UNIVERSIDAD NACIONAL DEL ALTIPLANO
FACULTAD DE INGENIERÍA ESTADÍSTICA E INFORMÁTICA

Ingeniería de Software

Métricas de Software: Evaluación Integral del Código, Desempeño, Fiabilidad, Mantenibilidad y Adaptabilidad

Capítulo IV : Métricas de Mantenibilidad y Usabilidad

Friday 21st June, 2024

Contents

1 **7**

2 **9**

3 **11**

4 Métricas de Mantenibilidad y Usabilidad **13**
4.1 Índice de Casos de Uso (UCP) 13
4.1.1 Definición 13
4.1.2 Método 13
4.1.3 Calculo de las Ponderaciones de Actores No Ajustadas (UAW) 14
4.1.4 Identificación de Casos de Uso y Actores 14
4.1.5 Pesos de Casos de Uso No Ajustados (UUCW) 14
4.1.6 Calculando UUCP 15
4.1.7 Factores de Complejidad Técnica (TCF) 15
4.1.8 Factores de Complejidad Ambiental (ECF) 15
4.2 Acoplamiento en el Software 16
4.2.1 Tipos de Acoplamiento 16
4.2.2 Otras Dimensiones del Acoplamiento 17
4.2.3 Métricas de Acoplamiento 17
4.2.4 Para el Acoplamiento de Datos y Flujo de Control . . . 17
4.2.5 Para el Acoplamiento Global 18
4.2.6 Para el Acoplamiento de Entorno 18
4.2.7 Fórmula de Acoplamiento 18
4.2.8 Interpretación 19
4.3 Índice de Reutilización de Componentes (IRC) 19
4.3.1 Definición 19

4.3.2 Uso de Componentes Desarrollados para ser Reutilizados 19
4.3.3 Aspectos a Considerar en la Reutilización 20
4.3.4 Desafíos y Consideraciones 20
4.3.5 Estrategias para Implementar el Índice de Reutilización de Componentes 21
4.4 Tiempo Promedio de Localización de Defectos (MTTD) 22
4.4.1 Definición 22
4.4.2 Importancia del MTTD 22
4.4.3 Cálculo del MTTD 22
4.4.4 Fórmula para el MTTD 23
4.4.5 Ejemplo 01 23
4.4.6 Código en Python para calcular MTTD: 24
4.4.7 Segunda fórmula 25
4.4.8 Ejemplo 02 25
4.4.9 Código en Python para Calcular MTTD 26
4.5 Tasa de Actualización 26
4.5.1 Definición 26
4.5.2 Importancia 26
4.5.3 La Tasa de Actualización se utiliza para: 27
4.5.4 ¿Qué se necesita para calcular la Tasa de Actualización? 27
4.5.5 Fórmula 27
4.5.6 Ejemplo 01 28
4.5.7 Código en Python 01 28
4.5.8 Código en Python 02 29
4.5.9 Ejemplo 02 29
4.5.10 Recolección de datos 29
4.5.11 Cálculo de la Tasa de Actualización 29
4.5.12 Código en Python III 30
4.6 Satisfacción del Usuario 30
4.6.1 Definición 31
4.6.2 Aspectos Clave de Evaluación 31
4.6.3 Aplicación 31
4.6.4 Limitaciones 32
4.6.5 Ejemplo de código en Python 32
4.7 Consumo de Memoria 33
4.7.1 Definición 33
4.7.2 Dimensiones de Evaluación 34
4.7.3 Aplicación 34

4.7.4 Limitaciones 35
4.8 Facilidad de Aprendizaje 35
4.8.1 Definición 35
4.8.2 Tipos 36
4.8.3 Aplicaciones 36
4.8.4 Limitaciones 37
4.8.5 Métricas 38
4.8.6 Facilidad de Aprendizaje en Software 39
4.8.7 Definición 39
4.8.8 Factores que Influyen en la Facilidad de Aprendizaje . 39
4.8.9 Métricas para Evaluar la Facilidad de Aprendizaje . . . 40
4.8.10 Ejemplo Práctico 40
4.8.11 Utilidad 41
4.9 Vulnerabilidades Encontradas 41
4.9.1 Tipos 41
4.9.2 Aplicaciones 42
4.9.3 Limitaciones 43
4.9.4 Métricas 43
4.10 Conclusión del capitulo 44

5 **47**

Chapter 1

Chapter 2

Chapter 3

Chapter 4

Métricas de Mantenibilidad y Usabilidad

4.1 Índice de Casos de Uso (UCP)

4.1.1 Definición

Los puntos de casos de uso (UCP) se utilizan para estimar el esfuerzo y el tiempo requeridos en un proyecto de software. Este método fue desarrollado por Gustav Karner en 1993 y se basa en el método de puntos de función. UCP es útil porque proporciona una forma estructurada y objetiva de medir la complejidad de un sistema de software a partir de sus casos de uso y actores [8].

4.1.2 Método

El método UCP utiliza actores y casos de uso para estimar el tamaño del software. A cada uno se le asigna una ponderación basada en su complejidad. Los actores se clasifican en simples, medios y complejos, mientras que los casos de uso se evalúan en función del número de transaccione[8]s.

La fórmula básica para calcular los puntos de casos de uso es:

$$UCP = UUCP \times TCF \times ECF$$

donde: - $UUCP$ es la suma de las ponderaciones de actores no ajustadas (UAW) y las ponderaciones de casos de uso no ajustadas (UUCW). - TCF

es el Factor de Complejidad Técnica. - ECF es el Factor de Complejidad Ambiental.

4.1.3 Calculo de las Ponderaciones de Actores No Ajustadas (UAW)

Las ponderaciones de actores no ajustadas (UAW) se calculan sumando los pesos de cada tipo de actor multiplicado por la cantidad de actores de ese tipo:

$$UAW = \sum(\text{cantidad de un tipo de actor} \times \text{peso})$$

Los actores se clasifican en tres categorías:

- **Actor Simple**: Interactúa con el sistema a través de una API. Peso: 1.
- **Actor Promedio**: Interactúa a través de un protocolo o red. Peso: 2.
- **Actor Complejo**: Interactúa a través de una interfaz gráfica de usuario. Peso: 3.

4.1.4 Identificación de Casos de Uso y Actores

Identificar y clasificar correctamente los actores y casos de uso es fundamental para la precisión del cálculo UCP. Los actores representan entidades externas que interactúan con el sistema, como usuarios finales o sistemas externos. Los casos de uso describen cómo estos actores utilizan el sistema para lograr un objetivo específico.

4.1.5 Pesos de Casos de Uso No Ajustados (UUCW)

Los pesos de casos de uso no ajustados (UUCW) se calculan sumando los pesos de cada tipo de caso de uso multiplicado por la cantidad de casos de uso de ese tipo:

$$UUCW = \sum(\text{cantidad de un tipo de caso de uso} \times \text{peso})$$

Los casos de uso se clasifican en tres categorías basadas en el número de transacciones:

- **Caso de Uso Simple**: Hasta 3 transacciones. Peso: 5.
- **Caso de Uso Promedio**: De 4 a 7 transacciones. Peso: 10.
- **Caso de Uso Complejo**: Más de 7 transacciones. Peso: 15.

