**Curso de desarrollo de software**

**Examen Parcial**

# **Normas**:

1. No compartir respuestas/consultas con tus compañeros a través de chats, redes sociales u otros medios digitales. Se va a eliminar la evaluación de manera automática si un estudiante es sorprendido.
2. Todo acto anti-ético será amonestado y registrado en el historial del estudiante.
3. Pese a lo útil que es la Inteligencia Artificial con el chatGPT o herramientas similares no se permite el uso de herramientas IA en el examen.
4. Presenta imágenes de tus respuestas junto con el código desarrollado. Todo examen que solo presenta código sin nada que lo soporte o respuestas sin explicación o puntuales será no considerada en la calificación. Mira un ejemplo:

Pregunta: ¿Las pruebas también sirven como parte de la documentación?

Respuesta de un estudiante: SI si respondes así , la nota es 0

Ejemplo de respuestas: Si las pruebas también pueden servir como documentación para las funciones que se implementa en una aplicación. Otros desarrolladores del equipo pueden leer tus pruebas y descubrir fácilmente qué busca lograr tu código.

Al crear las pruebas primero, puede asegurarse de que la función que está probando funcione de la forma prevista y evitar sorpresas en el futuro.

Sube tus respuestas a un repositorio llamado ExamenParcial-CC3S2 y dentro escribe las carpetas de cada una de las partes (preguntas) del examen. Escribe un archivo .md para explicar tus respuestas.

**Parte 1 Fundamentos de Ruby**

(3 puntos) El comando grep de Unix busca en archivos líneas que coincidan con una expresión regular. Tu tarea es implementar un comando grep simplificado, que admita la búsqueda de cadenas fijas. Busca en archivos líneas que coincidan con una cadena de búsqueda y devuelva todas las líneas coincidentes.

El comando grep toma tres argumentos:

* La cadena a buscar.
* Cero o más indicadores para personalizar el comportamiento del comando.
* Uno o más archivos para buscar.

Luego lee el contenido de los archivos especificados (en el orden especificado), busca las líneas que contienen la cadena de búsqueda y finalmente devuelve esas líneas en el orden en que fueron encontradas. Al buscar en varios archivos, cada línea coincidente va precedida por el nombre del archivo y dos puntos (':').

El comando grep admite los siguientes indicadores:

* -n Antepone el número de línea y dos puntos (':') a cada línea en la salida, colocando el número después del nombre del archivo (si está presente).
* -l Muestra solo los nombres de los archivos que contienen al menos una línea coincidente.
* -i Coincidencia utilizando una comparación que no distingue entre mayúsculas y minúsculas.
* -v Invierte el programa: recopila todas las líneas que no coinciden.
* -x Busca solo líneas donde la cadena de búsqueda coincida con la línea completa.

(3 puntos) El principio abierto-cerrado dice que una clase debe estar abierta a la extensión, pero cerrada a la modificación. Una extensión en el caso de Ruby sería agregar variables y métodos de instancia y una modificación sería modificar o eliminar variables o métodos de instancia existentes. El principio abierto-cerrado se escribió principalmente para abordar problemas con el software compilado escrito en lenguajes de programación que son menos expresivos que Ruby. En Ruby, prácticamente todas las clases están abiertas tanto para extensión como para modificación.

El propio Ruby ignora por completo el principio de abierto-cerrado y trabaja activamente para asegurarse de que las clases no estén cerradas para modificaciones. Uno de los cambios más significativos en el modelo de objetos de Ruby ocurrió con la adición de clases de origen. Las clases de origen son clases internas que se utilizan para permitir la implementación de Module#prepend. Las clases de origen agregaron una gran cantidad de complejidad al modelo de objetos de Ruby, con el único propósito de facilitar aún más la modificación por parte de los programadores para sobreescribir métodos singleton y llamar a super para obtener el comportamiento predeterminado.

