Документация и Описание Проекта: Менеджер Задач на Python с и без Factory Method

1 Описание Проекта

Данный проект представляет собой приложение "Менеджер Задач", разработанное на языке Python с использованием библиотеки Tkinter для создания графического интерфейса пользователя (GUI). Целью проекта является демонстрация принципа проектирования **Factory Method** (Фабричный Метод) в контексте разработки программного обеспечения.

Приложение позволяет пользователям управлять списком задач, добавлять новые задачи, удалять существующие и отмечать задачи как выполненные. Особенностью проекта является реализация двух версий Менеджера Задач:

- **Версия с Factory Method:** Эта версия использует паттерн проектирования Factory Method для создания объектов задач. Фабрика абстрагирует процесс создания задач, позволяя добавлять новые типы задач без изменения существующего кода пользовательского интерфейса. Это достигается за счет использования абстрактной фабрики `TaskFactory` и конкретных реализаций, таких как `ConcreteTaskFactory` и `WorkTaskFactory`.
- **Версия без Factory Method:** Эта версия демонстрирует традиционный подход к созданию объектов задач, используя условные операторы (if/elif/else) для выбора типа задачи и прямого создания объектов. Это позволяет наглядно сравнить подходы и увидеть преимущества использования Factory Method.

Проект включает в себя следующие типы задач:

- **Обычная задача (Task):** Базовый тип задачи, содержащий только описание и статус выполнения.
- **Задача с дедлайном (DeadlineTask):** Расширение обычной задачи, добавляющее атрибут "дедлайн" (дата и время выполнения).
- **Задача-покупка (ShoppingTask):** Еще одно расширение обычной задачи, включающее список покупок.

Приложение предоставляет простой и интуитивно понятный интерфейс, позволяющий пользователям легко управлять своими задачами. Основной целью проекта является образовательная демонстрация паттерна Factory Method, а не создание полнофункционального менеджера задач для повседневного использования.

2 Структура Проекта и Документация по Файлам

Проект состоит из следующих файлов:

- `main.py`
- `UI/`
  - `taskManagerGUI_Factory.py`
  - `taskManagerGUI_NoFactory.py`
  - `workTaskManagerGUI_Factory.py`

- entity/
  - task.py
  - deadLineTask.py
  - shoppingTask.py
  - base/
    * concreteTaskFactory.py
    * taskFactory.py

2.1 Файл main.py

- **Назначение:** Главный файл, точка входа в приложение. Отвечает за создание главного окна Tkinter и кнопок для запуска различных версий Менеджера Задач.
- **Функции:**
  - run_with_factory(): Создает новое окно Tkinter и запускает версию Менеджера Задач с использованием Factory Method (TaskManagerGUI_Factory).
  - run_without_factory(): Создает новое окно Tkinter и запускает версию Менеджера Задач без использования Factory Method (TaskManagerGUI_NoFactory).
  - run_work_factory(): Создает новое окно Tkinter и запускает версию "Рабочего" Менеджера Задач с использованием Factory Method (WorkTaskManagerGUI_Factory).
- **Основные компоненты:**
  - root: Главное окно Tkinter.
  - Кнопки (ttk.Button): Для запуска каждой версии Менеджера Задач.

2.2 Файл UI/taskManagerGUI_Factory.py

- **Назначение:** Реализация графического интерфейса Менеджера Задач, использующая Factory Method для создания объектов задач. Предназначена для управления **основными** задачами (Regular, Deadline, Shopping).
- **Класс:** TaskManagerGUI_Factory
- **Методы:**
  - __init__(self, root): Конструктор класса. Инициализирует GUI, создает фабрику задач (ConcreteTaskFactory), список задач и элементы управления (Listbox, кнопки).
  - update_task_list(self): Обновляет содержимое Listbox, отображая текущий список задач.
  - open_add_task_window(self): Открывает новое окно Tkinter для добавления новой задачи.
  - show_specific_fields(self, event=None): Динамически отображает специфические поля ввода в окне добавления задачи, в зависимости от выбранного типа задачи (Deadline, Shopping).
  - add_task(self): Получает данные из полей ввода в окне добавления задачи, использует ConcreteTaskFactory для создания объекта задачи соответствующего типа и добавляет задачу в список.
  - delete_task(self): Удаляет выбранную задачу из списка и обновляет Listbox.
  - mark_task_completed(self): Отмечает выбранную задачу как выполненную и обновляет Listbox.
  - show_task_details(self, event): Отображает детали выбранной задачи во всплывающем окне (messagebox).
- **Основные компоненты:**
  - task_factory_regular: Экземпляр ConcreteTaskFactory для создания задач.

