TASK

Software Design

Visit our website

Introduction

In this task, you'll learn some of the most powerful techniques for designing object-oriented software systems. These methods will help you simplify and manage complex processes. We'll explore class diagrams, sequence diagrams, separation of concerns, use case analysis, and CRUD matrices.

MODULARISATION

Modularisation involves breaking down complex systems into independent, interchangeable, and self-contained modules. These modules, or components, can function autonomously, contributing to the overall system's functionality. One of the key advantages of modularisation is its ability to enhance development efficiency, code reusability, and collaboration among developers.

In the realm of applications, this modular approach becomes especially valuable. Applications often undergo rapid changes, updates, and improvements to meet evolving user needs and take advantage of technological advancements. Modularisation allows for the swift removal and replacement of specific modules without disrupting the entire system. This adaptability is crucial in dynamic environments where quick adjustments are required to keep pace with market demands or to address emerging issues.

Furthermore, the ability to reuse modules across different applications can significantly streamline the development process. Developers can leverage existing, well-tested modules, saving time and effort by avoiding the need to recreate functionality from scratch. This not only accelerates development timelines but also contributes to a more consistent and reliable codebase.

In summary, modularisation's strength lies not only in its ability to decompose complexity and enhance collaboration, but also in its support for the agile development of applications. The rapid removal, replacement, and reuse of modules empowers developers to respond swiftly to changing requirements, ensuring that applications remain adaptable and resilient in a dynamic technological landscape.

KEY PRINCIPLES OF MODULARISATION

Independence: Modules (self-contained units such as functions or classes) should operate independently, minimising dependencies on other components. This independence facilitates easier testing, maintenance, and updates.

Interchangeability: Modules should be interchangeable (such as a Database Connector module), allowing developers to replace or upgrade one module without affecting the entire system. This flexibility is crucial for system evolution.

Reusability: Design modules with a focus on reusability. A well-designed module (such as the `print` function) can be applied in different parts of a system or even in entirely different projects, saving development time and effort.

Encapsulation: Each module should encapsulate a specific functionality (such as user authentication), hiding its internal workings. This encapsulation promotes a clear understanding of each module's purpose without delving into the intricacies of its implementation.

Scalability: Modular design supports scalability by enabling the addition or removal of modules to accommodate changes in system requirements (consider a web server). This adaptability is essential for systems that may evolve.

WHAT DOES MODULARISED SOFTWARE DESIGN LOOK LIKE?

In practical terms, modular design involves creating distinct, independent components within a system. These components communicate through well-defined interfaces (think of an interface as a socket through which your program can plug into), allowing seamless interaction. Let's explore a generic example.

Example: A building automation system

Consider a building automation system that controls heating, ventilation, and air conditioning (HVAC), as well as lighting. In a modular design approach, we would have a module for each aspect:

Temperature-control module:

- Responsible for monitoring and controlling the HVAC system.
- Independent functionalities include temperature sensing, fan control, and temperature adjustment.

Lighting-control module:

- Manages the lighting system based on occupancy and time of day.
- Independent functionalities include occupancy sensing, light intensity adjustment, and scheduling.

Security module:

- Deals with security aspects, such as access control and surveillance.
- Independent functionalities include door access management, camera control, and alarm systems.

Communication interface:

- Acts as a central communication hub.
- Independent functionalities include data transmission between modules, error handling, and system-wide notifications.

Each module operates independently, focusing on a specific aspect of the building automation system. This modular approach enhances the system's maintainability, flexibility, and overall robustness.

ADVANTAGES OF MODULAR DESIGN

Ease of maintenance: Modular code makes it easier to locate, fix, or update specific parts without affecting the entire system. This not only streamlines debugging but also facilitates teamwork, as different modules can be assigned to different developers. Additionally, it promotes code reusability, reducing redundancy and ensuring consistent updates across the system.

Enhanced reusability: Well-designed modules can be reused in different projects, promoting efficiency and consistency. This accelerates development and ensures reliable components are used across applications, saving time and minimising bugs.

Improved collaboration: Developers can work on different modules concurrently, fostering collaboration and speeding up development. Teams can efficiently divide the workload and address issues independently, enhancing responsiveness to project requirements.