Digamos que realmente deseas intentar aplicar el principio de abierto y cerrado en Ruby. ¿Cómo lo harías? Cerrar una clase para su extensión y modificación es tan fácil como llamar a ClassName.freeze, pero cerrarla para su modificación y dejarla abierta para su extensión es más difícil.

Hay tres formas generales de agregar métodos a las clases. Una es agregarlos incluyendo un módulo que los defina en la clase, y otra es anteponer un módulo que los defina a la clase. Por lo tanto, si deseas evitar la modificación, puede sobreeescribir include y prepend y hacer que generen una excepción si alguno de los módulos pasados tiene métodos de instancia que se superponen con los métodos de instancia de la clase. Querrás considerar métodos públicos, protegidos y privados al hacerlo. Primero puede crear la clase OpenClosed y agregarle un método singleton meths, devolviendo todos los métodos de instancia en la clase dada.

Ten en cuenta que instance\_methods devuelve métodos públicos y protegidos, por lo que debe agregarle los métodos privados:

class OpenClosed

def self.meths(m)

m.instance\_methods + m.private\_instance\_methods

end

Completa lo siguiente:

* Sobreescribe el método isingleton include: def self.include(\*mods)
* Incluye un alias include como prepend. Cuando se usan alias, el método super usa el nombre del alias, por lo que esto no rompe nada: singleton\_class.alias\_method :prepend,…
* Realiza lo mismo con el método de extend que hiciste con prepend e include.
* La tercera forma de agregar métodos a una clase es definirlos directamente en la clase. Puedes utilizar method\_added, que se llama después de que se haya agregado el método, momento en el que la clase o módulo ya se ha modificado. Dado que el method\_added se llama directamente después de cada método, puede deshacer la adición del método y generar una excepción siempre que tengas un alias para el método sobreescrito del método recién definido con el alias. Primero, debes crear alias para todos los métodos agregando un doble guión bajo al método:

meths(self).each do |method|

alias\_name = :"\_\_#{method}"

alias\_method alias\_name, method

end

Luego puedes definir method\_added. Utiliza una variable local que se define fuera del método y que se modifica dentro del método.

* Si el método comienza con un guión bajo doble y aún no está definido, alguien está intentando sobreescribir los métodos con alias. Sobrescribe el método con alias volviendo a asignar un alias al método original.
* Si el método no comienza con el doble guión bajo y ya está definido, puede solucionar el problema asignando un alias al método con alias nuevamente al método original. Realiza este opción.
* Podrías explotar una condición de carrera en la implementación intentando sobreescribir métodos en diferentes subprocesos en un bucle, ya sea explotando los intervalos de tiempo en los que check\_method está establecido en falso, o haciendo que un subproceso sobreescrine el método regular y otro subproceso sobreescribe el método de doble guión en al mismo tiempo. Remueve todo e por completo.

Debido a que un usuario siempre puede encontrar una manera de sobreescribir los métodos que tu estás sobreescribiendo para intentar evitar que modifiquen la clase, no tiene sentido intentar que las clases de Ruby estén abiertas para extensión y cerradas para modificación. Tus opciones están congeladas y cerradas tanto para modificación como para extensión, o descongeladas y abiertas tanto para modificación como para extensión.

(2 puntos) Responde las siguientes preguntas

(Escoge dos preguntas)

