UNIVERSIDAD DE SAN CARLOS DE GUATEMALA
FACULTAD DE INGENIERIA
ESCUELA DE CIENCIAS Y SISTEMAS
LABORATORIO ESTRUCTURA DE DATOS

Manual Técnico Fase 1

Pixel Print Studio

Nombre: Sergio Joel Rodas Valdez
Carné: 202200271
Sección: C

Instrucción

"Pixel Print Studio", dedicada a la impresión de figuras de pixel art, enfrenta el desafío de optimizar sus procesos operativos ante el crecimiento de su clientela. La solución propuesta es el desarrollo de una aplicación que, aplicando principios de ingeniería de sistemas, utiliza estructuras de datos como listas simples, listas doblemente enlazadas y colas para mejorar la eficiencia en la recepción y producción de pedidos. Este enfoque no solo busca agilizar las operaciones sino también ofrecer una representación visual de los procesos a través de herramientas como Graphviz, facilitando una gestión más efectiva y transparente de los flujos de trabajo. Este proyecto destaca la importancia crítica de las estructuras de datos en la optimización y automatización de procesos en el ámbito empresarial.

Manual Técnico

Requisitos del Sistema

Software y Entorno de Desarrollo:

- Sistema operativo Windows 64
- Compilador gnu fortran 12.0.3
- Visual Studio Code -> Extensiones: Fortran Moderno

- Para la lectura del json se utilizó json-fortran: Es una biblioteca para Fortran que permite la serialización y deserialización de datos JSON. Facilita la lectura y escritura de datos JSON, permitiendo a las aplicaciones Fortran manipular fácilmente este formato de intercambio de datos ampliamente utilizado.

- cmake versión 3.29.0-rc1

Para la lectura del Json se utilizó la biblioteca json-fortran mediante ella pude leer los datos y almacenarlos en variables para después hacer la llamada de encolar en cliente recepción a continuación se muestra código fuente.

```
call json%initialize()    ! Se inicializa el módulo JSON
call json%load(filename='data.json')  ! Se carga el archivo JSON llamado 'data.json'

! Se obtiene el tamaño del arreglo [] json
call json%info('',n_children=size)

call json%get_core(jsonc)              ! Se obtiene el núcleo JSON para acceder a sus funciones
call json%get('', p_list, found)

do j = 1, size                         ! Se inicia un bucle sobre el número de elementos en el
    call jsonc%get_child(p_list, j, p_cliente, found = found)  ! Se obtiene el i-ésimo hijo de

    nombre = ''
    id_cliente = 0
    img_g = 0
    img_p = 0

    ! Nombre
    call jsonc%get_child(p_cliente, 'nombre', p_atributos, found = found)  ! Se obtiene el valor
    if (found) then
        call jsonc%get(p_atributos, nombre)
    end if

    ! Para 'id'
    call jsonc%get_child(p_cliente, 'id', p_atributos, found = found)
    if (found) then
        call jsonc%get(p_atributos, id_cliente)
    end if

    ! Para 'img_g'
    call jsonc%get_child(p_cliente, 'img_g', p_atributos, found = found)
    if (found) then
        call jsonc%get(p_atributos, img_g)
    end if

    ! Para 'img_p'
    call jsonc%get_child(p_cliente, 'img_p', p_atributos, found = found)
    if (found) then
        call jsonc%get(p_atributos, img_p)
    end if

    call encolar(reception_tail, Client(id_cliente, trim(nombre), img_p, img_g, img_p, img_g))
end do
```

Es importante enfatizar de primero en definir como esta conformado este proyecto e ir mediante incisos explicando el flujo de este programa para que otro desarrollador se le haga mas legible y entendible el proyecto.

