PBE PUZZLE 2

1.Configuración

1.1 Instalación gtk3

Para desarrollar el segundo proyecto, es necesario utilizar una librería que permita crear una interfaz gráfica. Esta interfaz inicialmente constará de una ventana con dos botones. Existen varias opciones para esto, pero en este caso se ha optado por usar la librería “gtk3”, la cual instalaremos como una gema. Para hacerlo, solo es necesario ejecutar los siguientes comandos en la terminal.

```
Python
sudo apt-get install gtk3
sudo gem install
```

1.2 Creación gema Puzzle 1

El puzzle 2 reutiliza las funcionalidades programadas en el primer puzzle, por lo que es necesario crear una gema que permita reutilizar las clases y funciones ya implementadas. A continuación, se explica el procedimiento: primero, se debe instalar la gema "bundler", que nos permitirá crear nuestras propias gemas. Para ello, desde el terminal se ejecuta el comando:

```
Python
sudo gem install bundler
```

Una vez que "bundler" esté instalado, podemos proceder a crear nuestra propia gema con el siguiente comando.

```
Python
bundle gem puzzle1
```

Este proceso creará varios archivos que compondrán la estructura de la gema, siendo los más importantes el Gemfile, el archivo .gemspec (que contiene las especificaciones e información de la gema), y el README, donde se detalla cómo instalar y utilizar la gema.

Lo primero que se debe hacer es modificar el archivo .gemspec, ya que, por defecto, incluye código pendiente de implementar, como el nombre de la gema, la información del desarrollador, el correo electrónico, etc. Estos campos vienen marcados con "TODO", lo que indica que necesitan ser completados antes de compilar la gema correctamente.

Lo siguiente fue entrar al directorio lib y modificar el archivo puzzle1.rb, donde se añadió el código del ejercicio. También es importante mencionar que, en el archivo version.rb, podemos actualizar la versión de la gema según las modificaciones que realicemos; en este caso, se estableció la versión 1.0.

Finalmente, para instalar la gema en nuestro ordenador, lo primero que debemos hacer es construirla (hacer el "build") y luego proceder con la instalación. Para hacerlo, se ejecutan los siguientes comandos en la terminal:

```
Python
gem build puzzle1.gemspec
```

El proceso de construcción y compilación utiliza el código especificado en el archivo .gemspec, por lo que era crucial modificar los campos predeterminados, como el nombre de la gema, la versión y la información del autor.

Cuando veamos el mensaje "Successfully built RubyGem", eso indicará que la gema se ha construido correctamente. En ese momento, podremos comprobar que en el directorio ha aparecido un archivo con extensión .gem, que es la gema que hemos creado. Finalmente

```
Python
gem install ./puzzle1-1.0.gem
```

2. Puzzle 2

2.1 Apartado grafico

Primero, se escribió el código que permite obtener una vista preliminar de la interfaz gráfica que mostrará nuestro programa. Es importante recordar que la interfaz debe incluir dos botones: uno azul y otro rojo, ambos contenidos dentro de una ventana.

```
Python
@blue = Gdk::RGBA::new(0,0,1,1)
```

```
@white = Gdk::RGBA::new(1,1,1,1)
@red = Gdk::RGBA::new(1,0,0,1)
```

Comenzamos definiendo las variables constantes globales blue, white y red utilizando instancias de la clase Gdk, proporcionando los parámetros correspondientes en formato RGBA (red, green, blue, alpha).

Después de establecer los colores, creamos la ventana del programa instanciando la clase Gtk::Window con el título deseado. A continuación, ajustamos el tamaño de la ventana y el ancho del borde según nuestras necesidades utilizando las instrucciones set_size_request y set_border_width.

```Python
@rf = Rfid.new
@window = Gtk::Window.new("Rfid Window")
@window.set_size_request(600,150)
@window.set_border_width(5)
```

Para crear los botones utilizando la gema gtk3, se generan dos instancias de la clase Gtk::Button y se les asigna la etiqueta correspondiente, que es el texto que se mostrará en cada botón.

