Trabajo Practico N° 2

Paradigmas y lenguajes de programación II

Profesor: Ing. Carlitos

Integrantes: ¿?¿?¿?

Facultad: de Ingenieria y ¿????

Año: Nuevo Judio

Contenido

[Clase Cliente 4](#_Toc178960005)

[Clase Proyecto 4](#_Toc178960006)

[Clase Tarea 5](#_Toc178960007)

[Clase Empleado (abstracta) 6](#_Toc178960008)

[Subclase Desarrollador extend Empleado 6](#_Toc178960009)

[Subclase Tester extend Empleado 7](#_Toc178960010)

[Clase EstrategiaAsignacionTareas (interfaz) 7](#_Toc178960011)

[Clase AsignacionAleatoria (implementación de interfaz) 8](#_Toc178960012)

[Clase GestorDeProyectos (Singleton) 8](#_Toc178960013)

[Clase FabricaDeEmpleados (Factory Method) 9](#_Toc178960014)

[Clase AdaptadorDeBaseDeDatos (Adapter) 9](#_Toc178960015)

[¿Por qué el uso de estos patrones de diseño? 10](#_Toc178960016)

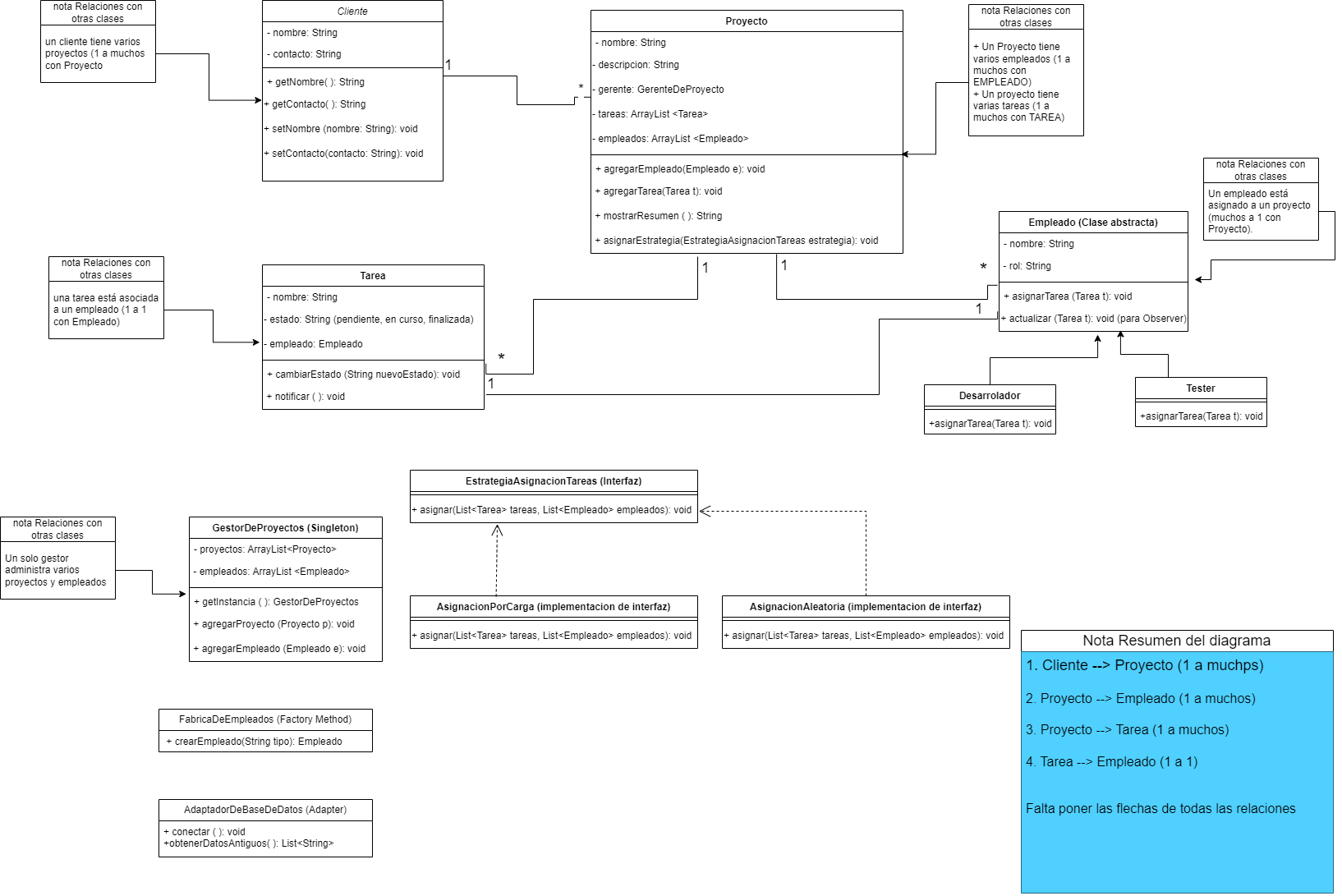
[Singleton (GestorDeProyectos) 10](#_Toc178960017)

