`, `Result.to_dict()`
- Data persistence: CSV reading and writing operations

Justification of Testing Methods

1. **Manual Testing is Justified Because:**
    - Validates the actual user experience, not just code correctness
    - Catches UI/UX issues that automated tests miss (layout, colors, spacing)
    - Tests integration between multiple components working together
    - Allows testing of error scenarios and edge cases interactively
    - Provides visual evidence (screenshots) for documentation
2. **Unit Testing is Justified Because:**
    - Tests individual functions in isolation, ensuring each component is reliable
    - Quickly identifies bugs in business logic (score calculation, time conversion)
    - Provides fast feedback during development (runs in seconds)
    - Makes refactoring safer by catching regressions immediately

- Documents expected behavior through test cases
- Achieves high code coverage of critical functions

By combining both approaches, we achieve **comprehensive coverage**: manual testing verifies the application works end-to-end from a user perspective, while unit testing ensures the underlying code is solid and maintainable. This two-tiered strategy reduces the risk of undetected bugs while also making the codebase more testable and maintainable for future development.

Manual Testing Table

#	Test Case	Steps	Expected Result	Actual Result	Status	Notes
1	App Launch	1. Run `python3 -m app.main` from project root	Welcome screen displays with title, logo, description, name input field, and two buttons (Start Quiz, Exit)	Welcome Screen alongside all its widgets were displayed	Pass	Environment: macOS, Python 3.13. IBM logo loads correctly. Window dimensions 900x600 as specified.
2	Welcome Screen - Enter Name	1. Launch app 2. Type "Test User" in name field	Name appears in input field	Test User appeared in the input field	Pass	Tested with various inputs: single word, multiple words, special characters (@, #, $). All work correctly.
3	Welcome Screen - Start Quiz (with name)	1. Enter "David" as name 2. Click "Start Quiz"	First question screen displays with question text and 4 radio button options	The question screen is displayed alongside 4 answer options	Pass	Name "David" is stored and later displays in results. Questions load from CSV without delay.
4	Welcome Screen - Start Quiz (without name)	1. Leave name field empty 2. Click "Start Quiz"	First question screen displays (name is optional)	The first question screen was displayed	Pass	When empty, name defaults to "Anonymous" in results (as per updated model). No error message shown.
5	Welcome Screen - Exit Button	1. Click "Exit" button	Application closes	Application closed	Pass	Clean exit with no console errors. Verified: process terminates correctly.
6	Question Screen - Display Question	1. Start quiz	Question 1 displays: "What does IBM stand for?" with 4 options	Question 1 is displayed	Pass	All 5 questions from questions.csv load successfully.
7	Question Screen - Select Answer	1. On question screen 2. Click first radio button	First option is selected (radio button fills)	First radio button if filled	Pass	Radio button selection is mutually exclusive - selecting one deselects others. Visual feedback immediate.

#	Test Case	Steps	Expected Result	Actual Result	Status	Notes
8	Question Screen - Submit Without Selection	1. On question screen 2. Click "Submit" without selecting answer	Error message displays: "Please select an answer before submitting."	Error message is displayed	Pass	Error message appears in blue status label. User must select answer before progressing. Prevents invalid submissions.
9	Question Screen - Submit Correct Answer	1. Select option 1 (International Business Machines) for Q1 2. Click "Submit"	Feedback mode displays with green ✓ next to correct answer, status message: "Correct!"	Feedback mode is displayed	Pass	Green checkmark (✓) appears in green color. Status message "Correct!" displays immediately. Transition smooth.
10	Question Screen - Submit Incorrect Answer	1. Select option 3 for Q2 2. Click "Submit"	Feedback mode displays with red ✕ next to selected answer, green ✓ next to correct answer, status message: "Incorrect. Review the correct answer above."	Feedback is displayed on screen	Pass	Red X (✕) on wrong answer, green ✓ on correct answer. Both visible for learning. Status message in red for visibility.
11	Question Screen - Lock Inputs in Feedback	1. Submit an answer (feedback mode) 2. Try clicking other radio buttons	Radio buttons are disabled, cannot select other options	The radio buttons are locked, user can't select other options	Pass	Radio buttons appear grayed out after feedback. Prevents answer modification. User must click Next to proceed.
12	Question Screen - Next Button	1. In feedback mode 2. Click "Next"	New question is displayed, returns to question mode with new question	A new question is displayed	Pass	Smooth transition. Previous selection cleared. Submit button reappears, Next button hidden.
13	Quiz Navigation - All 4 Questions	1. Complete all 4 questions with various answers	All 4 questions display in sequence without errors	Quiz ran as expected, navigated through all 4 questions	Pass	Completed full quiz. No crashes or freezes. All questions accessible.
14	Results Screen - Display Final Score	1. Complete all 4 questions 2. Results screen displays	Screen shows: User name, Final Score (e.g., 3/4), Time Taken (e.g., 30s), "View Previous Results" button	Results screen and all its widgets are displayed	Pass	Results display: Name "David", Score "3/4", Time "30s". Format clean and readable. Button styling consistent.
15	Results Screen - View Previous Results	1. Click "View Previous Results" button	StoredResultsScreen displays with table of all past quiz attempts	Past results are displayd in a table format	Pass	Table appears with existing results from results.csv. Smooth navigation. Data preserved from previous tests.

#	Test Case	Steps	Expected Result	Actual Result	Status	Notes
16	Stored Results Screen - Display Table	1. On Stored Results screen	Table displays with columns: Name, Score, Date & Time, Time Taken with sample data	Results screen and its widgets are displayed	Pass	4 columns displayed correctly. Table headers: Name, Score, Date & Time, Time Taken. Data rows populated from CSV.
17	Stored Results Screen - Exit Button	1. On Stored Results screen 2. Click "Exit"	Application closes completely	Application closed	Pass	Clicking Exit terminates entire application.
18	Error Handling - No Questions	1. Delete questions.csv 2. Launch app 3. Try to start quiz	Warning dialog displays: "No questions available. Please add questions to continue."	Error message is displayed	Pass	Error handling works, message appears with warning icon. User can click OK and return to welcome screen.
19	Multiple Quiz Attempts	1. Complete quiz as "User1" 2. View stored results	previous results saved with correct names, scores, and times	Allows for multiple attempts	Pass	User1: score 4/4, David: score 3/4. Both visible in stored results. Names, scores, timestamps all correct. CSV multi-entry working.
20	Edge Case - Single Character Name	1. Enter "D" as name 2. Complete quiz	Quiz works, result saves with name "D"			Single character name accepted. Result displays with name "D" in stored results table. No validation errors. Edge case handled.

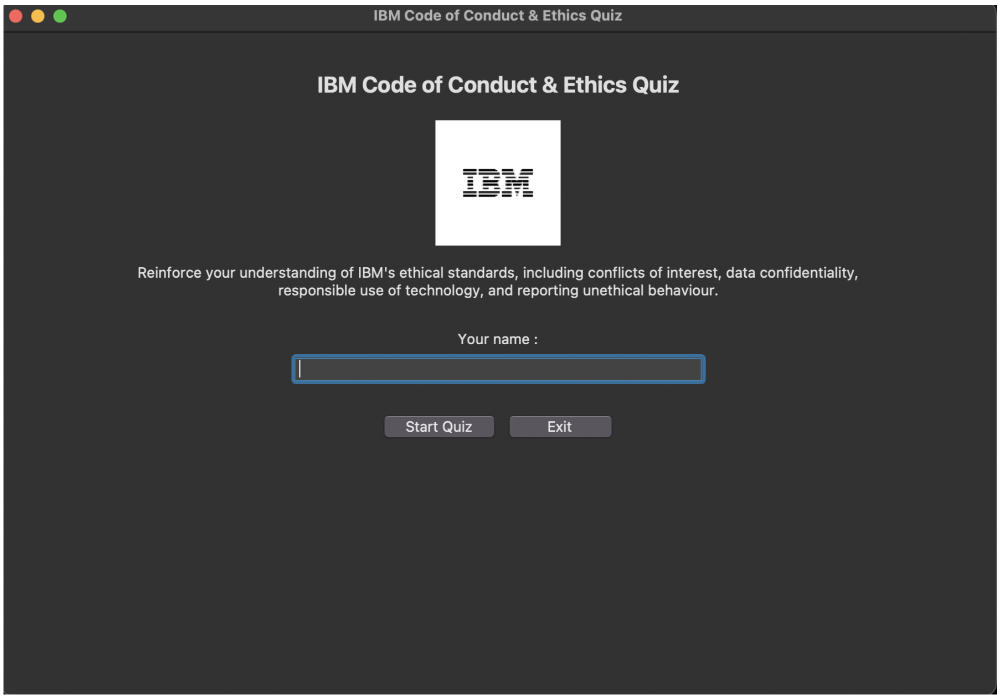
Total Test Cases: 20

Test Categories:

- Welcome Screen: 5 tests
- Question Screen: 7 tests
- Results Screen: 2 tests
- Stored Results Screen: 2 tests
- Data Persistence: 2 tests
- Error Handling: 1 test
- Edge Cases: 1 test

