
# Análisis y diseño de la solución - Sistema de Inscripciones para Instituto Via Diseño

Mateo Minghi - A01711231
Enrique Ayala - A01711235
Leonardo Cervantes - A07184003
Carlos Vazquez - A01711730
Fermín Nieto - A01707069

Querétaro, Qro. a 28 de febrero de 2025

# Problemática

El manejo actual de inscripciones del Instituto Via Diseño carece de estructura, involucra múltiples herramientas independientes y requieren de demasiados inputs manuales, lo cual se presta a cometer errores.

Actualmente, la gestión de la inscripción comienza 2 meses antes del inicio del ciclo de inscripción. Los profesores comparten su disponibilidad de horarios, y las coordinadoras de carrera diseñan manualmente el calendario de clases, estableciendo qué clase de qué materia será impartida en qué aula a qué hora. Tal proceso resulta poco eficiente y exhaustivo para las coordinadoras, pues la cantidad de variables a considerar (salones disponibles, cupo de los salones, alumnos inscritos, etc) lo hacen muy complejo. Añadido a esto, se tiene que registrar manualmente a cada alumno individual y gestionar a los alumnos que no son regulares.

# Propuesta

A partir de la necesidad de un mejor sistema de inscripción, se ideó un sistema informático que centralice las operaciones y gestión relacionadas a las inscripciones del Instituto Via Diseño.

Tal sistema será una aplicación web disponible para coordinadores y alumnos. Los coordinadores podrán semi-automatizar la generación de horarios, establecer disponibilidad de profesores e identificar aquellos alumnos que requieren de atención personalizada para su proceso de inscripción.

Por su parte, los alumnos tendrán un turno de inscripción en el que podrán acceder al sistema y realizar su inscripción de la manera más independiente posible. En caso de ser necesario, pueden solicitar un ajuste a su horario con sus coordinadoras.

</> ceams

# Requisitos

- **Tipo Objetivo**
  - Desarrollar un software que genere horarios de clases de manera semi-automática, con el objetivo de centralizar la gestión de los horarios e inscripciones, reducir el tiempo de creación de horarios para las coordinadoras, reducir el tiempo de inscripción para los alumnos, minimizar los errores de transcripción en los registros de inscripción, y optimizar la disponibilidad de aulas y profesores antes del próximo período de inscripciones.
- **Funcionales**
  - Usuario
    i. Usuario se autentica
  - Coordinador
    **i. C1)Coordinador ingresa disponibilidad de profesores.**
    **ii. C2)Coordinador genera turnos de inscripción.**
    **iii. C3)Coordinador genera horario por periodo académico.**
    **iv. C4)Coordinador consulta horario por periodo académico.**
    v. C5)Coordinador modifica horario por periodo académico.
    vi. C6)Coordinador guarda horario por periodo académico.
    vii. C7)Coordinador consulta turno de inscripción por alumno.
    viii. C8)Coordinador consulta estatus de inscripción por alumno.
    ix. C9)Coordinador modifica horario individual por alumno.
  - Alumno
    i. Alumno consulta turno de inscripción.
    **ii. Alumno consulta horario.**
    **iii. Alumno registra aceptación de propuesta de horario.**
    iv. Alumno registra solicitud de cambios de horario.

| Requisito | Complejidad(1-5) | Riesgo(1-5) | Estabilidad(1-5) | Prioridad Total |
|---|---|---|---|---|
| C1 | 4 | 4 | 3 | 11 |
| C2 | 3 | 3 | 4 | 10 |
| C3 | 5 | 5 | 3 | 13 |

| | | | | |
|---|---|---|---|---|
| C4 | 2 | 2 | 5 | 9 |
| C5 | 3 | 2 | 3 | 8 |
| C6 | 1 | 2 | 5 | 8 |
| C7 | 2 | 2 | 3 | 7 |
| C8 | 2 | 2 | 4 | 8 |
| C9 | 3 | 3 | 3 | 9 |
| A1 | 1 | 2 | 5 | 8 |
| A2 | 2 | 2 | 5 | 9 |
| A3 | 2 | 3 | 4 | 9 |
| A4 | 3 | 2 | 4 | 9 |

Tabla 1. Priorización de Requisitos Funcionales

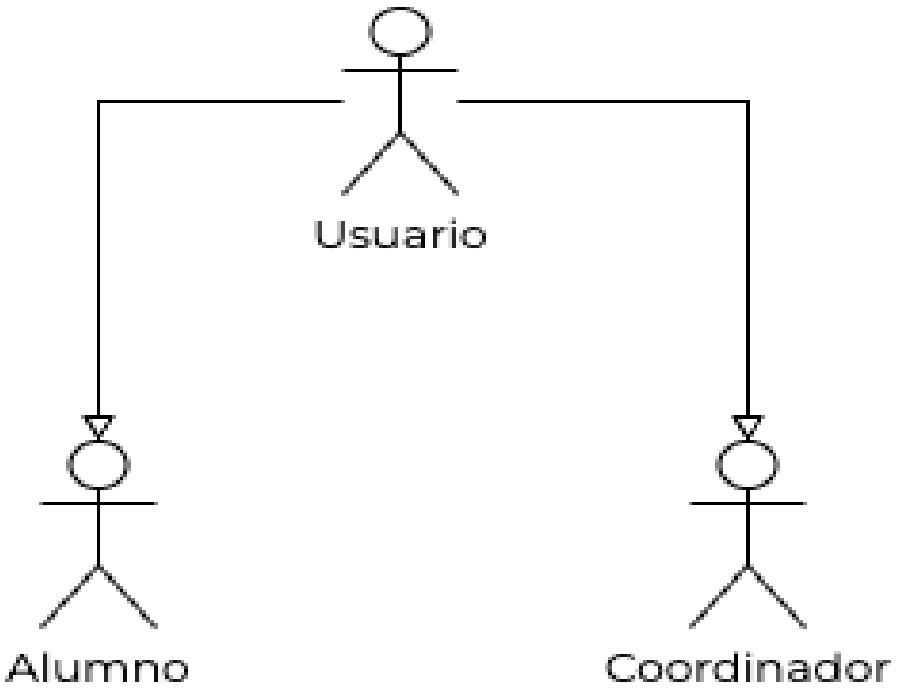


Figura 1. Diagrama de Casos de uso de “Alumno”

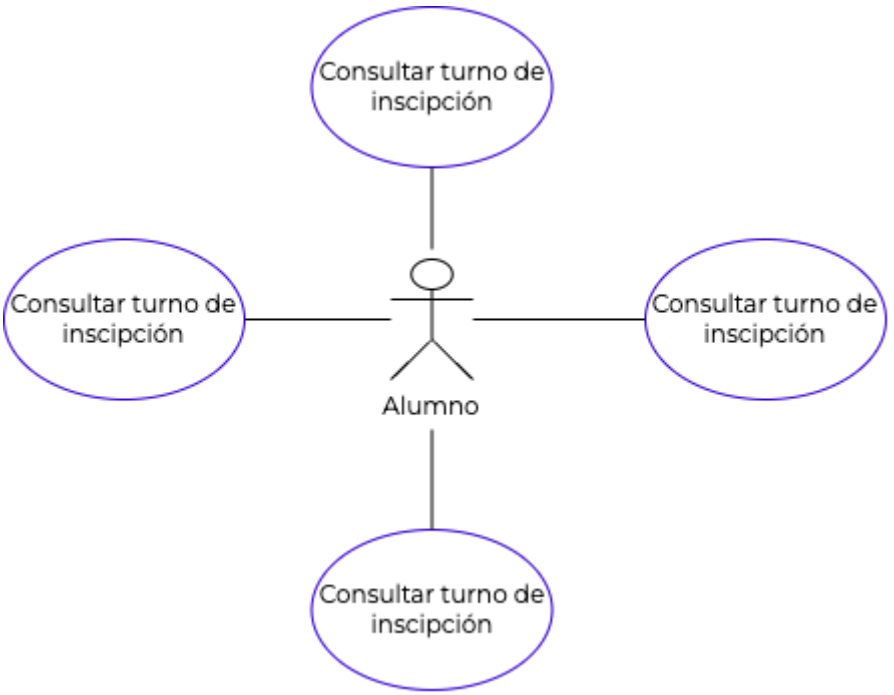


Figura 2. Diagrama de Casos de uso de “Alumno”


Figura 3. Diagrama de Casos de uso de “Alumno”