[Factory Method (FabricaDeEmpleados) 10](#_Toc178960018)

[Adapter (AdaptadorDeBaseDeDatos) 10](#_Toc178960019)

[Observer (Empleado y Tarea) 11](#_Toc178960020)

[Strategy (Asignación de Tareas en Proyecto) 11](#_Toc178960021)

[ 12](#_Toc178960022)

## Clase Cliente

Texto

Descripción generada automáticamente

**Descripción:** Representa a los clientes de la empresa que solicitan proyectos. Cada cliente Tiene un nombre y un contacto que lo identifica.

* **Relaciones:**
  + **Cliente** está relacionado con **Proyecto** por una relación de “uno a muchos”. Un cliente puede tener varios proyectos, pero un proyecto pertenece a un solo cliente.

## Clase Proyecto

Texto

Descripción generada automáticamente

**Descripción:** Un proyecto representa un conjunto de tareas a realizar para un cliente. Cada proyecto tiene un nombre, una descripción, un gerente de proyecto, empleados asignados y tareas.

* **Relaciones:**
  + **Proyecto** está relacionado con **Cliente**, donde un cliente puede tener varios proyectos (1 a muchos).
  + Está relacionado con **Empleado,** donde un proyecto tiene muchos empleados asignados (1 a muchos).
  + Está relacionad con **Tarea**, donde un proyecto tiene muchas tareas (1 a muchos).
  + Implementa el patrón **Strategy** para la asignación de tareas a empleados, mediante la interfaz **EstrategiaAsignacionTareas.**
  + **Patrón Strategy:** https://refactoring.guru/es/design-patterns/strategy

## Clase Tarea

Interfaz de usuario gráfica, Texto

Descripción generada automáticamente

**Descripción:** Representa una tarea que debe completarse dentro de un proyecto. Tiene un nombre, un estado (pendiente, en curso, finalizada) y está asignada a un empleado.

* **Relaciones:**
  + **Tarea** está relacionada con **Proyecto** en una relación (1 a muchos), donde un proyecto puede tener varias tareas.
  + Está relacionada con **Empleado** en una relación (uno a uno), donde cada tarea está asignada a un único empleado.
  + Utiliza el patrón **Observer** para notificar a los empleados cuando el estado de la tarea cambia
  + **Patrón Observer:** https://refactoring.guru/es/design-patterns/observer

## Clase Empleado (abstracta)

Texto, Aplicación

Descripción generada automáticamente con confianza media

**Descripción:** Clase base para todos los tipos de empleados. Incluye atributos como nombre y rol, y define métodos abstractos para asignar tareas y actualizar el estado.

* **Relaciones:**
  + **Empleado** está relacionado con **Proyecto**, donde un proyecto puede tener varios empleados (1 a muchos)
  + Tiene 2 subclases como **Desarrollador** y **Tester** que heredan sus métodos.
  + Un empleado puede recibir notificaciones a través del patrón **Observer** cuando una tarea cambia de estado.
  + **Patrón Observer:** https://refactoring.guru/es/design-patterns/observer

### Subclase Desarrollador extend Empleado

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

**Descripción:** Subclase de **Empleado** que representa el rol específico dentro de un proyecto. Implementa los métodos definidos en la clase abstracta **Empleado** para asignar tareas y actualizar su estado.