Scalability: As system requirements evolve, additional modules can be integrated or existing ones replaced, ensuring adaptability. This modular design allows for seamless growth, accommodating more users, increased data volumes, or new features without a complete overhaul.

Simplified testing: Testing individual modules in isolation simplifies the debugging process, ensuring each component functions as intended. This focused approach enhances reliability and accelerates development by allowing thorough testing of each module before integration.

Summary

By adhering to modular design principles, software engineers create systems that are robust, adaptable, and conducive to collaborative development. This design philosophy promotes the creation of independent and reusable modules, each responsible for a specific functionality. The modularity enhances the system's robustness by containing errors within isolated modules, preventing them from affecting the entire system. Moreover, adaptability stems from the ease with which new modules can be added or existing ones replaced, ensuring the software can evolve to meet changing requirements.

WHAT IS USE CASE ANALYSIS?

Use case analysis is a vital software engineering technique aimed at comprehensively defining and explaining how users interact with a system to accomplish specific objectives. This method captures real-world workflows and requirements, employing use case analysis diagrams to visually map plausible user-system interactions. These diagrams offer a high-level overview illustration of the functionalities users can engage with in the system.

The significance of use case analysis lies in its role as a valuable tool for discussions about stakeholder needs, identification of edge cases, and the planning of rigorous tests to ensure that the coded system delivers required behaviours. In essence, use case analysis serves as a facilitative tool, guiding us in exploring how users will utilise a system. The resulting use case analysis diagrams provide a visual model that aids in the effective alignment and planning of the development process.

USE CASE ANALYSIS DIAGRAM COMPONENTS

A use case analysis diagram represents the interactions between **actors** (**users** or **external systems**) and the application. It helps us to visualise and understand how different actors interact with the system through various use cases. Let's look at some of the components used to construct use case diagrams.

Actors

- **Users:** Actors represent entities outside the system that interact with it. Actors can be categorised into different roles, each having distinct interactions with the application. For instance, administrators, registered users, and anonymous visitors are common actor types.

- **External systems:** Other systems or services that interact with the application, such as third-party APIs or databases.

Use cases

- Use cases describe specific functionalities or tasks that actors can perform within the system. Each use case should be focused on a single action. Use cases often align with specific views or functions, such as user authentication or data manipulation. Examples of use cases in an application include user authentication, data submission, and content retrieval.

System boundary

- The system boundary defines the scope of the application, encapsulating all actors and use cases. It helps establish a clear distinction between the internal components of the application and external entities.

Relationships between use cases

- **Associations** represent relationships between actors and use cases, showing which actors are involved in each use case. This might indicate which types of users are associated with specific actions, such as administrators managing user accounts.

- **Dependencies** show the relationships between different use cases. For instance, one use case may depend on the successful completion of another. This could represent dependencies between user authentication and access to specific features.

- **Generalisations** represent inheritance relationships between use cases. It's particularly relevant when different types of users share common functionalities. For example, both administrators and regular users might share the use case of adding tasks to a to-do list.

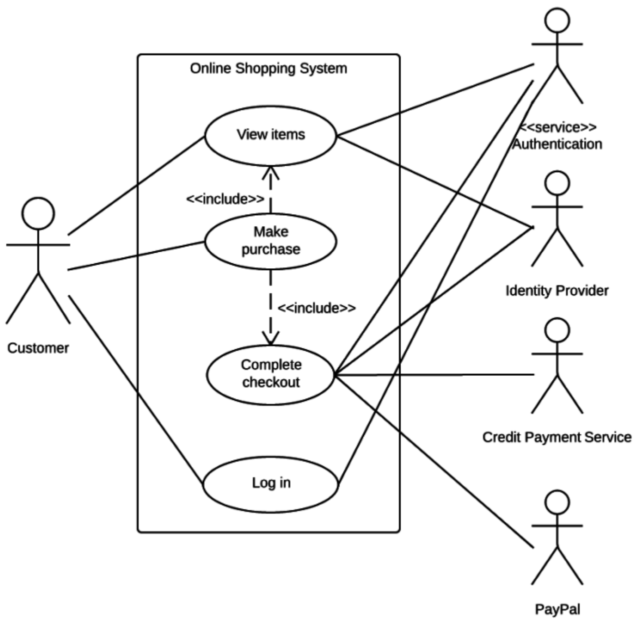
Include and extend relationships

- **Include relationship:** “Include” relationships indicate that one use case includes the functionality of another. This could be seen when a higher-level use case, like “Manage Tasks”, includes lower-level functionalities like “Add Task” and “Mark Task as Complete”.

