



BACHILLERATO EN INGENIERIA EN SISTEMAS DE INFORMACION
DIVISION DE CIENCIAS EXACTAS, NATURALEZ Y TECNOLOGIA
ESCLUELA DE INFORMATICA
SISTEMAS OPERATIVOS

FELIPE ALVARADO ALTAMIRANO
BYRON QUESADA JIMÉNEZ
CRISMAN MENA GOMEZ
I CICLO 2013

Ascensores EIF212

Manual Técnico

Tabla de contenido

Introducción	4
Ámbito de Sistema	5
Equipo de Trabajo	6
Herramientas de Desarrollo	7
Entorno de Desarrollo Integrado (IDE).....	7
Lenguaje de Programación.....	8
Herramienta de Planificación.....	8
Herramienta de Subversión	8
Herramienta de Diseño Multimedia	8
Estructura de Directorios	9
Clases.....	9
Controles	9
Recursos	9
Formas.....	9
Referencias (References)	9
Lógica.....	10
Componentes	10
Puerta (ucPuerta.cs).....	10
Panel de Botones (ucPanel.cs)	10
Indicador de Posición (ucIndicador.cs).....	11
Proceso de Cola Grafica (ucEjecutando.cs)	12
Cola de Procesos Por Realizar (ucCola.cs)	13
DataGridView's (LOAP).....	14
Edificio (ucEdificio.cs).....	15
Clases.....	25
clFuente.....	25
clProceso	25
clSolicitud	25
clFunciones.....	25
Hilos.....	29

Uso de la clase Threading	29
Función Invoke	29
Encapsulación de Hilos.....	30
Algoritmo Base de Planificación.....	30

Introducción

Con el presente documento se pretende, detallar ampliamente las partes y procesos lógicos del sistema Ascensores EIF212, para ser utilizado en cualquier caso de mantenimiento, mejora o auditoría que se le quisiese dar.

Ascensores EIF212 es un proyecto de software pequeño, desarrollado durante el tercer ciclo de la Carrera de Ingeniería en Sistemas de Información, curso de Sistemas Operativos cursado el año 2013 en la Sede Regional Brunca de la Universidad Nacional de Costa Rica, el tutor de curso, Licenciado Luis Alberto Arias Víquez presenta la propuesta con el objetivo de dar a los alumnos un ejemplo de que es trabajar con procesos, planificarlos y llegar a un resultado óptimo por medio de la ingeniería de software.

Es un sistema que ejemplifica el funcionamiento de uno a tres edificios y las solicitudes de ascensores en diferentes pisos, además muestra un historial de las solicitudes que se han y se van realizando *.

El manejo de cada solicitud será llamado como proceso, los movimientos de los ascensores y el de abrir y cerrar las puertas es manejado por un hilo, es decir cada ascensor posee un vector de hilos de cuatro campos, y cada vez que se atienda una solicitud nacerá y morirá un subproceso *.

Posee una interface que manipula uno a uno cada edificio, sus ascensores y las solicitudes respectivas asociadas al ascensor seleccionado **.

Finalmente se pretende también con esta documentación brindar un soporte y una idea de lo tolerante que es el sistema a futuros cambios, para eso se brindara la tabla detallada de los niveles de cambio.

Tabla de detalle de los niveles de cambio

Esta tabla pretende dar una vista detallada de los niveles de cambio de código que se podrían realizar en Ascensores EIF212, esto sería el trabajo de una sola persona.

Nivel	Descripción	Duración	Código	Gráficos
Bajo	Cambio posible	1 hora	2 - 5 %	2 – 5 %
Medio	Cambio difícil	6 horas	5 – 15 %	5 – 10 %
Alto	Cambio muy difícil	10 a 15 horas	15 – 25 %	10%
Siguiente versión	El sistema cambiaría notoriamente.	15 a más horas	Más de 25 %	--

* Se detalla más en el desarrollo de este documento.

** Se detalla más en el manual de Usuario.

Ámbito de Sistema

En esta versión el sistema no va más allá de tres ascensores por pantalla, pero por la estructura que posee permite incrementar, si se desea claro, se tendría que realizar un cambio de **Bajo Nivel**, ya que esta creado por componente y cada edificio queda como una clase con todas sus características (se hablara más detalladamente en el desarrollo del documento en la sección de Componente Edificio).

El sistema permite únicamente una solicitud simultánea de jefe por edificio y no se puede devolver o cancelar una solicitud de ascensor una vez pedida.

No se puede detener el proceso de cada edificio por aparte, solo se permite detener todos los edificios a la vez, este cambio representaría un trabajo mucho menor que el **Bajo Nivel**.

Los ascensores no se intercambian procesos o solicitudes entre ellos, el ámbito del número de pisos es de planta baja a azotea, en total suman diez pisos, el número de pisos no se puede aumentar o disminuir y sí se quisiese modificar representaría un cambio de **Siguiente Versión** por los cambios gráficos que representan muchos.

No se permite modificar el tamaño de los edificios, cambio de **Siguiente Versión**.

...

Equipo de Trabajo

El Tutor del Sistema; Es el licenciado **Luis Alberto Arias Víquez**, el cual presentó la propuesta del proyecto, realizará la auditoria y calificación del proyecto.

El análisis y desarrollo; Estuvo a cargo de los desarrolladores:

Luis Felipe Alvarado Altamirano, encargado de análisis compactación del algoritmo de planificación, investigación de hilos, componente de la puerta he indicador, programación de cola de solicitudes, compactación del edificio y pruebas unitarias, implementación y depuración final del proyecto.

Byron Quesada Jiménez, encargado de análisis, diseño multimedia, desarrollo de funciones de planificación, programación de solicitudes en las colas, documentación de requerimientos y manual de usuario, organizador de sesiones de trabajo.

Crisman Andrey Mena Gómez, encargado de análisis, componente de panel, investigación de hilos, funciones del algoritmo de planificación, compactación del algoritmo de planificación, programación de solicitudes en las colas, documentación de requerimientos y manual de usuario.

En el proceso de análisis inicial y pseudocódigo del algoritmo base de planificación los tres analistas el día jueves 21 de febrero realizaron una tormenta de ideas para plantearse el proyecto, plan, tareas, métodos, herramientas y teorías a aplicar, en lo anterior se llegó a la conclusión de utilizar las herramientas detalladas en la sección de “Herramientas de Desarrollo”.

Herramientas de Desarrollo

Las herramientas que permitieron la creación total del proyecto se dividen por la tarea que realizaron durante el proceso de análisis, diseño e implementación del proyecto Ascensores EIF212, pertenecen al fabricante Microsoft y Adobe, en seguida se detallarán.