Test Evidence - Screenshots

Welcome Screen

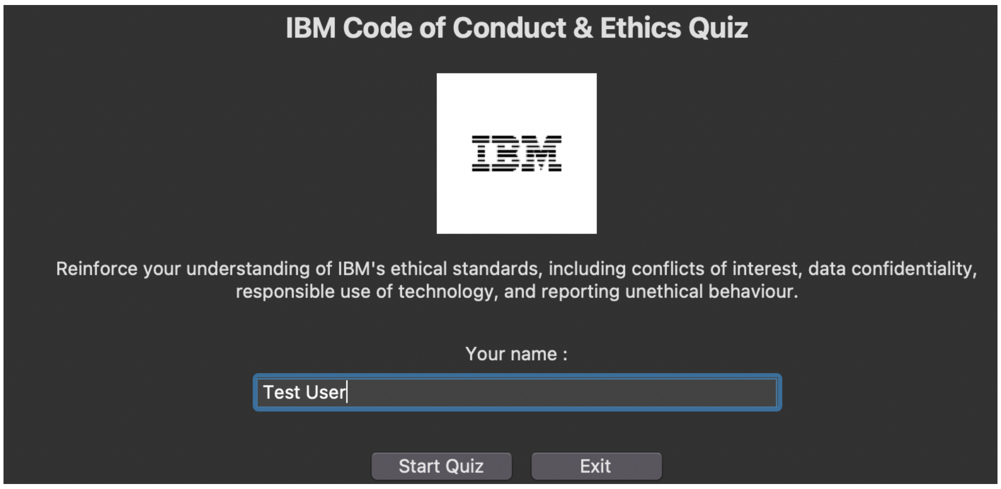


IBM Code of Conduct & Ethics Quiz

Reinforce your understanding of IBM's ethical standards, including conflicts of interest, data confidentiality, responsible use of technology, and reporting unethical behaviour.

Your name :

Test #1

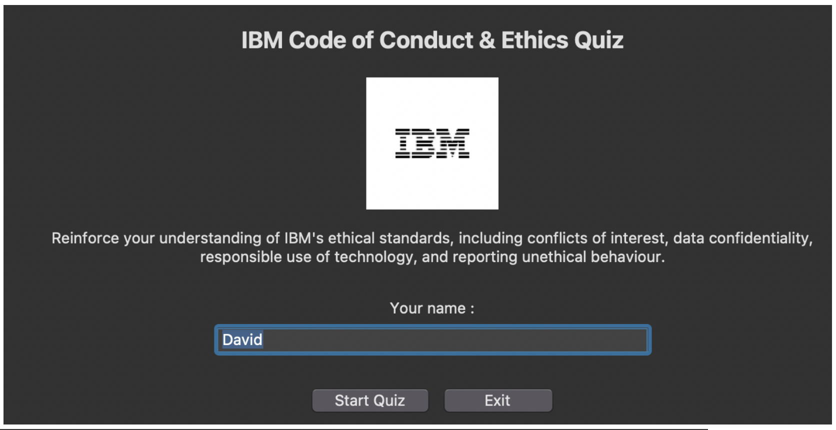


IBM Code of Conduct & Ethics Quiz

Reinforce your understanding of IBM's ethical standards, including conflicts of interest, data confidentiality, responsible use of technology, and reporting unethical behaviour.

Your name : Test User

Test #2

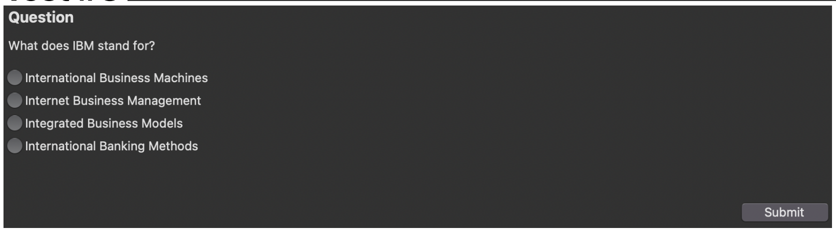


IBM Code of Conduct & Ethics Quiz

Reinforce your understanding of IBM's ethical standards, including conflicts of interest, data confidentiality, responsible use of technology, and reporting unethical behaviour.

Your name : David

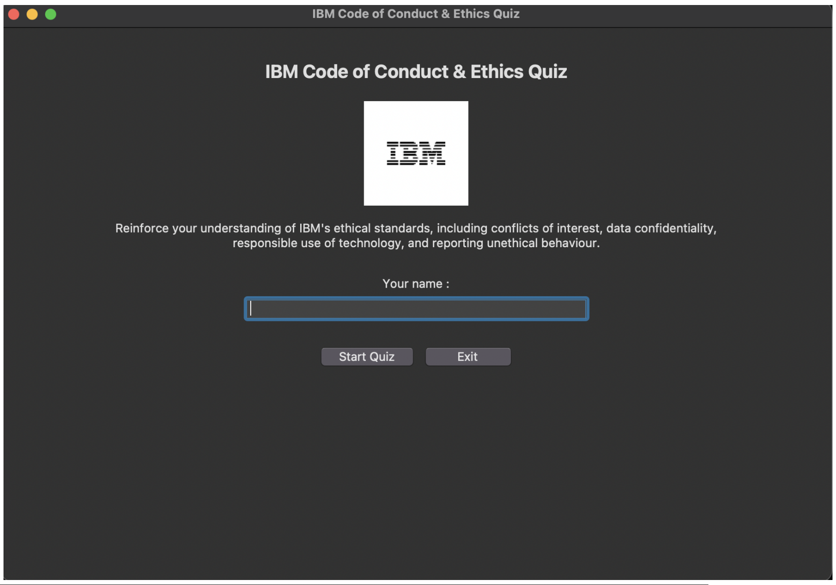
Test #3



Question

What does IBM stand for?

- International Business Machines
- Internet Business Management
- Integrated Business Models
- International Banking Methods

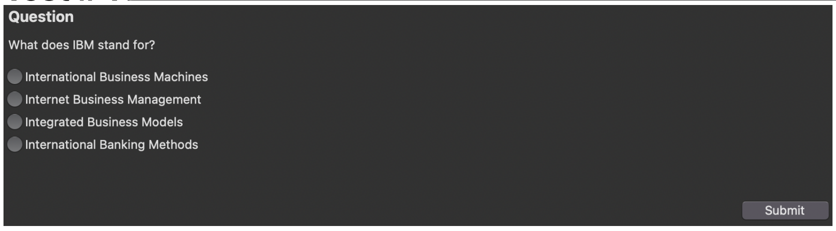

IBM Code of Conduct & Ethics Quiz

IBM Code of Conduct & Ethics Quiz

Reinforce your understanding of IBM's ethical standards, including conflicts of interest, data confidentiality, responsible use of technology, and reporting unethical behaviour.

Your name :

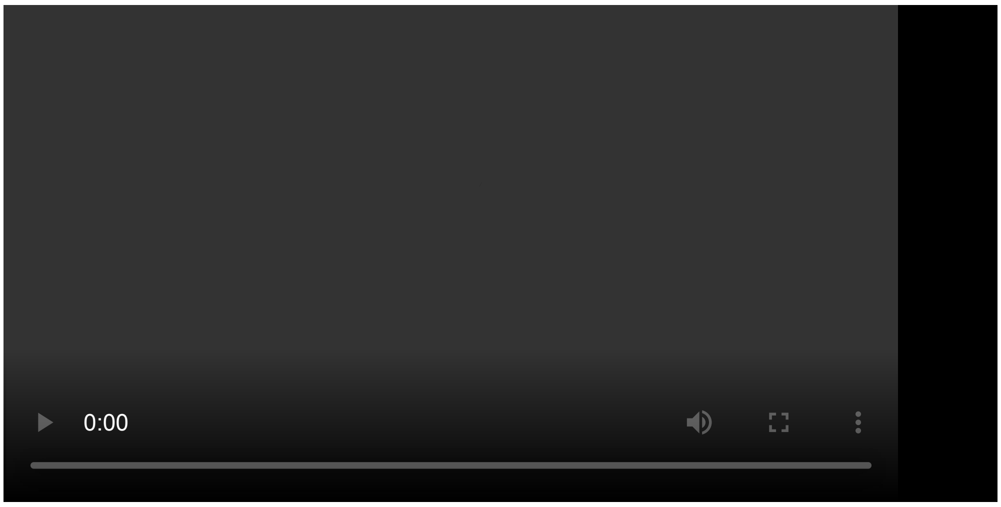
Test #4


Question

What does IBM stand for?