* **Relaciones:**
  + Hereda de **Empleado.**
  + Está asociado a **Tarea**, donde un empleado es asignado a una tarea específica.

### Subclase Tester extend Empleado

Interfaz de usuario gráfica

Descripción generada automáticamente

**Descripción:** Subclase de **Empleado** que representa el rol específico dentro de un proyecto. Implementa los métodos definidos en la clase abstracta **Empleado** para asignar tareas y actualizar su estado.

* **Relaciones:**
  + Hereda de **Empleado.**
  + Está asociado a **Tarea**, donde un empleado es asignado a una tarea específica.

## Clase EstrategiaAsignacionTareas (interfaz)

Interfaz de usuario gráfica

Descripción generada automáticamente con confianza media

**Descripción:** Define la estructura para las estrategias de asignación de tareas a empleados. Implementa el patrón **Strategy**, perimitiendo usar diferentes algoritmos de asignación.

* **Relaciones:**
  + **Proyecto** usa esta interfaz para implementar diferentes formas de asignar tareas, dependiendo de la estrategia utilizada.
  + **Patrón Strategy:** <https://refactoring.guru/es/design-patterns/strategy>
  + Clase AsignacionPorCarga (implementación de intefaz)

Interfaz de usuario gráfica

Descripción generada automáticamente con confianza baja

**Descripción:** Implementa la interfaz **EstrategiaAsignacionTareas** para definir diferentes formas de asignar tareas. **AsignacionPorCarga** asigna tareas según la carga de trabajo de los empleados

* **Relaciones:**
  + Implementa la interfaz **EstrategiaAsignaccionTareas.**
  + Utilizada por **Proyecto** para la asignación de tareas.

### Clase AsignacionAleatoria (implementación de interfaz)

Interfaz de usuario gráfica

Descripción generada automáticamente con confianza media

**Descripción:** Implementa la interfaz **EstrategiaAsignacionTareas** para definir diferentes formas de asignar tareas. **AsignacionAleatoria** asigna tareas de manera aleatoria.

**Relaciones:**

* + Implementa la interfaz **EstrategiaAsignaccionTareas.**
  + Utilizada por **Proyecto** para la asignación de tareas.

## Clase GestorDeProyectos (Singleton)

Texto

Descripción generada automáticamente

**Descripción:** Esta clase gestiona la lista de proyectos y empleados dentro del sistema. Implementa el patrón **Singleton**, lo que garantiza que solo haya una instancia de esta clase en toda la aplicación.

* **Relaciones:**
  + Administra la creación y el acceso a los objetos **Proyecto** y **Empleado.**
  + Está relacionada con **Proyecto** y **Empleado**, ya que gestiona ambos conjuntos de objetos.
  + **Patrón Singleton:** https://refactoring.guru/es/design-patterns/singleton

## Clase FabricaDeEmpleados (Factory Method)

Interfaz de usuario gráfica

Descripción generada automáticamente

**Descripción:** Clase que implementa el patrón **Factory Method** para crear diferentes tipos de empleados (como **Desarrolador** o **Tester**) según el rol que se solicite.

* **Relaciones:**
  + Está asociada a **Empleado** y sus subclases, ya que crea instancias de estos objetos.
  + **Factory Method:** https://refactoring.guru/es/design-patterns/factory-method

## Clase AdaptadorDeBaseDeDatos (Adapter)

Interfaz de usuario gráfica, Aplicación

Descripción generada automáticamente

**Descripción**: Implementa el patrón **Adapter** para permitir la integración con un sistema de base de datos antigua, que almacena la información en un formato diferente. Este adaptador convierte el formato antiguo al formato actual.

