**Documentation of Robotic Cell Management**

The Robotic Cell Management System is designed to automate and streamline the management of a robotic workforce in a production environment. This system allows users to add and remove robots, workers, and tasks, as well as monitor their statuses. The core functionality involves managing the availability of robots and workers, creating tasks, assigning tasks to available resources, and tracking the completion of these tasks. The system ensures efficient use of resources and provides real-time status updates.

**Key Features**

* **Resource Management:** The system manages both human workers and robots, ensuring that each is available and assigned to the correct tasks based on their capabilities.
* **Task Management:** The system supports creating tasks, assigning them to available resources and monitoring their status thoroughly.
* **Task Queue**: If resources are currently occupied then tasks are added to a FIFO (First-In-First-Out) queue and processed when sufficient resources are available.
* **Product Management:** The system supports creating products with a series of tasks involved in the product development, the system then uses the Task Management logic to handle the flow of completion.
* **Dynamic Workflow:** The system supports the creation of complex workflows by linking tasks to product steps and ensuring that all necessary resources are allocated efficiently

**Application of Procedural Concepts**

Procedural programming concepts such as variables, loops, and functions are central to the design and operation of the robotic cell management system. Let’s break down how each of these concepts is applied.

**Variables**

Variables used to store various data required by the system. For Example:

* **Robots:** A dictionary that stores data of all robots individually, including their ID, status and current task data.
* **Workers:** A Dictionary that stores data of all workers individually, including their ID, status and current task data.
* **Tasks:** A Dictionary that contains the details of the tasks, including the resources required (workers and robots) and the task status
* **Products:** A dictionary that contains the details and steps of a product.
* **Task\_queue:** List implementation of a queue to hold awaiting tasks.

These variables help in tracking and managing the state of robots, workers, tasks, and the queue, ensuring smooth operation of the system.

**Loops**

Loops are crucial for iterating over collections of data, such as workers, robots, and tasks. The system uses loops to perform various tasks, including:

* **Adding workers and robots**: Loops are used to prepopulate the system with a set number of workers and robots for ready to use flow of execution.
* **Assigning tasks**: A loop is used to update the status of workers and robots when they are assigned to tasks, the system checks for resource availability and assigns them dynamically as soon as resources become available.
* **Task status updates**: after each task is completed, the system checks if there are any tasks enqueued or not.

**Functions**

Functions are used to implement the divide and conquer technique for each core operation encapsulated within it.

* **addRobot():** Adds a robot to the system with a unique ID.
* **removeRobot():** removes a robot from the system while keeping in check whether it exists or not and whether it is idle or not.
* **addWorker():** Adds a worker to the system with a unique ID.
* **removeWorker():** removes a worker from the system while keeping in check whether it exists or not and whether it is idle or not.
* **createTask():** creates a new task while checking it’s requirements, ensuring the labour required doesn’t exceed the system’s current workforce.
* **assignTask():** Assigns task to available workforce. If the workforce is available but currently busy with some other tasks in the system then the task is enqueued to the task\_queue.
* **viewWorkForce()**: This function prints all the available workforce with their current status.
* **checkQueue()**: checks the tasks\_queue to see if any tasks can be started based on the available resources.
* **checkStatus()**:Monitors the current status of all workers, robots, tasks and products.
* **completeTask()**: Marks a given task complete, updating its status and calling the **checkQueue()** method to start any task enqueued.
* **addProduct()**: adds a product to the system with its required steps (ensuring that all the tasks including in its steps are then added to the tasks dictionary with its label)

**Testing and Error-Handling Procedures**

The system relies on error-handling mechanisms to ensure smooth operation and robustness. Here’s a breakdown of the approach to testing and error handling:

**Task Creation and Resource Availability**

When creating tasks, the system checks if there are enough resources (workers and robots) available before the task is created. This is done with the following checks:

* If the number of available workers and robots is insufficient, the task creation process is aborted, and an error message is displayed.
* If the task ID already exists, an error message is shown to prevent duplicates.

**Task Assignment**

When assigning tasks, the system checks if there are enough idle workers and robots available. If sufficient resources are available, the task is started, and the statuses of the workers and robots are updated. If not enough resources are available, the task is added to the queue for later execution.

* Error handling ensures that tasks cannot be assigned if the resources are unavailable.
* If a task is already in progress or has been completed, the system ensures that no further actions can be performed on it.

**Task Completion**

Once a task is complete, the system checks if the task exists and if it is in the "Started" state. If the task is not started or does not exist, an error message is displayed. This ensures that incomplete or non-existent tasks are not marked as completed.

