High-level Design:

The system is built upon a Flask application, a lightweight web framework in Python. This application serves as the backbone of the system, orchestrating the interaction between the frontend and the backend components.

On the backend side, Python files contain the logic necessary for generating test cases. These files interact with the OpenAI API, a powerful tool for natural language processing and generation. When a user inputs a description or other relevant information through the frontend, the Flask application routes this data to the appropriate Python file. This file then processes the input, makes the necessary requests to the OpenAI API, and receives the generated test cases as a response. The backend logic further refines this data before sending it back to the frontend.

Meanwhile, on the frontend, HTML pages provide the user interface for interacting with the system. These pages display the generated test cases and offer forms or input fields where users can input descriptions or other necessary information. When a user submits their input, the Flask application ensures that it is properly handled by the backend logic, and then displays the results back to the user through the HTML pages.

In summary, the Flask application acts as the intermediary between the frontend and the backend, enabling seamless communication and interaction. This architecture allows for the creation of a web application where users can easily input descriptions, generate test cases using the OpenAI API, and view the results through a user-friendly interface.

Low-level Design:
For implementing our project, which involves building a web-based system for automated test case generation, the Behavioral design pattern family would be helpful. This family is suited for managing complex interactions between the frontend and backend. Specifically:

- Observer Pattern: Useful for updating the frontend in real-time when new test cases are generated or when there's a change in the backend, ensuring the UI is always up-to-date without manual refreshes.
- Command Pattern: Helps in queuing requests to the OpenAI API, especially useful if there are rate limits or operations need to be managed asynchronously.
- Strategy Pattern: Allows switching between different algorithms for test case generation based on user input.

Code:

Frontend:

```
import React, { useState } from "react";
import "./App.css"; // Assuming you have an App.css file for styles

function App() {
  const [currentPage, setCurrentPage] = useState("");

  const navigateToPage = (pageId) => {
    setCurrentPage(pageId);
  };

  const analyzeText = () => {
    const text = document.getElementById("textInput").value;
    console.log("Text to be analyzed:", text);
    // You can send this text to your NLP model for analysis here
  };

  return (
    <div>
      <header
        style={{
          backgroundColor: "#333",
          color: "#fff",
          padding: "20px",
          textAlign: "center",
        }}
      >
        <h1>Automated Test Case Generation</h1>
      </header>
      <div
        className="container"
        style={{ maxWidth: "800px", margin: "20px auto", padding: "0 20px" }}
      >
        <section id="intro-section" style={{ marginBottom: "30px" }}>
          <h2>Introduction</h2>
          <p>
            Automated test case generation is a process of automatically
            creating test cases for software applications. These test cases are
            generated based on various inputs, such as requirements,
            specifications, or even existing code. This helps in ensuring the
            quality and reliability of the software product.
          </p>
        </section>
        <div
          className="button-layout"
          style={{ textAlign: "center", marginBottom: "20px" }}
```

```
>
  <button
    onClick={() => navigateToPage("page1")}
    style={{
      padding: "10px 20px",
      margin: "0 10px",
      border: "none",
      borderRadius: "5px",
      backgroundColor: "#007bff",
      color: "#fff",
      cursor: "pointer",
      fontSize: "16px",
    }}
  >
    Enter Text
  </button>
  {/* Add more buttons for navigation here */}
</div>
{currentPage === "page1" && (
  <div id="page1">
    <h2>Enter Text</h2>
    <textarea
      id="textInput"
      placeholder="Enter text here..."
      style={{
        width: "100%",
        height: "200px",
        padding: "10px",
        fontSize: "16px",
        borderRadius: "5px",
        border: "1px solid #ccc",
      }}
    ></textarea>
    <button
      className="submit-button"
      onClick={analyzeText}
      style={{
        display: "block",
        margin: "20px auto",
        padding: "10px 20px",
        border: "none",
        borderRadius: "5px",
        backgroundColor: "#007bff",
        color: "#fff",
              cursor: "pointer",
              fontSize: "16px",
            }}
          >
            Submit
          </button>
        </div>
      )}
      {/* Add more pages for navigation here */}
    </div>
  </div>
 );
}

export default App;
```

Backend code:

```
import openai
import os
# Set your API key
openai.api_key = os.getenv("OPENAI_API_KEY")
# Initialize the conversation with a system message
messages = [{"role": "system", "content": "You are an intelligent software engineering
assistant."}]

while True:
    message = input("User : ")
    if message:
        messages.append({"role": "user", "content": message})
        try:
            chat = openai.ChatCompletion.create(model="gpt-3.5-turbo", messages=messages)
            reply = chat.choices[0].message.content
            print(f"ChatGPT: {reply}")
            messages.append({"role": "assistant", "content": reply})
        except openai.error.RateLimitError as e:
            print("Rate limit exceeded. Please try again later.")
            break
        except Exception as e:
            print("An error occurred:", e)
            break
```

Informal Class Diagram:

TestGenerationService
- Manages test generation using different strategies.
- Notifies observers (e.g., UI components) about test generation updates.
- Attributes:
  - strategy: Strategy (Active test generation strategy.)
  - observers: List[Observer] (Observers to be notified.)
- Methods:
  + setStrategy(Strategy): void (Change test generation strategy.)
  + attach(Observer): void (Add an observer.)
  + detach(Observer): void (Remove an observer.)
  + notifyObservers(): void (Notify all observers of updates.)

