



*Calidad  
del  
Software*



# Tema 1. Calidad del software

## 1. Calidad del software

### Dimensiones de la calidad según Garwin

- **Calidad de desempeño:** Nos entrevistamos con el cliente, apuntamos las funcionalidades y vemos si se han implantado correctamente.
- **Calidad de las características:** Cuando hay una interfaz, que los usuarios se sorprendan y agraden desde el primer vistazo.
- **Confiabilidad:** El software funciona correctamente (sin fallos).
- **Conformidad:** Debe tener ciertos estándares de codificación.
- **Durabilidad:** Depurarse de manera sencilla sin que se generen efectos colaterales.
- **Servicio:** Actualizaciones/cambios en ciclos cortos.
- **Estética**
- **Percepción:** Si existe prejuicios sobre el desarrollador que alteren la calidad del software.

### Factores de la calidad según McCall

#### Operación

- **Corrección:** Grado de satisfacción de especificaciones y objetivos.
- **Confiabilidad:** Grado de cumplimiento de funcionalidad y precisión requerida.
- **Eficiencia:** Cantidad recursos de cómputo y código requeridos para llevar a cabo la funcionalidad.
- **Integridad:** Grado en el que es posible controlar el acceso de personas no autorizadas.
- **Usabilidad:** Curva de aprendizaje del programa.

#### Revisión

- **Facilidad de recibir mantenimiento**
- **Flexibilidad:** esfuerzo necesario para modificar un programa que ya opera.
- **Susceptibilidad de someterse a pruebas:** esfuerzo para garantizar que se cumple con la funcionalidad.

#### Transición

- **Portabilidad:** Esfuerzo para transferir un programa de un ambiente de sistema hardware a otro.
- **Reusabilidad:** Grado en el que el software puede volver a utilizarse en otras aplicaciones.
- **Interoperabilidad:** Esfuerzo requerido para acoplar un sistema con otro.

### Ejercicio clase: Test de comprobación de atributos.

Confiabilidad: ¿Lee correctamente la tarjeta al pasarlala por el dispositivo?

Integridad: ¿Una persona sin la tarjeta requerida puede pasar la asistencia por el dispositivo?

Portabilidad: ¿Se puede utilizar la aplicación en cualquier tipo de dispositivo?

Flexibilidad: ¿Es posible pasar la asistencia fuera del horario de clases?

## 2. El dilema de la calidad del software

### Situaciones que pueden comprometer la calidad

**Software ‘suficientemente bueno’:** Cuando en un software su versión más temprana tiene errores que son conocidos y que más adelante se solucionaron mediante una actualización.

### 3. El costo de la calidad del software

#### Tipos de costes

- **Costos de prevención:**

- Actividades de administración para planear y coordinar las actividades de control y aseguramiento de la calidad.
- Actividades técnicas para desarrollar modelos de requerimientos y de diseño.
- Planear pruebas.

- **Costos de evaluación:**

- Revisiones técnicas.
- Recopilación de datos y métricas.
- Hacer pruebas y depurar.

- **Costos de defecto o falla:**

- **Internos:** Se detecta un error en un producto antes del envío a los consumidores.
- **Externos:** Se detecta un error después de que el producto se envió a los consumidores.

# Tema 2. técnicas de Revisión

## 1. Técnicas de revisión

Descubrir errores para no producir defectos.

- **Error** → problema que encontramos en el análisis o toma de requisitos, por ejemplo.
- **Defecto** → se considera defecto cuando el error pasa a la siguiente fase, por ejemplo, encuentra el error el cliente.
- **Revisões formales** → planificación, preparación y estructuración de la reunión, después resaltamos errores, hacemos correcciones y las verificamos.
- **Revisões informales** → en el momento hacemos un debate, por ejemplo.

## 2. Métricas de revisión

Debemos poder medir las revisiones, para ello utilizamos métricas.

- **Esfuerzo de preparación** → revisión técnica con planificación antes del día de la reunión. Se mide en horas-personas.
  - **Esfuerzo de evaluación** → se revisa el producto de trabajo. Se mide en horas-personas.
  - **Esfuerzo de repetición** → cuando se describen los errores, los solucionamos para que no se conviertan en defectos. Se mide en horas-personas.
  - **Tamaño del producto de trabajo** → se mide según el tamaño del producto. Por ejemplo: líneas de código, páginas de un documento.
  - **Errores menos detectados** → #errores menores, se consideran menores cuando es un tiempo pequeño.
  - **Errores mayores detectados** → #errores mayores, se consideran mayores cuando es un tiempo grande.
  - **Esfuerzo total de la revisión** →  $E_p + E_a + E_r$
  - **Número total de errores descubiertos** →  $E_{total} = E_{mayor} + E_{menor}$
  - **Errores encontrados por TPT** → densidad del error =  $E_{total}/TPT$
- 
- **Densidad de error** → hacer estimaciones de los errores que esperas recibir.
  - **TPT** → producto de trabajo.  
Ejemplo →  $E_{promedio} = E_{menor} * E_{menortotal} + E_{mayor} * E_{mayortotal}$

### Eficacia del costo de las revisiones

Hay que aplicar revisiones para evitar entregar el proyecto más tarde que si hubiéramos hecho las revisiones.

### 3. Tipos de revisiones

- **Formales**

- Tenemos claro los roles de la organización
- Planificación adecuada hace que sea más formal
- Estructura correcta
- Seguimiento de las correcciones

- **Informal**

- Verificación de escritorio, se habla en el café, descanso... > se habla de como va el producto

¿Cómo mejorar las revisiones informales?