Entorno de Desarrollo Integrado (IDE)

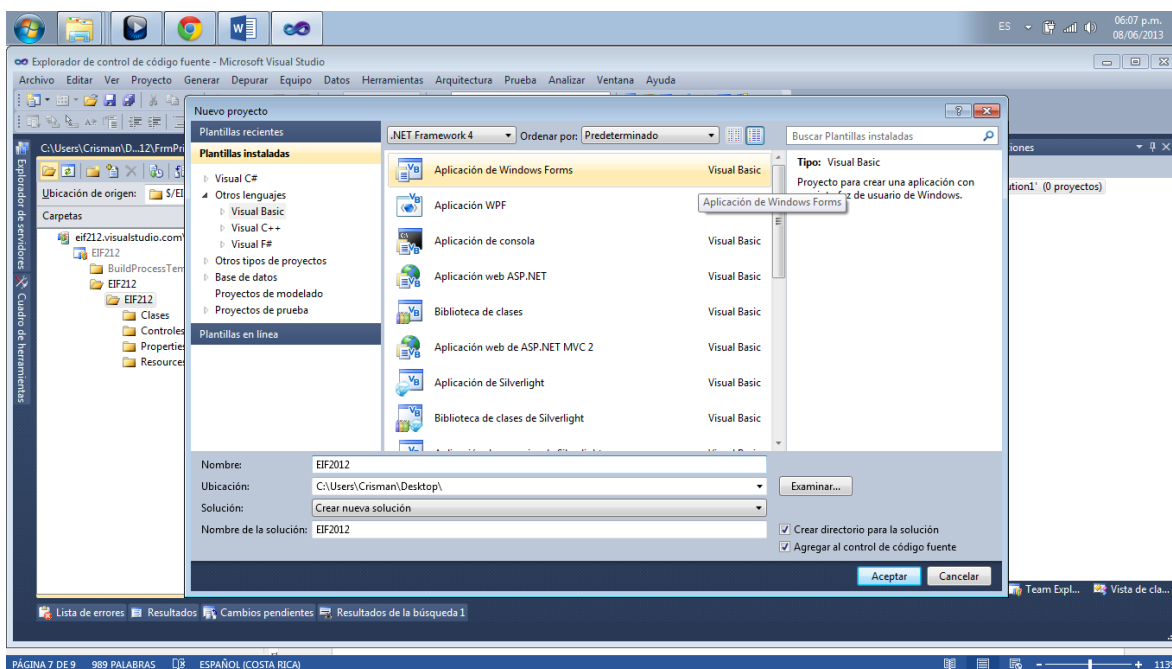
Corrector ortográfico de Análisis de código de Microsoft Visual Studio 2010

Partes de International CorrectSpell™ spelling correction system © 1993 por Lernout & Hauspie Speech Products N.V. Reservados todos los derechos.



Lenguaje de Programación

Visual Studio proporciona un (IDE), que soporta en su versión .NET Framework 4 múltiples lenguajes, el utilizado en este proyecto es Visual Basic, con la plantilla de Aplicación de Windows Forms.



Herramienta de Planificación

Microsoft proporciona muchas aplicaciones de oficina en su paquete de Microsoft Office, el utilizado en el proyecto es la herramienta de planificación de proyectos Project 2013.



Herramienta de Subversión

Visual Studio proporciona un (IDE), que soporta en su versión .NET Framework 4 los equipos de trabajo, manejados por una herramienta de Subversión que se conecta al servidor de Team Foundation Server, donde se puede crear un repositorio para el proyecto y así todos los integrantes pueden editar el proyecto sin problemas de compactación de código.



Herramienta de Diseño Multimedia

Adobe Systems Incorporated posee esta herramienta de simulación de taller de pintura para la creación y modificación de fotografías e imágenes, en el proyecto se le dará uso para crear y modificar las imágenes utilizadas en la interface gráfica.



Estructura de Directorios

Los directorios de código del proyecto se unirán a la plantilla de Windows Forms la cual agrego la solución y el proyecto EIF212, con los respectivos Name Spaces, las carpetas Properties, References y los archivos de Program.cs.

Ha esta plantilla se le agregan las carpetas:

Clases

Será la sección encargada de encapsular lo archivos de las diferentes clases del proyecto, las clases creadas son las cuatros que aparecen representadas en la imagen.

Controles

Será la sección encargada de encapsular lo archivos de los diferentes controles de usuario, vienen hacer clases pero con funciones programables y efectos gráficos, en total son 6 y cada uno posee su interface gráfica y código como se muestra en la expansión del control Edificio.

Recursos

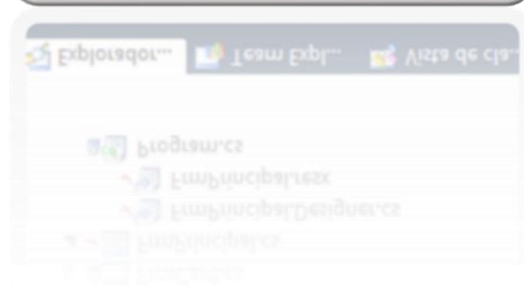
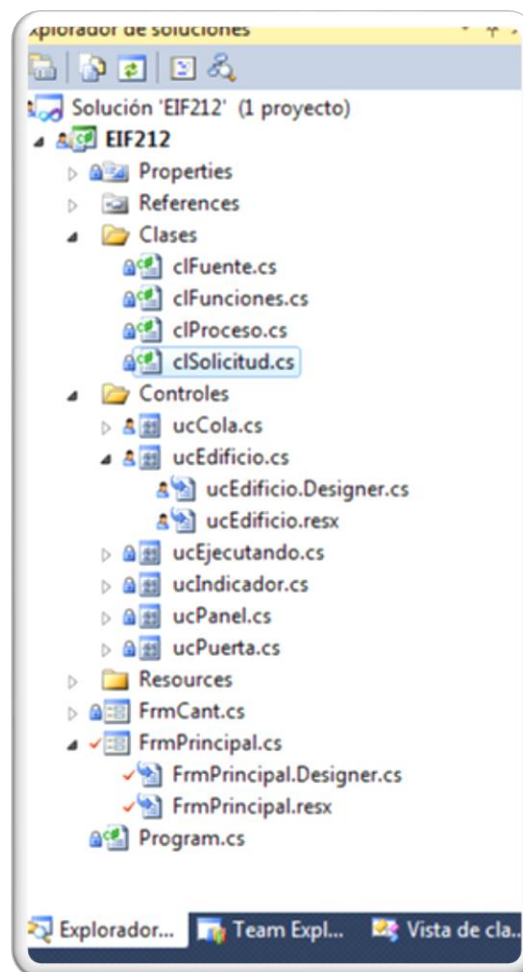
Será la sección encargada de encapsular los diferentes archivos multimedia como imágenes y música utilizados en los controles y Forma Principal.

Formas

Las formas son dos FrmCant y FrmPrincipal que se encargan de la presentación de datos al usuario, y ese encargan de la navegación y comando del sistema final.

Referencias (References)

Sección que maneja las librerías (.dll), que se ocupan para la plantilla de Windows Forms, además se encarga de las referencias a objetos de recursos y las clases.



Lógica

En esta sección del Manual Técnico de Ascensores EIF212, se detalla la lógica implantada de todo el proyecto de forma detallada y meticulosa, la forma en que da flujo el compilador y finalmente lo dará la CPU una vez compilado y ejecutado el sistema final.

Por lo extenso se adentrara en los temas lógicos más importantes y complejos del sistema, tomando en cuenta la estructura de directorios y finalmente un flujo de datos de solo el ascensor uno de un edificio uno, el funcionamiento total queda claro una vez comprendido este proceso.

Componentes

Los componentes son la parte esencial del sistema y representan las partes más segmentadas del problema de la planificación de las solicitudes de ascensores en diferentes pisos.