- **Extend relationship:** “Extend” relationships represent optional or conditional behaviour. This might be illustrated when a use case for adding a task can be extended with optional features, like setting reminders or attaching files.

- **System controller (optional):** The system controller represents the mechanism or component responsible for coordinating the flow of information between actors and the application. This often corresponds to views (components within a system that are responsible for specific functions related to user interaction and application functionality), which handle user requests, process data, and manage the application's response.

Here's a simplified example of a use case diagram:



UML use case analysis diagram (Lucidchart, n.d.)

This image shows a unified modelling language (**UML**) use case diagram for an online shopping system, depicting the interactions between a customer and various system components like viewing items, making purchases, and completing checkout, along with external services for authentication, identity, and payment processing. In this diagram, you would detail each use case with specific actions and interactions within the application. The relationships and actors would depend on the specific functionalities and requirements of your project.

WHAT IS A SEQUENCE DIAGRAM?

Sequence diagrams are a type of interaction diagram in UML that visually represents the chronological flow of messages and interactions among different components or objects in a system. These diagrams are particularly useful for illustrating the dynamic aspects of a system, showcasing how various entities collaborate over time to achieve a specific functionality or respond to an event.

In a sequence diagram, the participants, which can be objects or components, are depicted as vertical lifelines. The timeline runs horizontally, representing the progression of time from top to bottom. Arrows and messages between the lifelines illustrate the order and nature of interactions, showcasing the exchange of information or calls between different parts of the system.

Key elements of sequence diagrams

- **Lifelines:** Vertical lines representing the entities involved in the sequence. Each lifeline corresponds to an object or component (user, view, or database).
- **Messages:** Arrows indicating the flow of communication between lifelines. Messages can be synchronous (denoted by solid arrows) or asynchronous (denoted by dashed arrows), depending on whether they occur in a blocking or non-blocking manner.
- **Activation bars:** Rectangular vertical boxes along a lifeline that represent the duration of time during which an object is active or performing a task.

Sequence diagrams are valuable tools for understanding and documenting the dynamic behaviour of a system. They aid in visualising the order of interactions, identifying potential bottlenecks, and ensuring that the system behaves as intended during different scenarios. These diagrams are particularly helpful in the design and communication phases of software development, providing a clear and concise way to illustrate the flow of control and communication between system components.

