# TRABAJO PRÁCTICO N° 2

Universidad de la Cuenca del Plata

**CARRERA**

Ingeniería de Sistemas de Información

**CÁTEDRA**

Ingeniería de Software II

**Comisión**

“A”

**Profesor**

Kutz Gabriel René

**Estudiantes**

Monzón, Sebastián | Verón, Octavio

## Introducción

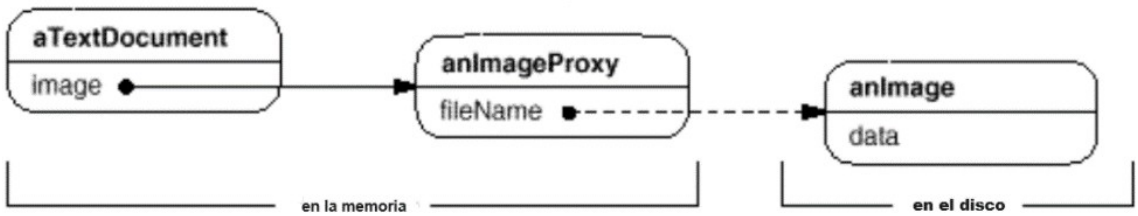
El presente informe tiene como objetivo proporcionar un análisis detallado de dos partes fundamentales en el desarrollo de software. La primera parte se centra en el análisis detallado de dos patrones diseño, el patrón de diseño estructural “Proxy” y el creacional “Prototipo”. Estos patrones representan enfoques fundamentales en el desarrollo de software para mejorar la gestión, la eficiencia operativa y la seguridad de los sistemas. Mientras que más adelante abordaremos la especificación de confiabilidad y seguridad en el desarrollo de software, la cual desempeña un papel crucial para garantizar la integridad y el funcionamiento adecuado de los sistemas.

## Desarrollo

- **Patrón de diseño estructural “Proxy”**

  Este es un patrón de diseño estructural donde se utiliza un *proxy* que actúa como un *placeholder* de otro objeto. Este proxy se puede utilizar de múltiples formas. Puede actuar como un representante de un objeto en una dirección de espacio remota, también puede usarse para proteger el acceso a objetos sensibles. Los proxies en si proporcionan un acceso indirecto a ciertas propiedades de los objetos, por lo que pueden alterar, restringir o alterar de cualquier forma estas propiedades.

  La razón principal por la que se podría controlar el acceso a un objeto es que es útil para aplazar su creación e inicialización hasta que realmente necesitemos usarlo. Como ejemplo podemos analizar el caso de un editor de texto que puede insertar objetos gráficos en un documento, debido a que algunos de estos pueden llegar a ser costosos para el sistema de crear pueden omitirse estos objetos cuando no estén siendo mostrados, y en cambio podemos colocar un reemplazo que sea más ligero y que ocupe su lugar mientras que no sea visible, y que desaparecerá cuando se supone que el objeto original sea visible y ya esté creado e inicializado, ese objeto de reemplazo es lo que llamamos “proxy”.




  Siguiendo el mismo ejemplo, el proxy solo crea la imagen verdadera cuando el editor de texto le ordena que lo haga a través de una operación llamada “Dibujar”. El proxy de imagen reenvía solicitudes posteriores a la creación directamente a la imagen. Por lo que debe mantener una referencia a la imagen después de crearla.

## Tipos de proxys

El patrón proxy es especialmente útil cuando se necesita una referencia a un objeto que sea más versátil o sofisticado que un puntero. Algunos tipos de proxys comúnmente utilizados son:

- Proxy remoto: proporciona un representante local para un objeto que encuentra en otro almacenamiento. Esto significa que puede esconder el hecho de que el objeto real se encuentra en otra dirección.
- Proxy virtual: crea objetos pesados cuando se le ordena hacerlo, como el que se comento en el ejemplo del editor de texto.
- Proxy de protección: controla el acceso al objeto original, estos son especialmente útiles cuando se necesita que los objetos reales tengan un diferente nivel de acceso.