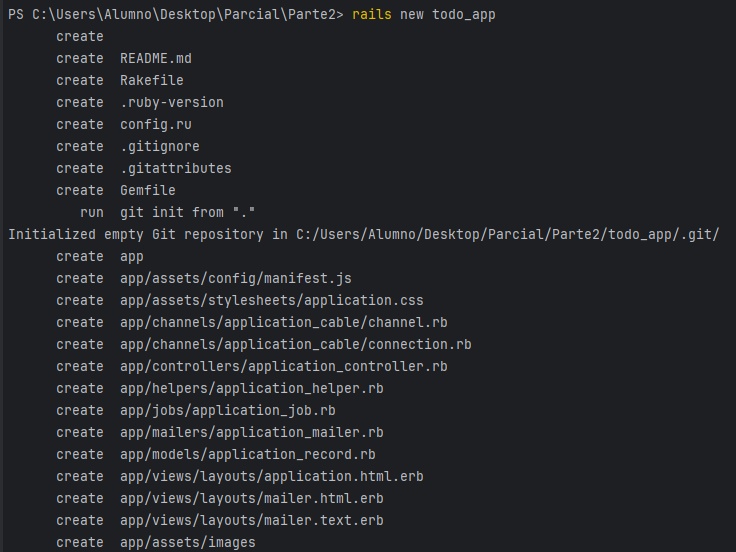
1. Si los métodos de clase son métodos de instancia, ¿qué clase contiene esos métodos de instancia?
2. ¿Cuándo es una mala idea implementar una abstracción mediante metaprogramación?
3. ¿Siempre es malo utilizar abstracciones en el código de prueba?
4. ¿Cuál es el requisito previo más importante antes de comenzar una refactorización?
5. ¿Es mejor utilizar un enfoque de desarrollo del lado del cliente o del lado del servidor al diseñar una aplicación que implica la entrada de datos a través de formularios HTML?

**Parte 2: Introducción a Rails (4 puntos)**

**Esta pregunta para que puntúe tienes que responder todos los ítems faltantes. Es evidencia de lo avanzado en el curso.**

Para crear una aplicación Rails, usaremos el comando rails new <app\_name>- Escribe lo siguiente en el símbolo del sistema e ingresa:

>> rails new todo\_app



El comando se habrá ejecutado correctamente si ves una lista de <create> <nombre de archivo>, seguida de otra lista de Using <gem>. El comando rails new crea el esqueleto de una aplicación simple que se ejecuta.

Este resultado detalla los archivos y gemas que se instalaron. Si ejecutas ls, deberías ver un nuevo directorio llamado "todo\_app".

Ahora que tenemos el framework de nuestra aplicación, es hora de definir los modelos de la aplicación. En otras palabras, ¡vamos a crear la clase "To Do"! El comando rails Generate es una poderosa herramienta que nos ayuda a hacer precisamente eso.

Cambie el directorio a "todo\_app", luego escriba e ingrese lo siguiente en el símbolo del sistema:

>> rails generate scaffold todo description:string

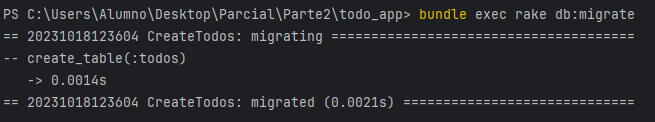
¿Qué está pasando aquí ? Si el comando se ejecutó correctamente, deberías ver una vez más una lista de resultados tanto para create <nombre de archivo> como para invoke <generator>.

Los argumentos de los comandos de invoke son "generadores". Los generadores (generators) pueden considerarse como scripts para generar elementos útiles en Rails. Para crear el modelo de una entidad, se recurre a varios generadores. La tercera palabra, "scaffold", indica que nos gustaría crear un recurso llamado "todo" con un único atributo "descripción" de tipo "cadena". La ejecución de este comando conduce a la creación de nuevos archivos de migración, base de datos, vista (ERB, HTML, CCS) y controlador.

**Base de datos**

Bien, ahora que tenemos un modelo, creemos una base de datos para que podamos conservar cualquier información que la aplicación quiera almacenar. Sin crear una base de datos, ningún cambio que realicemos se registrará en el esquema de la base de datos. Puedes considerar este paso como construir la conexión entre el modelo y la base de datos del diagrama general. Aún dentro de la raíz del directorio "todo\_app", escriba e ingresa lo siguiente en el símbolo del sistema (recuerda la versión de tu trabajo):