1. Generar listas de revisión.
2. Programación por pares.
3. Tiempo dedicado es importante, máximo una hora o dos.

**Revisión técnica formal:** Sus objetivos es reducir el alcance donde puedan existir errores en el software.

#### **Pasos para realizar una RTF:**

1. Primero hay que asignar los roles
  - a. Productor (desarrollador del producto SW)
  - b. Líder del proyecto
  - c. Líder de la revisión
  - d. Dos o tres revisores
2. Preparar un escenario previo donde el productor informa al líder del proyecto.
3. Cuando llega el día, estará el líder de la reunión, el productor y todos los revisores donde será un secretario que irá anotando todo lo acontecido en la reunión.
4. En la misma se analiza la agenda del día y el productor explica detalladamente el producto del trabajo a revisar.
5. Se realiza el RTF (1 página) donde se especifica quien lo ha revisado y que se ha descubierto y este se entregará al líder del proyecto.
6. También se hace el reporte técnico formal en le cual se identifican áreas de problemas en el producto y sirve como lista de verificación de acciones que guie al productor cuando se hagan las correcciones.

**Reporte técnico formal (1 página)** → que se revisó, quien, que se descubrió, conclusiones y se entrega al líder proyecto.

**Reporte técnico formal (Anexo: Lista de pendientes)** → se explican los errores menores que se han encontrado.

### 4. Consejos para la revisión

1. Revisar el producto, no al productor
2. Establecer una agenda y seguirla
3. Limitar los debates y las contestaciones
4. Comentar las áreas con problemas, pero no resolver ninguno
5. Tomar notas por escrito
6. Limitar el número de participantes en la reunión
7. Revisar otras revisiones realizadas anteriormente

# Tema 3. Aseguramiento de Calidad del Software

## 1. Concepto de ACS

**Administración y actividades:** específicas del proceso de SW para garantizar SW de calidad y a tiempo (llamada Actividad Sombrilla).

### Incluye:

- **Proceso** → actividades para llegar al objetivo
- **Tareas** → específicas de aseguramiento y control de calidad
- **Métodos y herramientas** de ISW
- **Control** de todos los productos del SW y sus cambios
- **Procedimiento** para que se cumplan todos los estándares del desarrollo de SW
- **Mecanismo de medición y reporte**

### ¿Cuáles son las etapas?

- Definir calidad del SW en varios niveles de abstracción
- Identificar actividades de ACS que filtren los errores antes de convertirse en defectos
- Desarrollar el control de calidad y actividades
- Usar métricas para desarrollar estrategias para mejorar el proceso del SW

### ¿Cuál es el producto final?

- **Plan de Aseguramiento de la Calidad del SW:** Estrategia de ACS para el equipo de SW
- **Durante modelado y codificación:** salida de las revisiones técnicas
- **Durante las pruebas:** planes y procedimientos de prueba
- **Otros** productos del trabajo asociados a la mejora

### ¿Cómo me aseguro de que lo hice bien?

- Encontrar errores antes de que sean defectos
- Mejorar la eficiencia en la eliminación de defectos

## 2. Elementos del ACS

- **Estándares; IEEE, ISO** → una organización de SW los adopta o los impone el cliente.
  - El ACS asegura que los estándares se sigan para todos los productos de trabajo.
- **Revisiones y auditorías:** revisiones técnicas sirven para detectar errores.
  - Auditoria son un tipo de examen crítico y sistemático efectuado por el personal de ACS para que se sigan las buenas prácticas en el proceso de revisión técnica del SW.
- **Pruebas:** función del control de calidad para detectar errores.
  - El ACS garantiza que las pruebas se planifican de manera adecuada y que se realicen con eficiencia.

- **Colección y análisis de errores:** mejorar implica medio como se esta haciendo algo.
  - El ACS reúne y analiza errores y datos acerca de los defectos para entender como se cometan errores y que actividades del ISW son mas adecuadas para eliminarlos.
- **Administración del cambio:** el SW puede cambiar
  - El ACS asegura que se han seguido prácticas adecuadas para la administración del cambio.
- **Educación:** formación
  - El ACS debe proponer y patrocinar formaciones.
- **Administración de los proveedores:** El ACS asegura que el proveedor nos proporcione SW de alta calidad y se establezcan cláusulas de calidad en el contrato.
  - Shell personalizado
  - Licencias
  - SW contratado
- **Administración de la seguridad:** delitos cibernéticos, privacidad, cortafuegos en webapps, etc.
  - El ACS garantiza que para lograr la seguridad del SW se utilicen el proceso y la tecnología adecuados.
- **Seguridad:** consecuencia de defectos ocultos en SW crítico.
  - El ACS evalúa el efecto de los defectos de SW e indica los pasos para disminuir el riesgo.
- **Administración de riesgos:** análisis y mitigación de riesgos en responsabilidad de los ingenieros del SW.

### 3. Tareas, metas y métricas del ACS

#### El ACS se compone de varias tareas asociadas con:

- **Ingenieros del SW** → trabajo técnico
- **El grupo de ACS** → auxilian a los Ingenieros de SW

#### Tareas del grupo de ACS:

- **Prepara el plan de ACS para un proyecto:** reuniones > revisado por todos los participantes.
- **Descripción del proyecto:** estándares.

#### Metas:

- **Calidad de los requerimientos:** análisis de requisitos completo. El ACS garantiza que se ha revisado el análisis de requisitos.
- **Calidad del diseño:** ACS busca atributos de diseño que sean de calidad.
- **Calidad del código:** ACS identifica atributos que permitan hacer un análisis de calidad del código.
- **Eficacia del control de calidad:** el equipo de SW debe aplicar recursos limitados. El ACS analiza la asignación de recursos para las revisiones y las pruebas a fin de evaluar si se asignan eficazmente.