- **No Funcionales**
  - Rendimiento
    i. El sistema debe poder soportar a todos los alumnos usándolo simultáneamente.
    ii. El sistema debe abrir y estar completamente funcional en un tiempo de 5 segundos en un dispositivo con especificaciones mínimas al año de 2024, siendo estas 2 GB de memoria RAM y un procesador con velocidad mínima de 1.8 GHz.
  - Escalabilidad
    i. El sistema debe de ser capaz de manejar un incremento del 50% en el número de usuarios activos, hasta 225 usuarios activos simultáneos, sin que haya efectos en los tiempos de respuesta.
    ii. El sistema debe permitir la adición de nuevas funcionalidades sin afectar el rendimiento en más de un 10% de tiempo adicional de carga en la pantalla inicial.
    iii. El sistema debe permitir la adición de nuevos registros en el Sistema de Gestión General sin provocar errores en las funcionalidades establecidas.
  - Disponibilidad
    i. El sistema debe estar disponible durante todo el periodo de inscripciones, sin márgen para actualizaciones o mantenimiento durante este periodo.
  - Seguridad
    i. Los datos personales y académicos de los usuarios deberán estar cifrados en tránsito utilizando una versión TLS (Transport Layer Security) versión 1.2 o posterior, y en reposo con AES (Advanced Encryption Standard) a 256 bits.
    ii. El sistema debe de solicitar únicamente los permisos necesarios para el funcionamiento de la aplicación, siendo estos el acceso a conexiones inalámbricas, y notificaciones del dispositivo.
    iii. La aplicación debe cumplir con las normativas de privacidad y uso de datos, de acuerdo con la GDRP y la Ley de Protección de Datos Personales.
  - Compatibilidad
    i. El sistema debe estar disponible en los navegadores modernos más populares.

ii. El sistema debe de funcionar sin errores críticos en al menos el 99% de los dispositivos con las versiones de navegadores soportados, en diversas resoluciones.

- Mantenibilidad
  i. Durante el periodo de inscripciones, el tiempo de resolución de errores críticos debe ser menor a 24 horas.
  ii. Fuera del periodo de inscripciones, el tiempo de resolución de errores críticos debe ser de 48 horas y para errores menores de 3 días.
  iii. Las actualizaciones deben de ser aplicadas a los usuarios sin comprometer la integridad de sus datos ni su estatus de inscripción.
  iv. La totalidad del código debe de estar estructurado modularmente, asegurando que el 90% de las funciones principales puedan ser actualizadas sin afectar otras áreas.
- Usabilidad
  i. El diseño debe permitir a los coordinadores establecer un nuevo horario con la menor cantidad de clicks posibles. Una vez generado el horario, los coordinadores deben poder actualizarlo arrastrando componentes sin necesidad de registrar nueva información.
  ii. Se deberá capacitar a los administradores/coordinadores para hacer uso del sistema. El resto de usuarios debe ser capaces de realizar tareas claves dentro del sistema sin necesidad de asistencia.
  iii. El sistema debe de contar con una calificación de 3.5 o mayor en una escala de 5 puntos en pruebas de usabilidad realizadas a usuarios, midiendo la navegación por la aplicación y su facilidad de uso.
- Confiabilidad
  i. Los datos del usuario deben de sincronizarse con el servidor en tiempo real garantizando el registro del progreso y contar con un respaldo con una pérdida de 2 minutos en caso de fallo.
  ii. En caso de un fallo crítico, se deberá de notificar al usuario y permitirle realizar la operación en un lapso máximo de 1 minuto.

- **Reglas de Negocio**
  - Inscripción por tipo de alumno
    i. Alumnos regulares de 1er semestre deben inscribir exactamente 7 materias del primer bloque.
    ii. Alumnos regulares de 2do a 7mo deben inscribir 6 materias.
    iii. Alumnos regulares de 9no semestre deben inscribir 5 materias específicas para completar el plan de estudios.
    iv. Alumnos irregulares deben inscribir al menos una materia, según su kardex y disponibilidad.
  - Restricciones de seriación
    i. Un alumno no puede inscribir una materia seriada sin haber acreditado su prerrequisito.
    ii. Las materias "Desarrollo Empresarial" y "Desarrollo de Proyecto Integrador" deben inscribirse simultáneamente.
  - Gestión de horarios
    i. El horario de clases opera de 7:00 am a 4:00 pm únicamente.
    ii. No puede existir empalmes de materias en el horario.
  - Proceso para alumnos irregulares
    i. Se debe de verificar su kardex a través del API del sistema administrativo para determinar materias pendientes.
    ii. Se deben descartar las materias en trámite de equivalencia.
    iii. El pre-listado no debe de contar con empalmes con el horario.
  - Criterios de prioridad
    i. Materias prácticas reprobadas tienes prioridad de inscripción.
    ii. Cada semestre debe incluir al menos una materia de patronaje y una de confección cuando sea posible.
    iii. Alumnos regulares tienen prioridad sobre irregulares en la asignación de cupos limitados.
  - Apertura y cupo de grupos
    i. Se requiere un mínimo de 3 alumnos inscritos para abrir un grupo.
    ii. No se puede exceder la capacidad máxima del salón asignado.

iii. La asignación de salones debe considerar las características específicas requeridas por la materia.

- Restricciones temporales
  i. La inscripción debe realizarse únicamente en el turno de inscripción asignado.
  ii. Solicitudes extemporáneas deben ser aprobadas por parte de los coordinadores.

- **Requisitos de Información**

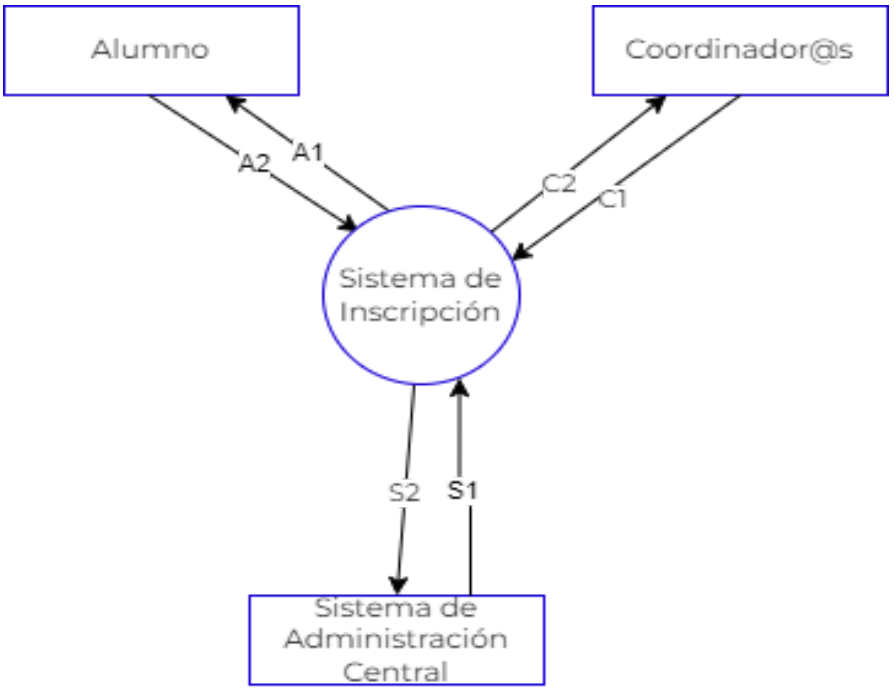


Figura 4. Diagrama de Contexto del Sistema "Vía Alta"

- A1) Recibe propuesta de horario (Nombre de Materias, id, Hora, Grupo, Salon, Profesor, Créditos, Carrera)
- A2.1) Alumno acepta horario (Nombre, Matrícula, Fecha de aceptación)
- A2.2) Alumno solicita cambios (Nombre, Matrícula, Carrera, Semestre, Estatus, Fecha de solicitud de cambios)
- C1) Coordinador Establece parámetros para la formación de horario (Horario de profesores, id, ciclo, horario de clases general (de 7:00am a 4:00pm))
- C2.1) Coordinador edita propuesta de horario general (día de la semana, hora, materia, id, profesor, semestre)
- Coordinador consulta estatus de inscripción de los alumnos (nombre apellido, matrícula, semestre, estatus de aceptación de horario,
- C2) Recibe propuesta inicial de horario general (día de la semana, hora, materia, id, profesor, semestre)
  - Usuario:
    i. ID_usuario

ii. Contraseña
- Alumnos:
  i. Nombre
  ii. Apellido_P
  iii. Apellido_M
  iv. ID_usuario
  v. Carrera
  vi.

- **Requisitos de Interfaz**
  - Interfaz de Usuario
    i. La interfaz debe poder ser utilizada desde laptops y computadoras de escritorio de diferentes tamaños.
    ii. La interfaz debe mantener un diseño cohesivo en toda la aplicación.
    iii. La interfaz debe ser intuitiva para usuarios que no hayan interactuando con el sistema anteriormente.
    iv. Los colores y tipografía de la interfaz deben estar alineados con el diseño del Instituto Vía Diseño.
    v. La interfaz debe estar disponible en idioma español.