>> bundle exec rake db:migrate



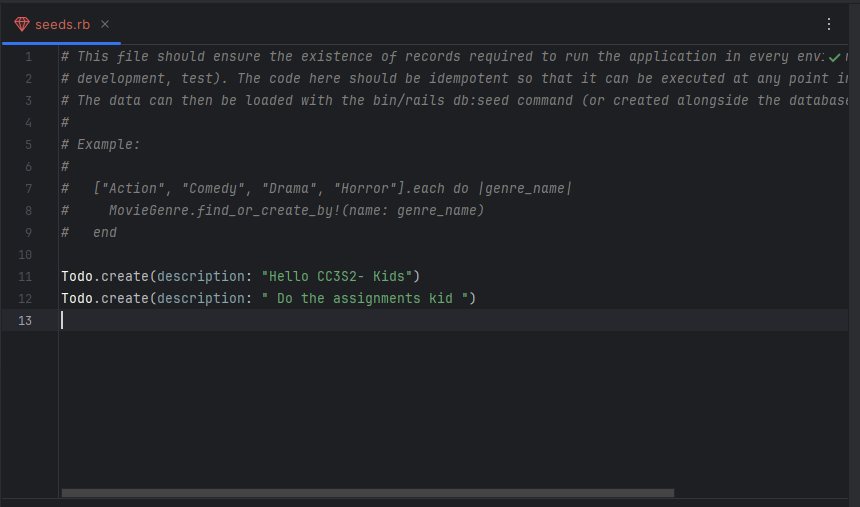
¿Que está pasando aqui?

Hay muchos términos nuevos aquí, ¡no dudes en hacer una pausa y leer más documentación en línea antes de continuar!). En este caso, estamos ejecutando la tarea de migración en la base de datos. Debería generar un resultado similar al siguiente:

De acuerdo. Ahora que tenemos la base de datos, ¡insertemos algunos registros! Abre db/seeds.rb en un editor de texto de tu elección, luego pega las dos líneas siguientes en la parte inferior del archivo, debajo de los comentarios:

Todo.create(description: "Hello CC3S2- Kids")

Todo.create(description: " Do the assignments kid ")



Luego inserta los datos en la base de datos. Comprueba si los registros se insertaron correctamente en la base de datos.

Puede ejecutar el servidor de rails para iniciar el servidor y ejecutar tu aplicación. Realiza eso y muestra evidencias.



Muestra todas las rutas generadas por el comando scaffold. Recomendamos realizar solicitudes a estas rutas para que puedas ver cómo responde tu aplicación, tanto en términos de comportamiento visual como de qué código se llama o se modifica.

**En este punto, siéntete libre de jugar un poco con la aplicación. Cambia partes del código base y ver cómo cambian las cosas. Muestra evidencias de esto como verificación de lo aprendido en clase.**

**Más migraciones**

La primera migración nos permitió crear la base de datos. Digamos que el cliente quiere que cada "tareas pendientes" tenga una fecha de vencimiento asociada. ¿Cómo hacemos que eso suceda?

¿Qué archivos nuevos o cambios en los archivos existentes han ocurrido? .

**Nuevas rutas**

En algún momento, es posible que desees agregar un nuevo servicio a tu aplicación existente. Para poder acceder a él, deberás crear una nueva ruta. Practiquemos un poco.

1. Agrega una ruta en config/routes.rb y asígnela a una acción del controlador. (Coloca esta línea dentro del cuerpo de route.draw).

get ’/hello’, to: ’todos#hello’

2. Añade una nueva vista correspondiente a la ruta. Si alguien navega a la página "hello" de tu aplicación (es decir, http://localhost:<número de puerto>/hello/), se mostrará una página web.