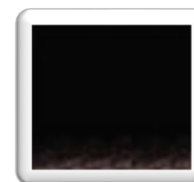
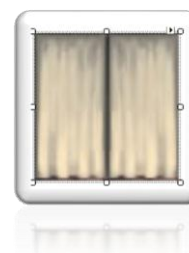
La creación del ascensor como entidad principal se divide en los componentes de:

Puerta (ucPuerta.cs)

Descripción general: Es una corrida de imágenes que reaccionan con una variable booleana y un hilo, estas se abren y cierran al ejecutar las funciones de Abrir () y Cerrar (), ambas se acceden al crear una instancia de la clase y llamar la función.

Lógica: La función principal es “función()”;

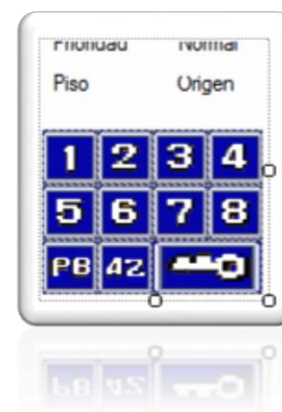
La cual va moviendo las imágenes una a una e intercalando un lapso de tiempo para que sea visible al usuario el efecto de abrir y cerrar, se activa con la variable bArrir, pues los movimientos de las imágenes son inversos.



Panel de Botones (ucPanel.cs)

Descripción general: Este componente trabaja sobre un solo edificio y ascensor. Su estructura funcional está dada por una serie de imágenes que hacen de botones, cambian de estado y generan la salida de un proceso hacia un edificio y ascensor específico, etiqueta un proceso con los datos de: ORIGEN y DESTINO, además verifica si es una solicitud de GEFE o no.

Lógica de Cargar: Al cargar a memoria y en la interfaz gráfica da uso de la clase fuente para darle a todos sus labels una fuente de 8 bits, por medio de las funciones “Panel_load” y “Fuente”, ubicadas al final del código, además traerá los apuntadores de edificio, cola y ascensor



correspondiente al panel.

Lógica de vinculación: las variables de “edificio, cola y ascensor” el componente queda apuntado a las regiones de memoria de las listas de solicitudes de un ascensor, sin afectar los hilos de otros ascensores (se detalla en el componente edificio).

Lógica de Creación de un Proceso: Se implementa en la función “actualOdestino (int x)”, Cuando se toca por primera vez un botón, será el origen y la segunda vez será el destino, para validar que no se solicite el mismo piso de destino se compara se usa la variable del parámetro “x” y “temporal”.

La primera vez no hay cierre de solicitud, la segunda sí, siempre y cuando el destino sea válido, afectando de la línea 14 a la 50, donde se asigna a la cola del ascensor de un edificio específico en la línea 15 el resto es para mostrar la información en los controles gráficos de las listas.

```
1 void actualOdestino(int x)
2 {
3
4     if (temporal != x)
5     {
6         temporal = x;
7         if (LabOrigeOdest.Text == "Origen")
8         {
9             LabOrigeOdest.Text = "Destino";
10            origen = x;
11        }
12        else
13        {
14            LabOrigeOdest.Text = "Origen"; // se termina la solicitud
15            edificio.fifo[asencensor].Add(new clProceso(origen,x,jefe)); // aca se agrega a la lista del acensor
16            if(jefe)
17                edificio.nuevaSolicitud(asencensor,"nSP " + origen.ToString()+"-"+x.ToString());
18            else
19                edificio.nuevaSolicitud(asencensor, "nS " + origen.ToString() + "-" + x.ToString());
20            if(jefe)
21                edificio.jefeSeMonto(false);
22            cola.update(edificio.fifo[asencensor]);
23
24            if (jefe)
25            {
26                LabPrioridad.Text = "Normal";
27                jefe = false;
28            }
29            origen = -1;
30            temporal = -1;
31        }
32    }
33 }
```

Indicador de Posición (ucIndicador.cs)

Descripción general: Este componente trabaja sobre cada piso y básicamente hace referencia de la posición del ascensor en el edificio. Su lógica no es tan complicada, solo consta de dos funcione que lo hacen cambiar de color. El color verde es por defecto y cuando cambia a rojo indica que el ascensor esta en ese piso.



Proceso de Cola Grafica (ucEjecutando.cs)

Descripción general: Este componente es parte de la cola (figura ucCola.cs, se detallara mas adelante), está compuesto por dos números (imágenes de recursos), el de la derecha es el piso de origen y el de la izquierda es el piso de destino, él reacciona dependiendo de los datos de los procesos de la cola de un ascensor, los estados que maneja son:





Solicitud nueva: Cada número cambia haciendo ejecutar la función “set (clProceso x)”, esta lee los enteros de origen y destino, agregando la imagen correspondiente; en caso de ser una solicitud de jefe reacciona diferente agregando números rojos quedando un resultado así:



Solicitud de origen completa: En caso de ser jefe se mantendría el 2 rojo solamente.



Solicitud terminada: Se borra el proceso de la cola de solicitudes por realizar.

Lógica: La función “set (clProceso)” es la principal: Habilitado y ejecutado, es un vector de las 9 imágenes de los numero, solo que uno con numero rojos  y otro con los blancos  ; x.origen y x.destino son los enteros del proceso y sirven de indise para cargar la imagen de los anteriores vectores. La ultima linea es para desactivar el origen por si ya fue ejecutado.

```
public void set(clProceso x)
{
    if (x.prioridad)
    {
        origen.Image = ejecutando[x.origen];
        destino.Image = ejecutando[x.destino];
    }
    else
    {
        origen.Image = habilitado[x.origen];
        destino.Image = habilitado[x.destino];
    }
    if (x.origenE)
        origen.Image = global::EIF212.Properties.Resources.menosicon;
}
```



Cola de Procesos Por Realizar (ucCola.cs)

Nota: Este componente no detalla todos los datos del proceso, ese trabajo lo llevaran a cabo los dataGridView's de información de BCP de cada solicitud (siguiente componente).

Descripción general: Este componente da uso del anterior para representar gráficamente los procesos o solicitudes a realizar.

Lógica:

Función “addItem (clProceso x)”:

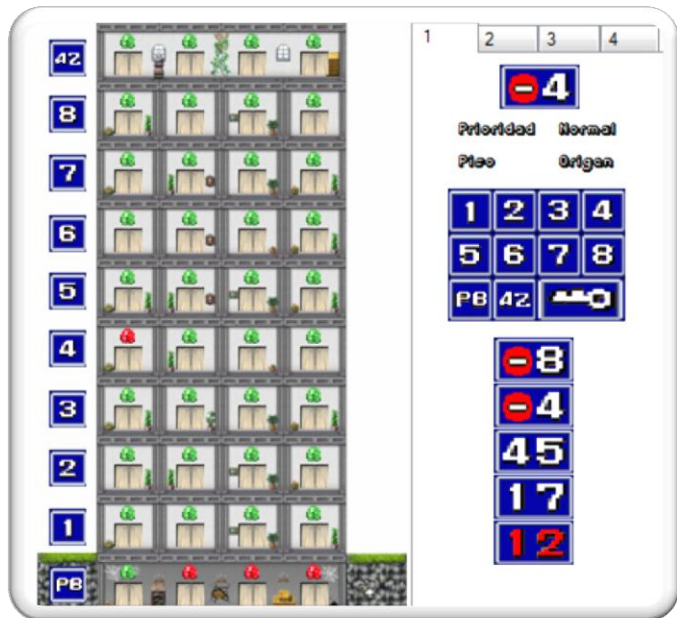
Es muy simple ya que lo único que hace es, en dos pasos, transformar el “clProceso x (parámetro)”, en un “ucEjecutando.cs (componente anterior)” y luego agregarlo al panel de la cola en una posición incrementada.