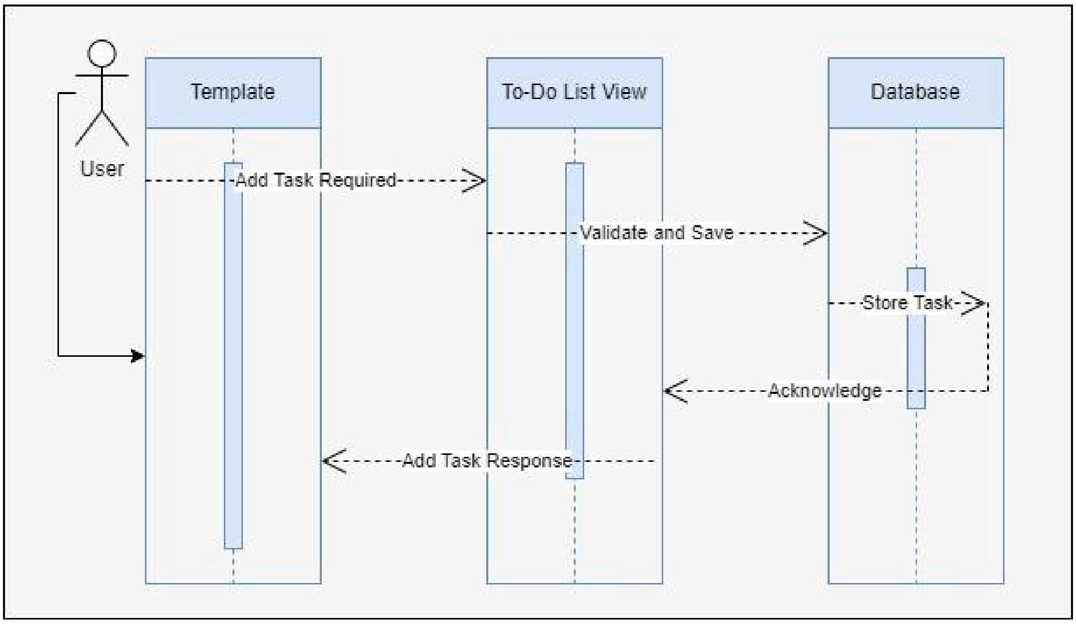
SEQUENCE DIAGRAM COMPONENTS

As a sequence diagram represents the flow of interactions between different components or objects over time, it is particularly useful for illustrating the chronological order of messages exchanged between different parts of an application during a specific process or scenario.

Actors/objects

- **Users:** Represented as the initiator of the sequence, sending a message to add a task.
- **Views/controllers:** Components that handle incoming requests and coordinate the flow of data and logic.
- **Models:** Representations of the application's software entities and business logic.
- **Templates:** Views responsible for rendering HTML and presenting data to users.
- **External systems:** Other systems or services that the application interacts with, such as databases, external APIs, etc.
- **Lifelines:** Lifelines represent the different participants (actors, components) in the sequence diagram. Each lifeline corresponds to a participant involved in the sequence of interactions.
- **Messages:** Messages depict the communication between lifelines. Messages could be HTTP requests, function calls, or any other communication between components, and can be synchronous or asynchronous.
- **Activation bar:** Activation bars represent the period during which a lifeline is active or engaged in processing a message. They show the duration of the interaction for a particular lifeline.
- **Return messages**: Return messages show the response or data flow back from the recipient to the sender.
- **System boundary:** Similar to what we did with the use case analysis diagram, you might include a system boundary to illustrate the scope of the application.

Here's a simplified example of a sequence diagram for adding a task to a web application:



Sequence diagram for adding a task to a web application

In this example, the sequence diagram shows the flow of messages between the user, the view, and the database during the authentication and profile rendering process. Each arrow represents a message, and the activation bars show when each component is actively processing the message.

The “Template” lifeline has an activation bar extending from the moment it initiates the “Add Task Request” message until it receives the “Add Task Response” message.

The “To-Do List View” lifeline has an activation bar that starts when it receives the “Add Task Request” message and continues through the “Validate and Save” process, ending after it sends the “Add Task Response” message to the user.

The “Database” lifeline has an activation bar starting when it receives the “Store Task” message and ending after it sends the acknowledgement back to the “To-Do List View”.

SEPARATION OF CONCERNS

Creating a software application involves more than just coding features – developers also bear important responsibilities around ethical design, quality assurance, and planning for long-term sustainability. At the forefront are security and privacy considerations, ensuring user data is handled safely and only in ways they have consented to. Accessibility standards must also be upheld so those with disabilities can effectively use the app.

Performance tuning through efficient algorithms and cloud infrastructure is key for responsiveness. Architecting a clean, well-documented codebase makes adding features and fixing bugs much easier down the road, and following platform best practices helps manage issues before they snowball. While balancing all these technical and social responsibilities adds complexity, prioritising users and system health leads to more robust and equitably useful applications. By proactively addressing areas of concern, developers can focus on innovation while creating trust.

Let's consider these concepts in the context of our modular design approach to building a web app. The responsibilities and concerns are effectively distributed across various components, models define the software entities and encapsulate business logic and interactions with the database, and views manage presentation logic and shape the visual presentation. Additionally, controllers handle user input, interact with models, and render the final output through views. This modular approach, following the model-view-controller (MVC) pattern, enhances code organisation, maintainability, and collaboration.

MVC is a design pattern commonly used in software development to organise code and separate concerns within an application. It is a conceptual framework that divides the application into three interconnected components: models, views, and controllers. Let's take a closer look at these components.

Models

- **Definition:** Models represent the application's software entities and business logic. They encapsulate the data and define the rules and operations for manipulating that data.
- **Responsibilities:**
    - Receive data from the controller, process it, and return the result.
    - Implement business logic related to the manipulation and processing of data.