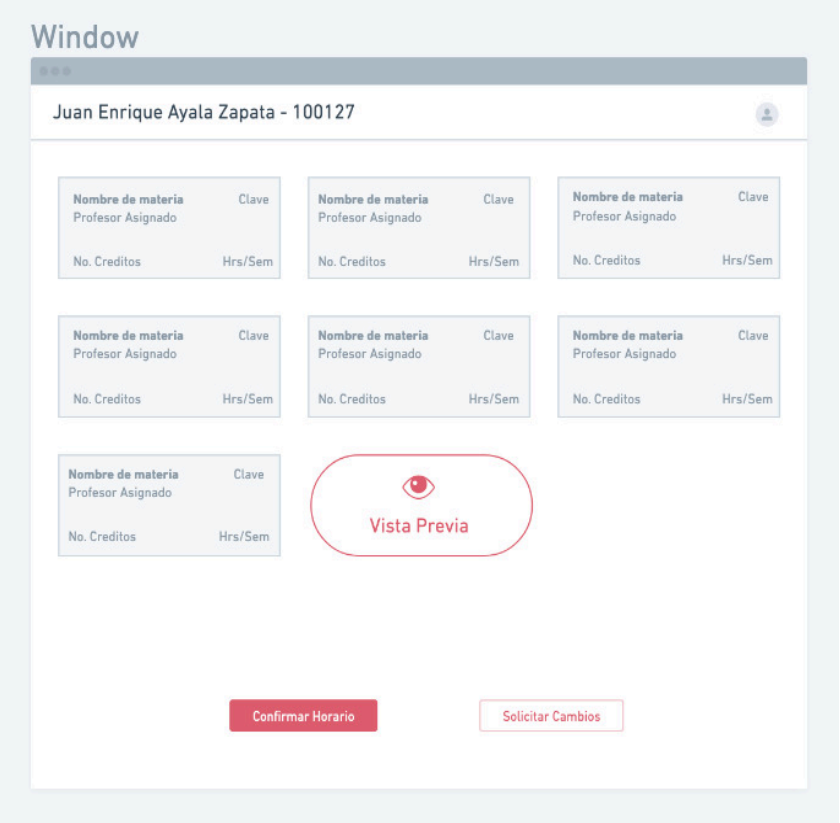


Figura 5. Interfaz de usuario para vista previa de materias del alumno.

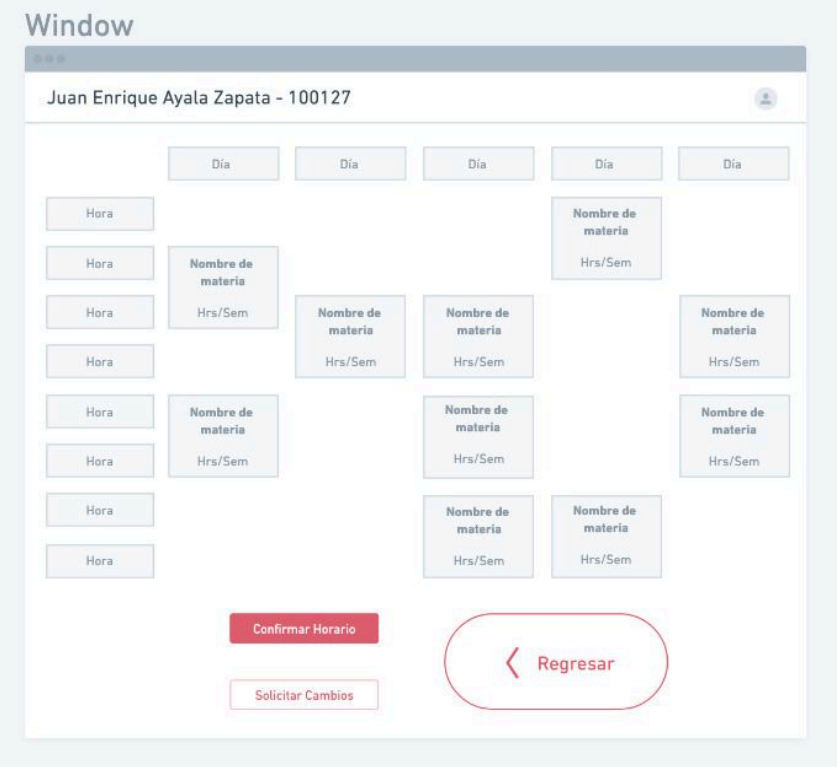


Figura 6. Interfaz de usuario de vista previa de horario del alumno.

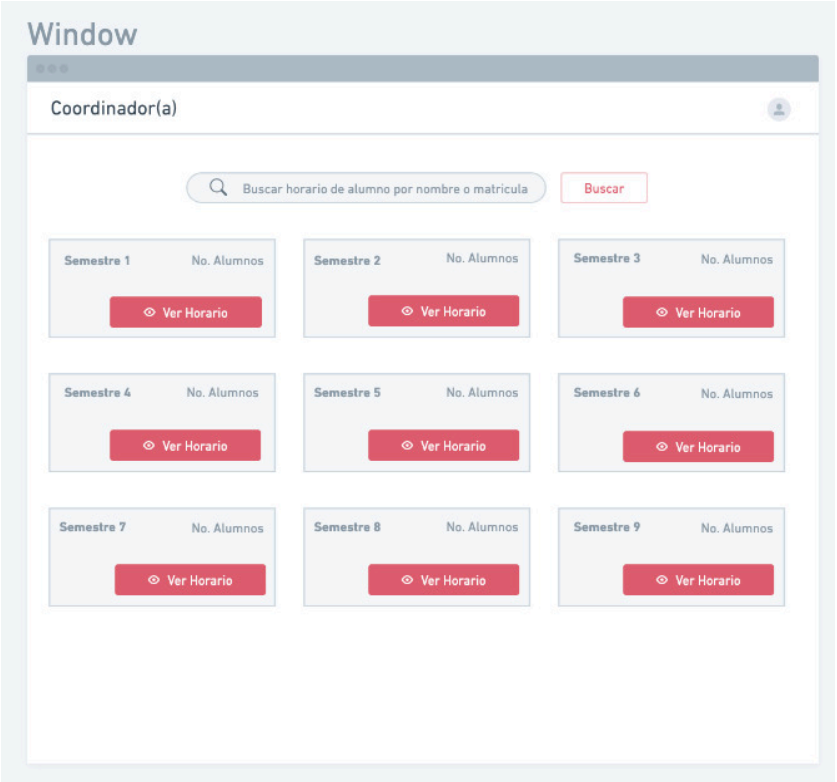


Figura 7. Interfaz de usuario de control de semestres del coordinador.

# Casos de Uso

- **Nombre:** Coordinador ingresa disponibilidad de profesores.
- **Descripción:** El coordinador es capaz de ingresar los horarios con los que se cuentan para que cada profesor imparte clases.
- **Precondiciones:**
    - No aplica.
- **Flujo:**
    1. Da inicio el caso de uso cuando el coordinador selecciona el tablero de profesores.
    2. El sistema lleva al coordinador al menú de maestros.
    3. El coordinador selecciona el profesor al que se le modificara su disponibilidad.
    4. Se despliega el menú de la disponibilidad del maestro.
    5. El coordinador modifica la disponibilidad.
    6. El coordinador selecciona “guardar”.
    7. El sistema guarda la disponibilidad del profesor.
    8. El sistema regresa al menú de maestros.
- **Flujo alterno:**
    1. Da inicio el caso de uso cuando el coordinador selecciona el tablero de profesores.
    2. El sistema lleva al coordinador al menú de maestros.
    3. El coordinador selecciona el profesor al que se le modificara su disponibilidad.
    4. Se despliega el menú de la disponibilidad del maestro.
    5. El coordinador modifica la disponibilidad.
    6. El coordinador selecciona “Cancelar”.
    7. El sistema no guarda datos.
    8. El sistema regresa al menú de maestros.
- **Requisitos especiales:**
    - No aplica.
- **Postcondiciones:**
    - Datos de los maestros almacenados para la producción del horario