Función “update (List<clProceso> lista)”:

Esta limpia por completo el panel y agrega todos los procesos de lista (parámetro), esta función será usada en el componente Edificio.cs, por el ascensor consultado.

```
public void update(List<clProceso> lista)
{
    p = 0;
    panel1.Size = new System.Drawing.Size(panel1.Size.Width, 10)
    panel1.Controls.Clear();
    for (int i = 0; i < lista.Count; i++)
        addItem(lista.ElementAt(i));
}
```

Ejemplo gráfico: de como el componente (cola grafica de solicitudes por realizar) muestra los datos de la lista de procesos del ascensor 1 que contiene 4 solicitudes de prioridad baja y una de jefe, 2 con el origen ejecutado y 3 sin atender.

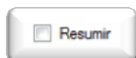


DataGridView's (LOAP)

LOAP: Lista de operaciones atendidas y pendientes.

Nota: Este componente no se encapsula en el Directorio, ya que su funcionalidad se adapta a las necesidades del proyecto, viene en la librería “System.Windows.Forms”, y su uso es para visualizar los BCP de las solicitudes y movimiento de los ascensores.

Descripción general: Es un componente muy conocido en la ingeniería así que no se detallara pero cabe rescatar que el presente se ve afectado por el “chckResumen”, gráficamente seria este:



En su evento “chckResumen_CheckedChanged(object sender, EventArgs e)”

Lo único que realiza es una validación que filtra el vector de solicitudes de forma que se generan las siguientes 2 salidas de datos, en ambos se genera observa el historial de solicitudes su orden de ejecución, porcentaje ejecutado.

Tabla 2

Tabla de simbología de las colas o lista de operaciones atendidas y pendientes

Solicitud	Descripción
nS	Entrada de nueva solicitud
nSP	Entrada de nueva con prioridad
eS 50%	Atendio la solicitud de origen (persona sube)
eS 100%	Atendio la solicitud de destino (persona baja)
aP	Indica el movimiento del ascensor a piso diferente

Ejemplo gráfico de Resumir activo e inactivo con las mismas colas de solicitudes.



Edificio (ucEdificio.cs)

Este componente absorbe todos los anteriores y es sin duda el componente lógico y grafico más importante del proyecto Ascensores EIF212 esta compactado en dos secciones:

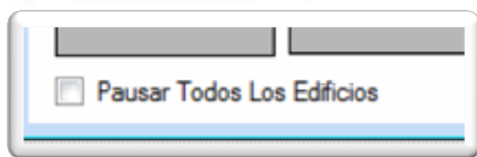
Sección Grafica:

Edificio: Es un panel con la imagen de un edificio de 10 pisos en formato de 8 bits (como fondo), los números de piso no son parte de la imagen, la imagen no posee indicadores ni puertas; al cargar un edificio el compilador genera dinámicamente esos componentes y crea las asignaciones a memoria respectivas de cada ascensor.

Panel de operaciones del ascensor: Controla las solicitudes nuevas, está ubicado en la parte derecha del edificio, y está integrado por la cola gráfica en su parte inferior, además posee cuatro pestañas que representan cada uno de los ascensores del edificio actual.

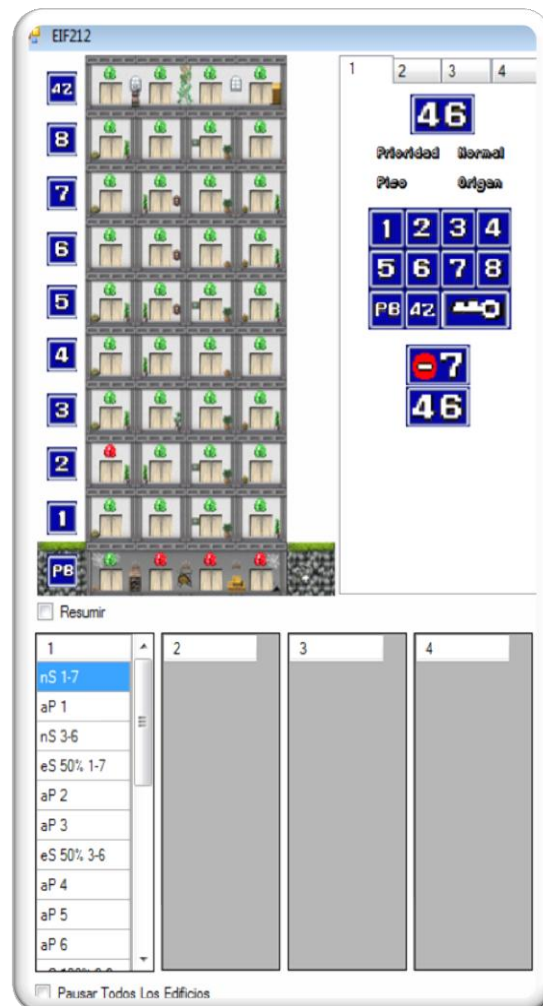
Lista de operaciones atendidas y pendientes: Son cuatro listas (una por ascensor), tienen el historial de todo lo realizado, permite dos vistas de datos (resumida y sin resumir), finalmente está ubicado en la parte inferior del edificio al que pertenece. (Detalle simbólico en [Tabla 2](#))

Checks: Resumir filtra las solicitudes para que no se muestren en el historial los datos de movimiento del edificio.



Restricciones:)

Pausar Todos Los Edificios detiene todos los hilos que se estén ejecutando en ese instante (detalle en [Funciones de](#)



Sección Lógica:

Describe el funcionamiento lógico y las funciones que se utilizaron para cumplir con los requisitos del sistema en lo que respecta a un solo edificio; en esta sección se hará mayor énfasis en la explicación del funcionamiento del ascensor uno del edificio uno, ya que lo demás es el mismo código con diferente índice en sus vectores y listas.

Funciones de Construir:

Descripción general: `contruirPuertas()` y `construirBoton()`, se encargan de construir dinámicamente las puertas y los indicadores de los cuatro ascensores respectivamente, son parte de la `ucEdificio()`, esta se detallará en seguida:

Descripción detallada: El primer bloque de código de líneas 3 y 4 son para iniciar los componentes de la plantilla de Forms y para validar el uso de llamadas al hilo grafico de este Form.

