Diseño e Implementación de Jamazon

Equipo de Desarrollo
Roger Peña Pérez

January 31, 2026

1 Introducción

El proyecto *Jamazon* se diseñó con el objetivo de gestionar y optimizar la planificación de tareas en un entorno de comida rápida. El dominio de la aplicación abarca la administración temporal de recursos limitados tales como freidoras, hornos, cocineros, repartidores

El problema central que resuelve el software es la asignación eficiente de intervalos de tiempo para realizar tareas operativas (eventos). Cada evento, como "Preparar Pedido", requiere el bloqueo simultáneo de una cantidad específica de múltiples recursos durante una duración determinada. El sistema debe garantizar que, en ningún momento del intervalo propuesto, la demanda acumulada de un recurso supere su capacidad máxima instalada. Además, se manejan restricciones de dependencia jerárquica; por ejemplo, el uso de un equipo informático bloquea automáticamente una cuota del recurso "Electricidad" y "Internet".

La aplicacion es posible de sugerir al usuario el momento más próximo posible para iniciar una tarea cuando los recursos están ocupados, calculando "huecos" disponibles en la agenda que satisfagan todas las restricciones de recursos simultáneamente.

2 Arquitectura

2.1 Estructura y Decisiones de Diseño

Para garantizar la mantenibilidad y la escalabilidad del sistema, se optó por una arquitectura modular que desacopla estrictamente la lógica de negocio de la interfaz de usuario. Se decidió utilizar archivos JSON como mecanismo de persistencia de datos en lugar de una base de datos relacional. Esta decisión se tomó para simplificar la portabilidad del proyecto y eliminar la necesidad de servicios externos corriendo en segundo plano, dado que el volumen de datos esperado para un local pequeño es manejable en memoria.

La estructura se divide claramente en:

1. **main.py:** Punto de entrada que orquesta el ciclo de vida de la aplicación.

2. **modules (Backend):** Contiene la lógica dura y algoritmos.

3. **guicore (Frontend):** Gestiona la interacción con la librería gráfica, consumiendo los servicios del backend sin conocer su implementación interna.

2.2 Entorno de Desarrollo

La implementación se ha llevado a cabo en un entorno Linux, utilizando Git y VSCode, se ha testeado la compatibilidad en sistemas Arch Linux, Ubuntu y Windows 11.

3 Lógica de módulos

3.1 utils.py

En este módulo solamente se encuentran las funciones útiles que se usan en todo el proyecto. Fue creado con el objetivo de evitar código repetido en el proyecto y tener más organizadas las funciones individuales que se usan en este.

- **log:** Esta función redirige los datos de entrada a un archivo, es utilizada para debug.
- **getsourcesdependency:** Esta función implementa un DFS el cual será explicado en siguientes secciones.
- **CheckISODate:** Retorna True si el ISODate de entrada está correcto.
- **getsavedjson:** Devuelve un diccionario con los datos de un JSON especificado.
- **readfile:** Carga un archivo directamente en memoria.
- **tominute:** Convierte una fecha en su correspondiente en minutos.
- **addtodict:** Esta función agrega a un diccionario recursivamente un dato (más detalles en utils.py).
- **eventoptionlabel:** Genera un pequeño texto descriptivo a partir de un evento.
- **buildeventoptionlabels:** Genera una lista con los datos retornados por la función anterior para cada evento.
- **formateventinfo:** Genera la información completa de un evento.

3.2 iohandler.py

Aquí se encuentra la clase BasicHandler, la cual se encarga de la mayoría de IO en archivos JSON.

- Guarda en un JSON.
- Lee datos de un JSON.
- Extrae la extensión de un archivo.

- Convierte un JSON en dict.
- Convierte un dict en JSON.

3.3 gvar.py

Este módulo, su principal función es cargar los datos de JSON y se encarga de contener los datos globales para todo el programa.

3.4 events.py

Se encuentra la clase Event; ésta hereda de BasicHandler y se encarga de guardar en ella una tarea específica, guardando los datos de esta. Además, implementa transformaciones a los datos:

- Convierte una tarea a dict.
- Convierte una tarea a str.
- Extrae qué recursos no se usan con uno especificado.

4 calendar.py

Como en este módulo hay mucho que explicar se dividió en varios puntos. Aquí se encuentra la clase Calendar, la que se encarga de:

- Agregar tareas. Uno de los retos principales enfrentados durante el desarrollo fue el manejo de rangos de tiempo basados en timestamps de Unix, los cuales generan números muy grandes (e.g., 167888888). Crear un arreglo de ese tamaño para el Segment Tree consumiría demasiada memoria. Para solucionar esto, se implementó dentro de esta lógica una técnica de **Compresión de Coordenadas**. El algoritmo identifica los puntos de interés (inicios y finales de eventos existentes) y los mapea a índices secuenciales pequeños (0, 1, 2...), permitiendo construir un árbol eficiente que solo considera los momentos donde el estado del sistema cambia, ignorando los periodos "vacíos" intermedios
- Ordenar las fechas.
- Eliminar tareas viejas al iniciar el sistema.