- **Example (in the context of a to-do list application):**
    - Define the task model to represent tasks with attributes such as title, description, and completion status.

Views

- **Definition:** Views are responsible for presenting the application's data to the user and receiving user input. Controllers render views using models.
- **Responsibilities:**
    - Handle user interface elements and interactions.
    - Interact with the models for data retrieval and modification.
    - Return appropriate responses, often by rendering templates.
- **Example (in the context of a to-do list application):**
    - Displaying a task list.
    - Accepting commands from the user to add tasks to the task list.

Controllers

- **Definition:** Controllers act as intermediaries between the models and views. They receive user input from the views, interact with the models to process that input, and update the views accordingly.
- **Responsibilities:**
    - Handle user input and invoke corresponding actions on the models.
    - Update views based on changes in the models.
- **Example (in the context of a to-do list application):**
    - Handle user requests to add or delete tasks, and update the task list accordingly.

Below is an example of a diagram depicting the MVC components:

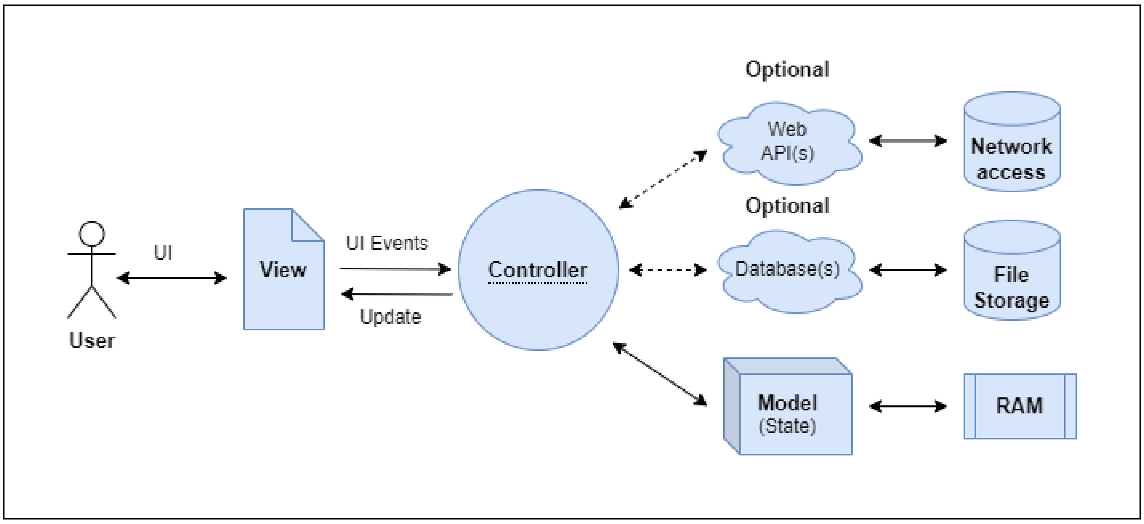


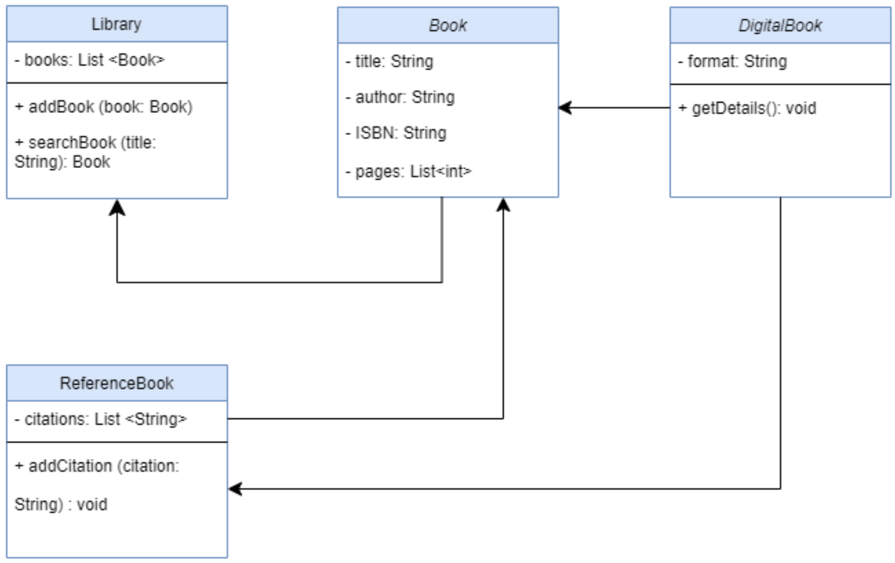
Diagram of MVC components

Understanding and appropriately managing these responsibilities and concerns in their respective components helps in building a well-organised, modular, and maintainable application. This separation of concerns is a fundamental principle in software design and contributes to the overall robustness and scalability of the application.

CLASS DIAGRAMS

A class diagram serves as a visual representation of the static structure within an application, illustrating classes, their attributes, and the relationships existing between them. To enhance the illustrative depth of the diagram, we aim to incorporate elements and details, such as relationships between classes, inheritance hierarchies, and attributes.

Below is an example of a class diagram:



Example of a class diagram

What follows is a breakdown of the classes.

Book class

Attributes:

- `title`: String (private)
- `author`: String (private)
- `ISBN`: String (private)

Methods:

- `getDetails()`: Void (public)

Library class

Attributes:

- `books`: List<Book> (private)

Methods:

- `addBook(book: Book)`: Void (public)
- `searchBook(title: String)`: Book (public)

DigitalBook class

Attributes:

- `title`: String (inherited)
- `author`: String (inherited)
- `ISBN`: String (inherited)
- `format`: String

Methods:

- `getDetails()`: Void (inherited from Book)

ReferenceBook class

Attributes:

- `title`: String (inherited)
- `author`: String (inherited)
- `ISBN`: String (inherited)
- `citations`: List<String>

Methods:

- `getDetails()`: Void (inherited from Book)
- `addCitation(citation: String)`: Void

In this simple representation:

- The arrows indicate the direction of associations.
- The solid arrow from Book to Library represents the Library having a list of books.