## Implementación en código

```
#include <stdio.h>
#include <stdbool.h>

// Interfaz para el recurso y el proxy
struct Resource {
    void (*access)(bool isAdmin);
};

// Implementación del recurso real
struct RealResource {
    void (*access)(bool isAdmin);
};

void realAccess(bool isAdmin) {
    if (isAdmin) {
        printf("Acceso al archivo de configuración permitido.\
n");
```

```
        // Simulamos la lectura del archivo de configuración
        printf("Leyendo archivo de configuración...\n");
    } else {
        printf("Permiso denegado para acceder al archivo de configuración.\n");
    }
}

// Implementación del proxy para el recurso
struct ResourceProxy {
    struct Resource resource;
};

void proxyAccess(bool isAdmin) {
    // Verificar permisos antes de permitir el acceso
    if (isAdmin) {
        // Si el usuario es administrador, permite el acceso
        realAccess(isAdmin);
    } else {
        printf("Permiso denegado para acceder al archivo de configuración.\n");
    }
}

int main() {
    // Crear un proxy para el recurso
    struct ResourceProxy resourceProxy;
    resourceProxy.resource.access = &proxyAccess;

    // Intentar acceder al archivo de configuración
```

```
    resourceProxy.resource.access(false); // Intento de acceso
por un usuario no administrador

    resourceProxy.resource.access(true);  // Intento de acceso
por un usuario administrador

    return 0;

}
```

En este código tenemos una interfaz llamada “Resource” que define una función “access”. Por otro lado, tenemos la implementación real “RealResource” con una función “realAccess” que simula el acceso al recurso real. Por último tenemos el *proxy* “ResourceProxy” con una función “proxyAccess” que verifica los permisos antes de permitir el acceso real. Durante la ejecución se creará una instancia del proxy y se realizan dos intentos de acceso, uno como usuario común y otro como administrador, siendo el resultado el siguiente:

```
Permiso denegado para acceder al archivo de configuración.

Acceso al archivo de configuración permitido.

Leyendo archivo de configuración...
```

- **Patrón de diseño creacional “Prototipo”**

  Este patrón de diseño se basa especificar los tipos de objetos que se van a generar usando instancias prototipadas, generando luego instancias de estos objetos copiando este prototipo. Se trata de un patrón de diseño donde se crean nuevos objetos mediante la clonación de un objeto ya existente, que es el prototipo. Este patrón suele ser utilizado cuando el costo de crear un nuevo objeto es excesivamente alto o complejo, siendo entonces la creación de copias del objeto una alternativa más eficiente. El uso de prototipos también te permite agregar y eliminar productos de forma dinámica durante la ejecución de un programa, ya que es posible introducir una nueva clase de un producto simplemente registrando una instancia prototípica con el cliente, por lo que no es necesario modificar el código existente o recompilar el programa para agregar nuevos tipos de productos.

  Para entenderlo mejor podríamos analizar el caso de un editor de partituras construido a partir de un *framework* hecho para editores gráficos al cual le agregariamos nuevos objetos que representarían las diferentes notas. Asumiendo que el *framework* nos provee una clase genérica para componentes gráficos podríamos usar esa clase para crear los objetos que necesitamos, pero está clase

no nos permite instanciar las clases musicales que necesitamos, como solución podríamos crear subclases para cada nota, pero haciéndolo terminaríamos con múltiples subclases que solo difieren en que tipo de nota se trata. Una solución mas eficiente sería clonar una instancia de una subclase, siendo esta instancia un prototipo. Por lo que entonces en nuestra clase cada nota musical es una instancia de esa clase de parte del *framework* pero con un prototipo diferente.

Este patrón de diseño puede ser especialmente útil en los siguientes casos:

- Cuando la creación de un objeto implica una serie de pasos complejos o costosos.
- Cuando se necesite la personalización de objetos de forma dinámica. Se puede tener un prototipo base que represente el estado inicial del objeto y los usuarios pueden clonarlo y modificarlo según lo necesiten sin afectar el prototipo original.
- Cuando se diseñe una aplicación donde la creación de objetos implique el uso de una gran cantidad de recursos, como puede ser la memoria RAM o el tiempo del CPU, ya que este patrón puede ayudar a reducir la sobrecarga debido a que clonar un objeto existente es más eficiente que crearlo desde cero.