- `tasks`: Список для хранения объектов задач.
- `task_listbox`: Listbox для отображения списка задач.
- Кнопки: "Добавить Основную Задачу", "Удалить Задачу", "Отметить выполненной".
- Окно добавления задачи (`add_window`): Временное окно для ввода данных новой задачи.

2.3 Файл `UI/taskManagerGUI_NoFactory.py`

- **Назначение:** Реализация графического интерфейса Менеджера Задач **без** использования Factory Method. Предназначена для управления задачами (Regular, Deadline, Shopping), создавая их напрямую с помощью условных операторов.
- **Класс:** `TaskManagerGUI_NoFactory`
- **Методы:** Аналогичны методам в `TaskManagerGUI_Factory.py` (конструктор, `update_task_list`, `open_add_task_window`, `show_specific_fields`, `delete_task`, `mark_task_completed`, `show_task_details`).
- **Отличия от** `TaskManagerGUI_Factory.py`
  - В методе `add_task(self)` объекты задач создаются напрямую с использованием условных операторов (if/elif/else) в зависимости от выбранного типа задачи. **Не используется Factory Method.**
  - Отсутствует атрибут `task_factory_regular`, так как фабрика не используется.

2.4 Файл `UI/workTaskManagerGUI_Factory.py`

- **Назначение:** Предположительно, реализация GUI для управления **рабочими** задачами с использованием Factory Method. **В текущей реализации функционально идентична** `TaskManagerGUI_Factory.py`.
- **Класс:** `WorkTaskManagerGUI_Factory`
- **Методы:** Аналогичны методам в `TaskManagerGUI_Factory.py`.
- **Основные компоненты:**
  - `task_factory_regular`: Экземпляр `WorkTaskFactory` для создания задач. **В текущей реализации** `WorkTaskFactory` **идентична** `ConcreteTaskFactory`**, поэтому функциональность не отличается от** `TaskManagerGUI_Factory.py`
  - Остальные компоненты аналогичны `TaskManagerGUI_Factory.py`.

2.5 Файл `entity/task.py`

- **Назначение:** Определение базового класса `Task` для представления задач.
- **Класс:** `Task`
- **Атрибуты:**
  - `description`: Описание задачи (строка).
  - `status`: Статус выполнения задачи (строка, по умолчанию "Не выполнена").
- **Методы:**
  - `__init__(self, description)`: Конструктор класса.
  - `get_description(self)`: Возвращает описание задачи.
  - `get_status(self)`: Возвращает статус задачи.
  - `mark_completed(self)`: Устанавливает статус задачи в "Выполнена".
  - `display_details(self)`: Возвращает строковое представление деталей задачи для отображения.
  - `get_task_type(self)`: Возвращает тип задачи ("Обычная").

- `get_specific_data(self)`: Возвращает словарь для хранения специфических данных задачи (по умолчанию пустой словарь).
- `set_specific_data(self, data)`: Устанавливает специфические данные задачи из словаря (в базовом классе не реализовано).
- `__str__(self)`: Возвращает строковое представление задачи для отображения в списке.

2.6 Файл `entity/deadLineTask.py`

- **Назначение:** Определение класса `DeadlineTask`, наследуемого от `Task`, для представления задач с дедлайном.
- **Класс:** `DeadlineTask`
- **Наследование:** Наследуется от класса `Task`.
- **Дополнительные атрибуты (помимо унаследованных от `Task`):**
  - `deadline`: Дедлайн задачи (строка, формат "ГГГГ-ММ-ДД").
- **Переопределенные и добавленные методы:**
  - `__init__(self, description, deadline)`: Конструктор класса. Вызывает конструктор родительского класса `Task` и инициализирует атрибут `deadline`.
  - `get_deadline(self)`: Возвращает дедлайн задачи.
  - `display_details(self)`: Переопределенный метод, добавляет информацию о дедлайне к деталям задачи.
  - `get_task_type(self)`: Переопределенный метод, возвращает тип задачи ("С дедлайном").
  - `get_specific_data(self)`: Переопределенный метод, добавляет дедлайн к специфическим данным задачи.
  - `set_specific_data(self, data)`: Переопределенный метод, устанавливает дедлайн из специфических данных.
  - `__str__(self)`: Переопределенный метод, добавляет информацию о дедлайне к строковому представлению задачи.