* Crea un archivo en el directorio app/views/todos llamado hello.html.erb.
* Puedes ingresar cualquier HTML que desees para mostrar tu página web, pero por ahora solo puedes escribir <h1>¡Hola!</h1>.
* Agrega el siguiente método a app/controllers/todos\_controller.rb.

def hello

respond\_to do |format|

format.html { render :hello }

format.json { render json: "hello world!" }

end

end

**Más ejercicios**

* Agrega un nuevo atributo al modelo Todo y actualiza las vistas para mostrar y editar el nuevo campo. Para propósitos de práctica digamos que queremos agregar un nuevo campo booleano llamado "done" con un valor predeterminado de falso.
* Cambia el esquema de enrutamiento. Debes suponer que queremos una nueva ruta new\_todo para ir a una página que crea un nuevo ítem Todo
* Establece todos#index como la página de inicio de la aplicación.
* Enumera todas las migraciones. ¿Cuál es el comando para volver a una versión anterior de la base de datos?. Muestra los resultados.

**Parte 3: Ruby on Rails (8 puntos)**

**Observación: en la sección de preguntas, debes completar todas las preguntas para que se puntue. No completar las preguntas hace que el puntaje sea 0. De igual modo con las preguntas que solo tienen respuestas puntual.**

En una tarea anterior creaste una aplicación web sencilla que juega Wordguesser. Reutilizarás el mismo código del juego pero lo “envolverás” en una aplicación Rails simple en lugar de una en Sinatra. El objetivo del ejercicios es:

Comprender las diferencias entre cómo Rails y Sinatra manejan diversos aspectos de la construcción de SaaS, que incluyen:

* Cómo se definen y asignan rutas a las acciones
* la estructura de directorios utilizada por cada framework
* Cómo se inicia y detiene una aplicación
* Cómo se puede inspeccionar el comportamiento de la aplicación mirando los registros o invocando un depurador.

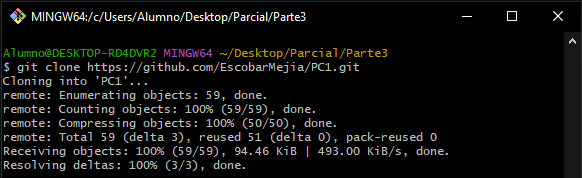
**Correr la aplicación**

NOTA: Puede resultarle útil tener a mano estas guías <https://guides.rubyonrails.org/v4.2/> y la documentación de referencia de Rails <https://api.rubyonrails.org/v4.2.11/> .

No clones el repositorio de la evaluación pasada sino usa el que has trabajado en la primera evaluación del curso. Muestra evidencia de ello.

Clona el repositorio

git clone <https://github.com/saasbook/hw-rails-wordguesser.git>



Inicia el servidor.

Preguntas (1 punto):

* ¿Cuál es el objetivo de ejecutar bundle install?

Su objetivo es instalar todas las gemas del archivo gemfile en el proyecto.

* ¿Por qué es una buena práctica especificar –without production al ejecutarlo en su computadora de desarrollo?

Permite instalar solo las gemas necesarias en el desarrollo y evitar las versiones específicas de alguna gema que pueda generar conflicto con la versión, entonces podemos decir que optimiza el uso de las gemas.

* (Para la mayoría de las aplicaciones Rails, también tendrías que crear y inicializar la base de datos de desarrollo, pero al igual que la aplicación Sinatra, esta aplicación no utiliza ninguna base de datos).

Juega para convencerte de que funciona igual que la versión de Sinatra.

Ambas aplicaciones tienen una estructura similar: el usuario activa una acción en un juego mediante una solicitud HTTP; se llama a un fragmento particular de código para "manejar" la solicitud según corresponda. Se llama a la lógica de la clase WordGuesserGame para manejar la acción y normalmente, se representa una vista para mostrar el resultado. Pero las ubicaciones del código correspondiente a cada una de estas tareas son ligeramente diferentes entre Sinatra y Rails.

