```
import concurrent.futures
from datetime import datetime
from typing import Callable, List

from loguru import logger
from pydantic import BaseModel, Field

from swarms.structs.agent import Agent


class AgentResponse(BaseModel):
    agent_name: str
    role: str
    message: str
    timestamp: datetime = Field(default_factory=datetime.now)
    turn_number: int
    preceding_context: List[str] = Field(default_factory=list)


class ChatTurn(BaseModel):
    turn_number: int
    responses: List[AgentResponse]
    task: str
    timestamp: datetime = Field(default_factory=datetime.now)
```

```
class ChatHistory(BaseModel):
    turns: List[ChatTurn]
    total_messages: int
    name: str
    description: str
    start_time: datetime = Field(default_factory=datetime.now)


SpeakerFunction = Callable[[List[str], "Agent"], bool]


def round_robin(history: List[str], agent: Agent) -> bool:
    """
    Round robin speaker function.
    Each agent speaks in turn, in a circular order.
    """
    return True


def expertise_based(history: List[str], agent: Agent) -> bool:
    """
    Expertise based speaker function.
    An agent speaks if their system prompt is in the last message.
    """
    return (
        agent.system_prompt.lower() in history[-1].lower()
```

```
        if history
        else True
    )


def random_selection(history: List[str], agent: Agent) -> bool:
    """
    Random selection speaker function.
    An agent speaks randomly.
    """
    import random

    return random.choice([True, False])


def custom_speaker(history: List[str], agent: Agent) -> bool:
    """
    Custom speaker function with complex logic.

    Args:
        history: Previous conversation messages
        agent: Current agent being evaluated

    Returns:
        bool: Whether agent should speak
    """
```

```
    # No history - let everyone speak
    if not history:
        return True

    last_message = history[-1].lower()

    # Check for agent expertise keywords
    expertise_relevant = any(
        keyword in last_message
        for keyword in agent.description.lower().split()
    )

    # Check for direct mentions
    mentioned = agent.agent_name.lower() in last_message

    # Check if agent hasn't spoken recently
    not_recent_speaker = not any(
        agent.agent_name in msg for msg in history[-3:]
    )

    return expertise_relevant or mentioned or not_recent_speaker


def most_recent(history: List[str], agent: Agent) -> bool:
    """
    Most recent speaker function.
```

```
    An agent speaks if they are the last speaker.
    """
    return (
        agent.agent_name == history[-1].split(":")[0].strip()
        if history
        else True
    )


class GroupChat:
    """
    GroupChat class to enable multiple agents to communicate in a synchronous group chat.
    Each agent is aware of all other agents, every message exchanged, and the social context.
    """

    def __init__(
        self,
        name: str = "GroupChat",
        description: str = "A group chat for multiple agents",
        agents: List[Agent] = [],
        speaker_fn: SpeakerFunction = round_robin,
        max_loops: int = 10,
    ):
        """
        Initialize the GroupChat.
```

```
    Args:
        name (str): Name of the group chat.
        description (str): Description of the purpose of the group chat.
        agents (List[Agent]): A list of agents participating in the chat.
        speaker_fn (SpeakerFunction): The function to determine which agent should speak next.
        max_loops (int): Maximum number of turns in the chat.
    """
    self.name = name
    self.description = description
    self.agents = agents
    self.speaker_fn = speaker_fn
    self.max_loops = max_loops
    self.chat_history = ChatHistory(
        turns=[],
        total_messages=0,
        name=name,
        description=description,
    )


def _get_response_sync(
    self, agent: Agent, prompt: str, turn_number: int
) -> AgentResponse:
    """
    Get the response from an agent synchronously.

    Args:
```

```
        agent (Agent): The agent responding.
        prompt (str): The message triggering the response.
        turn_number (int): The current turn number.

    Returns:
        AgentResponse: The agent's response captured in a structured format.
    """
    try:
        # Provide the agent with information about the chat and other agents
          chat_info = f"Chat Name: {self.name}\nChat Description: {self.description}\nAgents in Chat:
{[a.agent_name for a in self.agents]}"
        context = f"""You are {agent.agent_name}
                Conversation History:
                \n{chat_info}
                Other agents: {[a.agent_name for a in self.agents if a != agent]}
                Previous messages: {self.get_full_chat_history()}
        """  # Updated line

        message = agent.run(context + prompt)
        return AgentResponse(
            agent_name=agent.name,
            role=agent.system_prompt,
            message=message,
            turn_number=turn_number,
            preceding_context=self.get_recent_messages(3),
        )
```

```
    except Exception as e:
        logger.error(f"Error from {agent.name}: {e}")
        return AgentResponse(
            agent_name=agent.name,
            role=agent.system_prompt,
            message=f"Error generating response: {str(e)}",
            turn_number=turn_number,
            preceding_context=[],
        )

def get_full_chat_history(self) -> str:
    """
    Get the full chat history formatted for agent context.

