Justificación para la selección de estrategias de ejecución de tareas:

El documento explora diferentes estrategias para la ejecución de tareas periódicas en Java, analizando sus ventajas y desventajas.

1. Esperas Temporizadas:

- **`Thread.sleep()`:** Se explica que usar `sleep()` para tareas periódicas tiene el problema de que el periodo real de ejecución siempre es mayor al deseado debido a que el tiempo de espera es relativo. Esto se debe a que `sleep()` detiene la ejecución del hilo durante un tiempo determinado, pero no garantiza que la tarea se reanudará exactamente después de ese tiempo.
- **`Object.wait()`:** Similar a `sleep()`, pero se usa en conjunto con la sincronización de objetos. También presenta el problema de que el periodo real puede ser mayor al deseado.

2. Tareas basadas en tiempos absolutos:

- **`SleepUntil()` (hipotético):** El documento introduce este concepto para ilustrar la idea de programar tareas para que se ejecuten en un momento específico en el futuro. Aunque teóricamente podría lograr un periodo medio igual al programado, se menciona que puede presentar "jitter" (fluctuaciones en el tiempo de ejecución) debido a la competencia por recursos del sistema.

3. Clase `Timer`:

- Se presenta la clase `Timer` como una solución eficiente y segura para programar tareas periódicas.
    - **Ventajas:**
        - Permite programar tareas para ejecución futura o periódica.
        - Diseño eficiente que puede manejar miles de tareas.
        - Seguro para acceso multi-hilo.
    - **Desventajas:**
        - No ofrece garantías de tiempo real estricto.
        - Si una tarea tarda más de lo previsto, puede retrasar las demás.

4. `schedule()` vs. `scheduleAtFixedRate()`:

- Se comparan los métodos `schedule()` y `scheduleAtFixedRate()` de la clase `Timer`:
    - **`schedule()`:** Programa tareas con un retraso relativo al tiempo de inicio. Los retrasos se acumulan, lo que puede llevar a desviaciones significativas en el tiempo de ejecución.
    - **`scheduleAtFixedRate()`:** Programa tareas en instantes fijos. Los retrasos no se acumulan, lo que lo hace más adecuado para tareas que requieren precisión en el periodo de ejecución.

5. Clase `TimerTask`:

- Se describe la clase abstracta `TimerTask` como la base para definir tareas que se ejecutarán con `Timer`.

- Se destaca el método `scheduledExecutionTime()` para obtener el tiempo de ejecución programado y controlar posibles retrasos.

Conclusión:

El documento justifica la selección de la clase `Timer` con `scheduleAtFixedRate()` como la estrategia más adecuada para la ejecución de tareas periódicas en Java cuando se necesita un periodo de ejecución preciso y se manejan múltiples tareas. Se enfatiza la importancia de elegir la estrategia correcta según los requisitos de la aplicación, considerando factores como la precisión del periodo, la cantidad de tareas y la posibilidad de retrasos.

```
import java.util.Timer;
import java.util.TimerTask;


public class TareaPeriodica {


    public static void main(String[] args) {
        // Creamos una instancia de la clase Timer
        Timer timer = new Timer("MiTareaPeriodica");


        // Creamos una instancia de la clase TimerTask que define la tarea a ejecutar
        TimerTask tarea = new TimerTask() {
            @Override
            public void run() {
                System.out.println("Tarea ejecutada a las: " + new java.util.Date());
            }
        };


        // Programamos la tarea para que se ejecute cada 5 segundos (5000 milisegundos)
        timer.scheduleAtFixedRate(tarea, 0, 5000);
    }
}
```