- The solid arrow from subclasses (DigitalBook and ReferenceBook) to Book indicates inheritance.
- The DigitalBook class has an additional method, **`getDetails`**.
- The ReferenceBook class has an association with Book, indicating it inherits and has an "is-a" relationship; in other words, a ReferenceBook *is a* Book.

In class diagrams, attributes and methods can have different access levels. Public attributes and methods, indicated by a **+** symbol, are accessible from outside the class, meaning other classes can interact with them directly. Private attributes and methods, indicated by a **-** symbol, are only accessible from within the class itself, which helps encapsulate the internal state and behaviour. The lines connecting the classes represent associations.

To learn more about class diagrams, take a look at **this article**.

CRUD MATRIX

A **CRUD** matrix is a tool used to document the relationships between different entities (usually database tables) and the **CRUD** (**Create**, **Read**, **Update**, **Delete**) operations that can be performed on them. These entities are often represented by models. We will now look at how you might create a CRUD matrix for a simple application.

Example CRUD matrix

Let's consider a library application with a model called **Task**.

Entity	**Create**	**Read**	**Update**	**Delete**
Task	**X**	**X**	**X**	**X**

Task entity

Create (C): Creating a new task instance.
Read (R): Retrieving information about a task.
Update (U): Modifying information of an existing task.
Delete (D): Removing a task from the system.

The Python script below showcases the fundamental principles of CRUD operations within a simplified task management system. The script defines a `Task` class representing tasks with attributes such as `task_ID`, `title`, and `description`. A sample set of tasks is created, and various functions are implemented to perform CRUD operations on this data. The script then demonstrates the execution of these operations, offering a hands-on example of how to create, read, update, and delete tasks within a Python context.

```python
class Task:
    """Class representing a Task entity with attributes:
       task_id, title, and description."""
    def __init__(self, task_id, title, description):
        """Initialise a new Task instance."""
        self.task_id = task_id
        self.title = title
        self.description = description
```

Explanation:

- The `Task` class is a blueprint for creating instances representing tasks.
- Each task has the attributes `task_id`, `title`, and `description`.
- The `__init__` method is a special method that gets called when a new task instance is created. It initialises the attributes of the task.

```
# Sample objects
tasks_data = [
    Task(1, "Task 1", "Description for Task 1"),
    Task(2, "Task 2", "Description for Task 2"),
    Task(3, "Task 3", "Description for Task 3"),
]
```

Explanation:

- `tasks_data` is a list containing three sample tasks, each represented by a task instance.
- These instances have different `task_id`, `title`, and `description` values.