A) Una cola en estructura de datos se conoce el comportamiento en la primera en entrar la primera en salir y esta estructura es fundamenta en estructura de datos para comprender mejor el comportamiento de como de va manejando la memoria como también la creación de objetos mediante nodos y establecer nuestra cola, en esta primera parte establecimos una cola para el manejo de los clientes que ingresan a nuestro programa y a continuación definiremos como está compuesta para definir mejor esta estructura partimos de una clase de tipo cliente que tiene como atributos un identificador único, nombre y sus respectivas cantidades que el usuario desea imprimir, como también un nodo para la estructura de datos de cola, conteniendo un cliente y un puntero al siguiente nodo. Y por último definimos la cola de recepción con procesos básicos los cuales serán encolar, desencolar y uno adicional para verificar si todo se realizo correcto.

```
type :: Client
    integer :: id
    character(len=50) :: name
    integer :: img_p
    integer :: img_g
    integer :: copy_img_p , copy_img_g !copia de
end type

type :: NodoClient
    type(Client) :: data
    type(NodoClient), pointer :: next => null()
end type

!----- Cola Repecion
type :: Tail
    type(NodoClient), pointer :: head => null()
    type(NodoClient), pointer :: tail => null()
    contains
    procedure :: encolar
    procedure :: show
    procedure :: desencolar
end type
```

(Estructura de nuestra cola de recepción)

Operaciones principales

Para añadir nuevos clientes a la cola utilizamos una subrutina que se encarga de realizar un registro a la cola y mediante los punteros vamos dando un estructura a nuestra cola, para la parte de desencolar esto fue vital para eliminar un nodo o elemento de nuestra cola con el fin de pasar al siguiente paso que seria estar nuestro

cliente en una ventanilla, por ultimo realice una subrutina adicional para mostrar los datos del cliente para ver que cumpla las dos subrutinas anterior mente descritas

```
subroutine encolar(this, cliente)
    class(Tail), intent(inout) :: this
    type(Client), intent(in) :: cliente
    type(NodoClient), pointer :: tempClient
    allocate(tempClient)
    tempClient%data = cliente
    tempClient%next => null()

    if(associated(this%head)) then
        this%tail%next => tempClient
        this%tail => tempClient
    else
        this%head => tempClient
        this%tail => tempClient
    end if
end subroutine encolar
```

(Subrutina para agregar clientes a nuestra cola de recepción)

B) Después de gestionar nuestros clientes sigue el Paso numero 2 de nuestra aplicación realizar un lista simple para nuestra estructura de ventanilla y que a la vez, la lista contara con una pila para la agregación de imágenes, como primer punto se realizo una lista simple llamada VentanillaList donde primero declaramos un objeto de tipo Ventanilla que contiene como atributos su identificador, pila de imágenes, también definimos un puntero de tipo cliente para apuntar a memoria al cliente que esta en la ventanilla, esto para tener información del cliente para realizar otras acciones que describiré porterilmente y final mente definimos un Nodo que tiene la data de nuestra ventanilla y un apuntador next para definir el siguiente nodo.

```
!-------------- Ventanilla
type :: Ventanilla
    integer :: id
    integer :: id_cliente_asociado = 0
    integer :: paso = 0
    type(Client), pointer :: assing_client => null()
    type(PilaImagenes) :: pila_imgs
end type

type :: NodoVentanilla
    type(Ventanilla) :: dataV
    type(NodoVentanilla), pointer :: next => null()
end type

type :: VentanillaList
    type(NodoVentanilla), pointer :: head => null()
    contains
    procedure :: add_ventanilla
    procedure :: show_ventanilla
    !Asignar un cliente a Ventanilla
    !procedure :: assing_client
end type
```

(Estructura de la lista simple ventanilla)