## **Implementación en código**

```
#include <stdio.h>

#include <stdlib.h>


// Definición de la estructura Coordenada

typedef struct {

    int x;

    int y;

} Coordenada;


// Función para clonar una instancia de Coordenada

Coordenada* clonCoord(Coordenada* original) {

    Coordenada* clon =
(Coordenada*)malloc(sizeof(Coordenada));

    clon->x = original->x;
```

```
    clon->y = original->y;
    return clon;
}

int main() {
    // Creamos un objeto original
    Coordenada* originalCoord = (Coordenada*)malloc(sizeof(Coordenada));
    originalCoord->x = 10;
    originalCoord->y = 20;

    // Clonamos el objeto original
    Coordenada* clonCoordenada = clonCoord(originalCoord);

    // Imprimimos las coordenadas del objeto original y su clon
    printf("Objeto original: (%d, %d)\n", originalCoord->x, originalCoord->y);
    printf("Clon del objeto: (%d, %d)\n", clonCoordenada->x, clonCoordenada->y);

    return 0;
}
```

En este código de ejemplo podemos ver que definimos una estructura llamada “Coordenada” que representa la localización de un objeto según sus coordenadas “x” e “y”. También tenemos una función “clonCoord” que toma un objeto tipo “Coordenada” original y crea un clon de él copiando sus coordenadas. Durante la ejecución creamos un objeto original, lo clonamos y se muestran en pantalla las coordenadas originales y del clon. Siendo el resultado el siguiente:

```
Objeto original: (10, 20)
Clon del objeto: (10, 20)
```

En la próxima sección, nos adentraremos en la especificación de requisitos en el desarrollo de software.

## 1. Especificación de Requerimientos Dirigida por Riesgos (12.1):

La especificación de requerimientos dirigida por riesgos representa un enfoque proactivo y sistemático para identificar, evaluar y mitigar los riesgos potenciales que pueden comprometer la seguridad y confiabilidad de un sistema de software. Esta metodología implica una serie de etapas cruciales que merecen una atención detallada:

**Identificación del Riesgo:**
La primera fase consiste en la identificación exhaustiva de posibles amenazas y vulnerabilidades que podrían afectar el funcionamiento del sistema. Esto puede abarcar desde riesgos asociados al entorno operativo hasta fallos en el diseño arquitectónico del software.

**Análisis y Clasificación del Riesgo:**
Una vez que se han identificado los riesgos potenciales, es fundamental llevar a cabo un análisis riguroso para determinar su probabilidad de ocurrencia y el impacto que podrían tener en el sistema. Esta evaluación permite priorizar los riesgos según su gravedad y urgencia de acción.

**Descomposición del Riesgo:**
En esta etapa, se profundiza en las causas subyacentes de cada riesgo identificado, analizando las posibles fallas de diseño, errores de implementación o vulnerabilidades en el código fuente del software.

**Reducción del Riesgo:**
Para mitigar los riesgos identificados, se implementan estrategias preventivas y correctivas. Esto puede incluir desde mejoras en los controles de seguridad hasta la introducción de redundancias en el sistema para minimizar la posibilidad de fallos.

**Gestión Continua del Riesgo**:
La gestión del riesgo es un proceso dinámico y continuo que implica monitorear y revisar regularmente el estado de los riesgos identificados. Esto garantiza que las medidas de mitigación sigan siendo efectivas en el tiempo y permite abordar nuevas amenazas a medida que surjan.

**Ejemplo:**
Para ilustrar la aplicación práctica de la especificación de requerimientos dirigida por riesgos, consideremos el desarrollo de un sistema de gestión de datos para una institución financiera. En esta situación, los riesgos

potenciales podrían incluir la exposición de datos sensibles debido a vulnerabilidades en la infraestructura de red o la falta de controles de acceso adecuados. Al adoptar un enfoque de especificación dirigido por riesgos, el equipo de desarrollo puede identificar estas amenazas de manera proactiva y tomar medidas para mitigar su impacto, como implementar un cifrado robusto de datos y establecer políticas de acceso basadas en roles.

## 2. Especificación de Protección (12.2):

La especificación de protección se centra en definir requisitos para salvaguardar los activos del sistema contra posibles amenazas y ataques externos. Este enfoque integral requiere un análisis detallado de las vulnerabilidades potenciales y la implementación de medidas de seguridad adecuadas para prevenir o mitigar su explotación. Las etapas clave de este proceso incluyen:

**Identificación de Amenazas:**
Se identifican las posibles amenazas que podrían comprometer la seguridad del sistema, tanto desde el interior como desde el exterior. Esto puede abarcar desde ataques cibernéticos hasta amenazas físicas como el robo de equipos.