- **Diagrama:**

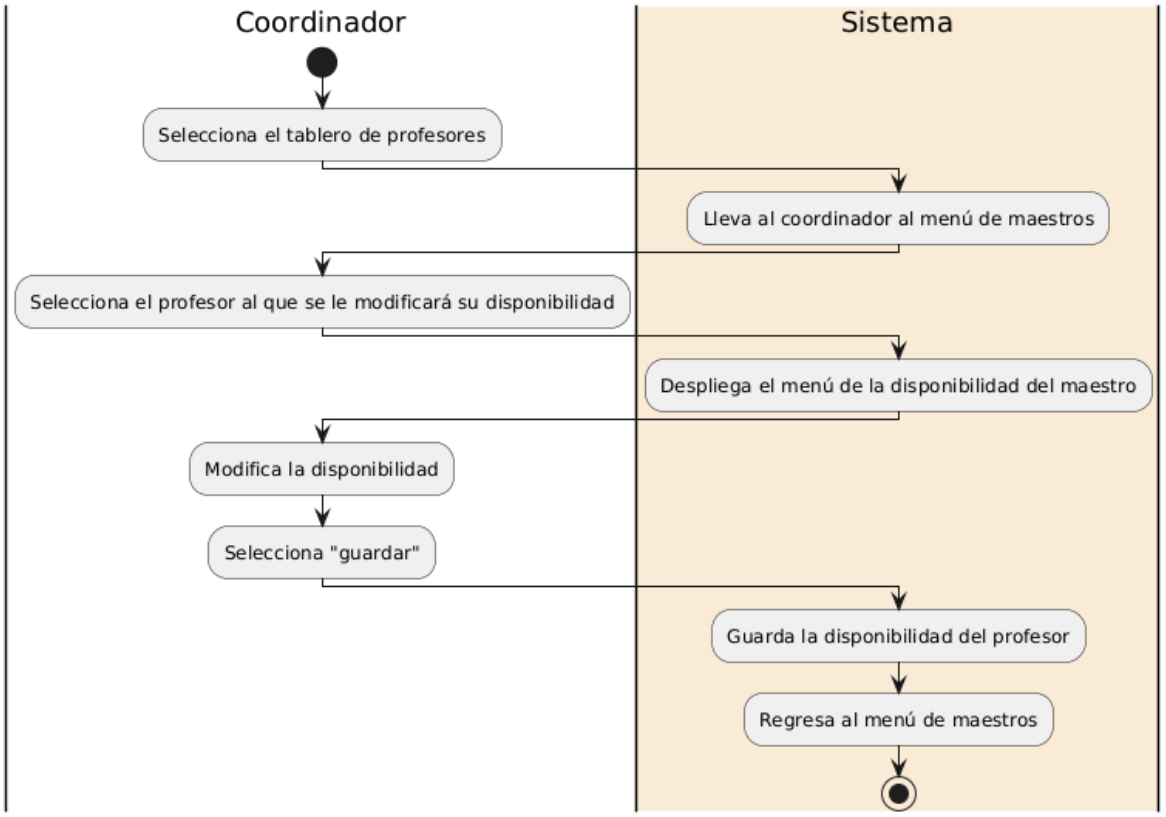


Figura 8. Diagrama de actividades para el caso de uso “Coordinador ingresa disponibilidad de profesores”.

- **Nombre:** Coordinador genera turnos de inscripción.
- **Descripción:** El coordinador tiene la capacidad de generar
- **Precondiciones:**
  - Horario previamente creado.
- **Flujo:**
  1. Da inicio cuando el coordinador selecciona “generar turnos de inscripción”.
  2. El programa asigna horarios a los alumnos.
  3. El programa guarda los horarios.
- **Flujo alterno:**
  - No aplica.
- **Requisitos especiales:**
  - No aplica.
- **Postcondiciones:**
  - Cada alumno tiene un turno asignado.
- **Diagrama:**

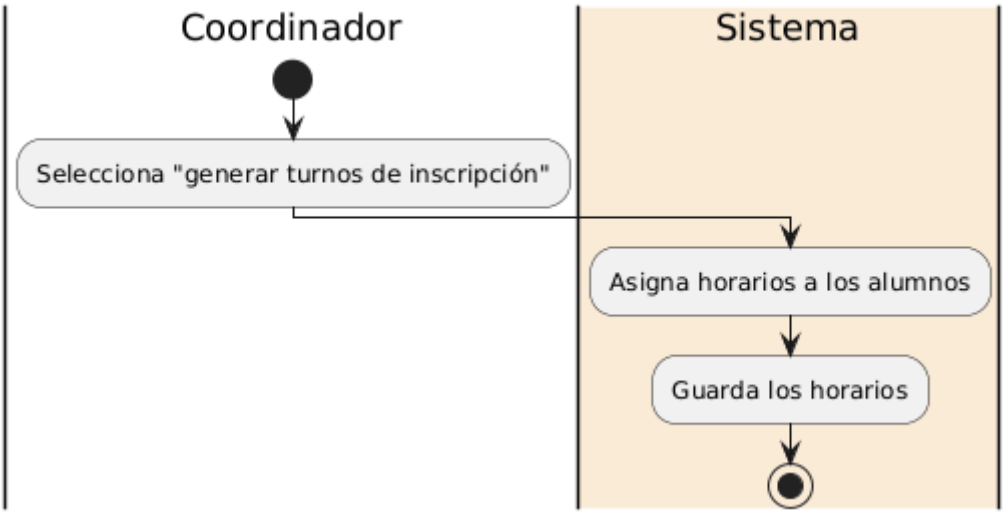


Figura 9. Diagrama de actividades para el caso de uso “Coordinador genera turno de inscripción”

- **Nombre:** Coordinador genera horario por periodo académico.
- **Descripción:** El coordinador tiene la capacidad de generar una propuesta de horario para el semestre.
- **Precondiciones:**
  - La previa modificación de la disponibilidad de los maestros.
- **Flujo:**
  1. Da inicio cuando el coordinador selecciona generar horarios.
  2. El sistema genera la propuesta de horario.
  3. El coordinador acepta la propuesta.
  4. El sistema guarda el horario.
- **Flujo alterno:**
  1. Da inicio cuando el coordinador selecciona generar horarios.
  2. El sistema genera la propuesta de horario.
  3. El coordinador modifica la propuesta.
  4. El coordinador acepta el horario actual.
  5. El sistema guarda el horario.
- **Requisitos especiales:**
  - No aplica.
- **Postcondiciones:**
  - Horario generado para la carrera correspondiente.
- **Diagrama:**

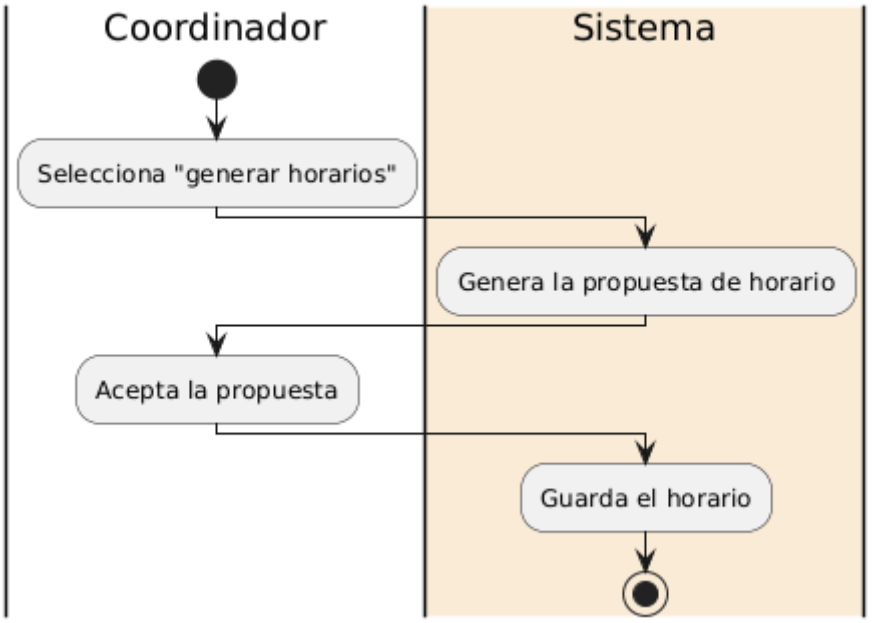


Figura 10. Diagrama de actividades para el caso de uso “Coordinador genera horario por periodo académico”

- **Nombre:** Coordinador consulta horario por periodo académico**.**
- **Descripción:** El coordinador consulta el horario generado previamente.
- **Precondiciones:**
  - Previa generación de horario.
- **Flujo:**
  1. Da inicio cuando el coordinador selecciona “consultar usuario".
  2. El sistema redirecciona al horario.
  3. El coordinador selecciona “regresar”.
  4. El sistema regresa a la interfaz anterior.
- **Flujo alterno:**
  - No aplica.
- **Requisitos especiales:**
  - No aplica.
- **Postcondiciones:**
  - Se puede ver el horario.
- **Diagrama:**