* After completing a task, the system updates the statuses of the workers and robots involved, and re-checks the task queue for any pending tasks that can now be started.

**Justifications for Design Choices**

**Data Structures**

The use of dictionaries (robots, workers, tasks, and products) is a design choice that supports fast lookups and updates. A dictionary allows for quick checking of resource availability (e.g., checking if a robot or worker exists, and retrieving or updating its status). The dictionary structure is ideal for managing large datasets of robots, workers, and tasks, ensuring efficient resource allocation and task management.

The task\_queue is implemented as a list, which makes it easy to append tasks that are waiting for resources and check tasks for execution.

**Functions and Modularity**

The system is designed to be modular, with each core operation encapsulated in a function. This modularity makes the system easy to maintain and extend. For instance, if new features (such as additional resource types or task management capabilities) are needed, they can be added as new functions without affecting the rest of the codebase.

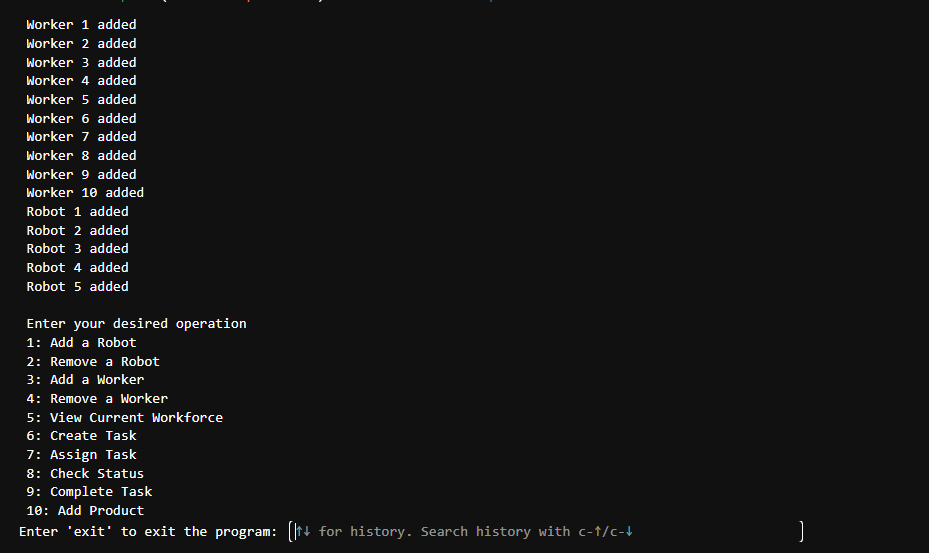
The use of functions also helps in improving code readability and reusability. Each function has a clear responsibility, making the system more maintainable and testable.

**Queue Management**

The task queue mechanism is a critical feature in this system. It ensures that tasks are not abandoned if resources are unavailable. Instead, tasks are added to the queue and executed when sufficient resources become available. This approach improves resource utilization and ensures that the system can handle variable workloads without overloading the available resources.

**Screenshots Demonstrating System in Action**

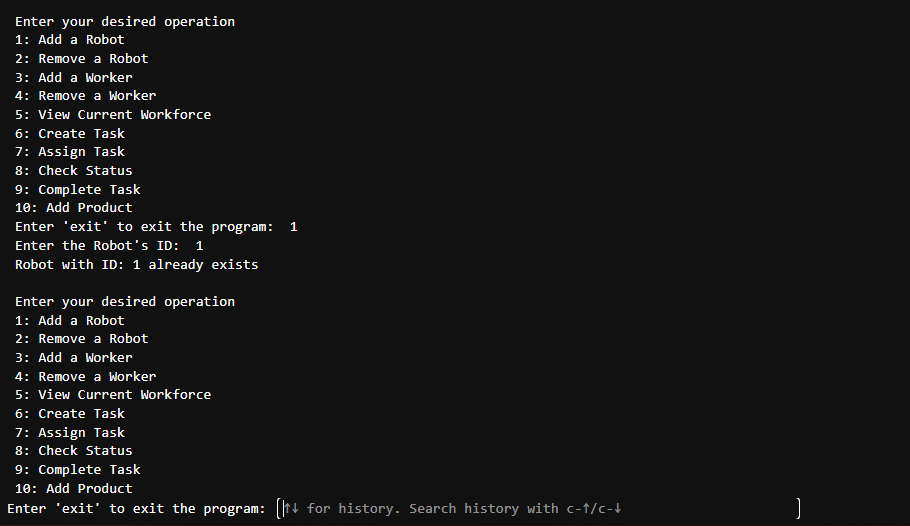
1. **Program Initialization**



The system initializes with Pre-Set number of workforce and displays the operations menue to the user for the user to input this desired operation.