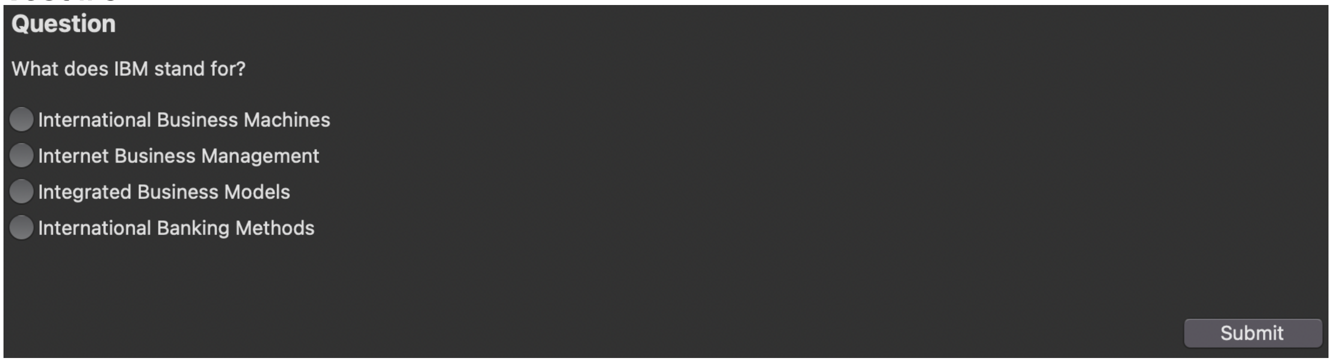
- International Business Machines
- Internet Business Management
- Integrated Business Models
- International Banking Methods

Test #5



Question Screen

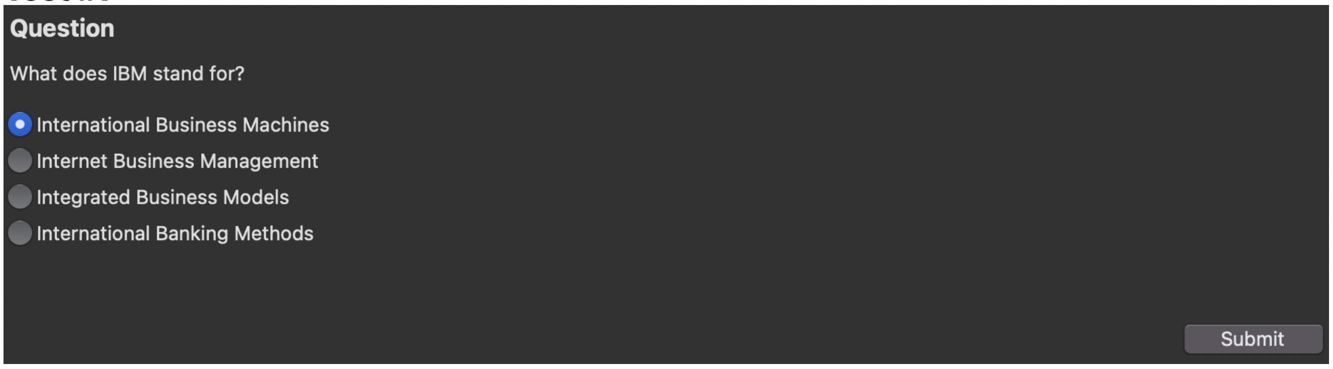
Test #6


Question

What does IBM stand for?

- International Business Machines
- Internet Business Management
- Integrated Business Models
- International Banking Methods

Test #7



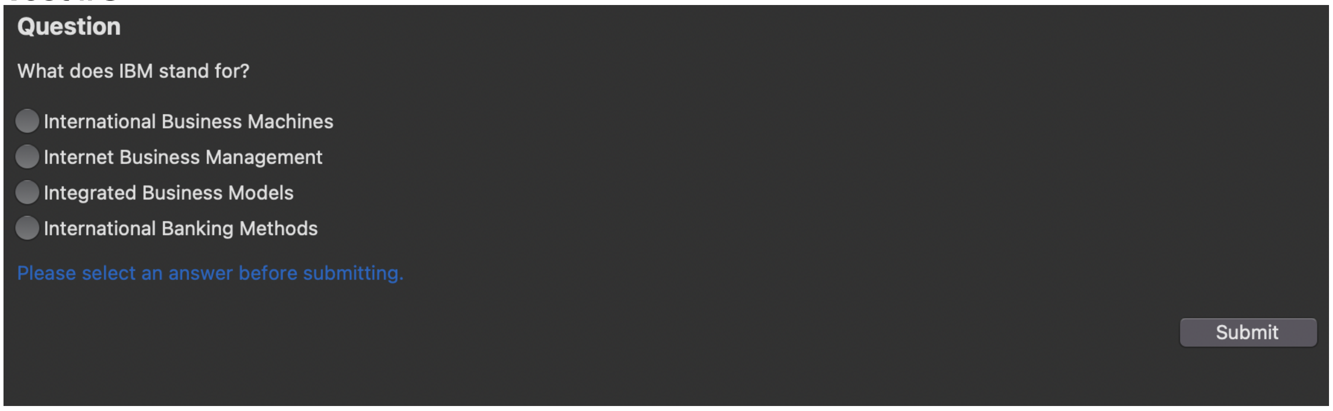
Question

What does IBM stand for?

[✓ International Business Machines]
Internet Business Management
Integrated Business Models
International Banking Methods

Submit

Test #8



Question

What does IBM stand for?

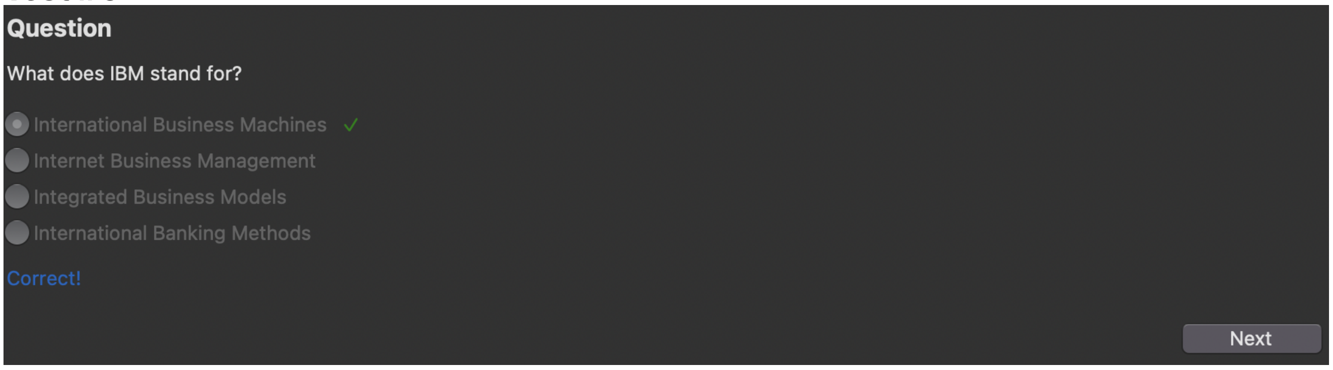
International Business Machines
Internet Business Management
Integrated Business Models
International Banking Methods

Please select an answer before submitting.

Submit

Feedback Mode - Correct Answer

Test #9



Question

What does IBM stand for?

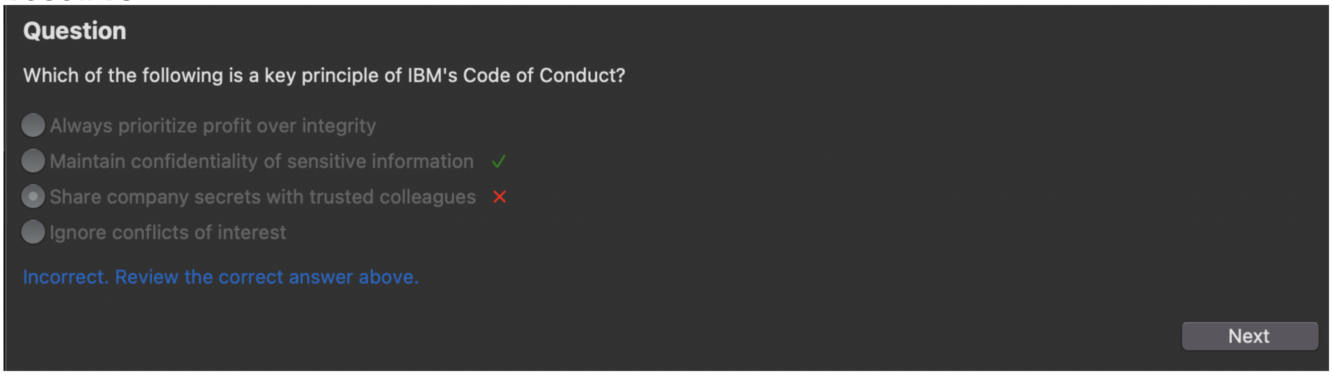
[✓ International Business Machines] ✓
Internet Business Management
Integrated Business Models
International Banking Methods

Correct!

Next

Feedback Mode - Incorrect Answer

Test #10



Question

Which of the following is a key principle of IBM's Code of Conduct?

Always prioritize profit over integrity
Maintain confidentiality of sensitive information ✓
[✓ Share company secrets with trusted colleagues] ✗
Ignore conflicts of interest

Incorrect. Review the correct answer above.