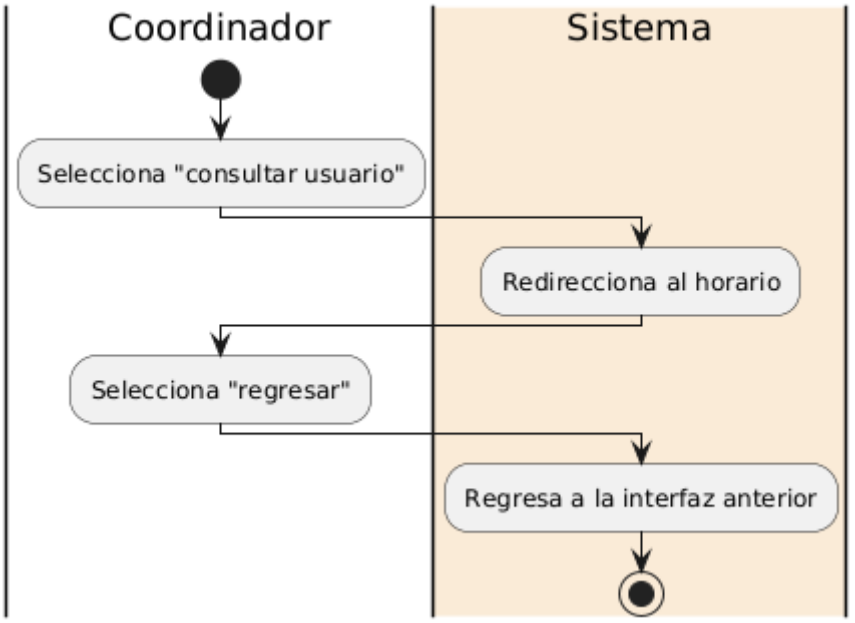


Figura 11. Diagrama de actividades para el caso de uso “Consulta horario por periodo académico”

- **Nombre:** Alumno registra aceptación de  propuesta de horario.
- **Descripción:** El alumno puede aceptar el horario que se le propone.
- **Precondiciones:**
    - Previa generación de horario.
    - Previa confirmación de inscripción.
- **Flujo:**
    1. Da inicio inmediatamente después que el alumno indique que quiere inscribirse.
    2. El sistema le despliega la propuesta de horario y la opción de aceptarlo.
    3. El alumno lo acepta.
    4. El sistema finaliza el proceso.
- **Flujo alterno:**
    - No aplica.
- **Requisitos especiales:**
    - No aplica.
- **Postcondiciones:**
    - Se puede ver el horario.
- **Diagrama:**

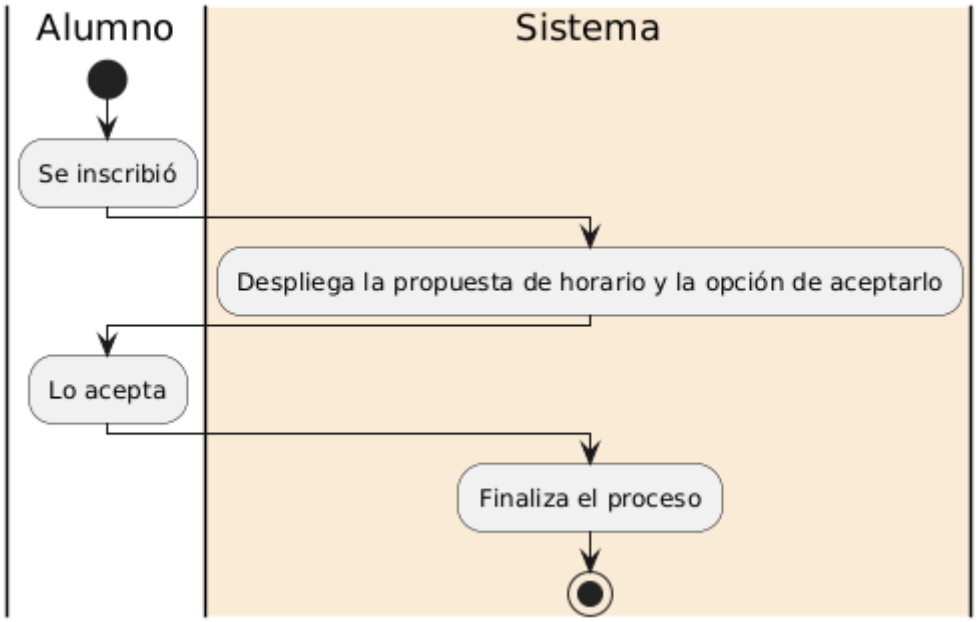


Figura 12. Diagrama de actividades para el caso de uso “Alumno registra aceptación de propuesta de horario”

- **Nombre:** Alumno consulta horario.
- **Descripción:** El coordinador consulta el horario generado previamente.
- **Precondiciones:**
  - Previa aceptación del horario.
- **Flujo:**
  1. Da inicio cuando el alumno selecciona “consultar usuario".
  2. El sistema redirecciona al horario.
  3. El alumno selecciona “regresar”.
  4. El sistema regresa a la interfaz anterior.
- **Flujo alterno:**
  - No aplica.
- **Requisitos especiales:**
  - No aplica.
- **Postcondiciones:**
  - Se puede ver el horario.
- **Diagrama:**

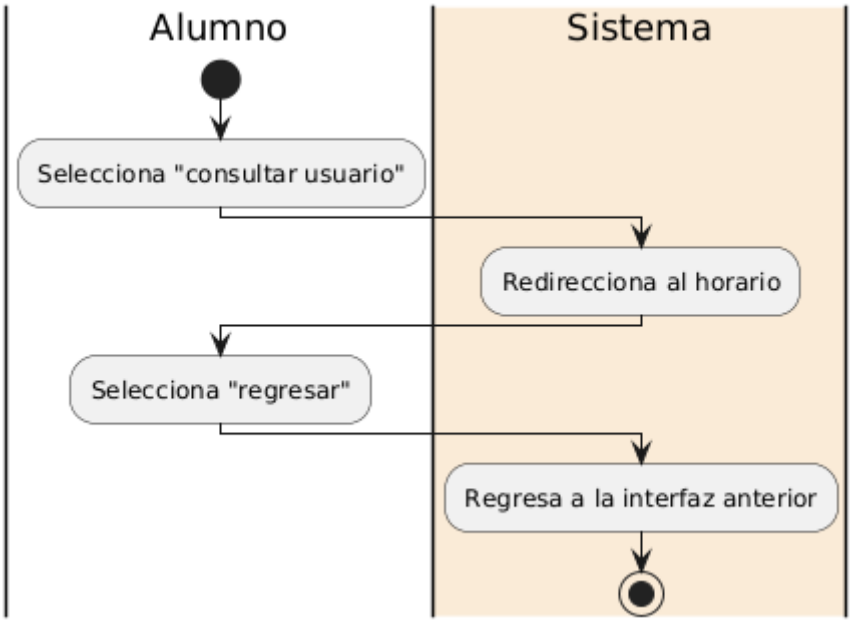


Figura 13. Diagrama de actividades para el caso de uso “Alumno consulta horario”

# Modelo Entidad-Relación

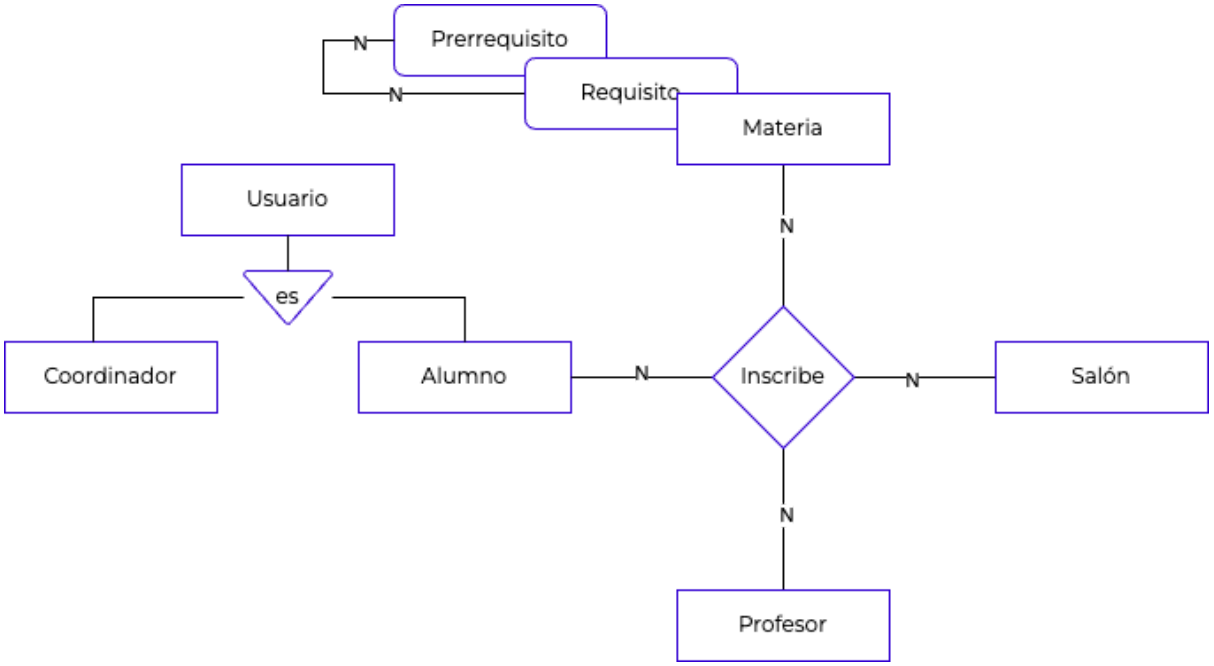


Figura 14. Modelo Entidad-Relación

## Diccionario de Datos

- Entidad: Usuario

| **Nombre atributo** | **Descripción** | **Tipo de dato** | **Ejemplo** |
|---|---|---|---|
| Contrasena | Contraseña para iniciar sesión. | Caracter(50) | “Ka2!saawd.?” |