**Análisis y Clasificación de Amenazas:**
Cada amenaza se evalúa en términos de su impacto potencial y su probabilidad de ocurrencia. Esta clasificación permite priorizar las amenazas más críticas y asignar recursos para su mitigación de manera eficiente.

**Descomposición de Amenazas:**
Se investigan las causas subyacentes de cada amenaza identificada, buscando entender los puntos débiles del sistema que podrían ser explotados por actores malintencionados.

**Reducción de Amenazas:**
Para mitigar las amenazas identificadas, se implementan medidas de seguridad adicionales, como firewalls, sistemas de detección de intrusiones y controles de acceso estrictos.

**Auditoría y Monitorización:**
La auditoría regular y la monitorización continua del sistema son fundamentales para detectar y responder rápidamente a posibles amenazas. Esto implica revisar los registros de actividad del sistema, realizar análisis de seguridad y llevar a cabo pruebas de penetración periódicas para evaluar la resistencia del sistema a ataques externos.

**Ejemplo:**

Consideremos el caso de una empresa de comercio electrónico que maneja grandes volúmenes de datos de clientes, incluidos detalles de tarjetas de crédito y otra información personal sensible. Para proteger estos datos contra posibles ataques, la empresa podría implementar medidas de seguridad como el cifrado de datos en tránsito y en reposo, la autenticación de dos factores para los usuarios y una estricta segregación de roles de acceso dentro de su infraestructura de red.

## 3. Especificación de Fiabilidad (12.3):

La especificación de fiabilidad se refiere a la capacidad del sistema para funcionar correctamente bajo diversas condiciones y durante un período de tiempo específico. Esto implica garantizar la disponibilidad del sistema, su capacidad para tolerar fallos y su capacidad de recuperación en caso de interrupciones inesperadas. Las siguientes etapas son fundamentales en este proceso:

**Identificación de Puntos de Fallo:**
Se identifican los posibles puntos de fallo que podrían afectar la fiabilidad del sistema, desde errores de software hasta fallos de hardware y problemas de conectividad de red.

**Análisis y Clasificación de Puntos de Fallo**:
Cada punto de fallo se evalúa en términos de su impacto potencial y su probabilidad de ocurrencia, permitiendo priorizar las áreas críticas que requieren atención inmediata.

**Descomposición de Puntos de Fallo:**
Se investigan las causas subyacentes de cada punto de fallo identificado, buscando entender las debilidades inherentes del sistema que podrían comprometer su fiabilidad.

**Reducción de Puntos de Fallo:**
Para mejorar la fiabilidad del sistema, se implementan medidas para mitigar los puntos de fallo identificados, como la introducción de redundancias de hardware y la implementación de procedimientos de recuperación de desastres.

**Pruebas de Fiabilidad:**
Las pruebas de fiabilidad son esenciales para verificar que el sistema cumpla con los requisitos de disponibilidad y rendimiento especificados. Esto puede incluir pruebas de carga, pruebas de estrés y simulaciones de fallos para evaluar la capacidad del sistema para manejar situaciones adversas.

**Ejemplo:**

En el sector de la atención médica, la fiabilidad de los sistemas de información es crítica para garantizar la entrega oportuna y precisa de servicios de atención al paciente. Un sistema de registro electrónico de salud debe ser capaz de mantenerse operativo incluso en situaciones de alta demanda, como emergencias médicas o picos de tráfico de usuarios. Al adoptar un enfoque de especificación de fiabilidad, los desarrolladores pueden identificar y abordar proactivamente los puntos de fallo potenciales, asegurando que el sistema sea robusto y confiable en todo momento.

## 4. Especificación de Seguridad (12.4):

La especificación de seguridad se enfoca en proteger los activos del sistema contra amenazas maliciosas o accidentales. Esto implica la implementación de medidas de seguridad física y lógica para garantizar la integridad y confidencialidad de la información. Las siguientes etapas son fundamentales en este proceso:

**Identificación de Vulnerabilidades:**
Se identifican las posibles vulnerabilidades en el diseño, la implementación y la configuración del sistema. Esto puede incluir errores de software, debilidades en la configuración del servidor y vulnerabilidades conocidas en el código del software.