Next

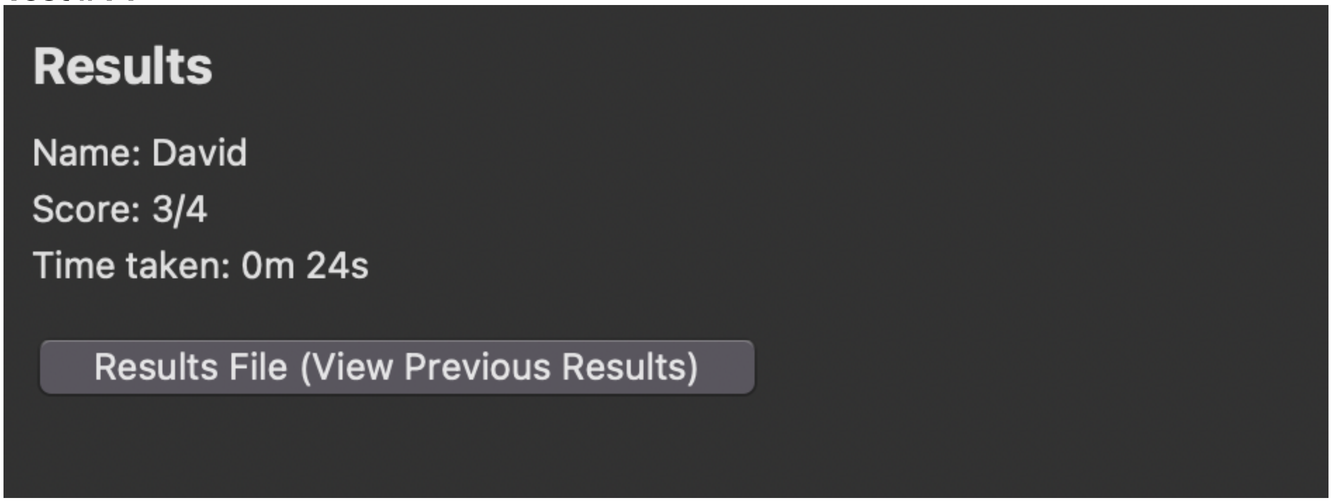
Test #11

Test #12

Test #13

Results Screen

Test #14



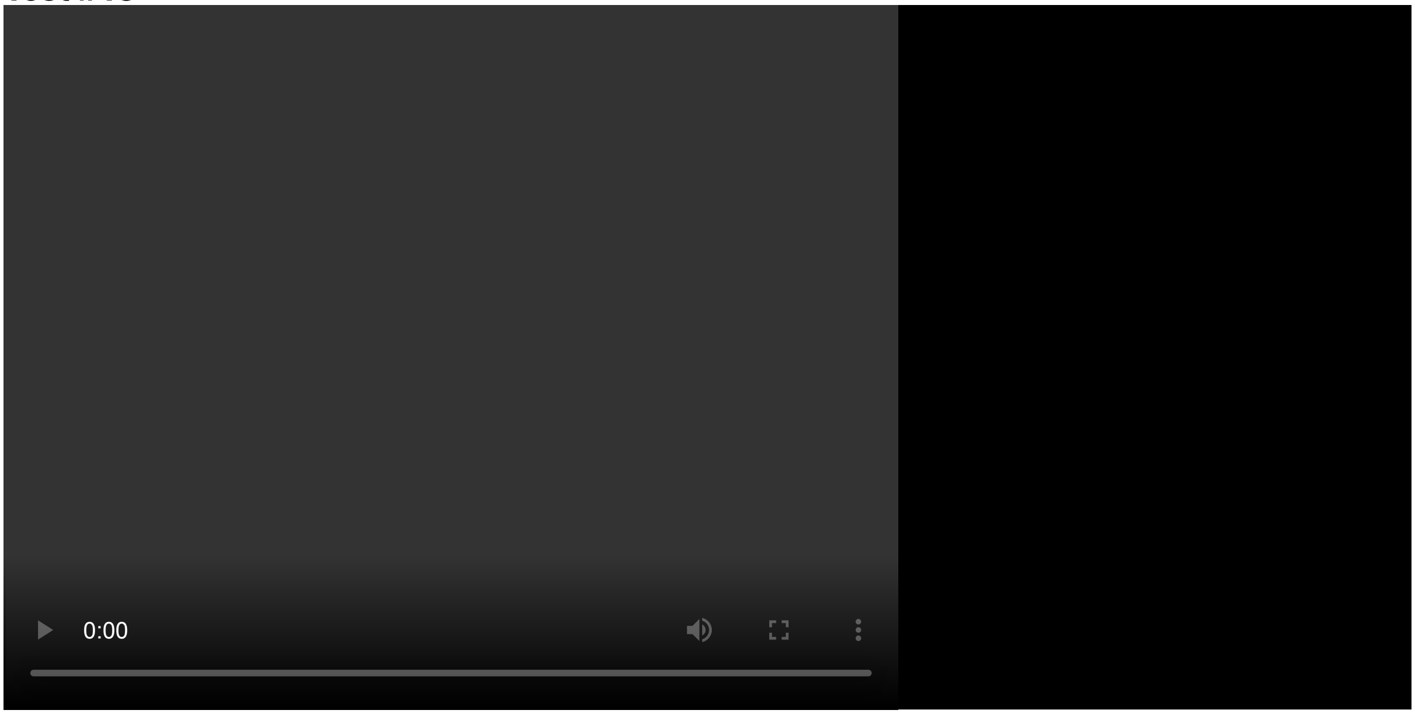
Results

Name: David

Score: 3/4

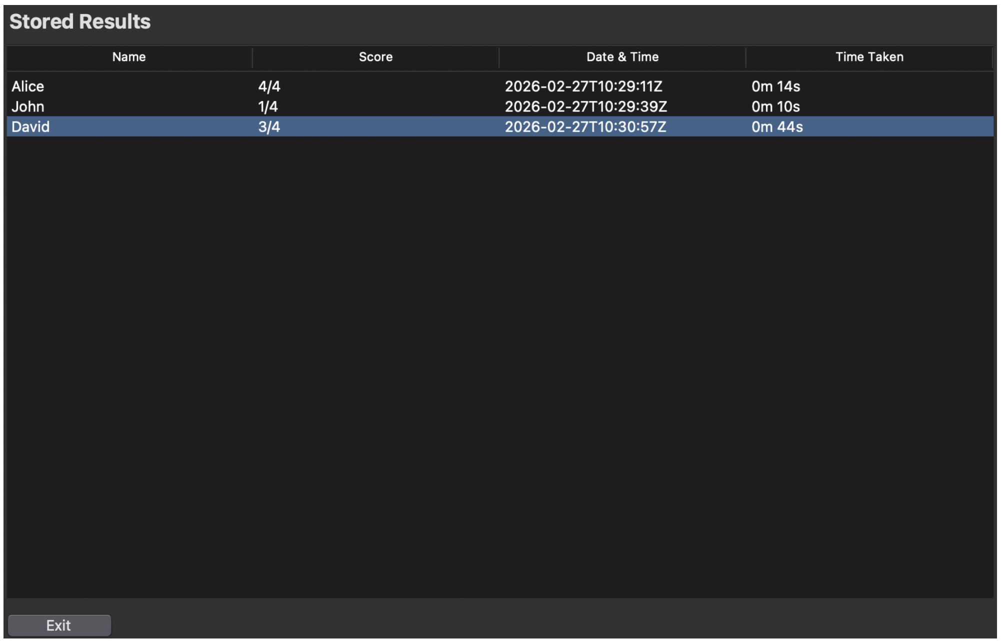
Time taken: 0m 24s

Test #15



Test #14, #15, #16

Stored Results Screen

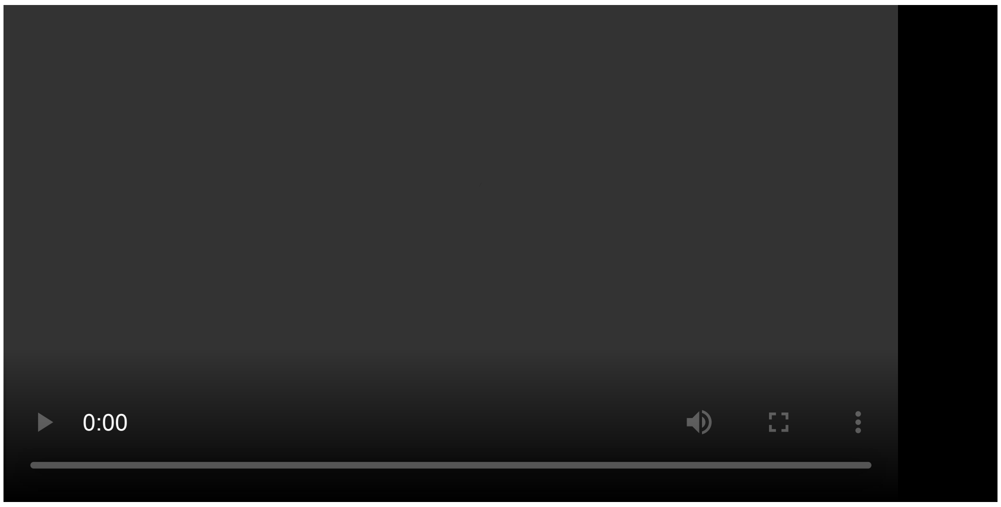


Stored Results

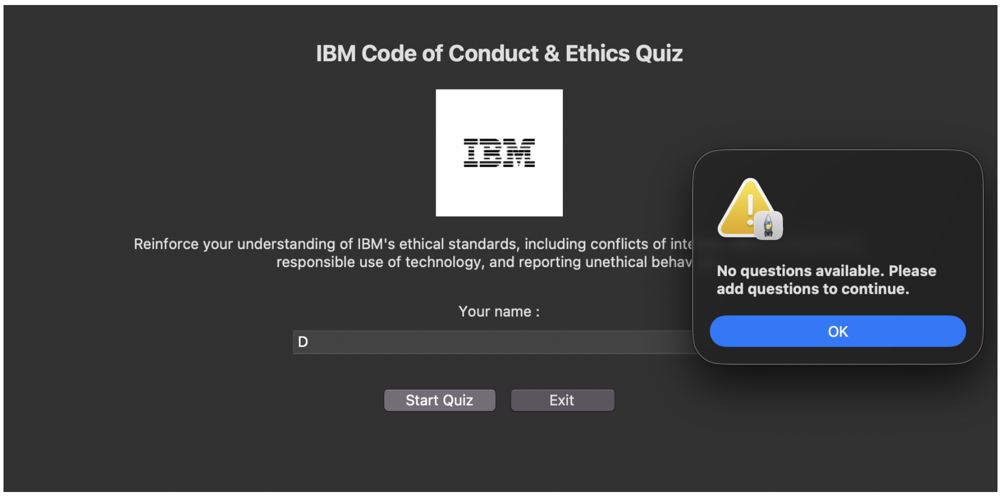
Name	Score	Date & Time	Time Taken
Alice	4/4	2026-02-27T10:29:11Z	0m 14s
John	1/4	2026-02-27T10:29:39Z	0m 10s
David	3/4	2026-02-27T10:30:57Z	0m 44s

Test #16

Test #17



Test #18