C) Después de definir mi lista de ventanilla procedemos a realizar la estructura de una pila la característica de una pila es la ultima en entrar la primera en salir y mediante clases declaramos nuestra pila de imágenes, partimos del primer punto de crear un objeto imagen que tiene como principales atributos su id, y que tipo de imagen es como también al cliente que pertenece, después creé un nodo para ir generando nuestra pila, añadí un apuntador de tipo NodoImagen que apunte a next para ir creando nuestra estructura en mi pila definí un puntero top para hacer referencia a la ultima imagen que se generó, Es importante mencionar que se realizó métodos para la creación de nuestras ventanillas, como también otras acciones para la asignación de un cliente a una ventanilla y también la asignación de imágenes a nuestra ventanilla establecí cuantas imágenes grandes y pequeñas tiene el cliente y mediante subrutinas fuimos creando la estructura para que un cliente se pueda asignar a una ventanilla si está disponible y mediante la ejecución coy asignando imágenes correspondientes a cada cliente de manera que cada ventanilla trabaje de forma independiente.

```
! Objeto imagen
type :: Imagen
integer :: id_img
character(len = 1) :: type_img ! tipo imagen
integer :: id_client  ! id_client id que pert
end type

type :: NodoImagen
type(Imagen) :: data_img
type(NodoImagen), pointer :: next => null()
end type

type :: PilaImagenes
type(NodoImagen), pointer :: top => null()
integer :: contador = 0
contains
procedure :: push_img
procedure :: show_img
procedure :: clear_stack
end type
```

(Estructura de la pila de imágenes)

```
!-- Subrutinas Ventanillas (add, assing, show)
subroutine add_ventanilla(this, id)
    class(VentanillaList), intent(inout) :: this
    integer, intent(in) :: id
    type(NodoVentanilla), pointer :: tempV => null()

    allocate(tempV)
    tempV%dataV%id = id
    tempV%dataV%assing_client => null()
    tempV%next => this%head
    this%head => tempV
end subroutine add_ventanilla

!-- agregar una imagen a la pila
subroutine push_img(this, image)
    class(PilaImagenes), intent(inout) :: this
    type(Imagen), intent(in) :: image
    type(NodoImagen), pointer :: newImg

    allocate(newImg)
    newImg%data_img = image
    newImg%next => this%top
    this%top => newImg
    this%contador = this%contador + 1
end subroutine push_img
```

(Subrutinas para agregar ventanilla y imágenes)

D) Después de definir nuestras lista siempre para nuestra ventanilla y nuestra pila como también el comportamiento y relación entre ellas , es importante mencionar de una subrutina importante para saber el momento exacto donde entendemos que la ventanilla tiene toda nuestras imágenes y pasamos al siguiente punto, el cliente pasa a la lista de cliente de espera que es una lista de listas doblemente enlazada circular y las imágenes pasan a sus respectivas colas de impresión, esto después lo definiré mas específico como se realizó este proceso como también la estructuras de cada una. La subrutina analizar_vetanilla es la encargada de manejar dos pasos importantes crean nuestra lista doblemente enlazada circular y asignar nuestras imágenes a cada respectiva cola de impresión para ello fue importante el uso de establecer una condicional donde miramos si nuestras imagenes pequeñas y grandes están en 0, significa que ya han sido entregadas todas las imágenes de nuestra pila y ya esta listo el cliente para pasar al siguiente punto.

```
subroutine analizar_ventanilla(this, listaClientesEspera, colaImpresionP, colaImpresionG)
    type(VentanillaList), intent(inout) :: this
    type(NodoVentanilla), pointer :: tempVentanilla

    type(lista_lista_cliente_espera), intent(inout) :: listaClientesEspera
    type(cliente_espera) :: clienteTemp

    type(Cola_Impresion_P), intent(inout) :: colaImpresionP
    type(Cola_Impresion_G), intent(inout) :: colaImpresionG

    type(ImpresoraP) :: data_impresora_p
    type(ImpresoraG) :: data_impresora_g

    type(NodoImagen), pointer :: tempNodoImagen ! temporal nodo imagen

    !MENSAJE
    character(len = 200) :: mensaje1, mensaje2, mensajeCompleto

    tempVentanilla => this%head
    do while(associated(tempVentanilla))
        if(associated(tempVentanilla%dataV%assing_client))then
            if((tempVentanilla%dataV%assing_client%img_p == 0 ) .and. (tempVentanilla%dataV%assing_client%img_g == 0)) then
```