4.1.6 Calculando UUCP

La suma de UAW y UUCW nos da los puntos de casos de uso no ajustados (UUCP):

$$UUCP = UAW + UUCW$$

4.1.7 Factores de Complejidad Técnica (TCF)

Los Factores de Complejidad Técnica (TCF) ajustan los puntos de caso de uso para reflejar la complejidad técnica del proyecto. Los factores TCF consideran elementos como el rendimiento del sistema, la complejidad de la instalación, y el uso de componentes específicos [9]. Se evalúan entre 0 (no relevante) y 5 (muy relevante).

La fórmula para calcular TCF es:

$$TCF = 0.6 + (0.01 \times \sum(\text{valor de cada TCF}))$$

Los factores de complejidad técnica típicos incluyen:

- Desempeño requerido
- Eficiencia del usuario final
- Procesamiento interno complejo
- Reutilización de código
- Instalación fácil

4.1.8 Factores de Complejidad Ambiental (ECF)

Los Factores de Complejidad Ambiental (ECF) ajustan los puntos de caso de uso para reflejar la complejidad del ambiente de desarrollo. Los factores ECF consideran elementos como la experiencia del equipo, las limitaciones

de tiempo, y la familiaridad con la tecnología [9]. Se evalúan entre 0 (no relevante) y 5 (muy relevante).

La fórmula para calcular ECF es:

$$ECF = 1.4 - (0.03 \times \sum(\text{valor de cada ECF}))$$

Los factores de complejidad ambiental típicos incluyen:

- Familiaridad con el proyecto
- Experiencia en la plataforma
- Habilidades del equipo
- Motivación del equipo
- Estabilidad de los requisitos

4.2 Acoplamiento en el Software

Definición

El acoplamiento es una métrica fundamental en la calidad del software, complementaria a la cohesión [3]. Se refiere a cómo están interconectadas las entidades dentro de un sistema de software, lo cual tiene un impacto significativo en la mantenibilidad, la evolución y la comprensión del programa.

4.2.1 Tipos de Acoplamiento

- **Acoplamiento de Contenido (Alto)**: Un módulo modifica o depende de los detalles internos de otro, como acceder a sus datos locales.
- **Acoplamiento Común (Global)**: Dos o más módulos comparten datos globales como variables.
- **Acoplamiento Externo**: Dos módulos dependen de un formato de datos externo, protocolo de comunicación o interfaz de dispositivo.
- **Acoplamiento de Control**: Un módulo controla el flujo de ejecución de otro, pasando información sobre qué acciones realizar.

- **Acoplamiento Sellado (Por Estructura de Datos)**: Los módulos comparten una estructura de datos compuesta, utilizando solo partes específicas.
- **Acoplamiento de Datos**: Los módulos intercambian información directamente, como parámetros de función. Facilita la comunicación pero introduce dependencias que deben gestionarse.
- **Acoplamiento de Mensajes (Bajo)**: Los módulos se comunican a través de mensajes o eventos, promoviendo la flexibilidad y reutilización del código. Fomenta un diseño más modular e independiente.
- **Sin Acoplamiento**: Módulos que operan de manera independiente sin comunicarse entre sí. Ideal para componentes que no requieren interacción directa.

4.2.2 Otras Dimensiones del Acoplamiento

- **Acoplamiento Conceptual**: Mide la relación conceptual entre los identificadores y los comentarios de diferentes clases.
- **Acoplamiento Dinámico**: Considera las conexiones que aparecen durante la ejecución del programa.
- **Acoplamiento Lógico**: Representa las dependencias de cambio entre clases basadas en datos históricos de modificaciones. Mide el grado en que dos o más clases cambian juntas o coevolucionan.

4.2.3 Métricas de Acoplamiento

El acoplamiento en ingeniería de software describe una variedad de métricas asociadas con este concepto. A continuación, se presentan las métricas para diferentes tipos de acoplamiento:

4.2.4 Para el Acoplamiento de Datos y Flujo de Control

- di: número de parámetros de entrada
- ci: número de parámetros de control

- *do*: número de parámetros de salida
- *co*: número de parámetros de control de salida

4.2.5 Para el Acoplamiento Global

- *gd*: número de variables globales usadas como datos
- *gc*: número de variables globales usadas como control

4.2.6 Para el Acoplamiento de Entorno

- *w*: número de módulos llamados (fan-out)
- *r*: número de módulos llamantes bajo consideración (fan-in)

4.2.7 Fórmula de Acoplamiento

El acoplamiento (C) se calcula utilizando la siguiente fórmula:

$$\text{Acoplamiento}(C) = 1 - \frac{1}{di + 2 \times ci + do + 2 \times co + gd + 2 \times gc + w + r}$$

Donde:

- *di*: número de parámetros de entrada
- *ci*: número de parámetros de control
- *do*: número de parámetros de salida
- *co*: número de parámetros de control de salida
- *gd*: número de variables globales usadas como datos
- *gc*: número de variables globales usadas como control
- *w*: número de módulos llamados (fan-out)
- *r*: número de módulos llamantes bajo consideración (fan-in)

4.2.8 Interpretación

El valor de *Acoplamiento* (C) aumenta con el nivel de acoplamiento del módulo. Este valor varía aproximadamente entre 0.67 (bajo acoplamiento) y 1.0 (alto acoplamiento).

4.3 Índice de Reutilización de Componentes (IRC)

4.3.1 Definición

El paradigma de desarrollo basado en componentes se centra en la reutilización efectiva de componentes de software como un criterio fundamental para el éxito en el desarrollo de software. En la investigación actual, se emplean enfoques basados en la experiencia de expertos para evaluar la reutilizabilidad de estos componentes, destacando su importancia en la eficiencia y calidad del desarrollo de software [7].

4.3.2 Uso de Componentes Desarrollados para ser Reutilizados

- CRI = (Número de componentes reutilizados) / (Número total de componentes)
- **Propiados o Propios**: Son ideales para ser reutilizados directamente sin necesidad de adaptación adicional. Generalmente se encuentran en componentes internos de la entidad y son fácilmente reutilizables para nuevos proyectos.
- **Adaptables**: Requieren ajustes y esfuerzo adicional para cumplir completamente con los requisitos del nuevo proyecto. Esto implica estudiar sus especificaciones y asegurar la compatibilidad con los otros módulos seleccionados previamente.
- **Rechazables**: No son adecuados para el proyecto actual debido a que no pertenecen a su dominio o no pueden ser adaptados eficientemente. Estos componentes deben ser descartados en la fase inicial de selección[7] [5].

4.3.3 Aspectos a Considerar en la Reutilización

- **Medición de Eficiencia**: Permite medir cuán eficientemente se están reutilizando los componentes de software en comparación con desarrollar nuevos componentes desde cero. Un índice alto sugiere que se está maximizando la reutilización, lo cual puede traducirse en ahorros de tiempo y recursos.