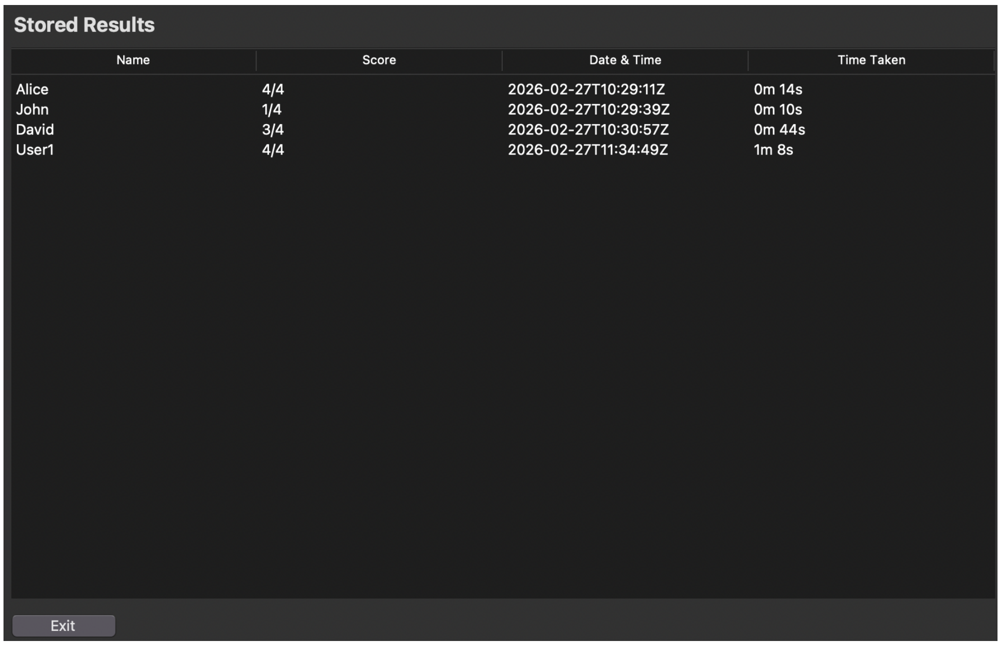

IBM Code of Conduct & Ethics Quiz

Reinforce your understanding of IBM's ethical standards, including conflicts of inte
responsible use of technology, and reporting unethical behav

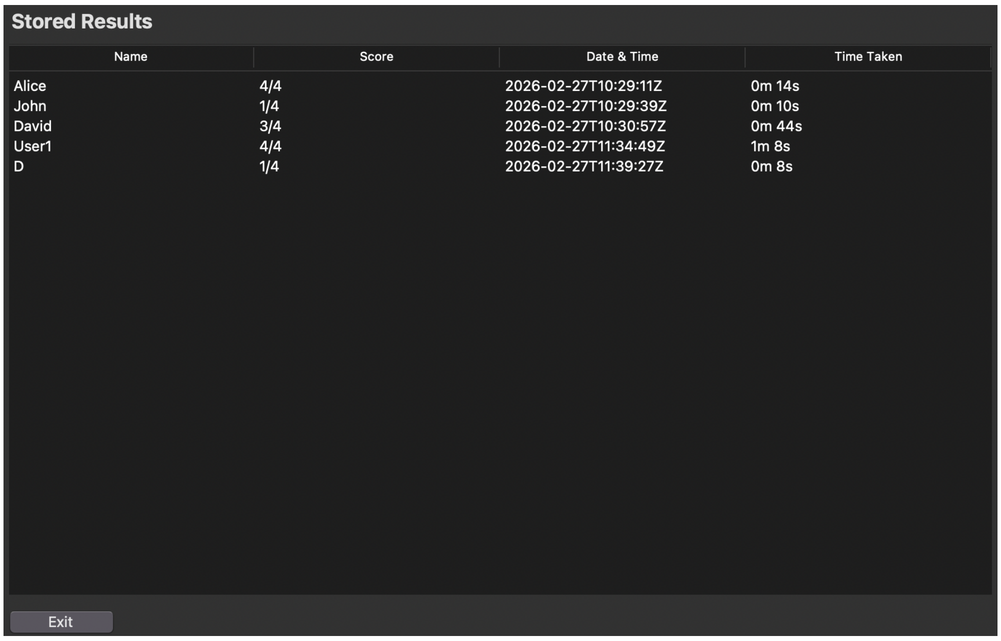
Your name : D

No questions available. Please add questions to continue.

Test #19


Stored Results

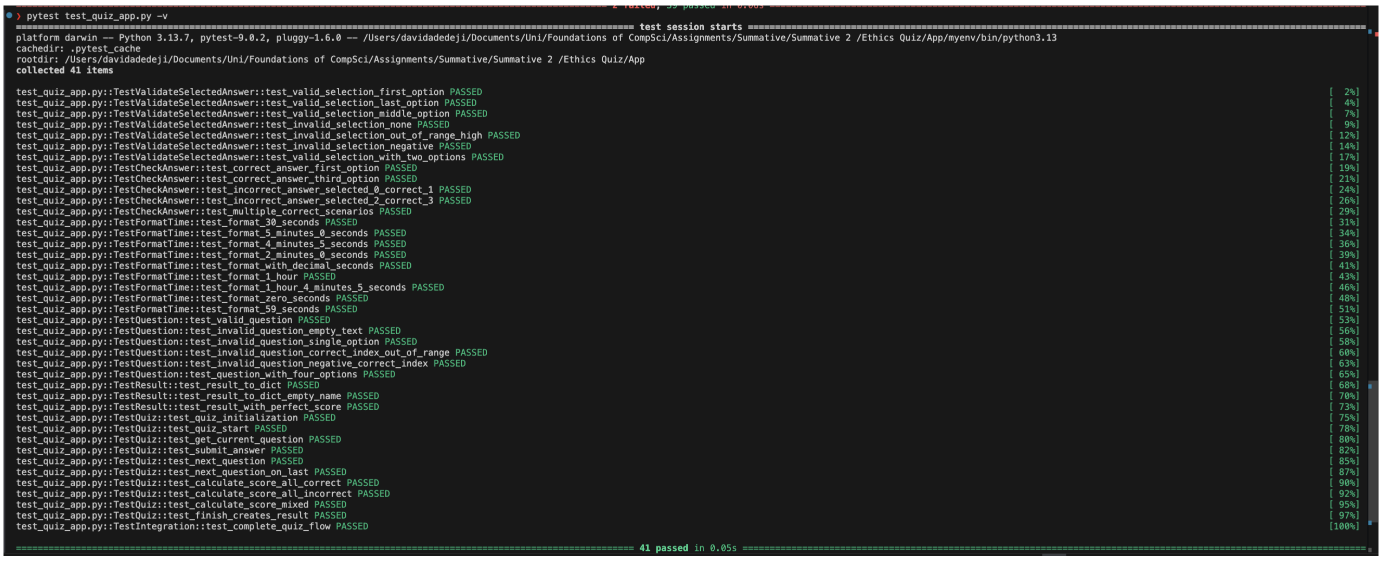
Name	Score	Date & Time	Time Taken
Alice	4/4	2026-02-27T10:29:11Z	0m 14s
John	1/4	2026-02-27T10:29:39Z	0m 10s
David	3/4	2026-02-27T10:30:57Z	0m 44s
User1	4/4	2026-02-27T11:34:49Z	1m 8s