Tabla 2. Diccionario de datos para entidad “Usuario”

- Entidad: Alumno

| **Nombre atributo** | **Descripción** | **Tipo de dato** | **Ejemplo** |
|---|---|---|---|
| Confirmacion | Indicador de confirmación de horario. | Bool | “true” |

Tabla 3. Diccionario de datos para entidad “Alumno”

- Entidad: Profesor

| **Nombre atributo** | **Descripción** | **Tipo de dato** | **Ejemplo** |
|---|---|---|---|
| horasDisponibles | Horas en las que el profesor puede impartir la materia. | JSON | {"lunes": ["08:00-10:00", "14:00-16:00"], "martes": ["09:00-11:00"]} |

| | | | |
|---|---|---|---|
| idProfesor | Número identificador del profesor | Entero | “300123” |

Tabla 4. Diccionario de datos para entidad “Profesor”

- Entidad: Salon

| **Nombre atributo** | **Descripción** | **Tipo de dato** | **Ejemplo** |
|---|---|---|---|
| idSalon | Número identificador de salón. | Entero | “2003” |
| Cupo | Cantidad máxima de estudiantes por salón. | Entero | “30” |
| Tipo | Tipo de salón. | Caracter(50) | “Taller manual” |

Tabla 5. Diccionario de Datos para entidad “Salón”

- Entidad: Materia

| **Nombre atributo** | **Descripción** | **Tipo de dato** | **Ejemplo** |
|---|---|---|---|
| Nombre | Nombre de la materia. | Caracter(50) | “Corte y Confección” |
| id | Número identificador de la materia. | Entero | “321” |
| HorasClase | Cantidad de horas de clase al día. | Flotante | (4.2) |
| Requisitos | El tipo de salón necesario para tomar la materia | Caracter(50) | “Taller de Joyería” |

Tabla 6. Diccionario de datos para entidad “Materia”

# Modelo Relacional

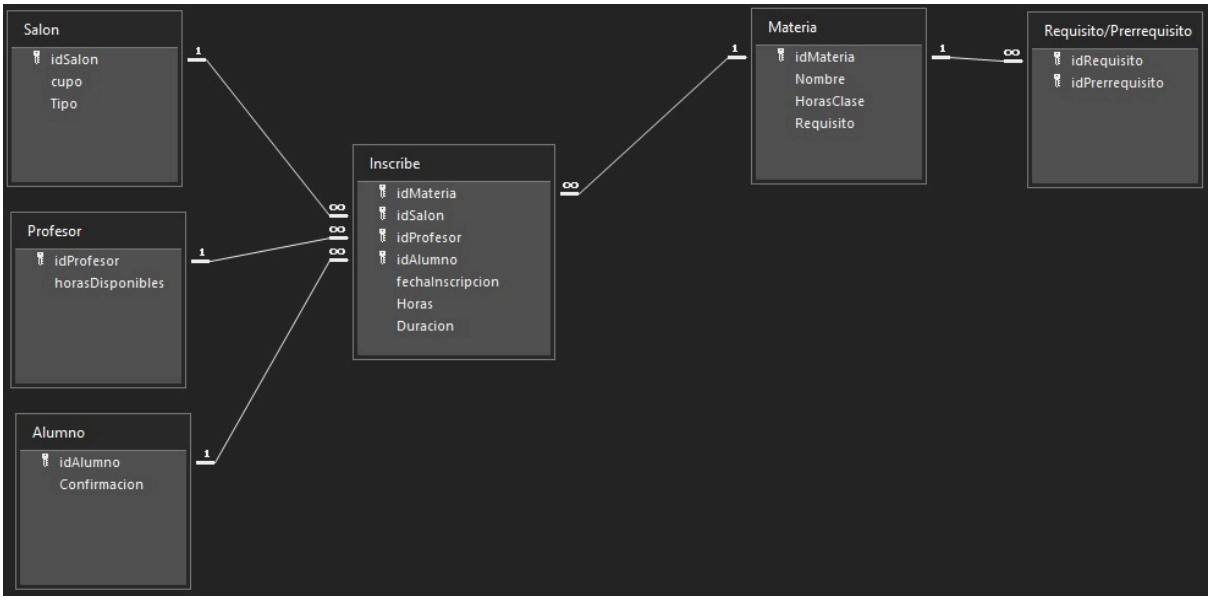


Figura 15. Modelo Relacional de “Vía Alta”

# Mapa del sitio

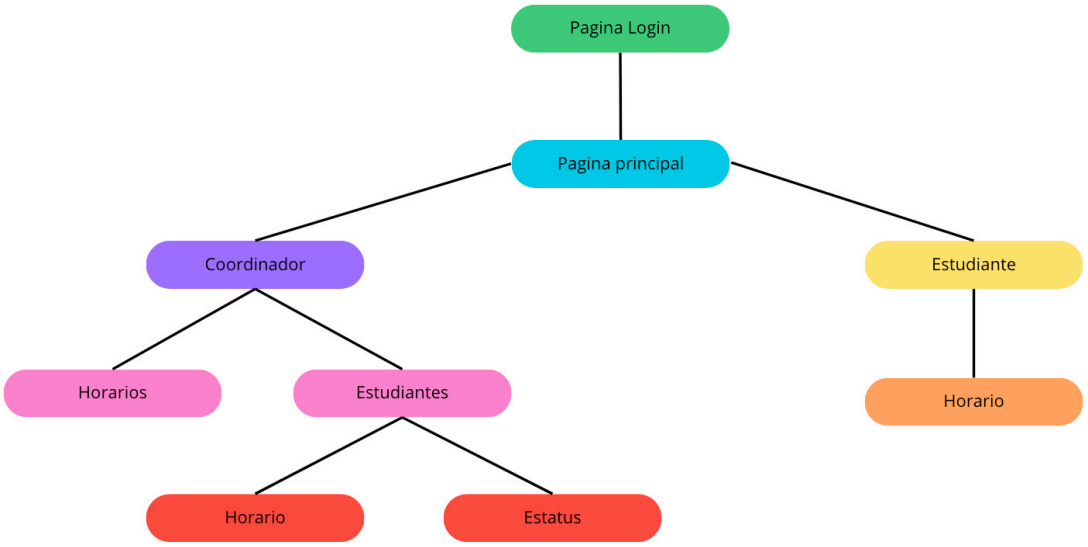


Figura 16. Mapa del Sitio de “Vía Alta”

# Guía de estilo de codificación

La guía de trabajo seleccionada para la realización de este proyecto es Airbnb JavaScript Style Guide.

- **Declaración de variable**
    a. Always use const or let to declare variables. Not doing so will result in global variables. We want to avoid polluting the global namespace.

```JavaScript
// bad
superPower = new SuperPower();

// good
const superPower = new SuperPower();
```

    b. Use one const or let declaration per variable or assignment.

```JavaScript
// bad
const items = getItems(),
    goSportsTeam = true,
    dragonball = 'z';

// bad
// (compare to above, and try to spot the mistake)
const items = getItems(),
    goSportsTeam = true;
    dragonball = 'z';

// good
const items = getItems();
const goSportsTeam = true;
const dragonball = 'z';
```

    c. Group all your consts and then group all your lets.

```JavaScript
// bad
let i, len, dragonball,
    items = getItems(),
    goSportsTeam = true;

// bad
let i;
```

```
const items = getItems();
let dragonball;
const goSportsTeam = true;
let len;

// good
const goSportsTeam = true;
const items = getItems();
let dragonball;
let i;
let length;
```

d. Assign variables where you need them, but place them in a reasonable place.

