File Breakdown: `src/core/task_manager.py`

File Location

`src/core/task_manager.py`

Overview

The `task_manager.py` file implements the task management system for the multi-agent framework. It defines the Task class to represent units of work and the TaskManager class to handle task creation, execution, and tracking. This module enables hierarchical task structures, allowing complex operations to be broken down into subtasks, and provides a mechanism to track the status and results of each task.

Key Responsibilities

- Define the Task class for representing work units
- Support hierarchical task structures with parent-child relationships
- Track task status (pending, in progress, completed, failed)
- Store task results for later retrieval
- Create and execute tasks with appropriate agents
- Manage task arguments and parameters
- Execute task hierarchies in the proper order

Core Functionality

Task Class Definition

```
class Task(BaseModel):
    """
    Represents a task or subtask for the agent system.
    Enhanced to support hierarchical task structures and status tracking.
    """
    name: str
    description: str
    agent: str                  # Name of the agent responsible (e.g., "Nova", "Emil", "Ivan", "Lola")
    function_name: Optional[str] = None  # Name of the function to call, if applicable
    args: Dict[str, Any] = {}   # Arguments for the function
    sub_tasks: List["Task"] = []  # List of subtasks
    status: str = "pending"     # Status of the task: pending, in_progress, completed, failed
    result: Any = None          # Result of the task execution

    class Config:
        orm_mode = True
        arbitrary_types_allowed = True
```

Task Methods

Methods for managing task state and structure:

```
def add_subtask(self, task: "Task"):
    """Add a subtask to this task"""
    self.sub_tasks.append(task)
    return task

def mark_in_progress(self):
    """Mark this task as in progress"""
    self.status = "in_progress"

def mark_completed(self, result=None):
    """Mark this task as completed with an optional result"""
    self.status = "completed"
    if result is not None:
        self.result = result

def mark_failed(self, error=None):
    """Mark this task as failed with an optional error message"""
    self.status = "failed"
    if error is not None:
        self.result = error

def update_args(self, new_args: Dict[str, Any]):
    """Update the arguments for this task"""
    self.args.update(new_args)
```

Methods for task serialization and visualization:

```
def to_dict(self):
    """Convert the task to a dictionary"""
    return {
        "name": self.name,
        "description": self.description,
        "agent": self.agent,
        "function_name": self.function_name,
        "args": self.args,
        "sub_tasks": [task.to_dict() for task in self.sub_tasks],
        "status": self.status,
        "result": str(self.result) if self.result is not None else None
    }

def pretty_print(self, indent=0):
    """Pretty print the task hierarchy"""
    indent_str = "  " * indent
    print(f"{indent_str}Task: {self.name} ({self.status})")
    print(f"{indent_str}  Agent: {self.agent}")
    print(f"{indent_str}  Function: {self.function_name}")
    print(f"{indent_str}  Args: {json.dumps(self.args, indent=2)}")
    if self.result:
        print(f"{indent_str}  Result: {self.result}")

    for subtask in self.sub_tasks:
        subtask.pretty_print(indent + 1)
```

Task Manager Class Definition

```
class TaskManager:
    """
    Manages task creation, execution, and tracking.
    """
    def __init__(self, kb):
        """
        Initialize the task manager.

        Parameters:
            kb: The knowledge base to use for storing task results
        """
        self.kb = kb
        self.tasks = []
```

Task Creation and Execution

Methods for creating and executing tasks:

```
def create_task(self, name, description, agent, function_name=None, args=None):
    """
    Create a new task.

    Parameters:
        name (str): Name of the task
        description (str): Description of the task
        agent (str): Name of the agent responsible
        function_name (str, optional): Name of the function to call
        args (dict, optional): Arguments for the function

    Returns:
        Task: The created task
    """
    task = Task(
        name=name,
        description=description,
        agent=agent,
        function_name=function_name,
        args=args or {}
    )
    self.tasks.append(task)
    return task

def execute_task(self, task, agents):
    """
    Execute a task using the appropriate agent.

    Parameters:
        task (Task): The task to execute
        agents (dict): Dictionary mapping agent names to agent instances
```

```
    Returns:
        Any: The result of the task execution
    """
    # Mark the task as in progress
    task.mark_in_progress()

    try:
        # Get the appropriate agent
        agent = agents.get(task.agent)
        if not agent:
            error = f"No agent found for name {task.agent}."
            task.mark_failed(error)
            return error

        # Execute the task
        result = agent.handle_task(task)

        # Mark the task as completed
        task.mark_completed(result)

        # Return the result
        return result
    except Exception as e:
        # Mark the task as failed
        error = f"Error executing task: {str(e)}"
        task.mark_failed(error)
        return error
```

Method for executing hierarchical tasks:

```
def execute_task_hierarchy(self, task, agents):
    """
    Execute a task and all its subtasks.

    Parameters:
        task (Task): The root task to execute
        agents (dict): Dictionary mapping agent names to agent instances

    Returns:
        Any: The result of the root task execution
    """
    # Execute all subtasks first
    for subtask in task.sub_tasks:
        self.execute_task_hierarchy(subtask, agents)

    # Execute the main task
    return self.execute_task(task, agents)
```

Key Features

1. **Hierarchical Task Structure**: Supports parent-child relationships between tasks
2. **Status Tracking**: Maintains the current status of each task

3. **Result Storage**: Captures and stores the results of task execution
4. **Agent Delegation**: Routes tasks to the appropriate agent
5. **Error Handling**: Tracks failures and stores error information
6. **Serialization**: Converts tasks to dictionaries for storage or transmission
7. **Visualization**: Pretty prints task hierarchies for debugging

Integration

- Used by Nova to create tasks from user prompts
- Coordinates with all agents (Nova, Emil, Ivan, Lola) for task execution
- Interfaces with the knowledge base to store task results
- Supports both flat task lists and hierarchical task structures
- Maintains the relationship between parent tasks and subtasks

Workflow

1. User prompt is processed by Nova and converted into tasks
2. Tasks are created with appropriate agent assignments and arguments
3. Tasks may be organized into hierarchies with subtasks
4. Tasks are executed by routing them to the appropriate agent
5. Task status is updated as execution progresses
6. Results are stored in both the task object and the knowledge base
7. Subtasks are executed before their parent tasks

Implementation Notes

- Uses Pydantic's BaseModel for the Task class for validation and serialization
- Implements a recursive execution strategy for task hierarchies
- Provides methods for updating task status throughout execution
- Includes error handling to capture and store exceptions
- Supports visualization of task hierarchies for debugging
- Enables complex workflows to be broken down into simpler subtasks