4.1 add_event

En el momento de hacer esta función se encontró el problema de que los rangos de tiempo podían ser muy grandes, por lo que se necesitó implementar una compresión de coordenadas para que (R - L) sea más pequeño para poder verificar más rápidamente si en ese rango se puede agregar la nueva tarea.

4.2 suggest_brute_lr

Esta función implementa a fuerza bruta la lógica detrás de sugerir un rango de fechas al usuario. Lo que hace es que por cada uno de los eventos...

4.3 check_available

Verifica si un rango de fecha dado está disponible para asignar recursos. Esta función revisa si, para el intervalo de tiempo especificado, la cantidad de recursos solicitados no excede la capacidad total disponible, asegurando que no existan conflictos con otras tareas ya programadas.

4.4 remove

Esta función elimina una tarea específica del calendario, liberando los recursos que estaban asignados a dicha tarea y actualizando la estructura de datos para reflejar la eliminación.

4.5 sort

Esta función ordena las fechas (claves del diccionario) por tiempo de inicio.

4.6 remove_old_events

Esta función se encarga de recorrer todas las tareas registradas y eliminar aquellas cuya fecha de finalización es anterior a la fecha y hora actual del sistema. Esto permite mantener el calendario limpio y optimizado. , reduciendo el tamaño del árbol de segmentos necesario para futuros cálculos.

5 Retos y Soluciones

Durante el ciclo de desarrollo se enfrentaron varios obstáculos técnicos que definieron la arquitectura final:

- **Validación de Dependencias:** Gestionar recursos que dependen de otros (ej: Internet requiere Electricidad) complicó la lógica de verificación. Se resolvió implementando una búsqueda en profundidad (DFS) en `utils.py` que recorre recursivamente el grafo de dependencias para asegurar que todos los recursos base estén disponibles antes de confirmar una tarea.
- **Persistencia de Datos:** Mantener la consistencia entre los archivos JSON y los objetos en memoria fue complejo. Se estandarizaron métodos en `iohandler.py` para serializar y deserializar objetos automáticamente, evitando la corrupción de datos al cerrar el programa inesperadamente.

6 gui_core

En esta carpeta se encapsulan todas las clases que heredan de `CTkToplevel` o componentes de la biblioteca gráfica, actuando como puente entre la interacción del usuario y la lógica.

6.1 EventCreator

Esta clase gestiona la interfaz para la definición de **tipos de eventos** en el sistema. Permite al usuario especificar el nombre de una nueva actividad operativa (ej. "Freír Papas") y asociarla con una lista de recursos necesarios para su ejecución.

6.2 TaskCreator

Esta clase implementa la interfaz de **agendamiento**. Proporciona campos para ingresar la fecha y hora de inicio y fin, así como un selector de las actividades definidas previamente. Interactúa directamente con el módulo `calendar` para verificar la disponibilidad de recursos en el intervalo solicitado y confirma la creación de la instancia del evento en el cronograma.

6.3 ResAdder

Clase responsable de la gestión del inventario de recursos base. Permite al administrador registrar nuevos recursos en el sistema (ej. "Freidora", "Personal", "Electricidad"), definiendo su capacidad máxima inicial. Actualiza el estructura de datos de recursos globales, haciéndolos disponibles para ser requeridos por futuras tareas.

6.4 EventShower

Visualizador de eventos activos y programados. Genera una lista dinámica de botones que representan cada tarea en la cola de ejecución del sistema, mostrando su ID, nombre y el intervalo de tiempo asignado. Al interactuar con un elemento, despliega información detallada sobre el evento específico, facilitando el monitoreo de las operaciones del local en tiempo real.

6.5 ResourceShower

Interfaz de consulta de recursos. Muestra un listado de todos los recursos registrados en el sistema junto con su cantidad total disponible. Permite a los administradores tener una visión clara de la capacidad instalada del local y verificar las propiedades de cada recurso.

6.6 TaskRemover

Herramienta de gestión para la cancelación de tareas. Proporciona una interfaz donde se listan los eventos programados en un menú desplegable, permitiendo al usuario seleccionar y eliminar una tarea específica. Al confirmar la acción, invoca los métodos de limpieza del backend para liberar los recursos reservados por dicho evento en el *Segment Tree*, actualizando inmediatamente la disponibilidad del sistema.

References

[1] Python Software Foundation. *Python 3.12.1 Documentation*. Disponible en: `https://docs.python.org/3/`.

[2] Tom Schimansky. *CustomTkinter: Official Documentation And Tutorial*. Disponible en: `https://customtkinter.tomschimansky.com/`.

[3] CP-Algorithms. *CP-Algorithms: Competitive Programming Library*. Disponible en: `https://cp-algorithms.com/`.