- **Evaluación de Estrategias de Desarrollo**: Ayuda a los equipos de desarrollo a evaluar la efectividad de sus estrategias de reutilización de componentes. Esto incluye decisiones sobre la arquitectura del software, la gestión de bibliotecas y frameworks, y la política de desarrollo de componentes reutilizables.

- **Optimización de Recursos**: Proporciona información para optimizar el uso de recursos, como personal de desarrollo y tiempo de programación. Al fomentar la reutilización de componentes, se pueden reducir los esfuerzos redundantes y enfocarse en el desarrollo de nuevas funcionalidades o mejoras.

- **Mantenibilidad del Software**: Contribuye a mejorar la mantenibilidad del software al fomentar el uso de componentes probados y bien documentados en lugar de soluciones ad-hoc. Esto facilita la corrección de errores, actualizaciones y modificaciones en el sistema.

- **Reducción de Costos**: Al reducir la necesidad de desarrollar nuevas funcionalidades desde cero, se pueden lograr ahorros significativos en costos de desarrollo y mantenimiento a lo largo del ciclo de vida del software.

4.3.4 Desafíos y Consideraciones

- **Gestión de Versiones y Dependencias**: A medida que crece la biblioteca de componentes reutilizables, la gestión de versiones y las dependencias entre componentes pueden volverse complejas. Es crucial establecer prácticas sólidas de control de versiones y gestión de configuraciones para garantizar la compatibilidad y estabilidad del sistema.

- **Cambio Cultural y Adopción Organizacional**: La implementación efectiva del IRC requiere una cultura organizacional que valore y promueva la reutilización de componentes. Esto puede requerir un cambio cultural y la adopción de nuevas prácticas y herramientas por parte de todo el equipo de desarrollo.

- **Calidad y Mantenimiento de los Componentes**: Es fundamental asegurarse de que los componentes reutilizables mantengan altos estándares de calidad y estén debidamente mantenidos y actualizados. La falta de mantenimiento puede llevar a problemas de seguridad, compatibilidad y rendimiento en el sistema.

- **Estrategias de Documentación y Comunicación**: Documentar adecuadamente los componentes reutilizables y comunicar su disponibilidad y capacidades dentro de la organización son aspectos clave para fomentar su adopción y uso efectivo por parte de los equipos de desarrollo.

4.3.5 Estrategias para Implementar el Índice de Reutilización de Componentes

- **Identificación y Catalogación**: Establecer un repositorio centralizado de componentes reutilizables y desarrollar criterios claros para identificar qué componentes son candidatos para la reutilización.

- **Evaluación de Impacto y Riesgos**: Antes de reutilizar un componente, es importante evaluar su impacto potencial en el rendimiento, la seguridad y la funcionalidad del sistema.

- **Formación y Capacitación**: Capacitar a los desarrolladores en las mejores prácticas de reutilización de componentes y en el uso adecuado de herramientas y tecnologías asociadas.

- **Monitoreo y Evaluación Continua**: Implementar mecanismos para monitorear y evaluar el rendimiento y la efectividad del IRC en el tiempo. Esto incluye la revisión periódica de métricas clave y la retroalimentación de los equipos de desarrollo [5].

4.4 Tiempo Promedio de Localización de Defectos (MTTD)

4.4.1 Definición

El Tiempo Promedio de Localización de Defectos (MTTD) se define como el promedio del tiempo transcurrido entre la introducción de un defecto y su detección. Esta métrica es crucial para evaluar la eficiencia en la detección de defectos y mejorar la calidad del software [10].

4.4.2 Importancia del MTTD

- **Calidad del software**: Un MTTD bajo sugiere un proceso ágil y eficiente, mejorando la calidad del software.
- **Mejora continua**: Ayuda a identificar áreas donde se pueden realizar mejoras en el proceso de detección de defectos.
- **Evaluación de la eficiencia de las pruebas**: Permite medir la efectividad de las estrategias y herramientas de prueba.

El MTTD se utiliza para:

- Evaluar la efectividad del proceso de pruebas: Un MTTD bajo indica que el equipo de pruebas es eficiente en identificar defectos rápidamente.
- Monitorear y mejorar el rendimiento del equipo: Al medir y analizar el MTTD, se pueden identificar áreas de mejora en el proceso de desarrollo y pruebas.
- Planificación y gestión de proyectos: Ayuda a los gerentes de proyecto a evaluar la calidad del software y ajustar los cronogramas y recursos en consecuencia.

4.4.3 Cálculo del MTTD

Para calcular el MTTD se requiere:

- Datos sobre la detección de defectos: Registro del tiempo en que cada defecto fue introducido y detectado.

- Herramientas de seguimiento de defectos: Software como JIRA, Bugzilla u otras herramientas de gestión de proyectos y defectos que permitan registrar y seguir el tiempo de vida de un defecto.
- Metodología para registrar tiempos: Un proceso claro para registrar cuándo se introduce un defecto y cuándo se detecta [2].

4.4.4 Fórmula para el MTTD

La fórmula para calcular el MTTD es:

$$\text{MTTD} = \frac{\sum_{i=1}^{n} \text{Tiempo para detectar cada defecto}}{\text{Número total de defectos}} \tag{4.1}$$

Donde:

- n es el número total de defectos.
- Tiempo para detectar cada defecto es el tiempo transcurrido desde la introducción hasta la detección de cada defecto.

Esta fórmula proporciona una medida cuantitativa del tiempo promedio que toma detectar un defecto en el proceso de desarrollo de software.

4.4.5 Ejemplo 01

Supongamos que trabajamos en un proyecto de desarrollo de software y utilizamos JIRA para rastrear los defectos. A continuación se muestra cómo podríamos calcular el MTTD:

- Registro de defectos en JIRA: Cada vez que se introduce un defecto, se crea un ticket en JIRA con la fecha y hora de creación.
- Detección de defectos: Cuando un defecto es detectado, se actualiza el ticket con la fecha y hora de detección.

Tenemos los siguientes defectos y sus tiempos de introducción y detección registrados en JIRA:

- Defecto 1: Introducido el 1 de enero a las 10:00, Detectado el 1 de enero a las 13:00

- Defecto 2: Introducido el 2 de enero a las 14:00, Detectado el 2 de enero a las 19:00
- Defecto 3: Introducido el 3 de enero a las 09:00, Detectado el 3 de enero a las 11:00
- Defecto 4: Introducido el 4 de enero a las 15:00, Detectado el 4 de enero a las 19:00

Calculamos el tiempo para detectar cada defecto:

- Defecto 1: 3 horas
- Defecto 2: 5 horas
- Defecto 3: 2 horas
- Defecto 4: 4 horas

Se tarda 3.5 horas en localizar un defecto haciendo uso de la primera fórmula.

