#### **Introdução**

O projeto envolve a simulação de uma fábrica de montagem de robôs. O sistema é organizado em esteiras, cada uma responsável por diferentes estágios da produção e montagem. Utilizamos programação concorrente para simular a produção contínua de peças e montagem dos robôs em paralelo. A aplicação utiliza Java e faz uso de threads para realizar operações simultâneas.

#### **Estrutura do Projeto**

1. **FabricaRobot.java**: Ponto de entrada principal que inicializa as threads de produção e montagem.
2. **GrupoEsteira1.java, GrupoEsteira2.java, GrupoEsteira3.java**: Implementações das esteiras responsáveis por diferentes estágios de produção.
3. **MembroRobo.java, ParteRobo.java, Robo.java**: Modelos que representam os componentes do robô e o robô completo.
4. **Logger.java**: Classe utilitária para salvar logs em um arquivo .txt.

### **Motivo das Escolhas das Funções Utilizadas**

#### **Uso de AtomicInteger**

**Motivo**: A classe AtomicInteger é usada para garantir operações atômicas em variáveis inteiras. Em um ambiente de programação concorrente, onde várias threads podem acessar e modificar uma variável simultaneamente, é crucial garantir que essas operações sejam atômicas para evitar condições de corrida e inconsistências de dados.

**Detalhamento**:

* **Operações Atômicas**: AtomicInteger fornece métodos como incrementAndGet() e getAndIncrement(), que são operações atômicas, garantindo que a variável seja incrementada de maneira segura, mesmo quando acessada por múltiplas threads simultaneamente.
* **Eficiência**: Utilizar AtomicInteger é mais eficiente do que usar bloqueios (locks) para garantir a sincronização, pois evita overhead desnecessário associado à sincronização explícita.
* **Simplicidade**: O uso de AtomicInteger simplifica o código, tornando-o mais legível e menos propenso a erros em comparação com a implementação manual de sincronização.

#### **Uso de BlockingQueue**

**Motivo**: A interface BlockingQueue é utilizada para implementar filas que suportam operações que esperam até que a fila seja não vazia ao recuperar um elemento, e que esperam até que a fila tenha espaço disponível ao inserir um elemento.

**Detalhamento**:

* **Sincronização**: BlockingQueue lida com a sincronização internamente, o que facilita a implementação de padrões de produtores-consumidores, onde múltiplas threads produtoras adicionam elementos à fila e múltiplas threads consumidoras removem elementos da fila.
* **Operações Bloqueantes**: Métodos como put() e take() são bloqueantes, o que significa que se a fila estiver cheia, a operação de inserção (put()) aguardará até que haja espaço disponível, e se a fila estiver vazia, a operação de remoção (take()) aguardará até que haja elementos disponíveis. Isso simplifica a coordenação entre threads produtoras e consumidoras.
* **Robustez**: BlockingQueue é uma estrutura robusta para situações onde a comunicação entre threads precisa ser coordenada de forma eficiente e segura, evitando a necessidade de implementar mecanismos de sincronização manualmente.

#### **Uso de Runnable e Thread**

**Motivo**: A interface Runnable e a classe Thread são utilizadas para permitir a execução de código em paralelo em diferentes threads.

**Detalhamento**:

* **Runnable**: A interface Runnable permite definir a tarefa que será executada em uma thread separada. Isso é útil para encapsular a lógica de produção e montagem de peças em unidades de trabalho distintas que podem ser executadas simultaneamente.
* **Thread**: A classe Thread é usada para iniciar a execução das tarefas definidas por Runnable. Cada esteira (grupo de produção) é executada em sua própria thread, permitindo que as operações de produção e montagem ocorram em paralelo.
* **Concorrência**: Utilizar threads separadas para cada grupo de produção permite que a fábrica funcione de maneira eficiente, com diferentes partes do processo sendo executadas simultaneamente sem bloqueios desnecessários.
* **Escalabilidade**: A implementação baseada em threads facilita a escalabilidade do sistema. Novas esteiras ou grupos de produção podem ser adicionados simplesmente iniciando novas threads, sem a necessidade de alterar significativamente a lógica existente.

#### **Uso de BlockingQueue nas Esteiras**

**Motivo**: A utilização de BlockingQueue nas esteiras (esteira1, esteira2, esteira3) é essencial para garantir a coordenação adequada entre as etapas de produção e montagem.

**Detalhamento**:

* **Coordenação**: As filas (BlockingQueue) garantem que as peças produzidas sejam armazenadas até que estejam prontas para serem consumidas pela próxima etapa do processo. Isso evita problemas de sincronização e coordenação manual entre as threads produtoras e consumidoras.
* **Thread-Safe**: BlockingQueue é segura para uso com múltiplas threads, eliminando a necessidade de implementar mecanismos de sincronização adicionais. Isso reduz a complexidade e o potencial de erros na implementação.
* **Bloqueio Automático**: As operações de inserção e remoção em BlockingQueue são bloqueantes, o que significa que uma thread produtora irá esperar até que haja espaço disponível na fila, e uma thread consumidora irá esperar até que haja elementos disponíveis na fila. Isso facilita a implementação de um fluxo de trabalho suave e contínuo.

### **Conclusão**

As escolhas de usar AtomicInteger, BlockingQueue, Runnable e Thread são fundamentadas na necessidade de garantir operações seguras e eficientes em um ambiente concorrente. Esses componentes simplificam a implementação, aumentam a robustez do sistema e garantem que a produção e montagem dos robôs ocorram de maneira coordenada e eficiente. A utilização de operações atômicas, filas bloqueantes e execução paralela permite que o sistema funcione de maneira suave, escalável e livre de condições de corrida ou inconsistências de dados.