El color de fondo de los botones se puede cambiar con la función predefinida override_background_color, mientras que el color del texto se ajusta utilizando la función override_color. Posteriormente, se modifica el tamaño de los botones según las especificaciones requeridas.

```Python
@window_button = Gtk::Button::new(:label => "Por favor, acerca tu tarjeta de identidad de la uni")
@window_button.override_background_color(:normal, @blue)
@window_button.override_color(:normal, @white)
@button = Gtk::Button::new(:label => "Clear")
```

En cuanto al diseño gráfico de la interfaz, se utilizó el método fixed, qur proporciona un contenedor con posiciones absolutas dentro de la ventana, lo que permite organizar los elementos de la interfaz de manera más controlada y precisa.

Para usarlo, solo es necesario calcular las dimensiones de los botones y su ubicación dentro del contenedor. Luego, se añaden los botones al objeto Fixed, y este se incorpora a la ventana, siguiendo un sistema de jerarquías de elementos.

```python
@fixed = Gtk::Fixed.new
@button.set_size_request(540,40)
@window_button.set_size_request(540,100)
@fixed.put(@window_button,30,0)
@fixed.put(@button,30,110)
@window.add(@fixed)
```

Finalmente, definimos el comportamiento de la ventana al cerrarse mediante la señal signal_connect("delete-event"), la cual generalmente se vincula a la función Gtk.main_quit para que el programa termine cuando se cierre la ventana.

Por último, se utiliza la instrucción show_all para que todos los elementos de la interfaz gráfica, como los botones y la ventana, se muestren correctamente en pantalla.

```python
@window.signal_connect("delete-event"){|_widget| Gtk.main_quit}
@window.show_all
Gtk.main
```

2.2 Gestión de threads y NFC

Lo primero que se debe programar es la acción del botón "clear". Aquí es importante verificar si el thread que procesa la tarjeta ya ha terminado. Esto se puede comprobar con la variable uid, que será nula si aún no se ha leído ninguna tarjeta. Si se pulsa "clear" mientras uid es nula, no ocurrirá nada. Sin embargo, si la tarjeta ya fue leída, se cambia el color y el texto del botón, se reinicia uid a una cadena vacía y se invoca el thread auxiliar encargado de gestionar la lectura de NFC, bloqueo y espera, mediante la función threads.

```
Python
@button.signal_connect"clicked" do |_widget|
    if @uid != ""
        @uid = ""
        @window_button.override_background_color(:normal, @blue)
        @window_button.set_label("Por favor, acerca tu tarjeta de identidad
de la uni")

        threads
    end
end
```

En la definición de la función threads, se crea una instancia de la clase Thread (incluida en la librería “threads”). Dentro de este thread, se programa su comportamiento llamando a la función lectura, que se encarga de la lectura de la tarjeta. Una vez que esta función termina, el thread se cierra y muestra un mensaje en la consola indicando que ha finalizado.

Es importante destacar que la función threads se invoca justo después de ser definida, al inicio del programa. Esto permite que el primer thread auxiliar se inicie de inmediato y comience a esperar la lectura de una tarjeta desde el principio.

```
Python
def threads
    thr = Thread.new {
        lectura
        puts "END THREAD"
        thr.exit
    }
end
threads
```

Finalmente, se define la función lectura y su subordinada gestion_UI. La función lectura muestra un mensaje en pantalla y espera a que el RFID lea una tarjeta. Una vez que la tarjeta es leída, su ID se guarda en la variable uid. Luego, se añade la función gestion_UI al Idle de Glib, lo que delega su ejecución al thread principal, ya que no queremos que el thread auxiliar maneje la interfaz de usuario. Aunque es posible asignar prioridades para que el thread principal gestione esta tarea de inmediato, en este caso no es necesario.

En la función gestion_UI, se verifica si uid no está vacía. Si no lo está, el texto del botón (su etiqueta) se actualiza con el valor de uid y se cambia el color del botón a rojo. Con esto, el programa queda completamente definido.