4.4.6 Código en Python para calcular MTTD:

A continuación se muestra el código en Python para calcular el MTTD utilizando los datos de tiempo de introducción y detección de defectos:

```python
from datetime import datetime

def calcular_mttd(tiempos_detectados):
    if not tiempos_detectados:
        return 0
    return sum(tiempos_detectados) / len(tiempos_detectados)

# Función para calcular la diferencia en horas entre dos timestamps
def calcular_diferencia_horas(inicio, fin):
    diferencia = fin - inicio
    return diferencia.total_seconds() / 3600

# Ejemplo de timestamps de introducción y detección de defectos
defectos = [
    {"introducido": "2023-01-01 10:00:00", "detectado": "2023-01-01 13:00:00"},
    {"introducido": "2023-01-02 14:00:00", "detectado": "2023-01-02 19:00:00"},
    {"introducido": "2023-01-03 09:00:00", "detectado": "2023-01-03 11:00:00"},
    {"introducido": "2023-01-04 15:00:00", "detectado": "2023-01-04 19:00:00"}
]
```

```
# Calcular tiempos de detección en horas
tiempos_detectados = []
for defecto in defectos:
    introducido = datetime.strptime(defecto["introducido"], "%Y-%m-%d %H:%M:%S")
    detectado = datetime.strptime(defecto["detectado"], "%Y-%m-%d %H:%M:%S")
    tiempo_detectado = calcular_diferencia_horas(introducido, detectado)
    tiempos_detectados.append(tiempo_detectado)

# Calcular MTTD
mttd = calcular_mttd(tiempos_detectados)
print(f"El Tiempo Promedio de Localización de Defectos (MTTD) es: {mttd} horas")
```

Este código Python toma los datos de tiempo de introducción y detección de defectos, calcula el tiempo para detectar cada defecto en horas, y luego calcula el MTTD como el promedio de estos tiempos. Es un ejemplo práctico para ilustrar cómo se puede implementar el cálculo del MTTD en un entorno real de desarrollo de software utilizando datos registrados.

4.4.7 Segunda fórmula

$$\text{MTTD} = \frac{\sum_{i=1}^{n} (t_{\text{detección},i} - t_{\text{introducción},i})}{n} \tag{4.2}$$

Donde:

- n es el número total de defectos.
- $t_{\text{detección},i}$ es el tiempo (en horas, días, etc.) en que se detectó el i-ésimo defecto.
- $t_{\text{introducción},i}$ es el tiempo (en horas, días, etc.) en que se introdujo el i-ésimo defecto.

4.4.8 Ejemplo 02

Supongamos que tenemos los siguientes datos sobre defectos en un archivo CSV (`defects.csv`):

id_defecto	fecha_introduccion	fecha_deteccion
1	2023-06-01	2023-06-05
2	2023-06-03	2023-06-07
3	2023-06-02	2023-06-10

4.4.9 Código en Python para Calcular MTTD

El código en Python lee el archivo CSV y calcula el Tiempo Promedio de Localización de Defectos:

```
import csv
from datetime import datetime

def calcular_mttd(archivo_csv):
    with open(archivo_csv, mode='r') as file:
        reader = csv.DictReader(file)
        total_tiempo = 0
        total_defectos = 0
        for row in reader:
            fecha_introduccion = datetime.strptime(row['fecha_introduccion'], '%Y-%m-%d')
            fecha_deteccion = datetime.strptime(row['fecha_deteccion'], '%Y-%m-%d')
            tiempo_deteccion = (fecha_deteccion - fecha_introduccion).days
            total_tiempo += tiempo_deteccion
            total_defectos += 1

        if total_defectos == 0:
            return 0

        mttd = total_tiempo / total_defectos
        return mttd

# Nombre del archivo CSV
archivo_csv = 'defects.csv'

mttd = calcular_mttd(archivo_csv)
print(f"El Tiempo Promedio de Localización de Defectos (MTTD) es: {mttd:.2f} días")
```

4.5 Tasa de Actualización

4.5.1 Definición

La Tasa de Actualización es una métrica que mide la frecuencia con la que se actualiza el software en un período de tiempo determinado [1]. Esta métrica es crucial para evaluar la agilidad del proceso de desarrollo y la capacidad del equipo para implementar mejoras, corregir errores y responder a las necesidades del usuario.

4.5.2 Importancia

- **Evaluación de la agilidad:** Permite medir la capacidad del equipo de desarrollo para entregar actualizaciones frecuentes.

- **Mejora continua:** Ayuda a identificar la rapidez con la que se pueden implementar mejoras y correcciones.

- **Satisfacción del usuario:** Una alta tasa de actualización puede indicar un equipo que responde rápidamente a las necesidades del usuario, mejorando la satisfacción.

4.5.3 La Tasa de Actualización se utiliza para:

- **Evaluar la agilidad del equipo de desarrollo:** Una alta tasa de actualización puede indicar que el equipo está trabajando de manera eficiente y respondiendo rápidamente a las necesidades de los usuarios.

- **Monitorear el ciclo de vida del software:** Ayuda a entender la frecuencia de cambios y mejoras que se están aplicando al software.

- **Planificación de recursos:** Permite a los gerentes de proyecto planificar mejor los recursos y el cronograma de desarrollo.

4.5.4 ¿Qué se necesita para calcular la Tasa de Actualización?

- **Datos sobre el número de actualizaciones:** Registro del número de actualizaciones realizadas en un período específico.

- **Periodo de tiempo definido:** Un período de tiempo durante el cual se cuenta el número de actualizaciones, como por mes, trimestre o año.

- **Herramientas de gestión de proyectos:** Software como JIRA, Git, o cualquier otra herramienta que registre las actualizaciones y cambios en el software.

4.5.5 Fórmula

$$\text{Tasa de Actualización} = \frac{\text{Número de actualizaciones en un período}}{\text{Período de tiempo}}$$

4.5.6 Ejemplo 01

Supongamos que en un proyecto de software se realizaron las siguientes actualizaciones en un mes:

- Actualización 1: 5 de junio
- Actualización 2: 12 de junio
- Actualización 3: 19 de junio
- Actualización 4: 26 de junio

Calculamos la Tasa de Actualización para el mes de junio (30 días) de la siguiente manera:

$$\text{Tasa de Actualización} = \frac{4}{30} = 0.133 \text{ actualizaciones por día}$$

Esto significa que, en promedio, se realiza una actualización aproximadamente cada 7.5 días.