CRUD functions

Display tasks:

```
def display_tasks():
    """Display information for all tasks."""
    print("Tasks:")
    for task in tasks_data:
        print(
            f"ID: {task.task_id}, Title: {task.title}, "
            f"Description: {task.description}"
        )
    print()
```

Create tasks:

```
def create_task(title, description):
    """Create a new task and add it to the tasks_data list."""
    new_id = len(tasks_data) + 1
    new_task = Task(new_id, title, description)
    tasks_data.append(new_task)
    print(f"Task '{title}' created successfully!\n")
```

Read tasks:

```
def read_task(task_id):
    """Read and display information for a task based on its ID."""
    for task in tasks_data:
        if task.task_id == task_id:
            print(
                f"Task found - ID: {task.task_id}, Title: {task.title},
"
                f"Description: {task.description}\n"
            )
            return
    print(f"Task with ID {task_id} not found.\n")
```

Update tasks:

```
def update_task(task_id, new_title, new_description):
    """Update the title and description of a task based on its ID."""
    for task in tasks_data:
        if task.task_id == task_id:
            task.title = new_title
            task.description = new_description
            print("Task updated successfully!\n")
            return
    print(f"Task with ID {task_id} not found.\n")
```

Delete tasks:

```
def delete_task(task_id):
    """Delete a task based on its ID."""
    for i, task in enumerate(tasks_data):
        if task.task_id == task_id:
            del tasks_data[i]
            print("Task deleted successfully!\n")
            return
    print(f"Task with ID {task_id} not found.\n")
```

Explanation:

- `display_tasks` prints information about all tasks.

- Each of the functions below corresponds to a CRUD operation.
  - `create_task` adds a new task to the `tasks_data` list.
  - `read_task` searches for a task by its `ID` and prints its information.
  - `update_task` finds a task by its `ID` and updates its `title` and `description`.
  - `delete_task` removes a task from `tasks_data` based on its `ID`.

Demonstration of CRUD operations:

```
# Demonstrate CRUD operations
display_tasks()
create_task("New Task", "Description for the new task")
display_tasks()
read_task(2)
update_task(1, "Updated Task 1", "New description for Task 1")
display_tasks()
delete_task(3)
display_tasks()
```

This matrix offers a concise summary of the operations applicable to each entity in the script. It serves as a helpful guide for grasping the fundamental functionalities of the code, and it can be referred to when implementing and understanding various aspects of the script, such as the creation, retrieval, update, and deletion of tasks.

Practical task

In this task, you will review the design principles you have learned and practise the skills you have been introduced to by **designing a task manager application** with features of your choice.

Need a quick and easy drawing tool? Check out **draw.io** for creating flowcharts, diagrams, and more. It's user-friendly and perfect for spicing up your projects visually. Give it a go and let your creativity shine!

Follow these steps:

1. Create a use case diagram for your **task manager application**. You have creative freedom here, so your diagram can be as simple or as complex as you choose, based on the use cases you decide for your application. However, plan for your application to have the full range of CRUD (Create, Read, Update, Delete) functionality which should be evident within your diagrams.
2. Create a sequence diagram for your task manager application. You may assume that your task manager application will utilise files to store its data.
3. In a plain text file, clearly outline the specific responsibilities and concerns of each component (models, views, and controllers) in your task manager application, following the **MVC** (model-view-controller) pattern.
4. Create a class diagram for your task manager application.
5. Create a CRUD matrix for your task manager application.

Important: Be sure to upload all files required for the task submission inside your task folder and then click "Request review" on your dashboard.

Share your thoughts

Please take some time to complete this short feedback **form** to help us ensure we provide you with the best possible learning experience.

Reference list

Lucidchart. (n.d.). *UML use case diagram tutorial*. **https://www.lucidchart.com/pages/uml-use-case-diagram**