    Returns:
        str: The full chat history with sender names.
    """
    messages = []
    for turn in self.chat_history.turns:
        for response in turn.responses:
            messages.append(
                f"{response.agent_name}: {response.message}"
            )
    return "\n".join(messages)

def get_recent_messages(self, n: int = 3) -> List[str]:
```

```
    """
    Get the most recent messages in the chat.

    Args:
        n (int): The number of recent messages to retrieve.

    Returns:
        List[str]: The most recent messages in the chat.
    """
    messages = []
    for turn in self.chat_history.turns[-n:]:
        for response in turn.responses:
            messages.append(
                f"{response.agent_name}: {response.message}"
            )
    return messages

def run(self, task: str) -> ChatHistory:
    """
    Run the group chat.

    Args:
        task (str): The initial message to start the chat.

    Returns:
        ChatHistory: The history of the chat.
```

```
"""
try:
    logger.info(
        f"Starting chat '{self.name}' with task: {task}"
    )

    for turn in range(self.max_loops):
        current_turn = ChatTurn(
            turn_number=turn, responses=[], task=task
        )

        for agent in self.agents:
            if self.speaker_fn(
                self.get_recent_messages(), agent
            ):
                response = self._get_response_sync(
                    agent, task, turn
                )
                current_turn.responses.append(response)
                self.chat_history.total_messages += 1
                logger.debug(
                    f"Turn {turn}, {agent.name} responded"
                )

        self.chat_history.turns.append(current_turn)
```

```
            return self.chat_history
        except Exception as e:
            logger.error(f"Error in chat: {e}")
            raise e

    def batched_run(self, tasks: List[str], *args, **kwargs):
        """
        Run the group chat with a batch of tasks.

        Args:
            tasks (List[str]): The list of tasks to run in the chat.

        Returns:
            List[ChatHistory]: The history of each chat.
        """
        return [self.run(task, *args, **kwargs) for task in tasks]

    def concurrent_run(self, tasks: List[str], *args, **kwargs):
        """
        Run the group chat with a batch of tasks concurrently using a thread pool.

        Args:
            tasks (List[str]): The list of tasks to run in the chat.

        Returns:
            List[ChatHistory]: The history of each chat.
```

```
        """
        with concurrent.futures.ThreadPoolExecutor() as executor:
            return list(
                executor.map(
                    lambda task: self.run(task, *args, **kwargs),
                    tasks,
                )
            )


# if __name__ == "__main__":

#     load_dotenv()

#     # Get the OpenAI API key from the environment variable
#     api_key = os.getenv("OPENAI_API_KEY")

#     # Create an instance of the OpenAIChat class
#     model = OpenAIChat(
#         openai_api_key=api_key,
#         model_name="gpt-4o-mini",
#         temperature=0.1,
#     )

#     # Example agents
#     agent1 = Agent(
```

```
#         agent_name="Financial-Analysis-Agent",
#         system_prompt="You are a financial analyst specializing in investment strategies.",
#         llm=model,
#         max_loops=1,
#         autosave=False,
#         dashboard=False,
#         verbose=True,
#         dynamic_temperature_enabled=True,
#         user_name="swarms_corp",
#         retry_attempts=1,
#         context_length=200000,
#         output_type="string",
#         streaming_on=False,
#     )

#     agent2 = Agent(
#         agent_name="Tax-Adviser-Agent",
#             system_prompt="You are a tax adviser who provides clear and concise guidance on tax-related queries.",
#         llm=model,
#         max_loops=1,
#         autosave=False,
#         dashboard=False,
#         verbose=True,
#         dynamic_temperature_enabled=True,
#         user_name="swarms_corp",
```

```
#         retry_attempts=1,
#         context_length=200000,
#         output_type="string",
#         streaming_on=False,
#     )

#     agents = [agent1, agent2]

#     chat = GroupChat(
#         name="Investment Advisory",
#         description="Financial and tax analysis group",
#         agents=agents,
#         speaker_fn=expertise_based,
#     )

#     history = chat.run(
#         "How to optimize tax strategy for investments?"
#     )
#     print(history.model_dump_json(indent=2))
```