JavaScript

```
// bad - unnecessary function call
function checkName(hasName) {
  const name = getName();

  if (hasName === 'test') {
    return false;
  }

  if (name === 'test') {
    this.setName('');
    return false;
  }

  return name;
}

// good
function checkName(hasName) {
  if (hasName === 'test') {
    return false;
  }

  const name = getName();

  if (name === 'test') {
    this.setName('');
    return false;
  }

  return name;
```

```
}
```

e. Don’t chain variable assignments.

```JavaScript
// bad
(function example() {
  // JavaScript interprets this as
  // let a = ( b = ( c = 1 ) );
  // The let keyword only applies to variable a; variables b and c become
  // global variables.
  let a = b = c = 1;
}());

console.log(a); // throws ReferenceError
console.log(b); // 1
console.log(c); // 1

// good
(function example() {
  let a = 1;
  let b = a;
  let c = a;
}());

console.log(a); // throws ReferenceError
console.log(b); // throws ReferenceError
console.log(c); // throws ReferenceError

// the same applies for `const`
```

f. Avoid using unary increments and decrements (++, --).

```JavaScript
// bad

const array = [1, 2, 3];
let num = 1;
num++;
--num;

let sum = 0;
let truthyCount = 0;
for (let i = 0; i < array.length; i++) {
  let value = array[i];
```

```
  sum += value;
  if (value) {
    truthyCount++;
  }
}

// good

const array = [1, 2, 3];
let num = 1;
num += 1;
num -= 1;

const sum = array.reduce((a, b) => a + b, 0);
const truthyCount = array.filter(Boolean).length;
```

g. Avoid linebreaks before or after = in an assignment. If your assignment violates max-len, surround the value in parens.

JavaScript

```
// bad
const foo =
  superLongLongLongLongLongLongLongFunctionName();

// bad
const foo
  = 'superLongLongLongLongLongLongLongString';

// good
const foo = (
  superLongLongLongLongLongLongLongFunctionName()
);

// good
const foo = 'superLongLongLongLongLongLongLongString';
```

h. Disallow unused variables.

JavaScript

```
// bad

const some_unused_var = 42;

// Write-only variables are not considered as used.
let y = 10;
```

```
y = 5;

// A read for a modification of itself is not considered as used.
let z = 0;
z = z + 1;

// Unused function arguments.
function getX(x, y) {
    return x;
}

// good

function getXPlusY(x, y) {
  return x + y;
}

const x = 1;
const y = a + 2;

alert(getXPlusY(x, y));

// 'type' is ignored even if unused because it has a rest property sibling.
// This is a form of extracting an object that omits the specified keys.
const { type, ...coords } = data;
// 'coords' is now the 'data' object without its 'type' property.
```

- **Strings**
    a. Use single quotes '' for strings.

JavaScript

```
// bad
const name = "Capt. Janeway";

// bad - template literals should contain interpolation or newlines
const name = `Capt. Janeway`;

// good
const name = 'Capt. Janeway';
```

b. Strings that cause the line to go over 100 characters should not be written across multiple lines using string concatenation.

```JavaScript
// bad
const errorMessage = 'This is a super long error that was thrown because \
of Batman. When you stop to think about how Batman had anything to do \
with this, you would get nowhere \
fast.';

// bad
const errorMessage = 'This is a super long error that was thrown because ' +
  'of Batman. When you stop to think about how Batman had anything to do ' +
  'with this, you would get nowhere fast.';

// good
const errorMessage = 'This is a super long error that was thrown because of
Batman. When you stop to think about how Batman had anything to do with
this, you would get nowhere fast.';
```

c. When programmatically building up strings, use template strings instead of concatenation.

```JavaScript
// bad
function sayHi(name) {
  return 'How are you, ' + name + '?';
}

// bad
function sayHi(name) {
  return ['How are you, ', name, '?'].join();
}

// bad
function sayHi(name) {
  return `How are you, ${ name }?`;
}

// good
function sayHi(name) {
  return `How are you, ${name}?`;
}
```

d. Never use eval() on a string; it opens too many vulnerabilities.

e. Do not unnecessarily escape characters in strings.

```JavaScript
// bad
const foo = '\'this\' \i\s \"quoted\"';

// good
const foo = '\'this\' is "quoted"';
const foo = `my name is '${name}'`;
```

- **Funciones**
  a. Use named function expressions instead of function declarations.

```JavaScript
// bad
function foo() {
  // ...
}

// bad
const foo = function () {
  // ...
};

// good
// lexical name distinguished from the variable-referenced invocation(s)
const short = function longUniqueMoreDescriptiveLexicalFoo() {
  // ...
};
```

b. Wrap immediately invoked function expressions in parentheses.

```JavaScript
// immediately-invoked function expression (IIFE)
(function () {
  console.log('Welcome to the Internet. Please follow me.');
}());
```

c. Never declare a function in a non-function block (if, while, etc). Assign the function to a variable instead. Browsers will allow you to do it, but they all interpret it differently, which is bad news bears.

d. ECMA-262 defines a block as a list of statements. A function declaration is not a statement.

```JavaScript
// bad
if (currentUser) {
  function test() {
    console.log('Nope.');
  }
}

// good
let test;
if (currentUser) {
  test = () => {
    console.log('Yup.');
  };
}
```

e. Never name a parameter arguments. This will take precedence over the arguments object that is given to every function scope.

```JavaScript
// bad
function foo(name, options, arguments) {
  // ...
}

// good
function foo(name, options, args) {
  // ...}
```

f. Never use arguments, opt to use rest syntax ... instead.

```JavaScript
// bad
function concatenateAll() {
  const args = Array.prototype.slice.call(arguments);
  return args.join('');
}

// good
function concatenateAll(...args) {
  return args.join('');
}
```

g. Use default parameter syntax rather than mutating function arguments.

```JavaScript
// really bad
function handleThings(opts) {
  // No! We shouldn't mutate function arguments.
  // Double bad: if opts is falsy it'll be set to an object which may
  // be what you want but it can introduce subtle bugs.
  opts = opts || {};
  // ...
}

// still bad
function handleThings(opts) {
  if (opts === void 0) {
    opts = {};
  }
  // ...
}

// good
function handleThings(opts = {}) {
  // ...
}
```

h. Avoid side effects with default parameters.

```JavaScript
let b = 1;
// bad
function count(a = b++) {
  console.log(a);
}
count();  // 1
count();  // 2
count(3); // 3
count();  // 3
```

i. Always put default parameters last.

```JavaScript
// bad
function handleThings(opts = {}, name) {
  // ...
```

```
}

// good
function handleThings(name, opts = {}) {
  // ...
}
```

j. Never use the Function constructor to create a new function.

```JavaScript
// bad
const add = new Function('a', 'b', 'return a + b');

// still bad
const subtract = Function('a', 'b', 'return a - b');
```

k. Spacing in a function signature.

```JavaScript
// bad
const f = function(){};
const g = function (){};
const h = function() {};

// good
const x = function () {};
const y = function a() {};
```

l. Never mutate parameters.

```JavaScript
// bad
function f1(obj) {
  obj.key = 1;
}

// good
function f2(obj) {
  const key = Object.prototype.hasOwnProperty.call(obj, 'key') ? obj.key :
1;
}
```

m. Never reassign parameters.

```JavaScript
// bad
function f1(a) {
  a = 1;
  // ...
}

function f2(a) {
  if (!a) { a = 1; }
  // ...
}

// good
function f3(a) {
  const b = a || 1;
  // ...
}
function f4(a = 1) {
  // ...
}
```

n. Prefer the use of the spread syntax ... to call variadic functions.

```JavaScript
// bad
const x = [1, 2, 3, 4, 5];
console.log.apply(console, x);

// good
const x = [1, 2, 3, 4, 5];
console.log(...x);

// bad
new (Function.prototype.bind.apply(Date, [null, 2016, 8, 5]));

// good
new Date(...[2016, 8, 5]);
```

o. Functions with multiline signatures, or invocations, should be indented just like every other multiline list in this guide: with each item on a line by itself, with a trailing comma on the last item.

```JavaScript
// bad
function foo(bar,
             baz,
             quux) {
  // ...
}

// good
function foo(
  bar,
  baz,
  quux,
) {
  // ...
}

// bad
console.log(foo,
  bar,
  baz);

// good
console.log(
  foo,
  bar,
  baz,
);
```

- **Comentarios**
    a. Use /** ... */ for multiline comments.

```JavaScript
// bad
// make() returns a new element
// based on the passed in tag name
//
// @param {String} tag
// @return {Element} element
function make(tag) {

  // ...
```

```
  return element;
}

// good
/**
 * make() returns a new element
 * based on the passed-in tag name
 */
function make(tag) {

  // ...

  return element;
}
```

b. Use // for single line comments. Place single line comments on a newline above the subject of the comment. Put an empty line before the comment unless it's on the first line of a block.

