```
from typing import List, Callable


from swarms.structs.agent import Agent

from swarms.utils.loguru_logger import logger

from swarms.structs.base_swarm import BaseSwarm

from swarms.structs.conversation import Conversation



# def select_next_speaker_bid(
#     step: int,
#     agents: List[Agent],
# ) -> int:
#     """Selects the next speaker."""
#     bids = []
#     for agent in agents:
#         bid = ask_for_bid(agent)
#         bids.append(bid)
#     max_value = max(bids)
#     max_indices = [i for i, x in enumerate(bids) if x == max_value]
#     idx = random.choice(max_indices)
#     return idx



def select_next_speaker_roundtable(
    step: int, agents: List[Agent]
) -> int:
```

```
    """Selects the next speaker."""
    return step % len(agents)


def select_next_speaker_director(
    step: int, agents: List[Agent], director: Agent
) -> int:
    # if the step if even => director
    # => director selects next speaker
    if step % 2 == 1:
        idx = 0
    else:
        idx = director.select_next_speaker() + 1
    return idx


def run_director(self, task: str):
    """Runs the multi-agent collaboration with a director."""
    n = 0
    self.reset()
    self.inject("Debate Moderator", task)
    print("(Debate Moderator): \n")

    while n < self.max_loops:
        name, message = self.step()
        print(f"({name}): {message}\n")
```

```
        n += 1



# [MAYBE]: Add type hints
class MultiAgentCollaboration(BaseSwarm):
    """
    Multi-agent collaboration class.

    Attributes:
        agents (List[Agent]): The agents in the collaboration.
        selection_function (callable): The function that selects the next speaker.
            Defaults to select_next_speaker.
        max_loops (int): The maximum number of iterations. Defaults to 10.
        autosave (bool): Whether to autosave the state of all agents. Defaults to True.
        saved_file_path_name (str): The path to the saved file. Defaults to
            "multi_agent_collab.json".
        stopping_token (str): The token that stops the collaboration. Defaults to
            "<DONE>".
        results (list): The results of the collaboration. Defaults to [].
        logger (logging.Logger): The logger. Defaults to logger.
        logging (bool): Whether to log the collaboration. Defaults to True.


    Methods:
        reset: Resets the state of all agents.
        inject: Injects a message into the collaboration.
```

inject_agent: Injects an agent into the collaboration.

step: Steps through the collaboration.

ask_for_bid: Asks an agent for a bid.

select_next_speaker: Selects the next speaker.

run: Runs the collaboration.

format_results: Formats the results of the run method.

Usage:

```
>>> from swarm_models import OpenAIChat
>>> from swarms.structs import Agent
>>> from swarms.swarms.multi_agent_collab import MultiAgentCollaboration
>>>
>>> # Initialize the language model
>>> llm = OpenAIChat(
>>>     temperature=0.5,
>>> )
>>>
>>>
>>> ## Initialize the workflow
>>> agent = Agent(llm=llm, max_loops=1, dashboard=True)
>>>
>>> # Run the workflow on a task
>>> out = agent.run("Generate a 10,000 word blog on health and wellness.")
>>>
>>> # Initialize the multi-agent collaboration
```

```
>>> swarm = MultiAgentCollaboration(
>>>     agents=[agent],
>>>     max_loops=4,
>>> )
>>>
>>> # Run the multi-agent collaboration
>>> swarm.run()
>>>
>>> # Format the results of the multi-agent collaboration
>>> swarm.format_results(swarm.results)


"""


def __init__(
    self,
    name: str = "MultiAgentCollaboration",
    description: str = "A multi-agent collaboration.",
    director: Agent = None,
    agents: List[Agent] = None,
    select_next_speaker: Callable = None,
    max_loops: int = 10,
    autosave: bool = True,
    saved_file_path_name: str = "multi_agent_collab.json",
    stopping_token: str = "<DONE>",
    logging: bool = True,
    *args,
```

```
    **kwargs,
):
    super().__init__(
        name=name,
        description=description,
        agents=agents,
        *args,
        **kwargs,
    )
    self.name = name
    self.description = description
    self.director = director
    self.agents = agents
    self.select_next_speaker = select_next_speaker
    self._step = 0
    self.max_loops = max_loops
    self.autosave = autosave
    self.saved_file_path_name = saved_file_path_name
    self.stopping_token = stopping_token
    self.results = []
    self.logger = logger
    self.logging = logging

    # Conversation
    self.conversation = Conversation(
        time_enabled=True, *args, **kwargs
```

```
    )


def default_select_next_speaker(
    self, step: int, agents: List[Agent]
) -> int:
    """Default speaker selection function."""
    return step % len(agents)


def inject(self, name: str, message: str):
    """Injects a message into the multi-agent collaboration."""
    for agent in self.agents:
        self.conversation.add(name, message)
        agent.run(self.conversation.return_history_as_string())
    self._step += 1


def step(self) -> str:
    """Steps through the multi-agent collaboration."""
    speaker_idx = self.select_next_speaker(
        self._step, self.agents
    )
    speaker = self.agents[speaker_idx]
    message = speaker.send()

    for receiver in self.agents:
        self.conversation.add(speaker.name, message)
        receiver.run(self.conversation.return_history_as_string())
```

```
        self._step += 1

        if self.logging:
            self.log_step(speaker, message)

        return self.conversation.return_history_as_string()

    def log_step(self, speaker: str, response: str):
        """Logs the step of the multi-agent collaboration."""
        self.logger.info(f"{speaker.name}: {response}")

    def run(self, task: str, *args, **kwargs):
        """Runs the multi-agent collaboration."""
        for _ in range(self.max_loops):
            result = self.step()
            if self.autosave:
                self.save_state()
            if self.stopping_token in result:
                break
        return self.conversation.return_history_as_string()

    # def format_results(self, results):
    #     """Formats the results of the run method"""
    #     formatted_results = "\n".join(
    #         [
```

```
#             f"{result['agent']} responded: {result['response']}"
#             for result in results
#         ]
#     )
#     return formatted_results


# def save(self):
#     """Saves the state of all agents."""
#     state = {
#         "step": self._step,
#         "results": [
#             {"agent": r["agent"].name, "response": r["response"]}
#             for r in self.results
#         ],
#     }

#     with open(self.saved_file_path_name, "w") as file:
#         json.dump(state, file)


# def load(self):
#     """Loads the state of all agents."""
#     with open(self.saved_file_path_name) as file:
#         state = json.load(file)
#     self._step = state["step"]
#     self.results = state["results"]
#     return state
```

```
# def __repr__(self):
#     return (
#         f"MultiAgentCollaboration(agents={self.agents},"
#         f" selection_function={self.select_next_speaker},"
#         f" max_loops={self.max_loops}, autosave={self.autosave},"
#         f" saved_file_path_name={self.saved_file_path_name})"
#     )
```