De la línea 6 a la 19 básicamente es la creación de vectores, listas de puertas y indicadores, si se mira con atención existe una lista y varios tipos de vectores que poseen 4 campos, esto es porque, cada edificio (como hemos visto) tiene 4 ascensores y ahora cada ascensor de estos tendrá que inicializar los siguientes datos:

Una lista de puertas, una lista de indicadores, una lista de solicitudes, un data grid para su historial y una lista de procesos lógicos.

```
1  public ucEdificio()  
2  {  
3      InitializeComponent();  
4      CheckForIllegalCrossThreadCalls = false;  
5  
6      mtzPuerta = new ucPuerta[10,4];  
7      mtzBoton = new ucIndicador[10, 4];  
8      solicitudes = new List<clSolicitud>[4];  
9      dtgSolicitudes = new DataGridView[4];  
10     contruirPuertas();  
11     contruirBoton();  
12     fifo = new List<clProceso>[4];  
13     hAscensor = new Thread[4];  
14     posicion = new int[4];  
15     Cola = new cola(updateCola);  
16     nSolicit = new nSolicitud(nuevaSolicitud);  
17     colas = new ucCola[4];  
18     ejecutandos=new ucEjecutando[4];  
19     panels = new Panel[4];  
20 }
```

```
30  
31  
32  
33  
34  
35  
36  
37  
38  
39  
40  
41  
42  
43  
44  
45  
46  
47  
48  
49  
50  
51  
52  
53  
54  
55  
56  
57  
58  
59  
60  
61  
62  
63  
64  
65  
66  
67  
68  
69  
70  
71  
72  
73  
74  
75  
76  
77  
78  
79  
80  
81  
82  
83  
84  
85  
86  
87  
88  
89  
90  
91  
92  
93  
94  
95  
96  
97  
98  
99  
100
```


El bloque de la línea 21 a 31 son las asignaciones de los datos anteriores a al ascensor 1, este código se repite 3 veces más con la diferencia de que será cada vez para un ascensor diferente.

```

15
16
17      *
18      *
19      *
20      //ascensor #1
21      panell.edificio = this;
22      panell.asencensor = 0;
23      panell.cola = ucCola1;
24      fifo[0] = new List<clProceso>();
25      hAscensor[0] = new Thread(fAscensor1);
26      colas[0] = ucCola1;
27      ejecutandos[0] = ucEjecutandol;
28      panels[0] = panell;
29      solicitudes[0] = new List<clSolicitud>();
30      dtgSolicitudes[0] = dataGridView0;

```

Funciones de Inicialización y Terminación de Hilos:

Descripción general: Son cinco pero las siguientes cuatro (fAscensor1 (), fAscensor2 (), fAscensor3 () y fAscensor4), realizan el mismo proceso para cada ascensor diferentes cuando este se desea iniciar desde la instancia creada de un edificio.

En detalle, primero las anteriores son funciones que parecen muy simples por contar con solo tres líneas de código, en realidad detrás de esta función esta toda la planificación, la enorme y compleja clase de Thread y finalmente es recursiva pero, no tienen un estado base sino un estado de pausa o freno ("while (true)")

```

private void fAscensor1()
{
    funciona(0);
    hAscensor[0] = new Thread(fAscensor1);
    hAscensor[0].Start();
}

public void iniciarEdificios()
{
    for (int i = 0; i < 4; i++)
        hAscensor[i].Start();
}

```

```

public void jefeSeMonto(Boolean x)
{
    for (int i = 0; i < 4; i++)
        panels[i].bJefe(x);
}

private void chckResumen_CheckedChanged(object sender, EventArgs e)
{
    for (int i = 0; i < 4; i++)
        dtgSolicitudes[i].DataSource = (from t in solicitudes[i]
                                         where t.Descripcion.Contains("S") || !chckResumen.Checked
                                         select t).ToList();
}

```

en

Función de Planificación se detalla ampliamente) para el hilo que se inicia en la tercera línea.

Esta misma función se aplica tres veces más pero para cada ascensor por separado. NOTA: si se cargan 3 edificios esta función sería llamada en un estado máximo de 12 veces.

Finalmente las funciones de `iniciarEdificios ()` y `death()`, lo que realizan es la inicialización y terminación de los cuatro ascensores a la vez.

Funciones de Actualización:

La primera es muy simple y se usa para actualizar la “Cola de Procesos Por Realizar (ucCola.cs)”, a través de la propiedad ya comentada de ese componente.

La segunda agrega solicitudes a al dataGridView y a la lista de solicitudes, correspondiente de cada ascensor.

```
private static void updateCola(ucCola ucCola1, List<clProceso> fifo)
{
    ucCola1.update(fifo);
    ucCola1.Refresh();
}

public void nuevaSolicitud(int i, string des)
{
    solicitudes[i].Add(new clSolicitud(des));
    dtgSolicitudes[i].DataSource = (from t in solicitudes[i]
                                     where t.Descripcion.Contains("S") || !chkResumen.Checked
                                     select t).ToList();
}
```

Funciones de Restricciones:

La primera restringe que en un mismo edificio solo se permita agregar una solicitud de jefe, y lo realiza a través de la función “bJefe(x)” para activar o desactivar el botón de jefe en todos los ascensores.

La siguiente restringe que los “DataGridView’s (LOAP)”, resuman sus resultados, agregando a los DataSource’s todas las solicitudes exceptuando las de los movimientos del ascensor (tema detallado en el componente LOAP).

```
public void jefeSeMonto(Boolean x)
{
    for (int i = 0; i < 4; i++)
        panels[i].bJefe(x);
}

private void chkResumen_CheckedChanged(object sender, EventArgs e)
{
    for (int i = 0; i < 4; i++)
        dtgSolicitudes[i].DataSource = (from t in solicitudes[i]
                                           where t.Descripcion.Contains("S") || !chkResumen.Checked
                                           select t).ToList();
}
```

Función de Planificación

Descripción general: Hasta acá todos los ascensores están inicializados lógicamente y gráficamente, están listos para planificar solicitudes, en la llamada de esta función, inicia el largo camino lógico de planificación de una sola solicitud (* detallado en **Algoritmo Base de Planificación** y **clFunciones**)

En palabras simples este algoritmo busca la siguiente solicitud a ejecutar, de todas las que tenga en su lista y una vez encontrado, se mueve un piso únicamente y verifica si llegó o no al destino, si lo hizo descarga y elimina la solicitud, sino el hilo vuelve a generar la función y el algoritmo se repite nuevamente, el proceso se detiene solo en el caso de que se desee pausar o que se llegue al destino y no hayan más solicitudes. (Ver **Algoritmo Base de Planificación** si desea más claridad).

Debemos recordar que esta función se ejecuta por ascensor, y que entonces por edificio serán cuatro ascensores planificando, cada uno por aparte lo que debe hacer con sus propias solicitudes, exceptuando el caso de la solicitud de jefe que es uno por edificio.

Descripción detallada del algoritmo: La línea 4, es el freno en caso de que se quiera detener el hilo o que la lista no tenga procesos.

La línea 5 solicita que la clase **clFunciones** le retorne el siguiente proceso a ejecutar por medio de la lógica de planificación*.

La línea 6, genera el proceso gráfico de la **Cola de Procesos Por Realizar (ucCola.cs)**, para saber hacia dónde mover el ascensor.