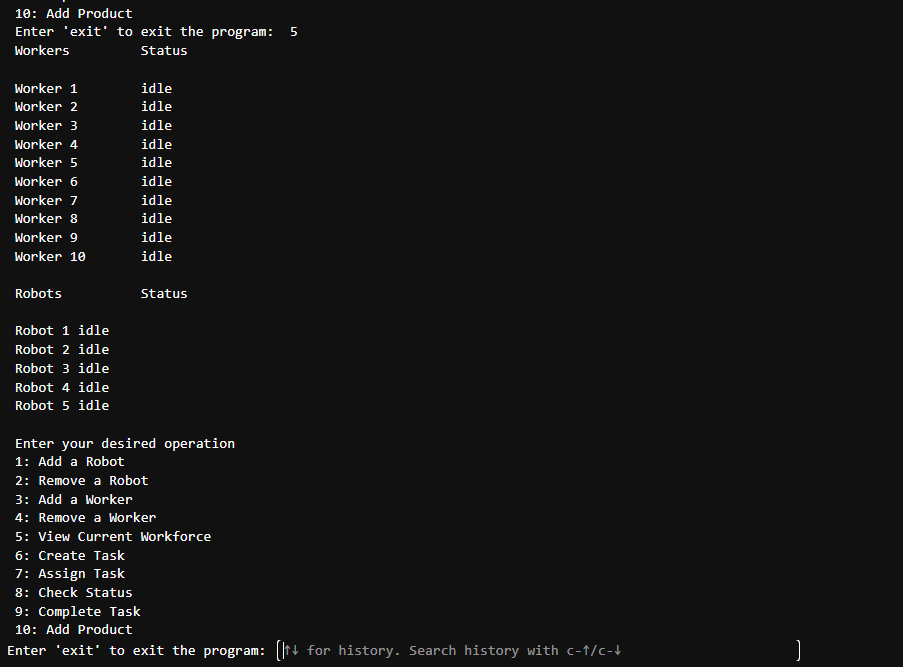
To exit the flow of execution the user simply inputs “exit”.

1. **Adding Labour to the system**



The system utilizes the addRobot() / addWorker() method to add new labour to the system. It also handles the uniqueness of the labour, ensuring only unique IDs are present in the system to avoid duplicates.

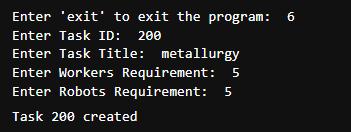
1. **Current Work Force**



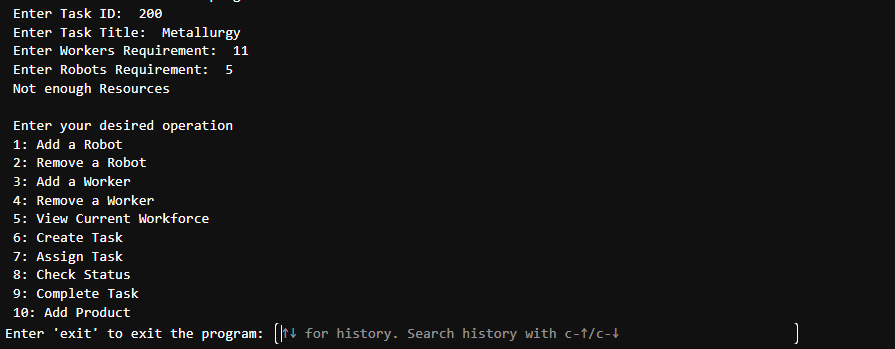
Upon accessing the view Current WorkForce method, the system displays the entire workforce with their respective status to the user.

1. **Task** **Creation**

The system accesses the create task method to create a task with a unique ID and requirements.

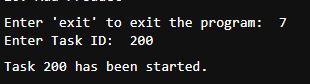


The System efficiently handles the test-case where the labour required exceeds the system’s total labour.



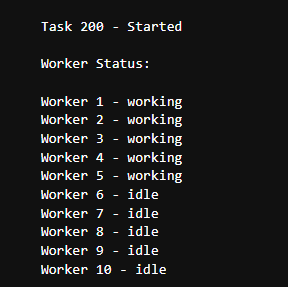
1. **Task Assignment**

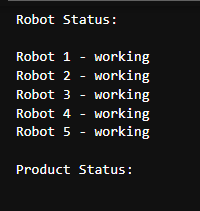
The system utilizes the assignTask method to assign the task to available labour, it takes the task ID as a parameter and then searches for that Task in the tasks dictionary.