## 4. El ACS estadístico

Pasos:

- **Recabar y clasificar** información de los **errores y defectos** de SW.
- **Rastrear** cada **error y defecto** hasta sus primeras causas.
- Utilizar el **Principio de Pareto** (80% de los defectos se debe al 20% de todas las causas posibles) y centrarnos en **identificar** el 20% de las **causas vitales**.
- **Analizar y corregir** problemas debido a las **causas vitales**.

## 5. Confiabilidad del SW

- Se debe medir la confiabilidad si un programa falla frecuentemente.
- La probabilidad que tiene un programa de cómputo de operar sin fallos en un ambiente específico por un tiempo específico
- Se define el tiempo medio entre fallas:

$$\text{TMEF} = \text{Tiempo medio para el fallo (TMPF)} + \text{el tiempo medio para la reparación (TMPR)}$$

$$\text{Disponibilidad} = 100 \times \text{TMPF} / (\text{TMPF} + \text{TMPR})$$

$$1\text{FET} = 1 \text{ fallo} / 1 \times 10^9 \text{ horas}$$

- Cada defecto tiene una tasa de fallos distinta.

## 6. Las normas de calidad ISO 9001

- **Describe en términos generales los elementos de aseguramiento de la calidad que se aplican a cualquier negocio, sin importar los productos o servicios ofrecidos**
- **Para que una empresa obtenga la certificación de calidad ISO 9001:**
  - Auditores externos revisan en detalle el sistema y las operaciones de calidad de una compañía, respecto del cumplimiento del estándar y de la operación eficaz
  - Los auditores pueden repetir la auditoría con periodicidad de 6 meses

## 7. El plan de ACS

- Mapa de ruta para aplicar las actividades el ACS
- Desarrollado por el grupo de ACS (o por el equipo de SW si no existe)

Contenido del plan de ACS

Preparar documento:

- **Propósito y alcance** del plan.
- Descripción de todos los **productos del trabajo** de ISW (documentos, código fuente, etc que se ubiquen dentro del ámbito del ACS).
- Todas las **normas y prácticas aplicables** que se utilicen durante el proceso del Software.
- **Acciones y tareas del ACS** (incluidas revisiones y auditorías) y su ubicación en el proceso del software.
- **Herramientas y métodos** que den apoyo a las acciones y tareas de ACS.
- **Procedimientos** para la administración de la configuración del software.
- **Roles y responsabilidades** relacionados con la calidad del producto.



# Tema 4. Estrategias de Prueba del Software

## 1. La prueba del software

- Descubrir errores para que no se conviertan en defectos.
- La estrategia de prueba de SW es una guía que describe los pasos que deben realizarse como parte de la prueba, cuando se planean y se llevan a cabo dichos pasos y cuánto esfuerzo, tiempo y recursos se requerirán.
- Es realizada por el gerente del proyecto y se busca no desperdiciar tiempo, hacer un esfuerzo necesario y detectar todos los errores cometidos.

## 2. Enfoque Estratégico para la Prueba del SW

Consejos para una EPSW efectiva

Realizar revisiones técnicas para eliminar errores antes de comenzar la prueba.

La prueba comienza en los componentes

- Pruebas de bajo nivel: Verificar un segmento de código.
- Pruebas de alto nivel: Validan las principales funciones del sistema vs. Especificaciones del cliente.

Las pruebas las realiza el desarrollador del software y un Grupo de Pruebas Independiente (GPI)

Prueba y depuración son actividades diferentes, se tiene que incluir dentro de la estrategia de prueba.

- **Verificación**

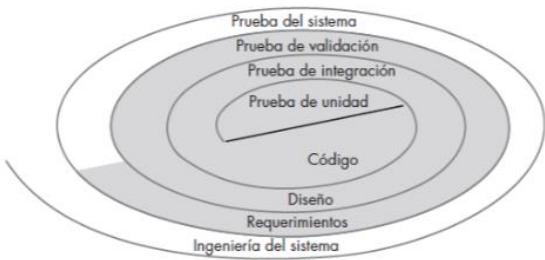
- Conjunto de tareas que garantizan que el software implementa correctamente una función específica.

- **Validación**

- Conjunto diferente de tareas que aseguran que el software que se construye sigue los requerimientos del cliente.

Las pruebas no deben verse como una red de seguridad, ya que la calidad se confirma durante las pruebas.

Visión general de la EPSW



- **Prueba de unidad:** se concentra en cada unidad (por ejemplo, componente, clase o un objeto de contenido de una webapp) del software: código fuente.
- **Prueba de integración:** el enfoque se centra en el diseño y la construcción de la arquitectura del software.
- **Prueba de validación:** donde los requerimientos establecidos como parte de su modelado se validan confrontándose con el software que se construyó.
- **Prueba del sistema:** donde el software y otros elementos del sistema se prueban como un todo.