De la línea 7 a la 14, se calcula en qué dirección caminará el ascensor, debemos tomar en cuenta que si el origen no se ha ejecutado quiere decir que el origen será el destino... (Si lo anterior no se entendió léalo de nuevo, debe quedar claro).

```
1
2 private void funciona(int a)
3 {
4     while (fifo[a].Count == 0 || pausa) { Thread.Sleep(500); }
5     clProceso proceso = clFunciones.planifica(fifo[a],posicion[a]);
6     ejecutandos[a].set(proceso);
7     int destino = proceso.destino;
8     if(!proceso.origenE)
9         destino = proceso.origen;
10    int dir=-1;
11    if (destino > posicion[a])
12    {
13        dir = 1;
14    }
15
16    *
17    *
18    *
```

Teniendo calculado el destino en la línea 18 se pregunta si no se ha llegado, es decir esta entre el origen y el destino, si eso es verdad se aumenta o disminuye la posición del ascensor y esto sucede porque la variable dir es igual a 1 o -1, afectando directamente la dirección del ascensor en la línea 20.

Las líneas de la 21 a 24 solo modifican la parte gráfica (el detalle de la línea 21 está en **Función Invoke**)

```
15      *
16      *
17      *
18      if (!(destino==posicion[a]))
19      {
20          posicion[a]+=dir;
21          BeginInvoke(nSolicit,a, "aP " + posicion[a].ToString());
22          mtzBoton[posicion[a], a].Rojo();
23          mtzBoton[posicion[a]-dir, a].verde();
24          Thread.Sleep(1000);
25      }
26      *
27      *
28      *
```

El último bloque en la línea 32 se pregunta si se llegó al destino (contrario al anterior bloque), si es verdad las líneas 34 a 37 realizan la animación de la puerta que es otro hilo encapsulado en el componente puerta (este se convierte en un subproceso).

En la línea 38 se da por terminada la solicitud del ascensor, cabe decir que hay dos tipos de terminación de solicitud del ascensor, la de llegada a donde se solicitó y la llegada de destino, en este proceso entra en juego la variable “origen ejecutado” en la función de **Solicitudes intermedias realizables**

Descripción general: Como en nuestra lista de solicitudes puede ocurrir este estado (El ascensor está en el piso 1, y se solicita de 1 a 10, luego de 2 a 4, de 5 a 3 y así por el estilo), se puede concluir que mientras se satisface la solicitud 1 a 10 se puede satisfacer los pisos 2, 4, 5 y a 3 no se podría devolver, ya que está subiendo; En fin este algoritmo es parte del proceso iterativo que permite la aceptación de los pisos en este orden para recoger en 1, 2, 5 y dejar personas en este orden 2, 10, 3. Pero intercalando los resultados sería así:

Tabla 3

Tabla de iteraciones de las diferentes llamadas positivas de la función a la lista de solicitudes {{1 a 10}, {2 a 4}, {5 a 3}}

En fin, la función devuelve la próxima solicitud más cercana que esté, entre la posición actual del ascensor y su destino (que puede ser el origen de una solicitud).

Descripción detallada: no se hará en imagen por su extensión, en seguida se verá una vista más quirúrgica de la función dividida por 3 bloques.

Bloque 1

Su objetivo es ir en camino a obtener la lista de solicitudes que estén entre origen y destino.

Este bloque inicializa y asigna valores, además en el if se permite intercambiar en los valores de origen y destino en el caso de que el ascensor tenga que bajar (de esta forma no repetimos el

Bloque 2 en dos ocasiones)

```
public static clProceso solicitudIntermedia(List<clProceso> listaTotal, int acensor)
{
    Boolean cambio=false;
    int destino = listaTotal.ElementAt(0).origen;
    if (listaTotal.ElementAt(0).origenE)
    {
        destino = listaTotal.ElementAt(0).destino;
    }
    List<clProceso> listaRango = new List<clProceso>();
    clProceso actual;

    // es por si acaso el ascensor baja
    if (acensor > destino)
    {
        cambio = true;
        int aux = destino;
        destino = acensor;
        acensor = aux;
    }
}
```

Bloque 2

Este “para”, recorre por completo la lista de procesos a realizar y obtiene el rango de los que están entre y a cada proceso le verifica el estado de su origen (“ejecutado” o “no ejecutado”) para ser agregado a la lista del rango.

```
for (int i = 0; i < listaTotal.Count; i++)
{
    actual = listaTotal.ElementAt(i);
    if (actual.origenE) // si es cierto ya se ejecuto el origen
    {
        if ((acensor <= actual.destino) && (actual.destino <= destino))
            //si es cierto este proceso si esta en el rango
        {
            listaRango.Add(actual);
        }
    }
    else
    {
        if ((acensor <= actual.origen) && (actual.origen <= destino))
        {
            listaRango.Add(actual);
        }
    }
}
```

Bloque 3

Este bloque ahora recorre la lista de procesos que están en el rango de la solicitud inicial y usando valores absolutos (`Math.Abs()`), se puede ir obteniendo el más cercano hasta llegar al último más cercano que será retornado, la función termina acá pero finalmente en la función de **Función de Planificación** se compara este resultado para verificar si se llegó al destino o se puede satisfacer una solicitud.

```
if (cambio)
{
    int aux = destino;
    destino = acensor;
    acensor = aux;
}

int absMenor=13;
clProceso masCercano=null;
for (int i = 0; i <= listaRango.Count - 1; i++)
{
    if(listaRango.ElementAt(i).origenE)
    {
        int calculo=Math.Abs(listaRango.ElementAt(i).destino - acensor);
        if (absMenor > calculo)
        {
            absMenor=calculo;
            masCercano = listaRango.ElementAt(i);
        }
    }
    else
    {
        int calculo = Math.Abs(listaRango.ElementAt(i).origen -
acensor);
        if (absMenor > calculo)
        {
            absMenor = calculo;
            masCercano = listaRango.ElementAt(i);
        }
    }
}
return masCercano;
}
```

Aceptar solicitudes

Es una función simple que limpia de la lista de solicitudes de un edificio específico, todas aquellas que se pueden satisfacer, y modifica en caso de solo satisfacer el origen de una solicitud.”.

Finalmente la línea 39, solo modifica la parte gráfica (se detalla en [Función Invoke](#)).

```
27
28
29      *
30      *
31      *
32      if (destino == posicion[a])
33      {
34          mtzPuerta[destino, a].Abrir();
35          Thread.Sleep(2000);
36          mtzPuerta[destino, a].Cerrar();
37          Thread.Sleep(2000);
38          clFunciones.aceptarSolicitud(fifo[a], posicion[a], this, a);
39          ucCola1.BeginInvoke(Cola, colas[a], fifo[a]);
40      }
41
```


Clases

clFuente

Maneja la lógica para aplicarle la fuente a las letras de los labels de nuestro panel de botones,

Nota: no se detalla por ser una clase común de aplicación de fuentes.

clProceso

Encapsula las características que tendrá nuestro proceso como tal, los números de piso de origen y destino, las variables booleanas que nos darán la información de prioridad y si el origen ya fue ejecutado, en lógica de requisitos diríamos que la persona ya fue recogida por el ascensor.

clSolicitud

Es una clase con un solo campo string, pero de gran uso para el historial de solicitudes de las listas DataGridView's (LOAP).

clFunciones

Sin duda una de las clases más importantes de todo el proyecto, todos los flujos de datos de cada ascensor y la información que generan sus listas, pasan por esta clase para ser planificados.