E) Para establecer la lista de lista doblemente enlazada circular fue importante saber como funcionada ya que esta estructura es un poco más difícil de entender que las demás estructuras esta lista se compone de un Nodo principal que en donde se crea nuestro cliente en espera, como también de un subNodo donde se almacena una imagen para realizar esta lista de lista establecí de primero una lista doblemente enlazada circular que su head apuntara al último Nodo “Tail” y el ultimo Nodo a la cabeza asi establecemos que sea circular después utilizamos punteros next (siguiente) prev (anterior), esta estructura esta compuesta de primero con un objeto llamado cliente_espera, como también un nodo_cliente_e que posee un puntero para referirme al siguiente nodo y un prev o anterior nodo como también un puntero de list_img_a para establecer un puntero que apunte a nuestra lista simple de imágenes.

```
type :: cliente_espera
    integer :: id_cliente_e
    character (len = 200) :: nombre_cliente
    integer :: img_g_e
    integer :: img_p_e
    integer :: img_g_copy = 0  ! sumar imagen grande
    integer :: img_p_copy = 0 ! sumar imagen pequeña
    integer :: id_ventanilla
    integer :: paso = 0
end type

! Nodo cliente en espera
type :: nodo_cliente_e
    type(cliente_espera) :: cliente_data_e
    type(nodo_cliente_e), pointer :: next => null()
    type(nodo_cliente_e), pointer :: prev => null()
    type(sub_nodo), pointer :: list_img_a => null()
end type

! Lista de Lista cliente espera
type :: lista_lista_cliente_espera
    type(nodo_cliente_e), pointer :: head => null()
    type(nodo_cliente_e), pointer :: tail => null()
contains
    procedure :: add_cliente_espera
    procedure :: show_cliente_espera
    procedure :: add_img_a_cliente
    procedure :: delete_cliente_en_espera
end type
```

(Estructura de nuestra lista de lista doble enlazada circular de clientes en espera)

Además, podemos notar en la imagen de arriba que contamos con procesos los cuales son *add_cliente_espera, add_img_a_cliente, delete_cliente_en_espera* los cuales son fundamentales para realizar la creación de un cliente en espera como también la eliminación al momento de realizar otra acción.

Otra estructura que es importante en nuestro flujo de aplicación son las dos colas de impresión una para imágenes pequeñas y otra para imágenes grandes cada una cuenta con una similar estructura lo único que cambia es el tipo de imagen, como también es importante mencionar que una imagen pequeña tarda 1 paso para imprimirse y asignarla a un cliente en espera y 2 pasos tarda una imagen grande para imprimirse y entregarle a su respectivo cliente que pertenece esa imagen. Para esto utilice como primer punto un objeto llamado ImpresoraP donde tiene como atributos id cliente, y el tipo de imagen, Como también un nodo donde se aguardará la información de nuestra imagen como también un puntero para establecer el siguiente nodo, después nuestra cola de impresionP que cuenta con dos punteros para hacer referencia a la cabeza y cola de la impresión. Es similar la

estructura para la cola de impresión G lo único que varia es el tipo que en este caso sería para una impresión Grande

```
type :: ImpresoraP
  integer :: id_cliente
  character(len = 1) :: tipo_imagen
end type

type :: NodoImpresionP
  type(ImpresoraP) :: impresoraP_data
  type(NodoImpresionP), pointer :: next => null()
end type

type :: Cola_Impresion_P
  type(NodoImpresionP), pointer :: head => null()
  type(NodoImpresionP), pointer :: tail => null()
  contains
  procedure :: encolar_imp_p
  procedure :: show_imp_p
end type
```

(Estructura de cola impresión P)