### 3. Estrategia de Prueba para SW imperativo

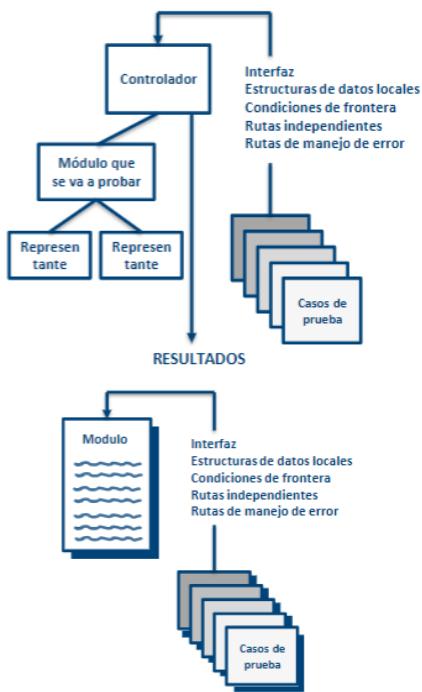
Esperar hasta que el sistema esté completamente construido y luego realizar las pruebas sobre el sistema total, con la esperanza de encontrar errores.

Realizarse pruebas diariamente

Visión incremental:

- Prueba de Unidad.
- Prueba de Integración.

#### Prueba de Unidad



Se concentra en cada unidad (Componente, clase u objeto de contenido de una webapp)

- **Interfaz del módulo:** Se ve el módulo como una caja negra y se estresa a la interfaz.
- **Estructura de datos locales:** Asegurar la integridad de las estructuras de datos.
- **Condiciones frontera:** Asegurar que el módulo opera adecuadamente en las fronteras para limitar. Ver problemas de asignación de memoria.
- **Pruebas de rutas:** Pasar por todos los caminos que vayan en el flujo de ejecución del módulo. Pasar por todos los casos del switch.
- **Componente controlador:** Es un main que acepta datos de casos de prueba, pasa tales datos al componente
- **Representante (o stub):** Evita llamar a módulos llamados por el componente que se va a probar.

#### Prueba de integración

Aunque los módulos en las pruebas de integración no den fallos, la integración entre ellos sí que puede dar fallos.

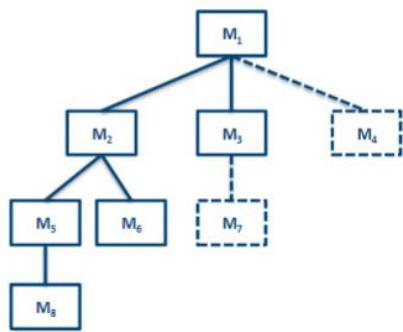
Enfoques de prueba de integración:

- **Big bang:** Todos los componentes se combinan por adelantado y todo el programa se prueba como un todo
- **Incremental:** El programa se construye y prueba en pequeños incrementos. El proceso puede ser descendente o ascendente

#### Tipos y fases de pruebas

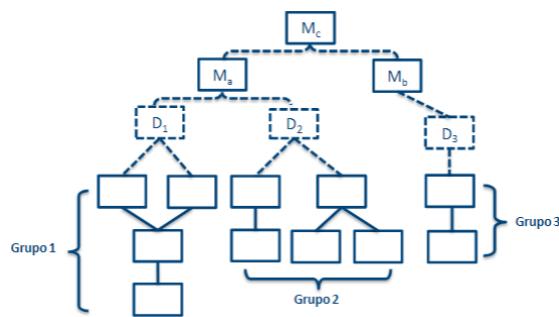
- **Integridad de interfaz.** Las interfaces internas y externas se prueban conforme cada módulo (o grupo) se incorpora en la estructura.
- **Validez funcional.** Se realizan pruebas diseñadas para descubrir errores funcionales ocultos.
- **Contenido de la información.** Se realizan pruebas diseñadas para descubrir errores ocultos asociados con las estructuras de datos locales o globales.
- **Rendimiento.** Se realizan pruebas diseñadas para verificar los límites del rendimiento establecidos durante el diseño del software.

## Integración descendente

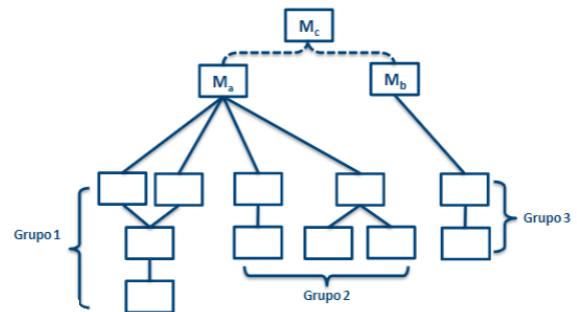
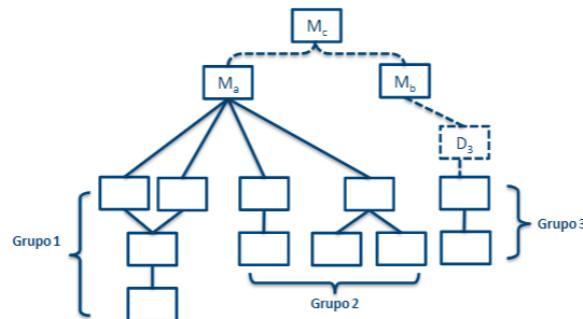


- Existen dos formas de recorrido:
  - Recorrido primero de profundidad
  - Recorrido primero en anchura
- Problema cuando se requiere procesamiento en niveles bajos en la jerarquía a fin de probar de manera adecuada los niveles superiores.
- **Stubs:** ningún dato significativo puede fluir hacia arriba en la estructura del programa.

## Integración descendente



Comienza con prueba de módulos atómicos (componentes en los niveles inferiores dentro de la estructura del programa) → No hacen falta **stubs**.



## Pruebas de regresión:

Nueva ejecución de algún subconjunto de pruebas que ya se realizaron a fin de asegurar que los cambios no propagaron efectos colaterales no deseados.

## Pruebas de validez funcional:

Son pruebas para verificar si hay errores funcionales.