4.5.7 Código en Python 01

Código en Python para Calcular la Tasa de Actualización del Ejemplo 01

```python
from datetime import datetime

def calcular_tasa_actualizacion(fechas_actualizaciones, periodo_dias):
    if not fechas_actualizaciones:
        return 0
    num_actualizaciones = len(fechas_actualizaciones)
    tasa_actualizacion = num_actualizaciones / periodo_dias
    return tasa_actualizacion

# Ejemplo de fechas de actualizaciones (formato: 'YYYY-MM-DD')
fechas_actualizaciones = [
    "2023-06-05",
    "2023-06-12",
    "2023-06-19",
    "2023-06-26"
]

# Convertir las fechas a objetos datetime
fechas_actualizaciones = [datetime.strptime(fecha, '%Y-%m-%d') for fecha in fechas_actualizaciones]

# Período en días (para este ejemplo, usamos un mes de 30 días)
periodo_dias = 30

tasa_actualizacion = calcular_tasa_actualizacion(fechas_actualizaciones, periodo_dias)
print(f"La Tasa de Actualización es: {tasa_actualizacion:.3f} actualizaciones por día")
```

4.5.8 Código en Python 02

Código en Python para Calcular la Tasa de Actualización desde un Archivo CSV

```
import csv
from datetime import datetime

def calcular_tasa_actualizacion(archivo_csv, periodo_dias):
    with open(archivo_csv, mode='r') as file:
        reader = csv.DictReader(file)
        fechas_actualizaciones = [datetime.strptime(row['fecha_actualizacion'], '%Y-%m-%d') for row in reader]

        if not fechas_actualizaciones:
            return 0

        num_actualizaciones = len(fechas_actualizaciones)
        tasa_actualizacion = num_actualizaciones / periodo_dias
        return tasa_actualizacion

# Nombre del archivo CSV
archivo_csv = 'actualizaciones.csv'

# Período en días (para este ejemplo, usamos un mes de 30 días)
periodo_dias = 30

tasa_actualizacion = calcular_tasa_actualizacion(archivo_csv, periodo_dias)
print(f"La Tasa de Actualización es: {tasa_actualizacion:.3f} actualizaciones por día")
```

4.5.9 Ejemplo 02

Supongamos que estamos utilizando un sistema de gestión de versiones como Git para rastrear las actualizaciones del software. Queremos calcular la Tasa de Actualización durante los últimos 6 meses.

4.5.10 Recolección de datos

Utilizamos comandos de Git para obtener el número de *commits* (que representan actualizaciones) realizados durante los últimos 6 meses.

4.5.11 Cálculo de la Tasa de Actualización

Contamos el número de *commits* y dividimos por el número de meses.

Supongamos que tenemos los siguientes datos:

- Mes 1: 5 actualizaciones

- Mes 2: 6 actualizaciones
- Mes 3: 4 actualizaciones
- Mes 4: 7 actualizaciones
- Mes 5: 3 actualizaciones
- Mes 6: 5 actualizaciones

Calculamos la Tasa de Actualización de la siguiente manera:

$$\text{Tasa de Actualización} = \frac{5+6+4+7+3+5}{6} = \frac{30}{6} = 5 \text{ actualizaciones por mes}$$

Esto significa que, en promedio, se realizan 5 actualizaciones al software por mes.

4.5.12 Código en Python III

Código en Python para Calcular la Tasa de Actualización del Ejemplo 02

```python
def calcular_tasa_actualizacion(num_actualizaciones, periodo_tiempo):
    if periodo_tiempo == 0:
        return 0
    return num_actualizaciones / periodo_tiempo

# Ejemplo: número de actualizaciones por mes
actualizaciones_por_mes = [5, 6, 4, 7, 3, 5]

# Calcular el total de actualizaciones y el periodo de tiempo
num_actualizaciones = sum(actualizaciones_por_mes)
periodo_tiempo = len(actualizaciones_por_mes)  # en meses

# Calcular la Tasa de Actualización
tasa_actualizacion = calcular_tasa_actualizacion(num_actualizaciones, periodo_tiempo)
print(f"La Tasa de Actualización es: {tasa_actualizacion:.2f} actualizaciones por mes")
```

4.6 Satisfacción del Usuario

Las métricas principales para evaluar la satisfacción del usuario con el software incluyen el Net Promoter Score (NPS), que mide la disposición de los usuarios a recomendar el software; las encuestas de satisfacción, que recopilan opiniones detalladas sobre la usabilidad y el rendimiento; la tasa de retención de usuarios, que indica la fidelidad de los usuarios al software; las calificaciones y reseñas públicas, que reflejan la percepción general del producto;

y las métricas de usabilidad, que evalúan la facilidad de uso y la eficiencia del software [6]. Estas métricas proporcionan una visión completa de la experiencia del usuario y ayudan a las empresas a mejorar continuamente su producto para satisfacer mejor las necesidades de sus usuarios.

4.6.1 Definición

Las métricas en la satisfacción del usuario son herramientas utilizadas para medir y evaluar cómo los usuarios experimentan y perciben un producto, servicio o experiencia específica. Estas métricas proporcionan datos cuantitativos y cualitativos que permiten a las empresas entender mejor las necesidades y expectativas de sus usuarios, identificar áreas de mejora y tomar decisiones informadas para optimizar la satisfacción general del usuario.

4.6.2 Aspectos Clave de Evaluación

Las métricas en satisfacción del usuario son herramientas para medir cómo los usuarios perciben un producto o servicio. Los tipos principales incluyen:

1. Net Promoter Score (NPS): Mide la disposición de los usuarios a recomendar el producto.
2. Encuestas de Satisfacción: Recopilan opiniones detalladas sobre la experiencia del usuario.
3. Tasa de Retención de Usuarios: Indica cuántos usuarios continúan utilizando el producto.
4. Calificaciones y Reseñas: Opiniones públicas que reflejan la satisfacción del usuario.
5. Métricas de Usabilidad: Evalúan la facilidad de uso y eficiencia del producto.

4.6.3 Aplicación

Las métricas de satisfacción del usuario se utilizan para evaluar y mejorar la experiencia que los usuarios tienen con productos o servicios. Por ejemplo, una empresa de software puede implementar encuestas periódicas de satisfacción del usuario para medir cómo los clientes perciben la usabilidad, el

rendimiento y el soporte del producto. Estas métricas proporcionan datos concretos que permiten a la empresa identificar áreas de mejora y tomar acciones específicas para optimizar la satisfacción del usuario.

4.6.4 Limitaciones

Sin embargo, estas métricas también tienen limitaciones importantes. Por ejemplo, la interpretación subjetiva de las respuestas puede variar entre diferentes usuarios, lo que puede llevar a conclusiones sesgadas o incompletas sobre la verdadera satisfacción del usuario. Además, la participación selectiva de los usuarios en las encuestas puede sesgar los resultados hacia experiencias extremas, como usuarios muy satisfechos o muy insatisfechos, lo que puede no representar de manera precisa la experiencia promedio del usuario.