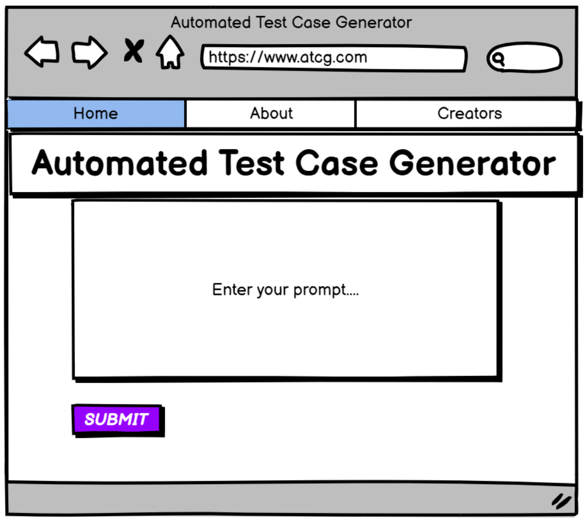
<<interface>> Strategy
- Interface for test generation strategies.
- Method:
  + generateTest(input): TestCases (Generate test cases.)

<<interface>> Observer
- Interface for entities observing TestGenerationService.
- Method:
  + update(TestGenerationService): void (React to updates.)

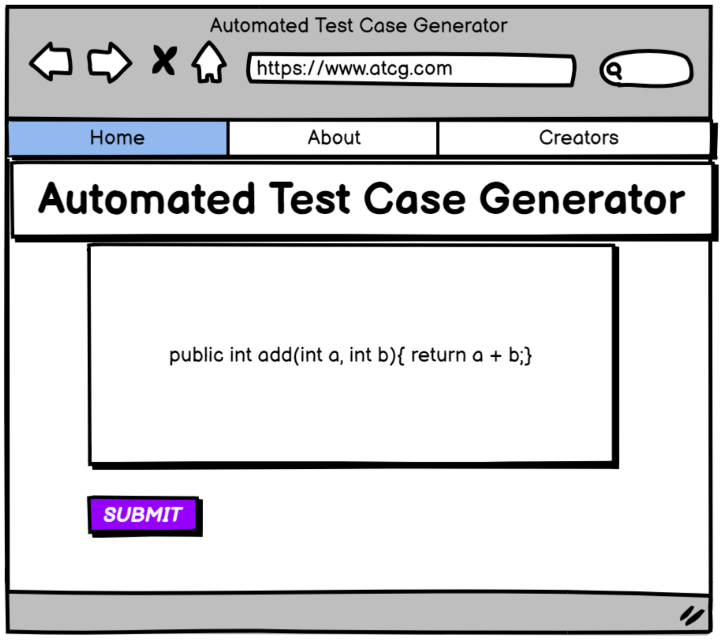
TestStrategy (implements Strategy)
- Concrete strategy for generating test cases.
- Method:
  + generateTest(input): TestCases (Implementation for generating test cases.)

FrontendComponent (implements Observer)
- UI component observing TestGenerationService for updates.
- Method:
  + update(TestGenerationService): void (Update UI based on service updates.)

Design Sketch:

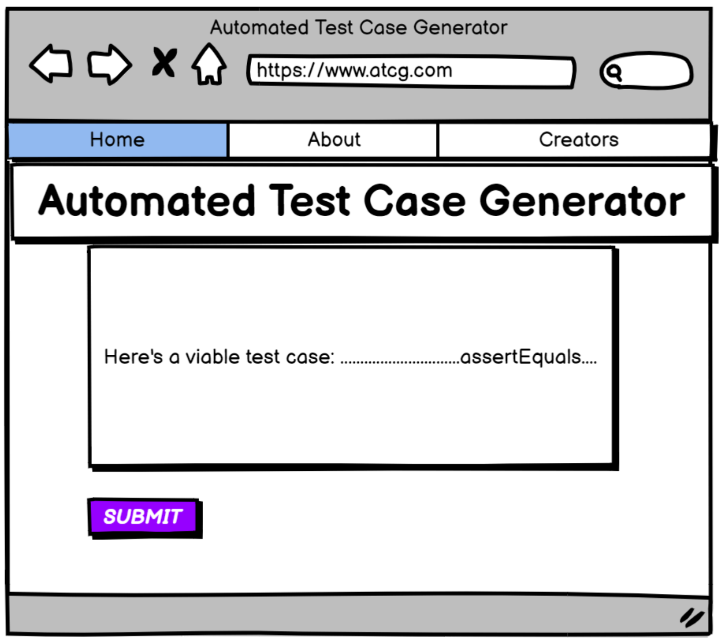


The user is prompted with a simplistic userface upon entering our website. This allows for easy accessibility and less complexity.



```
public int add(int a, int b){ return a + b;}
```

The user enters their code they desire to create test cases for..



Here's a viable test case: ....assertEquals....

Our model returns test cases based upon the input to fit the intended objectives.

We decided a simple interface, similar to OpenAI or Google Search Engine, would allow for ease of comprehension for the user. They can immediately enter their code, submit, and receive feedback. If they wish to learn more about the model, they can navigate to the ‘About’ page or the ‘Creators’ page.

Process Deliverable:
https://github.com/kojikoding/ATCG

The provided link is our github prototype.