**Análisis y Clasificación de Vulnerabilidades:**
Cada vulnerabilidad se analiza para determinar su impacto potencial en el sistema y su probabilidad de explotación. Las vulnerabilidades se clasifican según su gravedad y se priorizan para su posterior mitigación.

**Descomposición de Vulnerabilidades:**
Se investigan las causas subyacentes de cada vulnerabilidad identificada, buscando comprender las debilidades inherentes del sistema que podrían ser explotadas por actores malintencionados.

**Reducción de Vulnerabilidades:**
Para mitigar las vulnerabilidades identificadas, se implementan medidas de seguridad adicionales, como la instalación de parches de seguridad, la aplicación de controles de acceso más estrictos y la mejora de las políticas de seguridad del sistema.

**Gestión de Incidentes de Seguridad:**
La gestión de incidentes de seguridad es esencial para responder de manera efectiva a las amenazas y ataques contra el sistema. Esto incluye la implementación de planes de respuesta a incidentes, la coordinación con las autoridades competentes y la revisión post-incidente para mejorar las medidas de seguridad.

**Ejemplo:**
En el contexto de una institución financiera, la seguridad de la información es de suma importancia debido a la sensibilidad de los datos financieros y personales que manejan. Para protegerse contra amenazas como el robo de datos o el fraude financiero, la institución puede implementar medidas como sistemas de detección de intrusos, autenticación multifactor y cifrado de extremo a extremo en las comunicaciones. Estas medidas ayudan a mitigar el riesgo de ataques cibernéticos y proteger la confidencialidad e integridad de la información del cliente.

## 5. Especificación Formal (12.5):

La especificación formal implica el uso de lenguajes formales para definir los requisitos del sistema de manera precisa y no ambigua. Esto facilita el análisis y la verificación de los requisitos, ayudando a garantizar que el sistema cumpla con las expectativas del cliente y los estándares de calidad. Las siguientes etapas son fundamentales en este proceso:

**Identificación de Requisitos Críticos:**
Se identifican y especifican formalmente los requisitos críticos del sistema que deben cumplirse para garantizar su correcta implementación y verificación. Esto puede incluir requisitos de seguridad, fiabilidad, rendimiento, entre otros.

**Análisis y Especificación Formal de Requisitos:**
Se utiliza un lenguaje formal, como la lógica de primer orden o el cálculo de predicados, para especificar los requisitos del sistema de manera precisa y no ambigua. Esto facilita la comprensión y verificación de los requisitos por parte de todas las partes interesadas.

**Validación y Verificación de Requisitos:**
Los requisitos especificados formalmente se someten a un proceso de validación y verificación para garantizar su corrección y consistencia. Esto puede incluir pruebas de conformidad, análisis de modelos y demostraciones de cumplimiento.

**Gestión de Cambios y Versiones:**
La gestión de cambios y versiones es esencial para mantener la integridad y coherencia de la especificación de requisitos a lo largo del tiempo. Esto incluye la implementación de controles de versión, procedimientos de revisión y seguimiento de cambios para garantizar que la especificación se mantenga actualizada y relevante.

**Ejemplo:**

En el desarrollo de un sistema de control de tráfico aéreo, la especificación formal de requisitos desempeña un papel crucial en garantizar la seguridad y eficiencia del sistema. Mediante el uso de lenguajes formales, como la lógica temporal o la lógica de estados finitos, los requisitos críticos, como la capacidad de gestionar múltiples aeronaves simultáneamente y garantizar la separación mínima entre vuelos, pueden ser especificados de manera precisa y verificable. Esto ayuda a reducir el riesgo de errores de diseño y asegura que el sistema cumpla con los estándares de seguridad y rendimiento exigidos por la industria de la aviación.

# Bibliografía

Pressman, R. S. (2010). Ingeniería del software: Un enfoque práctico (7.a ed.). McGrwaw-Hill.

Sommerville, I. (2011). Ingeniería de software (9.a ed.). Pearson

Gamma, E., Johnson, R., Helm, R. (1994). Design Patterns: Element of Reusable Object-Oriented Software.