## Pruebas de diseño:

Descubren errores en las estructuras de datos.

## Pruebas de rendimiento:

Son pruebas para ver los límites de rendimiento del software.  
A la hora de hacer las pruebas hay que tener en cuenta un plan de prueba, un procedimiento de prueba y un reporte de prueba.

## 4. Pruebas de Humo

- **Se aplica a proyectos críticos en el tiempo.** Es una estrategia de integración continua. El software se reconstruye y se prueba cada día.
- Con estas pruebas se minimiza el riesgo de integración, la calidad del producto final mejora, etc...
- **Beneficios**
  - **Se minimiza el riesgo de integración**
  - **Mejora de la calidad final del producto**
  - **Diagnóstico y corrección de errores simplificados**
  - **Valoración del progreso más sencilla**

## 5. La prueba para aplicaciones imperativas

La meta es diseñar una serie de casos de prueba que tengan una alta probabilidad de encontrar errores.

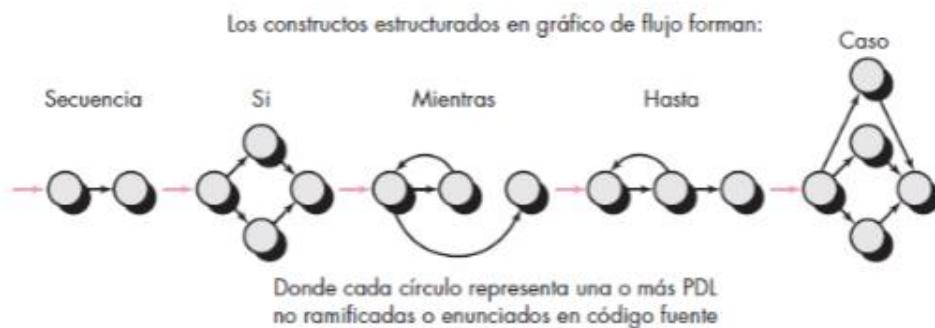
El SW se prueba desde dos perspectivas diferentes para ahorrar tiempo y esfuerzo:

- La lógica de programa interno se revisa usando técnicas de diseño de casos de prueba de '**cajablanca**'.
- Los requerimientos de software se revisan usando técnicas de diseño de casos de prueba de '**cajanegra**'.

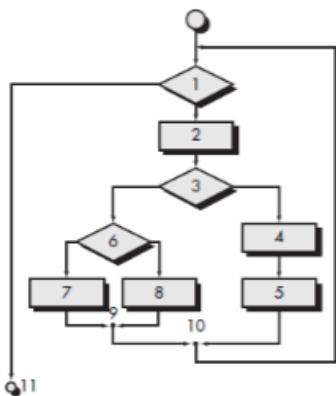
Comprobabilidad del software: ¿con cuánta facilidad puede probarse el código?

- **Operatividad.** "Mientras mejor funcione mejor puede probarse"
  - Si un sistema se diseña e implementa teniendo como objetivo la calidad, relativamente pocos errores bloquearán la ejecución de las pruebas.
- **Observabilidad.** "Lo que ve es lo que prueba"
  - Las entradas proporcionadas como parte de las pruebas producen distintas salidas. Los estados del sistema y las variables son visibles o consultables durante la ejecución. La salida incorrecta se identifica con facilidad.
- **Controlabilidad.** "Mientras mejor pueda controlar el software, más podrá automatizar y optimizar las pruebas"
  - Todo código es ejecutable a través de alguna combinación de entradas.
- **Descomponibilidad:** "Al controlar el ámbito de las pruebas, es posible aislar más rápidamente los problemas y realizar pruebas nuevas y más inteligentes"
  - El sistema de software se construye a partir de módulos independientes que pueden probarse de manera independiente.
- **Simplicidad:** "Mientras haya menos que probar, más rápidamente se le puede probar"
  - El programa debe mostrar simplicidad funcional; simplicidad estructural (arquitectura es modular para limitar la propagación de fallos) y simplicidad de código (se adopta un estándar de codificación para facilitar la inspección y el mantenimiento).
- **Estabilidad:** "Mientras menos cambios, menos perturbaciones para probar"
  - Los cambios al software son raros, se controlan cuando ocurren y no invalidan las pruebas existentes. El software se recupera bien de los fallos.
- **Comprendibilidad:** "Mientras más información se tenga, se probará con más inteligencia"
  - El diseño arquitectónico y las dependencias entre componentes internos, externos y compartidos son bien comprendidos. La documentación técnica es accesible al instante, está bien organizada, es específica, detallada y precisa.

## 6. Pruebas de caja blanca

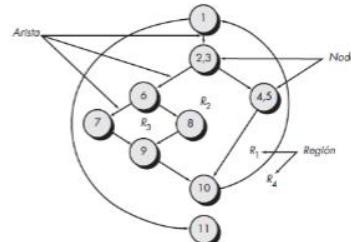


**Diagrama de Flujo:** estructura de control del programa.



### Gráfico de Flujo

- **Nodo:** uno o más enunciados de procedimiento.
- **Arista:** flujo de control.
- **Región:** área cortada por arista.



- Una ruta independiente es cualquiera que introduce al menos un nuevo conjunto de enunciados de procedimiento o una nueva condición en el programa.
- **Conjunto básico:** todo enunciado en el programa tendrá garantizada su ejecución al menos una vez, y cada condición se ejecutará en sus lados verdadero y falso.
- ¿Cómo saber cuántas rutas buscar? Cálculo de la Complejidad Ciclomática.
- **Complejidad Ciclomática (CC):** cota superior del número de casos de prueba para garantizar que cada enunciado en el programa se ejecuta al menos una vez

## 7. Prueba de caja negra

Se centra en encontrar errores.