1. **System Monitoring**

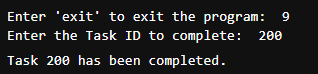
The checkStatus method is used to monitor and summarize the entire system.





1. **Task Completion**

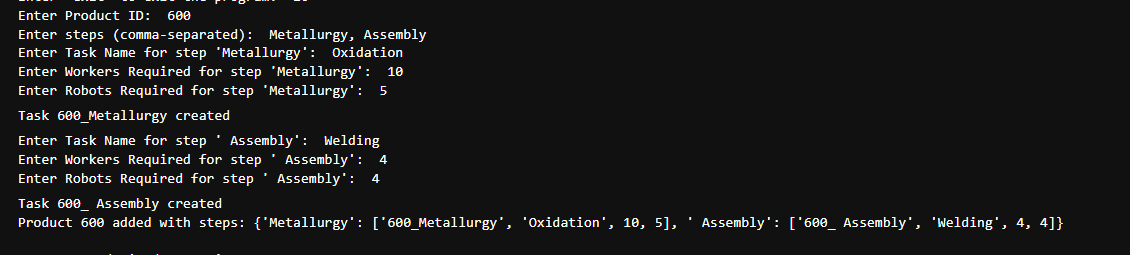
As observed completeTask method completes an existing started task and displays the output.



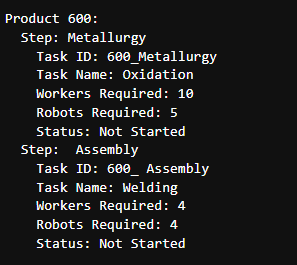
1. **Product Creation**

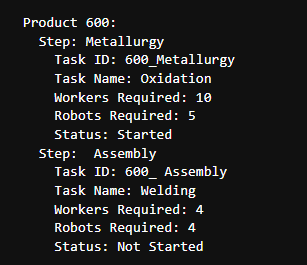
addProduct method takes product ID and steps as parameters and creates a product.

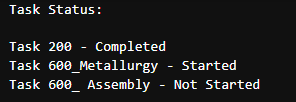
Then it adds these steps as tasks to the task dictionary with its respective label.



1. **Product Monitoring**

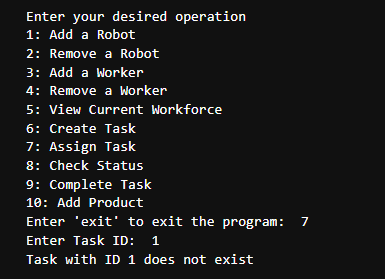






1. **Non-Existent Task case**

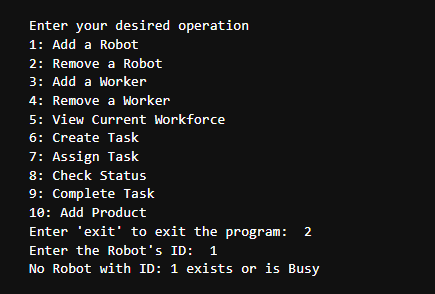
The system efficiently handles the case where a task that doesn’t exist is prompted to start. It Displays an error message and continues with flow of execution



1. **Deletion of Busy or Non-Existent Labour**

The system efficiently handles the exception where a non existing worker or robot is prompted to be deleted. The system displays an error message and continues with the execution flow.

The system also handles the exception where a non idle status worker or robot is prompted to be deleted and displays an error message following with the flow of execution.

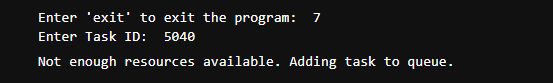


1. **Queue Utilization**

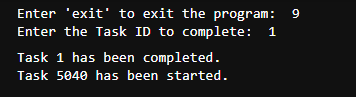
When a new task is created, the system checks if there are enough workers and robots available to assign to the task.

If there are sufficient idle workers and robots, the task is assigned, and it is marked as "Started."

If there are not enough resources available, the task is added to the task\_queue, where it waits for available resources.



The system starts the task as soon as the labour required by it is available.



**Summary**

In conclusion, the Robotic Cell Management System provides an efficient solution for managing robots, workers, and tasks in a production environment. The use of procedural programming concepts ensures that the system is flexible, easy to maintain, and scalable for larger operations. The queue management system and error handling mechanisms ensure smooth operation even in situations where resources are limited.