F) Ahora que ya hemos definido nuestras dos estructuras principales que es nuestra lista de listas y nuestras dos colas de impresión la ejecución se dará mediante una secuencia de pasos para 1 paso se agrega una imagen pequeña a cada cliente respectivo, sale de la cola de impresión P y ingresa al subnodo de un cliente en específico para saber a qué cliente pertenece cada imagen cuenta con id_cliente entonces recorremos nuestra estructura y encontramos el cliente y asignamos la imagen esto pasa para imágenes de tamaño pequeño para imágenes grandes se realizan dos ejecuciones para agregar una imagen grande a un cliente en específico para esto se creo dos rutinas principales una que es para signar imagen y la otra es la llamada de desencolar una imagen de la impresora P o impresora G y mediante la copia, antes de eliminar nuestro nodo asignamos esos valores a nuestra imagen que vamos a agregar y posteriormente desencolamos para quitar en memoria nuestra información todo esta parte la realiza nuestra subrutina add_img_a_cliente_espera que es la encargada de realizar estos pasos de manera correcta y desencolar cada respectiva cola y asignar imágenes a nuestros clientes en espera.

```
!Añadir imagen asociada cliente espera (desencolar - asingar imagen a cliente espera)
subroutine add_img_a_cliente_espera(this, print_tail_p, print_tail_g)
    class(lista_lista_cliente_espera), intent(inout) :: this

    class(Cola_Impresion_P), intent(inout) :: print_tail_p
    type(ImpresoraP) :: dataP

    class(Cola_Impresion_G), intent(inout) :: print_tail_g
    type(ImpresoraG) :: dataG

    type(nodo_cliente_e), pointer :: temp, aux

    character(len=100) :: mensaje1
    character(len=100) :: mensaje2

    temp => this%head
    contador_ejecutar = contador_ejecutar + 1
    print *, 'CONTADOR' , contador_ejecutar

    ! Desencolar imagen de impresora pequeña
    call desencolar_imp_p(print_tail_p, dataP)
  if(dataP%id_cliente /= -1)then
      call asignar_imagen_cliente(this, dataP%id_cliente, 'P')
      !MENSAJE   Se completa la impresión de una imagen p -- se le entrega al cliente 1
      write(mensaje1, '(">> SE COMPLETA LA IMPRESION DE UNA IMAGEN ""P"" SE LE ENTREGA AL CLIENTE ", I0)') &
      dataP%id_cliente

      print *, mensaje1
  end if

  ! Incrementar en  img_p_copy + 1 si ingresamos una imagen
  if (dataP%tipo_imagen == 'P')then
      do while(associated(temp))
          if(temp%cliente_data_e%id_cliente_e == dataP%id_cliente) then
              temp%cliente_data_e%img_p_copy = temp%cliente_data_e%img_p_copy + 1
              exit
          end if
          temp => temp%next
      end do
  end if
```

(Subrutina agregar imagen a cliente de espera)

El paso anterior descrito se hace consecutivamente hasta que un cliente llegue con todas sus imágenes. Cuando un cliente en espera obtiene todas sus imágenes pasa de estas en la lista de lista doble enlazada circular a ser un cliente atendido para esto definimos tres procesos los cuales son 1 eliminar el cliente de la lista de lista y 2 asignarlo a nuestra lista simple de clientes atendidos estas dos subrutinas, esta llamada, que verifica si el cliente posee todas sus imágenes entonces realiza la subrutina 1 y la subrutina 2

```
subroutine evaluar_cliente_espera(this, list_atendidos)
    type(lista_lista_cliente_espera), intent(inout) :: this
    type(list_finalizados), intent(inout) :: list_atendidos

    type(nodo_cliente_e), pointer :: currentCliente
    type(nodo_cliente_e), pointer :: nextCliente
    type(cliente_espera) :: data_cliente
    type(cliente_finalizado) :: data_atendido

    character (len = 100) :: mensaje1,  mensaje2
    character (len = 200) :: mensajeCompleto

    currentCliente => this%head

    if(.not. associated(currentCliente)) then
        return
    end if

    do while(associated(currentCliente))
        nextCliente => currentCliente%next
        if((currentCliente%cliente_data_e%img_g_e == currentCliente%cliente_data_e%img_g_copy) &
        .and. (currentCliente%cliente_data_e%img_p_e == currentCliente%cliente_data_e%img_p_copy))then

            call delete_cliente_en_espera(this, currentCliente%cliente_data_e%id_cliente_e, data_cliente)
            ! rellenar los datos para la lista de clientes atendidos
            data_atendido%id_cliente = data_cliente%id_cliente_e
            data_atendido%id_ventanilla = data_cliente%id_ventanilla
            data_atendido%nombre_cliente = data_cliente%nombre_cliente
            data_atendido%img_g_recividas = data_cliente%img_g_e
            data_atendido%img_p_recividas = data_cliente%img_p_e
            data_atendido%total_pasos = data_cliente%paso

            call add_list_f(list_atendidos, data_atendido)
            ! MENSAJE
            ! El cliente 1 ya posee todas sus imágenes impresas y sale de la empresa registrando
            !el tiempo total dentro de ella cantidad de pasos
            write(mensaje1, '(">> EL CLIENTE ", I0 " OBTIENE TODAS SUS IMAGENES IMPRESAS")') &
            data_atendido%id_cliente

            write(mensaje2, '("INGRESA A LA LISTA DE CLIENTES ATENDIDOS REALIZO: ", "(", I0,")", " PASOS")') &
            data_atendido%total_pasos
            mensajeCompleto = trim(mensaje1) // " ---- "// trim(mensaje2)
            print*, mensajeCompleto


        end if

        if(.not. associated(this%head)) then
            exit ! Salir del ciclo si nextCliente no está asociado, evitando el acceso a memoria inválida
        endif
```