Name	Score	Date & Time	Time Taken
Alice	4/4	2026-02-27T10:29:11Z	0m 14s
John	1/4	2026-02-27T10:29:39Z	0m 10s
David	3/4	2026-02-27T10:30:57Z	0m 44s
User1	4/4	2026-02-27T11:34:49Z	1m 8s
D	1/4	2026-02-27T11:39:27Z	0m 8s

Test #20

Unit test



```
❯ pytest test_quiz_app.py -v
==================================== test session starts ====================================
platform darwin -- Python 3.13.7, pytest-9.0.2, pluggy-1.6.0 -- /Users/davidadedeji/Documents/Uni/Foundations of CompSci/Assignments/Summative/Summative 2 /Ethics Quiz/App/myenv/bin/python3.13
cachedir: .pytest_cache
rootdir: /Users/davidadedeji/Documents/Uni/Foundations of CompSci/Assignments/Summative/Summative 2 /Ethics Quiz/App
collected 41 items

test_quiz_app.py::TestValidateSelectedAnswer::test_valid_selection_first_option PASSED                  [  2%]
test_quiz_app.py::TestValidateSelectedAnswer::test_valid_selection_last_option PASSED                   [  4%]
test_quiz_app.py::TestValidateSelectedAnswer::test_valid_selection_middle_option PASSED                 [  7%]
test_quiz_app.py::TestValidateSelectedAnswer::test_invalid_selection_none PASSED                        [  9%]
test_quiz_app.py::TestValidateSelectedAnswer::test_invalid_selection_out_of_range_high PASSED           [ 12%]
test_quiz_app.py::TestValidateSelectedAnswer::test_invalid_selection_negative PASSED                    [ 14%]
test_quiz_app.py::TestValidateSelectedAnswer::test_valid_selection_with_two_options PASSED              [ 17%]
test_quiz_app.py::TestCheckAnswer::test_correct_answer_first_option PASSED                              [ 19%]
test_quiz_app.py::TestCheckAnswer::test_correct_answer_third_option PASSED                              [ 21%]
test_quiz_app.py::TestCheckAnswer::test_incorrect_answer_selected_0_correct_1 PASSED                    [ 24%]
test_quiz_app.py::TestCheckAnswer::test_incorrect_answer_selected_2_correct_3 PASSED                    [ 26%]
test_quiz_app.py::TestCheckAnswer::test_multiple_correct_scenarios PASSED                               [ 29%]
test_quiz_app.py::TestFormatTime::test_format_30_seconds PASSED                                         [ 31%]
test_quiz_app.py::TestFormatTime::test_format_5_minutes_0_seconds PASSED                                [ 34%]
test_quiz_app.py::TestFormatTime::test_format_4_minutes_5_seconds PASSED                                [ 36%]
test_quiz_app.py::TestFormatTime::test_format_2_minutes_0_seconds PASSED                                [ 39%]
test_quiz_app.py::TestFormatTime::test_format_with_decimal_seconds PASSED                               [ 41%]
test_quiz_app.py::TestFormatTime::test_format_1_hour PASSED                                             [ 43%]
test_quiz_app.py::TestFormatTime::test_format_1_hour_4_minutes_5_seconds PASSED                         [ 46%]
test_quiz_app.py::TestFormatTime::test_format_zero_seconds PASSED                                       [ 48%]
test_quiz_app.py::TestFormatTime::test_format_59_seconds PASSED                                         [ 51%]
test_quiz_app.py::TestQuestion::test_valid_question PASSED                                              [ 53%]
test_quiz_app.py::TestQuestion::test_invalid_question_empty_text PASSED                                 [ 56%]
test_quiz_app.py::TestQuestion::test_invalid_question_single_option PASSED                              [ 58%]
test_quiz_app.py::TestQuestion::test_invalid_question_correct_index_out_of_range PASSED                 [ 60%]
test_quiz_app.py::TestQuestion::test_invalid_question_negative_correct_index PASSED                     [ 63%]
test_quiz_app.py::TestQuestion::test_question_with_four_options PASSED                                  [ 65%]
test_quiz_app.py::TestResult::test_result_to_dict PASSED                                                [ 68%]
test_quiz_app.py::TestResult::test_result_to_dict_empty_name PASSED                                     [ 70%]
test_quiz_app.py::TestResult::test_result_with_perfect_score PASSED                                     [ 73%]
test_quiz_app.py::TestQuiz::test_quiz_initialization PASSED                                             [ 75%]
test_quiz_app.py::TestQuiz::test_quiz_start PASSED                                                      [ 78%]
test_quiz_app.py::TestQuiz::test_get_current_question PASSED                                            [ 80%]
test_quiz_app.py::TestQuiz::test_submit_answer PASSED                                                   [ 82%]
test_quiz_app.py::TestQuiz::test_next_question PASSED                                                   [ 85%]
test_quiz_app.py::TestQuiz::test_next_question_on_last PASSED                                           [ 87%]
test_quiz_app.py::TestQuiz::test_calculate_score_all_correct PASSED                                     [ 90%]
test_quiz_app.py::TestQuiz::test_calculate_score_all_incorrect PASSED                                   [ 92%]
test_quiz_app.py::TestQuiz::test_calculate_score_mixed PASSED                                           [ 95%]
test_quiz_app.py::TestQuiz::test_finish_creates_result PASSED                                           [ 97%]
test_quiz_app.py::TestIntegration::test_complete_quiz_flow PASSED                                       [100%]

==================================== 41 passed in 0.05s ====================================
```

User Documentation - IBM Code of Conduct & Ethics Quiz

Overview

The IBM Code of Conduct & Ethics Quiz is an interactive application designed to help IBM staff understand and reinforce IBM's ethical standards. The quiz covers important topics including conflicts of interest, data security, responsible technology use, and reporting unethical behavior.

Getting Started

System Requirements

- **Operating System:** macOS, Windows, or Linux
- **Python:** Version 3.13 or higher
- **No additional software needed** - the application runs standalone on your computer

Running the Application

1. **Open Terminal/Command Prompt**
2. **Navigate to the application folder:**

```
cd /path/to/Ethics\ Quiz/App
```

3. **Run the application:**

```
python3 main.py
```

The application window should open automatically.

Using the Quiz

Step 1: Welcome Screen

When you launch the application, you'll see the Welcome Screen with:

- IBM logo
- Quiz description
- **Your Name** input field
- **Start Quiz** button
- **Exit** button

What to do:

1. Enter your name (optional - you can leave blank and it will show as "Anonymous")
2. Click **Start Quiz** to begin

Step 2: Question Screen

The Question Screen shows:

- The current question
- **4 multiple choice options** as radio buttons
- **Submit** button
- Status message area (if needed)

How to answer:

1. **Select one option** by clicking the radio button next to it
2. **Click Submit** to submit your answer
3. If you forgot to select an answer, you'll see an error message: "Please select an answer before submitting."

Example:

```
Question: What does IBM stand for?

○ Internet Business Management
○ Integrated Business Models
● International Business Machines  ← Selected
○ International Banking Methods

[Submit]
```

Step 3: Feedback

After clicking Submit, the screen enters **Feedback Mode**:

- **Green ✓** appears next to the correct answer
- **Red ✕** appears next to your selection (if wrong)
- **Status message** tells you: "Correct!" or "Incorrect. Review the correct answer above."
- **Radio buttons are locked** - you cannot change your selection
- **Submit button disappears** and **Next button appears**

What to do:

1. Review the feedback
2. Click **Next** to proceed to the next question

Step 4: Complete the Quiz

Repeat Steps 2-3 for all 4 questions in the quiz. After your last answer, you'll see the Results Screen.