4.6.5 Ejemplo de código en Python

```python
import numpy as np

# Función para calcular el Net Promoter Score (NPS)
def calcular_nps(respuestas):
    """
    Calcula el Net Promoter Score (NPS) a partir de una lista de respuestas.

    Args:
    respuestas (list): Lista de respuestas con valores entre 0 y 10.

    Returns:
    float: El NPS calculado.
    """
    # Clasificar las respuestas en Promotores, Pasivos y Detractores
    promotores = [r for r in respuestas if r >= 9]
    pasivos = [r for r in respuestas if 7 <= r <= 8]
    detractores = [r for r in respuestas if r <= 6]

    # Calcular los porcentajes
    total_respuestas = len(respuestas)
    porcentaje_promotores = (len(promotores) / total_respuestas) * 100
    porcentaje_detractores = (len(detractores) / total_respuestas) * 100

    # Calcular el NPS
    nps = porcentaje_promotores - porcentaje_detractores

    return nps

# Ejemplo de recopilación de respuestas de usuarios
respuestas_usuarios = [
    10, 9, 8, 7, 6, 5, 4, 3, 2, 1, 0, 10, 8, 7, 5, 6, 10, 9, 9, 8, 7, 6, 5, 4
]
```

```
# Calcular el NPS
nps = calcular_nps(respuestas_usuarios)

# Mostrar el resultado
print(f"El Net Promoter Score (NPS) es: {nps:.2f}")

# Análisis básico
promotores = len([r for r in respuestas_usuarios if r >= 9])
pasivos = len([r for r in respuestas_usuarios if 7 <= r <= 8])
detractores = len([r for r in respuestas_usuarios if r <= 6])
print(f"Promotores: {promotores}")
print(f"Pasivos: {pasivos}")
print(f"Detractores: {detractores}")
```

La satisfacción del usuario es clave para el éxito de cualquier producto o servicio. Las principales métricas para evaluarla incluyen el Net Promoter Score (NPS), encuestas de satisfacción, tasa de retención de usuarios, calificaciones y reseñas, y métricas de usabilidad. Estas herramientas ofrecen datos valiosos sobre la experiencia del usuario, permitiendo identificar áreas de mejora y tomar decisiones para optimizar el producto. Sin embargo, es importante considerar las limitaciones, como la subjetividad en las respuestas y la participación sesgada, para obtener una imagen precisa de la satisfacción general. Evaluar estas métricas de forma continua puede impulsar mejoras significativas en la experiencia del usuario.

4.7 Consumo de Memoria

El consumo de memoria se refiere a la cantidad de memoria RAM utilizada por un sistema en un momento dado. Las métricas clave incluyen el uso total de memoria, tanto física como virtual, la fragmentación que puede reducir la eficiencia, la tasa de alocación que indica la frecuencia de solicitudes de nueva memoria, y las fugas de memoria que pueden afectar el rendimiento al reservar memoria que no se libera correctamente [6]. Estas métricas son fundamentales para gestionar eficazmente los recursos de memoria, optimizar el rendimiento y garantizar la estabilidad del sistema informático.

4.7.1 Definición

El consumo de memoria se refiere a la cantidad de memoria RAM que un sistema informático utiliza en un momento específico para ejecutar programas y procesos. Es fundamental para determinar la eficiencia y el rendimiento

del sistema, ya que la falta de memoria puede ralentizar las operaciones o incluso hacer que el sistema falle.

4.7.2 Dimensiones de Evaluación

Las métricas en satisfacción del usuario son herramientas para medir cómo los usuarios perciben un producto o servicio. Los tipos principales incluyen:

- Uso Total de Memoria: La cantidad absoluta de memoria utilizada, incluyendo la física y la virtual.
- Uso de Memoria Física: La cantidad de RAM física ocupada por los procesos en ejecución.
- Uso de Memoria Virtual: El espacio de disco utilizado como memoria RAM adicional cuando la memoria física está llena.
- Fragmentación de Memoria: La dispersión de bloques de memoria que no se utilizan eficientemente.
- Fugas de Memoria (Memory Leaks): La pérdida gradual de memoria debido a la incapacidad de liberarla después de que ya no se necesite.
- Perfil de Uso de Memoria a lo Largo del Tiempo: La variación del uso de memoria a medida que los programas se ejecutan y se cierran.
- Memoria por Proceso: La cantidad de memoria utilizada por cada programa o proceso individualmente.
- Eficiencia de Caché: La medida en que se aprovecha la caché para reducir los accesos a memoria principal.

4.7.3 Aplicación

El monitoreo del consumo de memoria es esencial en la gestión eficiente de recursos informáticos, permitiendo optimizar el rendimiento y la estabilidad de sistemas operativos y aplicaciones. Proporciona información crítica para identificar y corregir problemas como fugas de memoria y fragmentación, que pueden impactar negativamente en la experiencia del usuario y la eficiencia operativa. Además, facilita la planificación de recursos al determinar cuánta

memoria se requiere para ejecutar aplicaciones y procesos sin comprometer el desempeño del sistema. En entornos de desarrollo de software, el monitoreo del consumo de memoria ayuda a los desarrolladores a mejorar la eficiencia del código y evitar errores relacionados con la gestión de memoria, contribuyendo así a la creación de aplicaciones más robustas y responsivas.

4.7.4 Limitaciones

El monitoreo del consumo de memoria presenta desafíos significativos. La interpretación precisa de las métricas puede ser complicada debido a la complejidad de los sistemas operativos y la interacción entre múltiples procesos concurrentes. Además, algunas herramientas de monitoreo pueden introducir una sobrecarga adicional de recursos, lo que podría afectar temporalmente el rendimiento del sistema durante su implementación. El contexto específico de uso de memoria también influye en la validez de las métricas, requiriendo un entendimiento profundo de las características y cargas de trabajo específicas de las aplicaciones monitorizadas. La implementación de herramientas avanzadas de monitoreo puede ser costosa en términos de recursos y capacitación, limitando su accesibilidad para organizaciones con presupuestos ajustados o recursos limitados de personal técnico.

4.8 Facilidad de Aprendizaje

4.8.1 Definición

La facilidad de aprendizaje se refiere a la rapidez y la eficiencia con la que una persona puede adquirir nuevas habilidades, conocimientos o comportamientos. Este concepto es esencial en varios campos como la educación, la psicología, el diseño de interfaces de usuario y la ingeniería de software. Un sistema, herramienta o material que posee una alta facilidad de aprendizaje permite a los usuarios entenderlo y usarlo de manera efectiva con un mínimo de esfuerzo.

4.8.2 Tipos

Facilidad de Aprendizaje Intrínseca

Se refiere a la facilidad inherente de un contenido o habilidad para ser aprendido, sin depender de ayudas externas. Ejemplo: Matemáticas básicas.

Facilidad de Aprendizaje Extrínseca

Depende de factores externos como métodos de enseñanza, herramientas de aprendizaje y entornos educativos. Ejemplo: Uso de software educativo interactivo para enseñar historia.

Facilidad de Aprendizaje del Usuario Final

Relacionada con la facilidad con la que los usuarios finales pueden aprender a usar un producto o sistema. Ejemplo: La facilidad con la que alguien puede aprender a usar una nueva aplicación móvil.

Facilidad de Aprendizaje en el Diseño de Interfaz de Usuario

Refleja cómo un diseño de interfaz facilita el aprendizaje y uso efectivo por parte del usuario. Ejemplo: Interfaz intuitiva de una página web.

4.8.3 Aplicaciones

Educación

Desarrollo de currículos y métodos de enseñanza que faciliten el aprendizaje. Ejemplo: Cursos online interactivos.

Tecnología y Software

Diseño de interfaces de usuario que sean intuitivas y fáciles de aprender. Ejemplo: Sistemas operativos con tutoriales integrados.

Capacitación Corporativa

Programas de entrenamiento diseñados para minimizar el tiempo de aprendizaje y maximizar la retención. Ejemplo: Software de gestión empresarial con módulos de capacitación interactivos.