* **Relaciones:**
  + **AdaptadorDeBaseDeDatos** interactúa con el sistema antiguo para obtener y convertir datos, pero no tiene relaciones directas con las otras clases del sistema actual.
  + **Patrón Adapter**: <https://refactoring.guru/es/design-patterns/adapter>

# ¿Por qué el uso de estos patrones de diseño?

## Singleton (GestorDeProyectos)

* **Razón de uso:** Este patrón garantiza que solo haya una única instancia de la clase **GestorDeProyectos** en toda la aplicación. Dado que el gestor maneja la lista centralizada de proyectos y empleados, es importante que exista una única fuente de verdad para evitar inconsistencias.
* **Ventajas:**
  + Control centralizado de proyectos y empleados.
  + Asegura que todas las partes del sistema accedan a la misma instancia, evitando la duplicación de datos.
  + Facilita la gestión de los recursos del sistema, como los proyectos, asegurando que todos los módulos trabajen sobre la misma lista de proyectos y empleados.

## Factory Method (FabricaDeEmpleados)

* **Razón de uso:** Este patrón permite la creación de diferentes tipos de empleados (Como Desarrollador o Tester en este caso) sin necesidad de especificar la clase exacta al momento de la instanciación. Esto promueve el desacoplamiento entre la lógica del sistema y la creación de los objetos.
* **Ventajas:**
  + Flexibilidad para agregar nuevos tipos de empleados sin modificar el código existente.
  + Centraliza la lógica de creación, simplificando la expansión del sistema en el futuro.
  + Facilita la personalización o adaptación de empleados según nuevas necesidades o roles sin afectar el código que usa estos objetos.

## Adapter (AdaptadorDeBaseDeDatos)

* **Razón para usarlo**: Este patrón se usa para integrar el sistema de gestión con una base de datos antigua que almacena la información en un formato diferente al utilizado en el nuevo sistema. Permite que el sistema nuevo siga trabajando con la base de datos antigua sin cambiar su lógica interna.
* **Ventajas**:
  + Permite la compatibilidad entre el sistema nuevo y la base de datos antigua sin necesidad de modificar el código del sistema actual.
  + Facilita la migración progresiva hacia un nuevo sistema o base de datos, ya que no es necesario realizar una conversión inmediata de todos los datos.
  + Reduce el impacto en los módulos del sistema que consumen los datos, ya que el adaptador convierte los formatos antiguos al nuevo sin afectar la lógica de negocio.

## Observer (Empleado y Tarea)

* **Razón para usarlo**: Este patrón nos permite que los **Empleados** sean notificados automáticamente cuando cambia el estado de una **Tarea** (por ejemplo, de "en curso" a "finalizada"). Esto implementa un sistema de actualización en tiempo real entre tareas y empleados, sin necesidad de consultas manuales o de que el empleado deba estar revisando el estado de la tarea continuamente.
* **Ventajas**:
  + Permite una mayor reactividad en el sistema: los empleados pueden ser notificados de cambios en las tareas sin intervención manual.
  + Facilita la comunicación entre tareas y empleados de manera desacoplada. Los empleados solo necesitan saber que están "observando" una tarea, sin preocuparse por cómo se implementan los cambios en la tarea.
  + Escalable: si en el futuro se requieren nuevas notificaciones o cambios de estado, el patrón **Observer** puede extenderse fácilmente para incluir más eventos o notificar a otros actores.

## Strategy (Asignación de Tareas en Proyecto)

* **Razón para usarlo**: Este patrón se utiliza para implementar diferentes algoritmos o estrategias de asignación de tareas a empleados. Esto permite que el sistema sea flexible y pueda cambiar fácilmente la lógica de asignación sin modificar la clase **Proyecto**.
  + Por ejemplo, se puede asignar tareas basándose en la carga de trabajo de los empleados (**AsignacionPorCarga**) o de forma aleatoria (**AsignacionAleatoria**), dependiendo de las necesidades del proyecto.
* **Ventajas**:
  + Proporciona flexibilidad para cambiar o añadir nuevas estrategias de asignación de tareas sin modificar el código existente.
  + Favorece el principio de **abierto/cerrado**: se pueden agregar nuevas estrategias sin alterar las clases existentes.
  + Separa la lógica de asignación de tareas de la clase **Proyecto**, lo que reduce la complejidad y facilita el mantenimiento.

# 