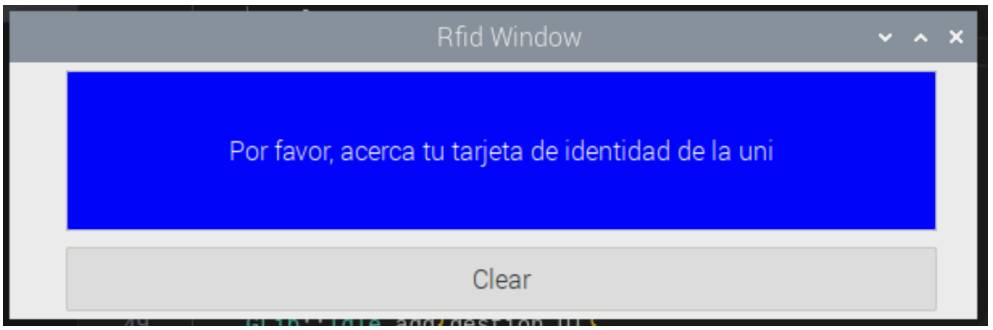
```Python
def lectura
    puts "Esperando id"
    @uid = @rf.read_uid
    GLib::Idle.add{gestion_UI}
end

def gestion_UI
    if @uid != ""
        @window_button.set_label(@uid)
        @window_button.override_background_color(:normal, @red)
    end
end
```

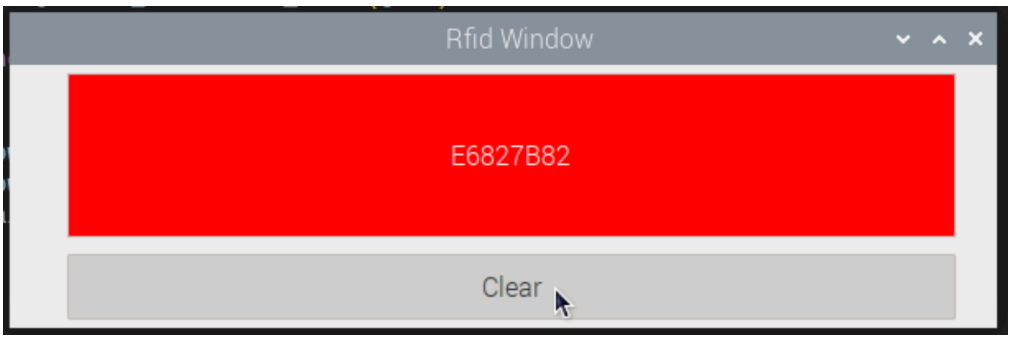
3. Código ruby

```Python
require "gtk3"
require "thread"
require "puzzle1"

@rf = Rfid.new
@window = Gtk::Window.new("Rfid Window")
@window.set_size_request(600,150)
@window.set_border_width(5)
@uid = ""

@blue = Gdk::RGBA::new(0,0,1,1)
@white = Gdk::RGBA::new(1,1,1,1)
@red = Gdk::RGBA::new(1,0,0,1)

@window_button = Gtk::Button::new(:label => "Por favor, acerca tu tarjeta de
identidad de la uni")
@window_button.override_background_color(:normal, @blue)
@window_button.override_color(:normal, @white)
@button = Gtk::Button::new(:label => "Clear")

@fixed = Gtk::Fixed.new
@button.set_size_request(540,40)
@window_button.set_size_request(540,100)
@fixed.put(@window_button,30,0)
@fixed.put(@button,30,110)
@window.add(@fixed)

@button.signal_connect"clicked" do |_widget|
```

```
    if @uid != ""
        @uid = ""
        @window_button.override_background_color(:normal, @blue)
        @window_button.set_label("Por favor, acerca tu tarjeta de identidad
de la uni")

        threads
    end
end

def threads
    thr = Thread.new {
        lectura
        puts "END THREAD"
        thr.exit
    }
end
threads

def lectura
    puts "Esperando id"
    @uid = @rf.read_uid
    GLib::Idle.add{gestion_UI}
end

def gestion_UI
    if @uid != ""
        @window_button.set_label(@uid)
        @window_button.override_background_color(:normal, @red)
    end
end

@window.signal_connect("delete-event"){|_widget| Gtk.main_quit}
@window.show_all
Gtk.main
```

4. Funcionamiento




Rfid Window
E6827B82
Clear