(subrutina encargada de evaluar si el cliente en espera posee todas sus imágenes, elimina el cliente e ingresamos a lista de clientes atendidos)

Describiré como esta compuesta nuestra lista simple de clientes atendidos partimos de un tipo objeto que es cliente_finalizado que tiene como parámetros id_cliente, id_ventanilla, nombre_cliente, imágenes recibidas (pequeñas y grandes) como también el numero total de pasos que el cliente estuvo en la ejecución, también un tipo Nod_finalizado que contiene la data del cliente como también un puntero next para referirnos al siguiente nodo, la lista simple tiene un puntero para definir cual es su head

```
type :: cliente_finalizado
  integer :: id_cliente
  integer :: id_ventanilla
  character (len = 200) :: nombre_cliente
  integer :: img_p_recividas
  integer :: img_g_recividas
  integer :: total_pasos
end type


type :: Nodo_finalizado
  type(cliente_finalizado) :: data_finalizado
  type(Nodo_finalizado), pointer :: next => null()
end type

type :: list_finalizados
  type(Nodo_finalizado), pointer :: head => null()
  contains
  procedure :: add_list_f
  procedure :: order_img_G
  !procedure :: order_img_P
end type
! ------ end Lista de clientes atendidos
```

(Estructura de la lista simple clientes atendidos)

Conclusiones

- **Integración de Tecnologías Modernas:** La implementación de la biblioteca json-fortran demuestra la capacidad del proyecto para integrar tecnologías modernas de manejo de datos con sistemas tradicionales, permitiendo una gestión eficiente de la información de clientes y pedidos a través de formatos de intercambio de datos ampliamente utilizados. Esto no solo mejora la flexibilidad y escalabilidad del sistema, sino que también facilita la interoperabilidad con otras aplicaciones y sistemas.

- **Optimización de Procesos mediante Estructuras de Datos:** El uso de estructuras de datos como colas, listas simplemente y doblemente enlazadas, y pilas, ha permitido crear un sistema altamente eficiente para gestionar la recepción de clientes, el procesamiento de pedidos de impresión y la asignación de recursos en Pixel Print Studio. Este enfoque estructurado no solo mejora la eficiencia operativa al minimizar los tiempos de espera y optimizar el manejo de la memoria, sino que también facilita la escalabilidad del sistema para adaptarse a un mayor volumen de pedidos y complejidades de procesamiento en el futuro.

- **Claridad en la Estructuración del Proyecto:** La detallada definición de las estructuras de datos y la secuenciación lógica de operaciones subrayan la importancia de una arquitectura de proyecto clara y bien organizada. Esto no solo hace que el sistema sea más legible y mantenible, sino que también proporciona una base sólida para la expansión futura y la incorporación de nuevas funcionalidades.