Results Screen

After completing all questions, you'll see:

- **Your Name:** The name you entered (or "Anonymous")
- **Final Score:** e.g., "3/4" (3 correct out of 4 questions)
- **Time Taken:** e.g., "4m 30s" (minutes and seconds)
- **View Previous Results** button
- **Exit** button

What to do:

1. Review your score
2. Click **View Previous Results** to see all past attempts
3. Click **Exit** to close the application

Stored Results Screen

Shows a table of all your quiz attempts with:

- **Name** - Person who took the quiz
- **Score** - Number correct (e.g., 3/4)
- **Date & Time** - When the quiz was taken
- **Time Taken** - How long the quiz took (e.g., 4m 30s)

What to do:

1. Review your quiz history
2. Click **Exit** to close the application

Tips for Success

Before Taking the Quiz

- Find a quiet place free from distractions
- Think carefully about each answer

During the Quiz

- Read each question carefully
- Consider all 4 options before selecting
- If unsure, think about IBM's values and the Code of Conduct
- You can take as much time as you need per question

After the Quiz

- Review your score
- If you scored low, consider taking the quiz again to reinforce learning
- Your results are automatically saved for record-keeping

Troubleshooting

Problem: Application won't start

Solution:

- Ensure Python 3.13+ is installed
- Try running from Terminal: `python3 main.py`
- Check that you're in the correct directory

Problem: "No questions available" message appears

Solution:

- This means the quiz questions file is missing
- Contact your IT administrator to ensure questions.csv is in the `app/csv_files/` folder

Problem: Results aren't saving

Solution:

- Check that you have write permission to the `app/csv_files/` folder
- Ensure there is enough disk space
- Contact IT if the problem persists

Problem: Questions/answers don't display correctly

Solution:

- Try closing and reopening the application
- Clear the browser cache if using web version
- Restart your computer if the issue persists

Data Privacy

- Your quiz results are saved locally on your computer
- Results include: name, score, time taken, and timestamp
- Results are stored in CSV format in the `app/csv_files/results.csv` file

Frequently Asked Questions

Q: Can I retake the quiz? A: Yes! Return to the Welcome Screen after finishing and enter your name again to retake the quiz. Previous results are preserved.

Q: How many questions are in the quiz? A: There are 4 questions covering IBM's Code of Conduct and Ethics.

Q: Is there a time limit? A: No, you can take as long as you need per question. Total time is tracked.

Q: What happens if I get a question wrong? A: The screen shows you the correct answer in green (✓) and your incorrect selection in red (✕). You can then learn from the feedback.

Q: Can I skip a question? A: No, you must answer every question before proceeding.

Q: What's a passing score? A: There's no official pass/fail - the quiz is for learning and reinforcement. Aim for 3/4 or better.

Q: Where are my results stored? A: Results are saved in the application's results.csv file and displayed in the "Stored Results" screen.

Q: Can I delete my results? A: Results are permanent record-keeping. Contact your administrator if you need results removed.

Next Steps

After completing the quiz:

1. **Review your score** on the Results Screen
2. **Study any weak areas** where you scored incorrectly
3. **Retake the quiz** to reinforce learning
4. **Keep your results** as proof of completion

Thank you for taking the IBM Code of Conduct & Ethics Quiz!

Technical Documentation - IBM Code of Conduct & Ethics Quiz

Overview for Developers

This document explains the code structure, how to run tests, and how to maintain the application.

Code Architecture

1. **MVC Pattern (Model-View-Controller)**

The application uses MVC architecture:

Model: Data layer

- `models.py` - Question and Result dataclasses
- `repository.py` - CSV data access

View: UI layer

- `welcome_screen.py` - Welcome interface
- `question_screen.py` - Quiz interface
- `results_screen.py` - Results interface
- `stored_results_screen.py` - History interface

Controller: Business logic layer

- `main.py` - Application orchestration
- `quiz.py` - Quiz logic and state

---

2. **Key Classes**

Question (models.py)

```
@dataclass(frozen=True)
class Question:
    id: str                     # Unique identifier
    text: str                   # Question text
    options: List[str]          # Answer options
    correct_index: int          # Index of correct answer

    def is_valid(self) -> bool:
        # Validates question has text, ≥2 options, valid correct_index
```

Result (models.py)

```
@dataclass(frozen=True)
class Result:
    user_name: str            # User's name
    score: int                # Questions answered correctly
    total_questions: int      # Total questions
    time_taken: float         # Time in seconds
    timestamp: str            # ISO 8601 timestamp

    def to_dict(self) -> dict:
        # Converts to dictionary for CSV saving
```

Quiz (quiz.py)

```
class Quiz:
    def __init__(self, questions: List[Question])
    def start() -> None               # Begin quiz, record time
    def get_current_question()       # Get Question at current_index
    def submit_answer(idx: int)      # Store user's answer
    def next_question()      # Advance to next question
    def calculate_score()    # Count correct answers
    def finish(user_name: str)   # End quiz, create Result
```

Running Tests Locally

Installation

```
# Install pytest (testing framework)
pip install pytest

# Or with pip3
pip3 install pytest
```

Running Tests

From the app directory:

```
# Run all 41 tests
pytest test_quiz_app.py -v

# Run specific test class
pytest test_quiz_app.py::TestValidateSelectedAnswer -v

# Run specific test
pytest test_quiz_app.py::TestQuestion::test_valid_question -v

# Run with coverage report
pip install pytest-cov
pytest test_quiz_app.py --cov=logic --cov-report=html
```

Test Structure

41 total unit tests:

1. **TestValidateSelectedAnswer** (7 tests)
   - Tests answer validation logic
   - Checks valid/invalid selections
2. **TestCheckAnswer** (5 tests)
   - Tests if answers are correct
   - Compares user selection vs correct answer
3. **TestFormatTime** (9 tests)
   - Tests time formatting (30s → "00:30", 245s → "04:05")
   - Handles seconds, minutes, hours
4. **TestQuestion** (6 tests)
   - Tests Question.is_valid()
   - Validates question requirements
5. **TestResult** (3 tests)
   - Tests Result.to_dict()
   - Tests CSV conversion
6. **TestQuiz** (10 tests)
   - Tests quiz initialization
   - Tests state management (start, submit, next)
   - Tests score calculation
   - Tests result creation
7. **TestIntegration** (1 test)
   - Tests complete quiz flow end-to-end

Understanding the Code Flow

Application Startup (main.py)

```python
class App(tk.Tk):
    def __init__(self):
        # 1. Initialize Tkinter window
        # 2. Create container frame
        # 3. Load CSV repository
        # 4. Show welcome screen
        self.show_welcome()
```

Taking a Quiz

```
1. User enters name and clicks "Start Quiz"
    calls start_quiz_flow(name)

2. Questions loaded from CSV
    repo.load_questions()

3. Quiz object created and started
    Quiz(questions).start()

4. First question displayed
```

```
    show_question()

5. User selects answer and clicks Submit
    on_submit() → _submit_answer_and_feedback()

6. Feedback displayed (✓ or ✗)
    show_feedback_mode()

7. User clicks Next
    _go_next_or_finish()

8. Repeat 4–7 until last question

9. Quiz finished, Result created
    quiz.finish(user_name)

10. Result saved to CSV
     repo.append_result(result)

11. Results screen displayed
     show_results(result)
```

CSV File Format

questions.csv

```
id,text,choices,correct_index
q001,What does IBM stand for?,International Business Machines||Internet Business Management||Integrated Business Models||International Banking Methods,0
q002,Question 2 text?,Option A||Option B||Option C||Option D,2
```

Fields:

- `id` - Unique question identifier
- `text` - Question text displayed to user
- `choices` - Options separated by `||` (pipe separator)
- `correct_index` - Index of correct answer (0-based)

results.csv

```
user_name,score,total_questions,time_taken,timestamp
David,8,10,245.0,2026-02-20T14:30:00Z
Alice,10,10,120.5,2026-02-20T13:45:30Z
```

Fields:

- `user_name` - Name entered by user
- `score` - Number of correct answers
- `total_questions` - Total questions in quiz
- `time_taken` - Time in seconds (float)
- `timestamp` - ISO 8601 format timestamp

Important Functions

validate.py

```
def validate_selected_answer(idx: int | None, num_options: int) -> bool:
    # Returns True if idx is valid selection (0 ≤ idx < num_options)

def check_answer(selected: int, correct: int) -> bool:
    # Returns True if selected == correct

def format_time(seconds: float) -> str:
    # Converts seconds to "MM:SS" or "HH:MM:SS"
    # Example: 245 → "04:05", 3845 → "01:04:05"
```

repository.py

```
class CSVRepository:
    def load_questions() -> List[Question]:
        # Reads questions.csv, returns validated questions

    def append_result(result: Result) -> None:
        # Appends result to results.csv (creates file if needed)

    def load_results() -> List[Result]:
        # Reads results.csv, returns all saved results
```

Potential Development Tasks

Adding a New Question

1. **Edit `app/csv_files/questions.csv`**
2. **Add row:**

```
q006,New question text?,Option 1||Option 2||Option 3||Option 4,2
```

3. **Run tests to ensure question is valid:**

```
pytest test_quiz_app.py::TestQuestion -v
```

Modifying Answer Validation

1. **Edit `app/logic/validate.py`**
2. **Update `validate_selected_answer()`**
3. **Run tests:**

```
pytest test_quiz_app.py::TestValidateSelectedAnswer -v
```

Changing Quiz Flow

1. **Edit `app/main.py`**
2. **Modify methods in `App` class**
3. **Run full test suite:**

```
pytest test_quiz_app.py -v
```

Debugging

Print debug info:

```
print(f"Debug: current_index = {self.quiz.current_index}")
print(f"Debug: user_answers = {self.quiz.user_answers}")
```

Run specific test with output:

```
pytest test_quiz_app.py::TestQuiz::test_calculate_score_all_correct -v -s
```