Solicitud con prioridad jefe

Esta función devuelve el proceso de prioridad jefe de la lista (parámetro)

Esta lista de procesos es la de cada ascensor y se ejecutara encada lapso del hilo de que planifica el ascensor, así que su retorno será directo a ese hilo sin afectar los demás.

```
//esto busca un proceso de prioridad jefe
private static clProceso chkpriprocess(List<clProceso> lista)
{
    for (int i = 0; i < lista.Count; i++)
    {
        if (lista.ElementAt(i).prioridad == true)
        {
            return lista.ElementAt(i);
        }
    }
    return null;
}
```

Solicitudes intermedias realizables

Descripción general: Como en nuestra

lista de solicitudes puede ocurrir este estado (El ascensor está en el piso 1, y se solicita de 1 a 10, luego de 2 a 4, de 5 a 3 y así por el estilo), se puede concluir que mientras se satisface la solicitud 1 a 10 se puede satisfacer los pisos 2, 4, 5 y a 3 no se podría devolver, ya que está subiendo; En fin este algoritmo es parte del proceso iterativo que permite la aceptación de los pisos en este orden para recoger en 1, 2, 5 y dejar personas en este orden 2, 10, 3. Pero intercalando los resultados seria así:

Tabla 3

Tabla de iteraciones de las diferentes llamas positivas de la función a la lista de solicitudes {{1 a 10}, {2 a 4}, {5 a 3}}

En fin, la función devuelve la próxima solicitud más cercana que esté, entre la posición actual del ascensor y su destino (que puede ser el origen de una solicitud).

Llamadas a la función	Recoger	Descargar
1	Piso 1	--
2	Piso 2	--
3	--	Piso 4
4	Piso 5	--
5		Piso 10
6	--	Piso 3

Descripción detallada: no se hará en imagen por su extensión, en seguida se verá una vista más quirúrgica de la función dividida por 3 bloques.

Bloque 1

Su objetivo es ir en camino a obtener la lista de solicitudes que estén entre origen y destino.

Este bloque inicializa y asigna valores, además en el if se permite intercambiar en los valores de origen y destino en el caso de que el ascensor tenga que bajar (de esta forma no repetimos el Bloque 2 en dos ocasiones)

```
public static clProceso solicitudIntermedia(List<clProceso> listaTotal, int acensor)
{
    Boolean cambio=false;
    int destino = listaTotal.ElementAt(0).origen;
    if (listaTotal.ElementAt(0).origenE)
    {
        destino = listaTotal.ElementAt(0).destino;
    }
    List<clProceso> listaRango = new List<clProceso>();
    clProceso actual;

    // es por si acaso el ascensor baja
    if (acensor > destino)
    {
        cambio = true;
        int aux = destino;
        destino = acensor;
        acensor = aux;
    }
}
```

Bloque 2

Este “para”, recorre por completo la lista de procesos a realizar y obtiene el rango de los que están entre y a cada proceso le verifica el estado de su origen (“ejecutado” o “no ejecutado”) para ser agregado a la lista del rango.

```
for (int i = 0; i < listaTotal.Count; i++)
{
    actual = listaTotal.ElementAt(i);
    if (actual.origenE) // si es cierto ya se ejecuto el origen
    {
        if ((acensor <= actual.destino) && (actual.destino <= destino))
            //si es cierto este proceso si esta en el rango
        {
            listaRango.Add(actual);
        }
    }
    else
    {
        if ((acensor <= actual.origen) && (actual.origen <= destino))
        {
            listaRango.Add(actual);
        }
    }
}
```

Bloque 3

Este bloque ahora recorre la lista de procesos que están en el rango de la solicitud inicial y usando valores absolutos (`Math.Abs()`), se puede ir obteniendo el más cercano hasta llegar al último más cercano que será retornado, la función termina acá pero finalmente en la función de **Función de Planificación** se compara este resultado para verificar si se llegó al destino o se puede satisfacer una solicitud.

```
if (cambio)
{
    int aux = destino;
    destino = acensor;
    acensor = aux;
}

int absMenor=13;
clProceso masCercano=null;
for (int i = 0; i <= listaRango.Count - 1; i++)
{
    if(listaRango.ElementAt(i).origenE)
    {
        int calculo=Math.Abs(listaRango.ElementAt(i).destino - acensor);
        if (absMenor > calculo)
        {
            absMenor=calculo;
            masCercano = listaRango.ElementAt(i);
        }
    }
    else
    {
        int calculo = Math.Abs(listaRango.ElementAt(i).origen -
acensor);
        if (absMenor > calculo)
        {
            absMenor = calculo;
            masCercano = listaRango.ElementAt(i);
        }
    }
}
return masCercano;
}
```

Aceptar solicitudes

Es una función simple que limpia de la lista de solicitudes de un edificio específico, todas aquellas que se pueden satisfacer, y modifica en caso de solo satisfacer el origen de una solicitud.

Planifica

Esta función encapsula las funciones `chkpriprocess` y `solicitudIntermedia`. Primero ejecuta `chkpriprocess`, si encuentra un proceso de prioridad lo retorna, en caso contrario ejecuta `solicitudIntermedia`, igual, en caso de encontrar alguna la retorna, en caso contrario retorna el primer elemento de la lista, en todo caso, la función planifica se usa para obtener la siguiente solicitud a ejecutar.

```
public static clProceso planifica(List<clProceso> lista,int ascensor){  
    clProceso proceso = chkpriprocess(lista);  
    if (proceso != null)  
        return proceso;  
    proceso = solicitudIntermedia(lista, ascensor);  
    if (proceso != null)  
        return proceso;  
    if (proceso != null)  
        return proceso;  
    return lista.ElementAt(0);  
}
```

Hilos

Son procesos auxiliares o de trabajo que permiten intercalar dos o más tareas completamente diferentes, en ocasiones muy complejas y diferentes, de forma que en un solo procesador el usuario percibe que se realizan al mismo tiempo, en varios procesadores estas tareas serían asíncronas dependiendo de la programación, el IDE utilizado y el lenguaje.

Uso de la clase Threading

Es una librería dll de .Net que permite la creación, uso y aplicación de subprocesos o hilos de trabajo, en este proyecto es fundamental su uso para poder planificar cada ascensor por aparte.

Función Invoke

Es una interesante función que actúa sobre componentes de un subproceso A, hacia otro B, de forma que A, puede invocar un proceso en B, o viceversa, según se desee, en este apartado se ejemplificará con el proyecto y se explicará cómo realiza el proceso de modificar componentes gráficos que le pertenecen a un subproceso de manejo gráfico de Windows Form, por parte de nuestros hilos.

Consta de 3 partes para su aplicación:

Inicialización de un delegate:

Se crea una función delegada con nombre y parámetros, en nuestro caso los parámetros serán el i = piso y des = descripción de la acción.

Se crea una instancia de un delegado con nombre.