La prueba de caja negra tiende a aplicarse durante las últimas etapas de la prueba.

**Unidad → Integración → Validación → Sistema**

## 8. Estrategia de Prueba para SW OO

Consejos para una EPSW efectiva

Ya no es posible probar una sola operación en aislamiento (la visión convencional de la prueba de unidad), sino más bien como parte de una clase.

- **Jerarquía de clases:** Superclase (Método X). X en subclases usa atributos de las subclases → El contexto en el que se usa X depende del contexto (superclase o subclases)

## Prueba de integración

Hay diferentes formas:

### 1. Prueba basada en hebra

- a. Integra el conjunto de clases requeridas para responder a una entrada o evento para el sistema.

### 2. Prueba basada en uso

- a. Clases independientes.
- b. Clases dependientes de las anteriores.
- c. Hasta la construcción del sistema.

### 3. Pruebas de grupo

- a. Descubrir errores en las colaboraciones entre clases.
- b. Se usan controladores y Stubs para poder realizarlas.

## 9. Estrategia de Prueba para Webapps

### Fundamentos

- El modelo de contenido para la Webapp se revisa para descubrir errores
- El modelo de interfaz se revisa para garantizar que todos los casos de uso pueden adecuarse
- El modelo de diseño para la Webapp se revisa para descubrir errores de navegación
- La interfaz de usuario se prueba para descubrir errores en los mecanismos de presentación (responsive)
- A cada componente funcional se le aplica una prueba de unidad
- Se prueba la navegación a lo largo de toda la arquitectura a Webappse implementa en varias configuraciones ambientales diferentes y se prueba en su compatibilidad con cada configuración.
- Las pruebas de seguridad se realizan con la intención de explotar vulnerabilidades en la Webapp dentro de su ambiente.
- Se realizan pruebas de rendimiento.
- La Webappse prueba mediante una población de usuarios finales

## 10. Pruebas de Validación

Comienzan tras las pruebas de integración

- El software está completamente ensamblado como un paquete y los errores de interfaz se descubrieron y corrigieron.
- Desaparece la distinción entre paradigmas de programación (imperativo, OO, Webapps, etc.).
- Las pruebas se enfocan en las acciones visibles para el usuario y las salidas del sistema reconocibles por el usuario que demuestran conformidad con los requerimientos establecidos por el cliente.

### Tipos de prueba de validación

#### • Prueba Alfa

- En el sitio del desarrollador por un grupo representativo de usuarios finales en un ambiente controlado.
- El desarrollador está presente.

#### • Prueba Beta

- Se realiza en uno o más sitios del usuario final.
- El desarrollador no está presente.

#### • Prueba de aceptación

- Tipo de prueba beta en la que se contrata a un cliente(s) para probar el SW antes de liberarlo. Puede ser muy formal y abarcar muchos días.

## 11. Pruebas del Sistema

Serie de pruebas cuyo propósito principal es ejercitar por completo el sistema SW-HW.  
Tipos de pruebas de sistema:

- **Pruebas de recuperación**

- Forzar al software a fallar en varias formas y verificar que la recuperación se realice de manera adecuada.
  - **Por restauración automática:** se evalúa el reinicio, los mecanismos de puntos de verificación, la recuperación de datos y la reanudación para correcciones
  - **Por intervención humana.** Se calcula el tiempo medio hasta la restauración. se evalúa el tiempo medio de reparación (TMR) para determinar si está dentro de límites aceptables

- **Pruebas de seguridad**

- Verificar que los mecanismos de protección que se construyen en un sistema en realidad lo protegerán de cualquier penetración impropia.

- **Pruebas de esfuerzo**

- Ejecuta un sistema de forma que demanda recursos en cantidad, frecuencia o volumen anormales.

- **Pruebas de rendimiento**

- Sistemas en tiempo real y sistemas embebidos además de satisfacer los requerimientos del cliente tienen que soportar unos mínimos de rendimiento.

- **Pruebas de despliegue**

- El software debe ejecutarse en varias plataformas y bajo más de un entorno de sistema operativo.

## 12. El Proceso de Depuración

- La depuración tiene un objetivo dominante: encontrar y corregir la causa de un error o defecto de software.
- Establezca un límite, por ejemplo, dos horas, en la cantidad de tiempo que emplea al intentar depurar un problema por cuenta propia. Después de eso, pida ayuda.

- Existen 3 tipos:

- **Fuerza bruta:** Deja que el ordenador encuentre el error.
- **Backtracking:** Comenzar en el sitio donde se descubrió un síntoma, el código fuente se rastrea hacia atrás.
- **Eliminación de causas:** Se hace una lista de las posibles causas y se realizan pruebas para eliminar cada una.



## 13. Estrategia de Prueba para Software OO

De lo pequeño hacia lo grande’:

- **Pruebas de clase:** pruebas que ejercitan las operaciones de clase y que examinan si existen errores conforme una clase colabora con otras clases
- **Pruebas de integración:** las clases se integran para formar un subsistema, se aplican pruebas de hebra, de uso y de grupo, mediante enfoques basados en fallo para ejercitar por completo clases colaboradoras
- **Pruebas de validación:** finalmente, se usan casos de uso (desarrollados como parte del modelo de requerimientos)

**Estrategia:**

- **Detección en fase de análisis** → Evitará:
  - Subclases para alojar el atributo innecesario o las excepciones
  - Relaciones de clase incorrectas o extrañas
  - Caracterización incorrecta del comportamiento del sistema
- **Propagación a fase de diseño:**
  - Asignación inadecuada de la clase a algunas tareas
  - Procedimientos para las operaciones sobre el atributo extraño
  - Modelo de mensajería será incorrecto
- **¡Propagación a la fase de codificación!**

## 14. Modelos de Prueba AOO y DOO

Los modelos de análisis (AOO) y diseño (DOO) no pueden probarse de la manera convencional porque no pueden ejecutarse.