JavaScript

```
// bad
const active = true;  // is current tab

// good
// is current tab
const active = true;

// bad
function getType() {
  console.log('fetching type...');
  // set the default type to 'no type'
  const type = this.type || 'no type';
  return type;
}

// good
function getType() {
  console.log('fetching type...');

  // set the default type to 'no type'
  const type = this.type || 'no type';

  return type;
}

// also good
```

```
function getType() {
  // set the default type to 'no type'
  const type = this.type || 'no type';

  return type;
}
```

c. Start all comments with a space to make it easier to read.

```JavaScript
// bad
//is current tab
const active = true;

// good
// is current tab
const active = true;

// bad
/**
 *make() returns a new element
 *based on the passed-in tag name
 */
function make(tag) {

  // ...

  return element;
}

// good
/**
 * make() returns a new element
 * based on the passed-in tag name
 */
function make(tag) {

  // ...

  return element;
}
```

d. Prefixing your comments with FIXME or TODO helps other developers quickly understand if you're pointing out a problem that needs to be revisited, or if you're suggesting a solution to

the problem that needs to be implemented. These are different than regular comments because they are actionable.

e. Use // FIXME: to annotate problems.

```JavaScript
class Calculator extends Abacus {
  constructor() {
    super();

    // FIXME: shouldn't use a global here
    total = 0;
  }
}
```

f. Use // TODO: to annotate solutions to problems.

```JavaScript
class Calculator extends Abacus {
  constructor() {
    super();

    // TODO: total should be configurable by an options param
    this.total = 0;
  }
}
```

- **Objetos**

a. Use the literal syntax for object creation.

```JavaScript
// bad
const item = new Object();

// good
const item = {};
```

b. Use computed property names when creating objects with dynamic property names.

```JavaScript

function getKey(k) {
  return `a key named ${k}`;
}
```

```
// bad
const obj = {
  id: 5,
  name: 'San Francisco',
};
obj[getKey('enabled')] = true;

// good
const obj = {
  id: 5,
  name: 'San Francisco',
  [getKey('enabled')]: true,
};
```

c. Use object method shorthand.

```
JavaScript
// bad
const atom = {
  value: 1,

  addValue: function (value) {
    return atom.value + value;
  },
};

// good
const atom = {
  value: 1,

  addValue(value) {
    return atom.value + value;
  },
};
```

d. Use property value shorthand.

```
JavaScript
// bad
const atom = {
  value: 1,

  addValue: function (value) {
    return atom.value + value;
```

```
  },
};

// good
const atom = {
  value: 1,

  addValue(value) {
    return atom.value + value;
  },
};
```

e. Group your shorthand properties at the beginning of your object declaration.

```
JavaScript
const anakinSkywalker = 'Anakin Skywalker';
const lukeSkywalker = 'Luke Skywalker';

// bad
const obj = {
  episodeOne: 1,
  twoJediWalkIntoACantina: 2,
  lukeSkywalker,
  episodeThree: 3,
  mayTheFourth: 4,
  anakinSkywalker,
};

// good
const obj = {
  lukeSkywalker,
  anakinSkywalker,
  episodeOne: 1,
  twoJediWalkIntoACantina: 2,
  episodeThree: 3,
  mayTheFourth: 4,
};
```

f. Only quote properties that are invalid identifiers.

```
JavaScript
// bad
const bad = {
  'foo': 3,
```

```
  'bar': 4,
  'data-blah': 5,
};

// good
const good = {
  foo: 3,
  bar: 4,
  'data-blah': 5,
};
```

g. Do not call Object.prototype methods directly, such as hasOwnProperty, propertyIsEnumerable, and isPrototypeOf.

JavaScript

```
// bad
console.log(object.hasOwnProperty(key));

// good
console.log(Object.prototype.hasOwnProperty.call(object, key));

// better
const has = Object.prototype.hasOwnProperty; // cache the lookup once, in
module scope.
console.log(has.call(object, key));

// best
console.log(Object.hasOwn(object, key)); // only supported in browsers that
support ES2022

/* or */
import has from 'has'; // https://www.npmjs.com/package/has
console.log(has(object, key));
/* or */
console.log(Object.hasOwn(object, key));
//https://www.npmjs.com/package/object.hasown
```

h. refer the object spread syntax over Object.assign to shallow-copy objects. Use the object rest parameter syntax to get a new object with certain properties omitted.

JavaScript

```
// very bad
const original = { a: 1, b: 2 };
```

```
const copy = Object.assign(original, { c: 3 }); // this mutates `original`
ಠ_ಠ
delete copy.a; // so does this

// bad
const original = { a: 1, b: 2 };
const copy = Object.assign({}, original, { c: 3 }); // copy => { a: 1, b: 2,
c: 3 }

// good
const original = { a: 1, b: 2 };
const copy = { ...original, c: 3 }; // copy => { a: 1, b: 2, c: 3 }

const { a, ...noA } = copy; // noA => { b: 2, c: 3 }
```

- **Estatutos de control**
    a. In case your control statement (if, while etc.) gets too long or exceeds the maximum line length, each (grouped) condition could be put into a new line. The logical operator should begin the line.

JavaScript

```
// bad
if ((foo === 123 || bar === 'abc') && doesItLookGoodWhenItBecomesThatLong()
&& isThisReallyHappening()) {
  thing1();
}

// bad
if (foo === 123 &&
  bar === 'abc') {
  thing1();
}

// bad
if (foo === 123
  && bar === 'abc') {
  thing1();
}

// bad
if (
  foo === 123 &&
  bar === 'abc'
) {
```

```
  thing1();
}

// good
if (
  foo === 123
  && bar === 'abc'
) {
  thing1();
}

// good
if (
  (foo === 123 || bar === 'abc')
  && doesItLookGoodWhenItBecomesThatLong()
  && isThisReallyHappening()
) {
  thing1();
}

// good
if (foo === 123 && bar === 'abc') {
  thing1();
}
```

b. Don't use selection operators in place of control statements.

JavaScript

```
// bad
!isRunning && startRunning();

// good
if (!isRunning) {
  startRunning();
}
```

# Plan de comunicación

| INTERESADOS EXTERNOS | | | | | |
|---|---|---|---|---|---|
| **Concepto** | **Descripción** | **Frecuencia** | **Canal** | **Interesados** | **Encargado** |
| Retroalimentación semanal | Sesión de presentación de avances para una retroalimentación personalizada | Semanal (miércoles a 5:00 PM) | Zoom | Todo el equipo de desarrollo, así como los interesados del proyecto, Desarrollador Bernardo y Responsable de Negocios Adrian | Mateo |
| Presentaciones de avance | Sesión de presentación de avances y clarificación de dudas para todos los equipos | Semanal | Presencial (Tec) | Todo el equipo de desarrollo, así como los interesados del proyecto, Desarrollador Bernardo y Responsable de Negocios Adrian | Carlos |
| Resolución de dudas asíncrona | Espacio para realizar preguntas entre sesiones | Como se requiera | WhatsApp | Todo el equipo de desarrollo, así como los interesados del proyecto y Desarrollador Bernardo y Responsable de Negocios Adrian | Todo el equipo de desarrollo |
| Revisión de uso de API | Sesión de educación sobre el uso de API de Vía Diseño | TBD | TBD | Todo el equipo de desarrollo, así como los interesados del proyecto y Desarrollador Bernardo | Desarollador Bernardo |
| Invitación a presentaciones | Comunicar datos relevantes a presentaciones de avance | Con dos días de antemano | WhatsApp | Todo el equipo de desarrollo, así como los interesados del proyecto, Desarrollador Bernardo y Responsable de Negocios Adrian | Kike |

Tabla 7. Plan de comunicación con los interesados externos.

| INTERESADOS INTERNOS | | | | | |
|---|---|---|---|---|---|
| **Concepto** | **Descripción** | **Frecuencia** | **Canal** | **Interesados** | **Encargado** |
| Chequeo de proyecto y su estado | Resolución de dudas para que todo el trabajo sea coherente | Diaria | Whatsapp | Todo el equipo de desarrollo | Leo |
| Reuniones de equipo | Junta de equipo para trabajar en conjunto en los avances necesarios | De Lunes a Viernes | Presencial | Todo el equipo de desarrollo | Fermin |

Tabla 8. Plan de comunicación con los interesados internos.

# Plan de trabajo actualizado

- Terminar el diseño de las interfaces restantes.
- Terminar la descripción de los casos de uso de prioridad media y baja.
- Realizar pruebas a nuestro modelo relacional para este proyecto.
- Realizar entrenamiento/capacitación sobre el uso de la RestfulAPI de Vía Diseño.
- Realizar los diagramas de secuencia.
- Comenzar con el desarrollo del sistema.

# Aprendizajes adquiridos

Durante estas primeras semanas del proyecto, todos los miembros del equipo fuimos aprendiendo sobre el modelo Entidad-Relación, así como todos los aspectos clave en la correcta traducción de este. Si bien, algunos de los miembros del equipo cuentan con conocimientos sobre el framework de desarrollo que se usará, se realizan sesiones de entrenamiento con aquellos miembros del equipo que no saben todavía muy bien trabajar con este framework.