Desarrollo Personal

Herramientas y técnicas para mejorar la capacidad de aprender de los individuos. Ejemplo: Aplicaciones de aprendizaje de idiomas que adaptan el contenido según el progreso del usuario.

4.8.4 Limitaciones

Diferencias Individuales

No todas las personas tienen la misma capacidad para aprender rápidamente debido a diferencias cognitivas, emocionales y educativas.

- **Ejemplo:** Personas con discapacidades de aprendizaje pueden necesitar adaptaciones especiales.

Complejidad del Contenido

Algunos temas o habilidades son inherentemente complejos y requieren más tiempo y esfuerzo para ser aprendidos.

- **Ejemplo:** Aprender a programar en un lenguaje de programación complejo.

Calidad del Diseño Educativo

La facilidad de aprendizaje puede verse afectada negativamente por un mal diseño de materiales educativos o herramientas de aprendizaje.

- **Ejemplo:** Un curso online mal estructurado puede dificultar el aprendizaje.

Recursos Limitados

La falta de recursos adecuados, como tiempo, dinero o acceso a tecnología, puede limitar la facilidad de aprendizaje.

- **Ejemplo:** Escuelas con pocos recursos tecnológicos.

4.8.5 Métricas

Para evaluar la facilidad de aprendizaje, se utilizan diversas métricas que ayudan a cuantificar la eficacia y eficiencia del proceso de aprendizaje. Algunas de las más comunes son:

Tiempo para Aprender

Mide el tiempo que tarda un usuario en aprender a usar una nueva herramienta o sistema hasta alcanzar un nivel de competencia específico.

Tasa de Error

Calcula la frecuencia de errores cometidos por los usuarios durante el proceso de aprendizaje. Una menor tasa de error indica una mayor facilidad de aprendizaje.

Retención del Conocimiento

Evalúa cuánto tiempo los usuarios pueden recordar y aplicar lo aprendido después de un período sin uso. Alta retención indica una alta facilidad de aprendizaje.

Satisfacción del Usuario

Se mide mediante encuestas y cuestionarios que evalúan la percepción de los usuarios sobre la facilidad de uso y el proceso de aprendizaje. Una alta satisfacción suele correlacionar con una mayor facilidad de aprendizaje.

Eficiencia en la Tarea

Mide la capacidad del usuario para completar tareas específicas después de un período de aprendizaje. Mayor eficiencia indica una mayor facilidad de aprendizaje.

Número de Recursos Necesarios

Cuenta la cantidad de recursos (manuales, tutoriales, asistencia) que los usuarios necesitan para aprender una nueva herramienta o sistema. Menos recursos necesarios indican una mayor facilidad de aprendizaje.

4.8.6 Facilidad de Aprendizaje en Software

4.8.7 Definición

La facilidad de aprendizaje se refiere a la rapidez y eficiencia con la cual los usuarios pueden aprender a utilizar una aplicación de software sin la necesidad de instrucciones extensas o formación especializada [4]. Es esencial en entornos donde la eficiencia y la reducción de errores desde el primer uso son prioritarias, como en el software empresarial y herramientas de administración.

4.8.8 Factores que Influyen en la Facilidad de Aprendizaje

Intuitividad de la Interfaz

- **Diseño Coherente y Familiar:** Una interfaz que sigue convenciones de diseño reconocidas facilita la navegación intuitiva.
- **Feedback Visual Inmediato:** Respuestas visuales claras ante las acciones del usuario ayudan a corregir errores rápidamente.

Estructura y Organización del Flujo de Trabajo

- **Navegación Lógica:** La disposición de menús y opciones debe reflejar el proceso de trabajo del usuario, minimizando la carga cognitiva.
- **Acceso Directo a Funcionalidades Clave:** Las funciones más utilizadas deben ser accesibles de manera rápida y directa.

Documentación y Soporte

- **Manuales de Usuario Claros:** Documentación bien redactada que explique las funciones del software y cómo realizar tareas comunes.
- **Tutoriales Integrados o Ayuda Contextual:** Guías dentro del software que orienten a los usuarios sobre tareas específicas.

Capacidades de Personalización y Configuración

- **Adaptabilidad a Preferencias del Usuario:** Permitir ajustes de configuración que se adapten a las preferencias individuales de los usuarios.

4.8.9 Métricas para Evaluar la Facilidad de Aprendizaje

Tiempo de Aprendizaje

Mide el tiempo necesario para que los usuarios nuevos realicen tareas básicas.

Tasa de Error Inicial

Calcula la cantidad de errores cometidos por los usuarios al realizar tareas simples.

Feedback de los Usuarios

Recopila retroalimentación directa sobre la dificultad percibida y los obstáculos encontrados durante el aprendizaje.

4.8.10 Ejemplo Práctico

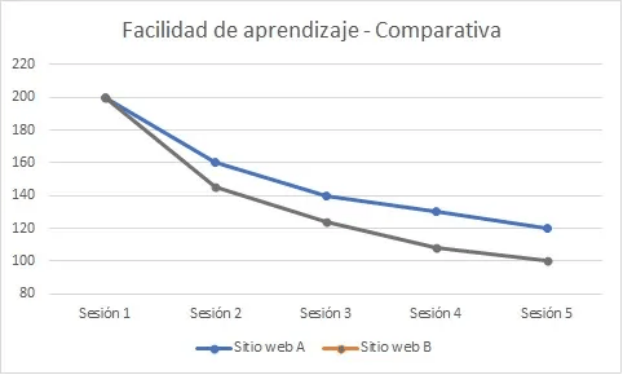

Figure 4.1: Comparativa de la facilidad de aprendizaje entre el sitio web A y B

Ejemplo de resultados de métricas de usabilidad y UX – Facilidad de aprendizaje (Tiempo por sesión): Vemos como en el primer uso son necesarios 200 segundos en ambos casos. Sin embargo, en usos posteriores ese tiempo va reduciéndose, y al cabo de 10 usos la media se sitúa en 120 segundos para el sitio web A y 100 segundos para el sitio web B. Podemos entonces concluir que la facilidad de aprendizaje del sitio web B es mejor que la del sitio A.

4.8.11 Utilidad

La facilidad de aprendizaje es esencial para optimizar la experiencia del usuario y maximizar la eficiencia operativa en aplicaciones de software complejas. Diseñar interfaces intuitivas, proporcionar documentación clara y ofrecer soporte efectivo son prácticas clave para mejorar este aspecto crucial de la usabilidad del software.

4.9 Vulnerabilidades Encontradas

Definición

Las vulnerabilidades encontradas se refieren a debilidades o fallos en un sistema, software o infraestructura que pueden ser explotados por atacantes para comprometer la seguridad, integridad, confidencialidad o disponibilidad del sistema [11]. Estas vulnerabilidades pueden estar presentes en el código fuente, configuraciones, políticas de seguridad, o incluso en los propios procesos operativos.

4.9.1 Tipos

Vulnerabilidades de Software

Errores o fallos en el código de un programa que pueden ser explotados.
Ejemplo: Desbordamiento de búfer.