2.7 Файл `entity/shoppingTask.py`

- **Назначение:** Определение класса `ShoppingTask`, наследуемого от `Task`, для представления задач-покупок.
- **Класс:** `ShoppingTask`
- **Наследование:** Наследуется от класса `Task`.
- **Дополнительные атрибуты (помимо унаследованных от `Task`):**
  - `shopping_list`: Список покупок (список строк).
- **Переопределенные и добавленные методы:**
  - `__init__(self, description, shopping_list)`: Конструктор класса. Вызывает конструктор родительского класса `Task` и инициализирует атрибут `shopping_list`.
  - `get_shopping_list(self)`: Возвращает список покупок.
  - `display_details(self)`: Переопределенный метод, добавляет список покупок к деталям задачи.
  - `get_task_type(self)`: Переопределенный метод, возвращает тип задачи ("Покупка").
  - `get_specific_data(self)`: Переопределенный метод, добавляет список покупок к специфическим данным задачи.
  - `set_specific_data(self, data)`: Переопределенный метод, устанавливает список покупок из специфических данных.
  - `__str__(self)`: Переопределенный метод, добавляет список покупок к строковому представлению задачи.

2.8 Файл `entity/base/concreteTaskFactory.py`

- **Назначение:** Реализация конкретных фабрик задач: `ConcreteTaskFactory` и `WorkTaskFactory`.
- **Классы:**
  - `ConcreteTaskFactory`: Конкретная фабрика для создания **основных** типов задач (Regular, Deadline, Shopping). Наследуется от `TaskFactory`.
  - `WorkTaskFactory`: Предположительно, конкретная фабрика для создания **рабочих** типов задач. **В текущей реализации идентична** `ConcreteTaskFactory`. Наследуется от `TaskFactory`.
- **Методы (в обоих классах):**
  - `create_task(self, task_type, kwargs)`: Реализация метода `create_task` из абстрактной фабрики `TaskFactory`. Принимает тип задачи (`task_type`) и дополнительные аргументы (`kwargs`). Использует условные операторы (if/elif/else) для создания объекта задачи соответствующего типа (Task, DeadlineTask, ShoppingTask) и возвращает его. В случае неизвестного типа задачи вызывает исключение `ValueError`.

2.9 Файл `entity/base/taskFactory.py`

- **Назначение:** Определение абстрактной фабрики `TaskFactory`, реализующей паттерн Factory Method.
- **Класс:** `TaskFactory`
- **Абстрактный класс (ABC):** Использует `abc.ABC` для определения абстрактного класса.
- **Абстрактные методы:**
  - `create_task(self, task_type, kwargs)`: Абстрактный метод, который должен быть реализован в конкретных фабриках. Предназначен для создания объектов задач.

3 Запуск Приложения

Для запуска приложения необходимо выполнить файл `main.py` с помощью интерпретатора Python.

```
python main.py
```

Listing 1: Запуск приложения из командной строки

После запуска появится главное окно с кнопками для запуска различных версий Менеджера Задач. Нажмите на соответствующую кнопку, чтобы открыть нужную версию приложения.

4 Factory Method Паттерн

Проект наглядно демонстрирует паттерн проектирования **Factory Method**. В версии с Factory Method, создание объектов задач делегируется фабрикам (`ConcreteTaskFactory` и `WorkTaskFactory`). GUI компоненты (`TaskManagerGUI_Factory` и `WorkTaskManagerGUI_Factory`) не зависят от конкретных классов задач (Task, DeadlineTask, ShoppingTask). Они взаимодействуют с фабрикой, запрашивая создание задачи определенного типа, но не зная, как именно эти объекты создаются.

Это обеспечивает следующие преимущества:

- **Слабая связанность (Decoupling):** GUI код не зависит от конкретных классов задач. Изменения в классах задач не потребуют изменений в GUI коде (если интерфейс фабрики остается неизменным).
- **Расширяемость (Extensibility):** Для добавления нового типа задачи достаточно создать новый класс задачи и обновить конкретную фабрику, добавив логику создания нового типа. GUI код остается неизменным.

- **Инкапсуляция процесса создания объектов:** Логика создания объектов задач инкапсулирована в фабриках, что делает код более организованным и легким для понимания.

Версия без Factory Method (`taskManagerGUI_NoFactory.py`) демонстрирует подход, где создание объектов задач осуществляется непосредственно в GUI коде с использованием условных операторов. Этот подход менее гибкий и сложнее в расширении, так как добавление нового типа задачи потребует изменения кода GUI.

5 Заключение

Данный проект является учебным примером, демонстрирующим применение паттерна Factory Method в Python для создания Менеджера Задач. Проект позволяет сравнить реализацию с и без Factory Method, наглядно показывая преимущества использования данного паттерна для повышения гибкости, расширяемости и слабой связанности кода.