**Tipos de modelo de prueba:**

- **Pruebas basadas en escenario:** Capturar las tareas que el usuario tiene que realizar y luego aplicar estas y sus variantes como pruebas.
- **Prueba aleatoria:** Escogemos una clase aleatoria donde verificamos la ejecución/orden de las funciones.
- **Prueba de partición:**
  - Si es en base a **estado**, serán casos de pruebas que cambian el estado.
  - En base al **atributo**, verán los métodos que consultan el atributo y verifica su funcionalidad.
  - En base a **categoría**, se mira el tipo de operaciones que tenemos.

## 15. Estrategias de Pruebas OO

- **Prueba de Unidad:** Cada clase y cada instancia de una clase (objeto) encapsulan los atributos (datos) y las operaciones (métodos o servicios) que manipulan dichos datos
- **Pruebas de integración:** El software orientado a objetos no tiene una estructura de control jerárquica, las estrategias de integración tradicionales, descendente y ascendente no sirven
- **Pruebas de validación:** Acciones visibles para el usuario y en las salidas del sistema reconocibles por él mismo:
  - Casos de uso del modelo de requerimientos: Escenario que tiene una alta probabilidad de descubrir errores en los requerimientos de interacción de usuario
  - Pruebas de caja negra
  - Casos de prueba del modelo de comportamiento del objeto
  - Diagrama de flujo de evento creado como parte del AOO

## Ejercicios de Ejemplo del Tema 2

En una revisión del análisis de requerimientos se detectan los siguientes errores:

Componentes	Unidades	Errores mayores	Errores menores
Diagramas de estructura	3	2	5
Modelos de datos	4	5	10
Modelos arquitectónicos	2	5	6
Modelos de componentes	2	1	6
Casos de uso	4	2	8
<b>Total modelado</b>	<b>15</b>	<b>15</b>	<b>35</b>

Se pide calcular:

1. Tamaño del producto del trabajo (TPT) para el modelado
2. Densidad del error por modelo de componentes
3. Calcule las anteriores densidades de errores contabilizando sólo los errores menores
4. Calcule las anteriores densidades de errores contabilizando sólo los errores mayores
5. Considere la densidad del error por modelo de componentes como la densidad del error promedio, si para un nuevo proyecto se contabilizan 50 componentes, ¿cuál sería el número de posibles errores que esperaríamos?

Para determinar la eficacia del costo de las revisiones se rellena la siguiente tabla:

Componentes	Esfuerzo preparación y evaluación	Esfuerzo repetición TODOS erroresMenores	Esfuerzo repetición TODOS erroresMayores
Diagramas de estructura	3	2	15
Modelos de datos	4	5	20
Modelos arquitectónicos	2	5	16
Modelos de componentes	2	1	16
Casos de uso	4	2	18
<b>Total modelado</b>	<b>15</b>	<b>15</b>	<b>85</b>

1. Calcule el esfuerzo total de la revisión del modelado indicando la unidad de medida:
2. Calcular el esfuerzo de repetición por error menor de los casos de uso
3. Calcule el esfuerzo de repetición promedio para la parte del modelado
4. Utilice el esfuerzo de repetición promedio calculado anteriormente para calcular el esfuerzo ahorrado por error en la fase de diseño. Para ello, si el esfuerzo de repetición promedio es de 25 horas-hombre/error si el error no se solucionara en la fase de requerimientos y se solucionara en la fase de diseño.
5. ¿Cuánto sería el ahorro si no se propagaran 10 errores de la fase de análisis a la de diseño?

En una revisión del análisis de requerimientos se detectan los siguientes errores:

Componentes	Unidades	Errores mayores	Errores menores
Diagramas de estructura	3	2	5
Modelos de datos	4	5	7
Casos de uso	4	2	8

**Se pide calcular:**

1. Tamaño total del producto del trabajo (TPT) para el análisis de requerimientos
2. Densidad del error del análisis de requerimientos
3. Considere la densidad del error del análisis de requerimientos como la densidad del error promedio. Si para un nuevo proyecto se contabilizan un total de 20 unidades, ¿cuál sería el número de posibles errores que esperaríamos?

Para determinar la eficacia del costo de las revisiones se rellena la siguiente tabla:

Componentes	Esfuerzo preparación y evaluación	Esfuerzo repetición TODOS erroresMenores	Esfuerzo repetición TODOS erroresMayores
Diagramas de estructura	3	2	15
Modelos de datos	4	5	20
Casos de uso	4	2	18

1. Calcule el esfuerzo total de la revisión del análisis de requerimientos indicando la unidad de medida:
2. Calcule el esfuerzo de repetición promedio para el análisis de requerimientos
3. Utilice el esfuerzo de repetición promedio calculado anteriormente para el análisis de requerimientos para calcular el esfuerzo ahorrado por error en la fase de diseño. Para ello, considere que el esfuerzo de repetición promedio es de 125 horas-hombre si el error no se solucionara en la fase de análisis de requerimientos y se solucionara en la fase de diseño.
4. ¿Cuánto sería el ahorro si no se propagaran 10 errores de la fase de análisis de requerimientos a la de diseño?