Preguntas (3 puntos):

* ¿En qué parte de la estructura del directorio de la aplicación Rails está el código correspondiente al modelo WordGuesserGame?
* ¿En qué archivo está el código que más se corresponde con la lógica del archivo app.rb de las aplicaciones Sinatra que maneja las acciones entrantes del usuario?
* ¿Qué clase contiene ese código?
* ¿De qué otra clase (que es parte del framework Rails) hereda esa clase?
* ¿En qué directorio está el código correspondiente a las vistas de la aplicación Sinatra (new.erb, show.erb, etc.)?
* Los sufijos de nombre de archivo para estas vistas son diferentes en Rails que en la aplicación Sinatra. ¿Qué información proporciona el sufijo situado más a la derecha del nombre del archivo (por ejemplo: en foobar.abc.xyz, el sufijo .xyz) sobre el contenido del archivo?
* ¿Qué información te brinda el otro sufijo sobre lo que se le pide a Rails que haga con el archivo?
* ¿En qué archivo está la información de la aplicación Rails que asigna rutas (por ejemplo, GET/new) a las acciones del controlador?
* ¿Cuál es el papel de la opción :as => 'name' en las declaraciones de ruta de config/routes.rb? .

Ambas aplicaciones garantizan que el juego actual se cargue desde la sesión antes de que se produzca cualquier acción del controlador y que el juego actual (posiblemente modificado) se reemplace en la sesión después de que se complete cada acción.

Preguntas (1 punto):

* En la versión de Sinatra, los bloques before do...end y after do...end se utilizan para la gestión de sesiones. ¿Cuál es el equivalente más cercano en esta aplicación Rails y en qué archivo encontramos el código que lo hace?
* Un formato de serialización popular para intercambiar datos entre aplicaciones web es JSON. ¿Por qué no funcionaría utilizar JSON en lugar de YAML? (Reemplaza YAML.load() con JSON.parse() y .to\_yaml con .to\_json para realizar esta prueba. Tendrás que borrar las cookies asociadas con localhost:3000 o reiniciar tu navegador con un nuevo Incognito/ Ventana de navegación privada, para borrar la sesión[]. Según los mensajes de error que recibe al intentar utilizar la serialización JSON, debería poder explicar por qué la serialización YAML funciona en este caso pero JSON no).

Preguntas (2 puntos):

* En la versión de Sinatra, cada acción del controlador termina con redirect (que, como puedes ver, se convierte en redirección\_to en Rails) para redirigir al jugador a otra acción, o con erb para representar una vista. ¿Por qué no hay llamadas explícitas correspondientes a erb en la versión Rails?
* En la versión de Sinatra, codificamos directamente un formulario HTML usando la etiqueta <form>, mientras que en la versión de Rails usamos un método Rails form\_tag, aunque sería perfectamente legal usar etiquetas HTML <form> sin formato en Rails. ¿Se te ocurre alguna razón por la que Rails podría introducir este "nivel de direccionamiento indirecto"?
* ¿Cómo se manejan los elementos del formulario, como campos de texto y botones, en Rails? (Nuevamente, el HTML sin formato sería legal, pero ¿cuál es la motivación detrás de la forma en que Rails lo hace?)
* En la versión de Sinatra, las vistas de show, win y lose reutilizan el código en la vista new que ofrece un botón para iniciar un nuevo juego. ¿Qué mecanismo de Rails permite reutilizar esas vistas en la versión de Rails?.

Los escenarios de Cucumber y las definiciones de pasos (todo lo que se encuentra en features/, incluido el soporte para Webmock en webmock.rb) se copiaron palabra por palabra de la versión de Sinatra, con una excepción: el archivo features/support/env.rb es más simple porque la gema cucumber-rails hace automáticamente algunas de las cosas que teníamos que hacer explícitamente en ese archivo para la versión de Sinatra.

Verifique la ejecución de los escenarios Cucumber y páselos ejecutando rake cucumber.

* ¿Cuál es una explicación cualitativa de por qué no fue necesario modificar los escenarios de Cucumber y las definiciones de pasos para que funcionaran igualmente bien con las versiones de la aplicación Sinatra o Rails? (1 punto)