Y se le asigna la función delegada al delegado, en nuestro caso el parámetro es la función nuevaSolicitud que recibe los mismos parámetros y se explica en el siguiente título.

```
public delegate void nSolicitud(int i, string des);  
public nSolicitud nSolicit;  
  
nSolicit = new nSolicitud(nuevaSolicitud);
```

Función solicitada

Como la función recibe los mismos parámetros, se sobrecarga y no tiene problemas al aplicar el código de la última posición de memoria referenciada, así que el código final a aplicar será el de esta función, a nivel de proyecto Ascensores lo que acá se hace es agregar una solicitud nueva a dtgSolicitudes (vector de DataGridView's (LOAP))

```
public void nuevaSolicitud(int i, string des)  
{  
    solicitudes[i].Add(new clSolicitud(des));  
    dtgSolicitudes[i].DataSource = (  
  
from t in solicitudes[i]  
where t.Descripcion.Contains("S") || !chckResumen.Checked  
select t  
)  
.ToList();
```

}

Aplicación de invoke

Acá se llama en dos momentos diferentes para su aplicación, y en nuestro ejemplo un correcto manejo grafico.

```
BeginInvoke(nSolicit,a, "aP " + posicion[a].ToString());
```

```
ucCola1.BeginInvoke(Cola, colas[a], fifo[a]);
```

Encapsulación de Hilos

Los hilos son subprocesos que una vez iniciados tener control de ellos es muy difícil si se le pierde cuidado, en nuestro proyecto optamos por aplicarlos de forma encapsulada en componentes para no preocuparnos en su control, sino que manejamos el componente.

El ejemplo más simple es el de la puerta, donde nace, crece y muere un subproceso y sin la mayor complicación, solo al llamar las funciones, sabremos que el compilador organiza el código.

El ejemplo más complejo es el edificio que posee una planificación de cuatro ascensores de forma asincrónica, multitarea (sin detalle pues la clase threading se encarga de eso), todo esto parece ser muy complejo y realmente no es así... ya que cada ascensor posee sus apuntadores y su propio hilo de modo que no invadirá ni se perderá entre todos.

Algoritmo Base de Planificación

El seudocódigo que aclara la forma de planificación planteada es este:

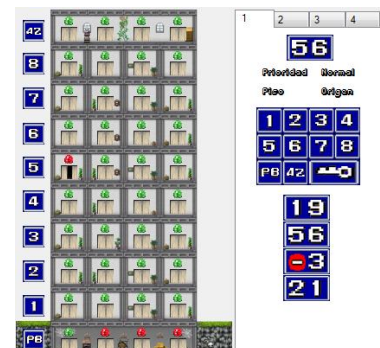
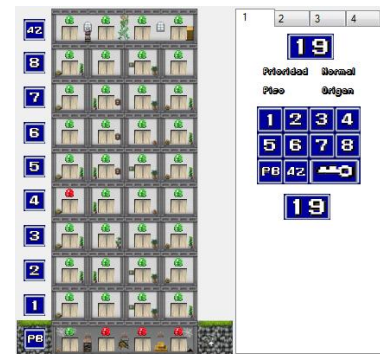
- 1) Es un ciclo infinito que se detiene cuando no haya procesos en la lista SOLICITUDES o se pause.
- 2) De la lista SOLICITUDES obtiene el rango que esta entre el ascensor y el destino y se guarda en LISTARANGO.

NOTA: En la siguiente imagen se muestra la solicitud 19 pero el ascensor esta bajado, lo que quiere decir que la información es = rango de 4, origen a 1, destino, en la solicitud es 1, origen a 9, destino.

- 3) LISTARANGO de la imagen dos en ese preciso instante sería igual a rango = 5 a 1 y estarían dentro las solicitudes de 5,3, 2 y 1

- 4) de LISTARANGO obtener el más cercano y preguntar si está en el piso para satisfacer las solicitudes, en este punto 5 se inicia a ejecutar.

- 5) el algoritmo termina, y se vuelve a ejecutar con 5 ya ejecutado, y en la próxima interacción no se atendería nada, caminaría hacia 4, luego en la siguiente iteración, atiende a 3 y



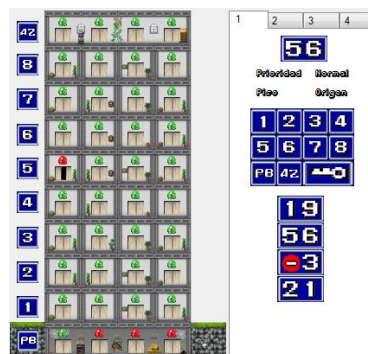
termina esa solicitud... Hasta terminar, en la **Tabla 4** se ve todo el recorrido y resultado final, con todas las iteraciones.

Tabla 4

Tabla de iteraciones totales de la solución de la imagen de la derecha su lista de solicitudes es {{1 a 9}, {5 a 6}, {7 a 3}, {2 a 1}}

Sin detalle de movimiento

Llamadas a Función	Recoger	Descargar
1	Piso 7	--
5	Piso 5	--
8	--	Piso 3
10	Piso 2	--
12,13	Piso 1	Piso 1
18	--	Piso 6
22	--	Piso 9



Con detalle de movimiento

Llamadas a la Función	Ascensor En PosAct	Rango	Recoger	Descargar	Cambiar a Piso	Solicitud Más Cercana
1	8	8 a 1	--	--	7	Piso 7 Origen
2	7	7 a 1	Piso 7	--	--	Piso 7 Origen
3	7	7 a 1	--	--	6	Piso 5 Origen
4	6	6 a 1	--	--	5	Piso 5 Origen
5	5	5 a 1	Piso 5	--	--	Piso 5 Origen
6	5	5 a 1	--	--	4	Piso 3 Destin
7	4	4 a 1	--	--	3	Piso 3 Destin
8	3	3 a 1	--	Piso 3	--	Piso 3 Destin
9	3	3 a 1	--	--	2	Piso 2 Origen
10	2	2 a 1	Piso 2	--	--	Piso 2 Origen
11	2	2 a 1	--	--	1	Piso 1 O/D
12	1	1 a 1	Piso 1	Piso 1	--	Piso 1 O/D
13	1	1 a 9	--	--	2	Piso 6 Destin
14	2	2 a 9	--	--	3	Piso 6 Destin
15	3	3 a 9	--	--	4	Piso 6 Destin
16	4	4 a 9	--	--	5	Piso 6 Destin
17	5	5 a 9	--	--	6	Piso 6 Destin
18	6	6 a 9	--	Piso 6	6	Piso 6 Destin
19	6	6 a 9	--	--	7	Piso 9 Desti
20	7	7 a 9	--	--	8	Piso 9 Desti
21	8	8 a 9	--	--	9	Piso 9 Desti
22	9	9 a 9	--	Piso 9	9	Piso 9 Desti

Conclusiones

Este material puede ser usado para apoyo en sistemas de desarrollo orientado a subproceso de alta y media complejidad, si se leer detenidamente y entiende de forma completa este manual, se produce conocimiento de gran valor en tema de ingeniería de software.

Se puede dar mantenimiento y auditoria a Ascensores EIF212 de una forma más eficaz y certera, sin posibles errores o resultados no deseados, todo depende de quién y cuánto tiempo tenga para entender todos sus procesos unido al código fuente.

Es un apoyo didáctico y tiene métodos, teorías y procesos que se pueden aplicar en la ingeniería en sistemas, dependiendo la aplicación que se desee construir.

Puede ser usada como material de investigación, experiencia y trabajo en equipo.