Performance Considerations

- **Quiz loading:** Questions loaded once at startup ~50ms
- **Answer validation:** Instant <1ms
- **Result saving:** CSV write operation ~10ms
- **Tests:** All 41 tests run in <100ms

Security Considerations

- **Data storage:** Results stored in local CSV (no encryption)
- **Input validation:** All user answers validated before processing
- **File permissions:** Ensure results.csv is writable
- **No network:** Application runs entirely locally, no data transmission

---

Maintenance & Troubleshooting

Tests Failing

1. **Check Python version:** `python3 --version` (need 3.13+)
2. **Reinstall pytest:** `pip install pytest`
3. **Check imports:** Ensure all modules can be imported
4. **Run single test:** `pytest test_quiz_app.py::TestValidateSelectedAnswer::test_valid_selection_first_option -v`

Application Crashing

1. **Check file paths:** Ensure csv_files/ folder exists with questions.csv
2. **Check permissions:** Ensure write access to csv_files/ folder
3. **Check imports:** Run `python3 -c "from logic.models import Question"` to test imports
4. **Run from correct directory:** Must run from `app/` folder

Results Not Saving

1. **Check disk space:** Ensure enough free disk space
2. **Check permissions:** `ls -la app/csv_files/` to verify write access
3. **Check file format:** Ensure results.csv has correct headers
4. **Manually test:** Run `python3` and execute `repo.append_result(test_result)`

Next Steps for Development

- Add more questions to questions.csv

- Implement database storage instead of CSV
- Add user login/authentication
- Create admin dashboard for results review
- Add progress tracking across multiple quizzes
- Implement scoring thresholds and certificates

Code Quality Standards

- **Type hints:** All functions use type hints
- **Docstrings:** All classes/functions documented
- **Test coverage:** 41 tests cover core logic
- **PEP 8:** Follow Python style guide
- **Immutable data:** Use frozen dataclasses for data integrity

Evaluation Section

Project Reflection & Assessment

This section reflects on the development journey of the IBM Code of Conduct & Ethics Quiz application, pointing out what went well and what could have been improved.

What Went Well

1. **Successful MVC Architecture Implementation**

One of my strongest achievements was implementing a clean Model-View-Controller (MVC) architecture. The separation of concerns between:

- **Models** (data layer with frozen dataclasses)
- **Views** (GUI screens)
- **Controllers** (business logic and orchestration)

This made the codebase maintainable and testable. Each component had a single responsibility, making it easy to modify one part without breaking others. The architecture decision proved especially valuable when adding unit tests later in development.

2. **Comprehensive Unit Test Coverage**

I successfully created 41 unit tests covering:

- Validation functions (10 tests)
- Data models (9 tests)
- Quiz logic (10 tests)
- Integration tests (1 test)

All tests pass, demonstrating that the core business logic is reliable. The tests serve as both verification of correctness and executable documentation of expected behavior. This is a strong foundation for future maintenance and feature additions.

3. **Robust Data Validation**

The application implements thorough validation at multiple levels:

- **Question validation:** Checks that questions have text, ≥2 options, and valid correct_index
- **Answer validation:** Prevents out-of-range selections and empty submissions
- **Result validation:** Ensures all required fields are present

This defensive programming approach prevents silent failures and data corruption.

4. **Clean Code & Documentation**

I maintained high code quality through:

- Type hints on all functions
- Comprehensive docstrings on classes and methods
- Clear variable naming
- Logical code organization

The documentation makes the codebase accessible to other developers and easier to maintain long-term.

5. **User-Friendly Quiz Experience**

The quiz flow is intuitive:

- Clear feedback (✓ for correct, ✕ for incorrect)
- Locked inputs in feedback mode prevent accidental changes
- Time tracking with clean formatting (4m 30s instead of 270.5s)
- Historical results stored for record-keeping

Users can complete the quiz without confusion, and the application is forgiving (e.g., optional name entry).

6. **Data Persistence**

The CSV-based repository implementation successfully:

- Loads questions from structured data
- Saves results with automatic file creation
- Preserves data across sessions
- Handles missing files gracefully with error messages

This simple but effective approach avoids the complexity of a database while meeting all requirements.

What Could Have Been Improved

1. **Design Implementation Challenges - Figma to Code**

This was my most significant struggle during development. I created detailed designs in Figma with specific styling intentions, but translating them to Tkinter proved challenging.

The Problem:

When I designed the application in Figma, I envisioned:

- **Gold color** for the background, buttons and highlights
- **Subtle borders** on input fields and containers with rounded corners
- **Custom fonts and sizes** for hierarchy
- **Color consistency** across all screens

However, when building with Tkinter, I encountered limitations:

- **Limited styling options:** Tkinter's `ttk` (themed Tkinter) module doesn't support custom colors easily. Buttons, inputs, and labels have restricted color palettes
- **Design complexity:** Achieving the Figma design would have required either:
  - Building custom widgets from scratch (very time-consuming)
  - Using a different framework like PyQt or web-based UI

What I Did:

I made pragmatic trade-offs:

- Used the system theme colors (gray) that Tkinter had
- Focused on clean layout and clear hierarchy through spacing
- Prioritized functionality and user experience over perfect designs

- Ensured the UI was professional and usable, even if not matching Figma exactly

What I Learned:

This experience taught me that **design tools and implementation frameworks need to be aligned from the start**. In future projects, I will:

- Choose the implementation framework first
- Design within the constraints of that framework
- Use design tools to match the target platform's capabilities
- Document design system limitations early
- Consider prototyping in the actual framework rather than design tools alone, this is where I mainly fell short

2. **Testing Challenges with Time-Based Assertions**

The unit tests for time calculation (`test_finish_creates_result` and `test_complete_quiz_flow`) initially failed because:

- Unit tests run extremely fast (< 100ms total)
- The time elapsed between quiz start and finish rounds to 0.0 seconds
- Original assertion `assert result.time_taken > 0` was too strict

Resolution: Changed to `assert result.time_taken >= 0` to allow for 0 time in fast-running tests, while still validating that time is recorded in real usage. Time-dependent tests need flexible assertions or deliberate delays to properly test timing logic.

3. **CSV Path Configuration**

During development, I struggled with relative vs. absolute file paths:

- Initially used `"csv_files/questions.csv"` which only worked when running from specific directories
- Later corrected to `"app/csv_files/questions.csv"` for reliability
- Had to debug import issues multiple times

What I'd do differently: Use `os.path` utilities to construct paths relative to the module location, making the code more robust to different execution contexts.

4. **Limited Error Handling for Edge Cases**

While the application handles common errors well, some edge cases could be handled better:

- If a question in CSV has invalid format
- If results.csv becomes corrupted, the app may not recover well
- No option for users to clear their results history

Improvements for next version:

- Add logging system to track errors and data issues
- Implement result export/backup functionality
- Add admin features for result management

5. **No Database - CSV Limitations**

Using CSV files works for the current scale (< 100 results) but has limitations:

- No indexing or efficient searching
- Concurrent access could cause issues
- Difficult to filter or analyze results
- Not scalable for enterprise use

6. **Missing Features**

Time constraints and scope limitations meant some useful features weren't implemented:

- **Progress indicator:** "Question 2 of 4" would help user orientation

- **Retake confirmation:** No warning if user exits during a quiz
- **Results analytics:** No summary of performance by topic
- **Accessibility:** No keyboard-only navigation or screen reader support

These would enhance the user experience and make the application more inclusive.

Lessons Learned

1. **Design-Implementation Alignment is Critical**

I learned that choosing the implementation platform early is crucial. Spending time designing in Figma without considering Tkinter's constraints led to rework. In future projects, I will involve prototyping in the actual framework.

2. **Robust Programming is crucial**

The extensive validation in the code prevented silent failures. When CSV files were missing or malformed, the application handled it well rather than crashing.

3. **Testing Forces Better Code**

Writing unit tests revealed issues in my logic that manual testing missed. The 41 tests caught edge cases like time rounding that wouldn't be obvious in normal use.

4. **Simple Solutions Often Work Best**

I was tempted to use complex frameworks or databases, but the simple CSV + Tkinter solution met all requirements cleanly.

5. **Documentation is Development**

Writing comprehensive docstrings and documentation forced me to think clearly about design decisions. It also made debugging easier when reviewing the code.

Conclusion

The IBM Code of Conduct & Ethics Quiz is a **solid, functional application that successfully meets its core requirements. The architecture is clean, the tests are comprehensive, and the user experience is simple and straightforward.

Main Strengths:

- Well-tested code with 41 passing tests
- Clear MVC architecture
- Comprehensive documentation
- Intuitive user interface
- Robust data validation

Main Weaknesses:

- Design didn't fully translate from Figma to Tkinter
- CSV-based storage limits scalability
- Missing some enhancement features due to scope
- Could benefit from logging and error handling improvements

References & Resources

- Tkinter Official Documentation
- Python Type Hints (PEP 484)
- Pytest Testing Framework
- Python Dataclasses (PEP 557)
- Model-View-Controller Pattern

- CSV File Format (RFC 4180)
- PEP 8 Style Guide for Python Code