Vulnerabilidades de Configuración

Errores en la configuración de sistemas que pueden dejar puertas abiertas a atacantes.
Ejemplo: Configuración incorrecta de permisos de usuario.

Vulnerabilidades de la Red

Debilidades en la infraestructura de red que pueden ser explotadas para interceptar o redirigir tráfico.
Ejemplo: Ataques de intermediario (MITM).

Vulnerabilidades Físicas

Puntos débiles en la seguridad física que permiten acceso no autorizado a sistemas o datos.
Ejemplo: Acceso físico no restringido a servidores.

Vulnerabilidades de Ingeniería Social

Técnicas que explotan las debilidades humanas para obtener acceso a información o sistemas.
Ejemplo: Phishing.

4.9.2 Aplicaciones

Auditoría de Seguridad

Evaluar y mitigar las vulnerabilidades para asegurar la integridad de los sistemas.
Ejemplo: Realización de pruebas de penetración (pentesting).

Desarrollo Seguro de Software

Implementar prácticas de codificación seguras para prevenir vulnerabilidades.
Ejemplo: Revisión de código y pruebas de seguridad.

Respuesta a Incidentes

Identificar y responder a vulnerabilidades explotadas en tiempo real.
Ejemplo: Implementación de un equipo de respuesta a incidentes de seguridad informática (CSIRT).

Cumplimiento Normativo

Asegurar que los sistemas cumplen con las regulaciones y estándares de seguridad.
Ejemplo: Implementación de controles de seguridad según ISO/IEC 27001.

4.9.3 Limitaciones

Actualización Continua

Las vulnerabilidades pueden surgir con cada nueva actualización de software, lo que requiere una vigilancia constante.
Ejemplo: Actualizaciones de seguridad periódicas.

Recursos Limitados

La identificación y mitigación de vulnerabilidades requiere tiempo, dinero y personal capacitado.
Ejemplo: Falta de presupuesto para un equipo de seguridad dedicado.

Falsos Positivos y Negativos

Las herramientas de detección pueden generar falsos positivos (detectando amenazas inexistentes) o falsos negativos (no detectando amenazas reales).
Ejemplo: Un sistema de detección de intrusiones (IDS) que no identifica una brecha de seguridad.

Complejidad de Sistemas

La creciente complejidad de los sistemas y redes puede dificultar la identificación de todas las posibles vulnerabilidades.
Ejemplo: Infraestructuras de nube híbrida con múltiples puntos de entrada.

4.9.4 Métricas

Para evaluar la severidad y el impacto de las vulnerabilidades encontradas, se utilizan diversas métricas. Algunas de las más comunes son:

Common Vulnerability Scoring System (CVSS)

CVSS es un estándar abierto para evaluar la severidad de las vulnerabilidades de seguridad. Se divide en tres métricas:

- **Base**: Representa las características intrínsecas de una vulnerabilidad.
- **Temporal**: Considera aspectos que cambian con el tiempo, como la disponibilidad de un parche.
- **Ambiental**: Refleja las características del entorno en el que se encuentra la vulnerabilidad.

Exploitability Metrics

Estas métricas evalúan la facilidad con la que una vulnerabilidad puede ser explotada. Consideran factores como la complejidad del ataque, la autenticación necesaria y los vectores de acceso.

Impact Metrics

Miden el impacto potencial de una vulnerabilidad en la confidencialidad, integridad y disponibilidad de un sistema comprometido.

Remediation Effort

Esta métrica considera el esfuerzo necesario para mitigar una vulnerabilidad, incluyendo la dificultad de aplicar parches y la necesidad de reiniciar sistemas.

Attack Surface Metrics

Evalúan la cantidad de puntos de entrada potenciales que una vulnerabilidad podría ofrecer a un atacante. Una mayor superficie de ataque indica un mayor riesgo.

4.10 Conclusión del capitulo

En este capítulo, hemos explorado una serie de métricas esenciales para evaluar y mejorar tanto la mantenibilidad como la usabilidad del software. La

integración y análisis de estas métricas ofrecen una visión holística de la calidad del software. Al combinar métricas de eficiencia técnica con la percepción del usuario, las empresas pueden desarrollar software que no solo sea robusto y fácil de mantener, sino también intuitivo y satisfactorio para los usuarios finales. Este enfoque integral garantiza que el software pueda evolucionar y mejorar continuamente, adaptándose a las nuevas demandas del mercado y manteniendo una alta satisfacción del usuario, lo cual es crucial para el éxito a largo plazo en un entorno competitivo.

Chapter 5

Bibliography

[1] Arpan Bhattacharjee et al. "Edge Assisted Over the Air Software Updates". In: *IEEE* (), pp. 1–10. URL: https://weisongshi.org/papers/Bhatta23-OTA.pdf.

[2] Jose Campos et al. "Evaluating & improving fault localization techniques". In: (2017), pp. 1–27. URL: https://homes.cs.washington.edu/~mernst/pubs/fault-localization-tr160803.pdf.

[3] Ioan Gabriel Czibula et al. "An aggregated coupling measure for the analysis of object-oriented software systems". In: *Journal of Systems and Software* (2019). URL: https://www.sciencedirect.com/science/article/abs/pii/S0164121218302371.

[4] Omar Salvador Gómez. "Evaluando Arquitecturas de Software". In: *Parte* 2 (2007), pp. 1870–0888.

[5] Alfonso Lozano Tello and Asunción Gómez-Pérez. "Factores de decisión para la reutilización de componentes software". In: (1998). URL: https://oa.upm.es/72622/1/CON_NAC_06.pdf.

[6] Maximiliano Agustín Mascheroni et al. "Calidad de software e ingeniería de usabilidad". In: *XIV Workshop de Investigadores en Ciencias de la Computación*. 2012.

[7] Michail D Papamichail, Theodoros Diamantopoulos, and Andreas L Symeonidis. "Software reusability dataset based on static analysis metrics and reuse rate information". In: *Data in Brief* (2019). URL: https://www.sciencedirect.com/science/article/pii/S235234091931042X.

[8] Ayu Putu Luh Primandari and Sholiq. "Effort distribution to estimate cost in small to medium software development project with use case points". In: *Procedia Computer Science* (2015). URL: https://www.sciencedirect.com/science/article/pii/S1877050915035681.

[9] Radek Silhavy, Petr Silhavy, and Zdenka Prokopova. "Evaluating subset selection methods for use case points estimation". In: *Information and Software Technology* (2018). URL: https://www.sciencedirect.com/science/article/pii/S0950584917305153.

[10] Faheem Ullah and Thomas R. Gross. "Profiling for Detecting Performance Anomalies in Concurrent Software". In: (2015), pp. 1–10. URL: https://ethz.ch/content/dam/ethz/special-interest/infk/inst-cs/lst-dam/documents/Publications/SEPS_2015.pdf.

[11] John Viega and Gary McGraw. *Building Secure Software: How to Avoid Security Problems the Right Way*. Addison-Wesley, 2001.