# Tema 3

Un programa tarda una media de 60 minutos en dar un error y se tarda una media de 20 minutos en solucionar el error. Calcule el tiempo medio entre fallas (TMEF).

$$\text{TMEF} = \text{TMPF} + \text{TPMR} =$$

Calcule la disponibilidad del programa anterior

$$\text{Disponibilidad} = \text{TMPF} / (\text{TMPF} + \text{TPMR}) =$$

# Tema 4

Dado el siguiente procedimiento:

```
void main()
{
    int ch, i=0, iter=0;
    printf ("Program to find nodes at maximum distance");
    while (iter < 1000) {
        printf("Enter your choice : ");
        scanf("%d", &ch);
        switch (ch) {
            case 1:
                insert();
                break;
            case 2:
                max = 0;
                count = 0;
                maxdistance(root);
                for(i = 1; i < z; i++)
                {
                    max2 = max1[0];
                    if (max2 < max1[1])
                        max2 = max1[1];
                    else{
                        max2 -=1
                    }
                }
                printf("Maximum distance nodes \nNodesi \tDistance ");
                for(i = 0; i < z; i++)
                {
                    if (max2 == max1[1])
                        printf("\n %dt %d ",v[i].max2);
                }
            } //End switch
        iter++;
    } //End while
    printf ("output achieved!\n");
}
```



# Tema 5. Modelado y Verificación Formal

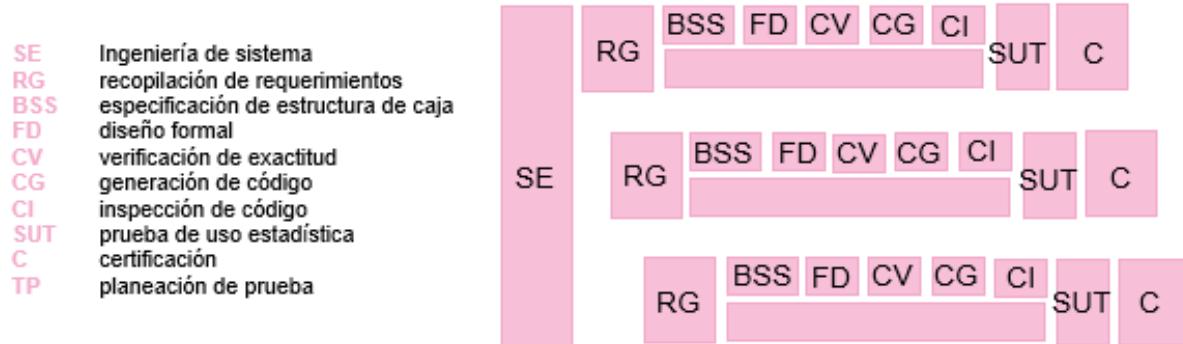
## 1. Métodos de Modelado y Verificación Formal

- Métodos de modelado y verificación formal se aplican **antes/durante el desarrollo de los modelos y código.**
  - A diferencia de las revisiones y pruebas que ocurren después.
  - Es una actividad llevada a cabo por los ingenieros de SW.
- Proporcionar un **enfoque basado en matemáticas** para programar el modelado y la capacidad de verificar que el modelo del producto SW es correcto.
  - Reducir el número de errores (bugs) mientras el SW se diseña y construye
- Los modelos de requisitos y de diseño se crean usando **notación especializada que es susceptible de verificación matemática.**
  - **Prueba de exactitud formal** se aplica al modelo de requisitos.
  - **Prueba de uso estadístico** ejerce los escenarios de uso para garantizar que se descubren y corrijen los errores en la funcionalidad del usuario.
- ¿Cuál es el producto final?
  - **Modelo formal** especializado de requisitos.
  - Se registran los **resultados de las pruebas de exactitud y de uso estadístico.**

## 2. Estrategia de Cuarto Limpio

- Enfoque para construir con exactitud el software conforme éste se desarrolla.
  - Incluye la **certificación de calidad estadística** de los incrementos de código conforme se desarrolla.
  - Se centra en la necesidad de probar la 'exactitud'.
- Versión especializada del modelo de software incremental de los métodos ágiles.
  - Pequeños equipos de software independientes desarrollan “**una tubería de incrementos de software**”.
  - Tras certificar el incremento se integra en el todo.

### Modelo de proceso de cuarto limpio



### Tareas

- **Planteamiento del incremento (SE).** Se desarrolla un plan de proyecto que adopte la estrategia incremental. Se crea la funcionalidad de cada incremento, su tamaño proyectado y un calendario de desarrollo de cuarto limpio. Debe tenerse especial cuidado para garantizar que los incrementos certificados se integrarán adecuadamente.
- **Recopilación de requisitos (RG).** Se desarrolla una descripción más detallada de los requisitos del cliente (para cada incremento)

- **Especificación de estructuras de cajas (BSS).** Se describe la funcionalidad de la aplicación mediante un método de especificación de estructuras de caja.
  - Las estructuras de caja se separan en cuanto a funcionalidad, datos y procedimientos en cada nivel de refinamiento.
- **Diseño formal (FD).** Una vez obtenido un diseño en base a estructuras de caja, se procede a definirlas mediante un proceso de diseño Top-Down:
  - Cada caja (caja negra) se refinan iterativamente para obtener cajas de estado y cajas claras.
- **Verificación de exactitud (CV).** El equipo de cuarto limpio realiza actividades de verificación de exactitud sobre el diseño y, luego, el código.
  - La verificación comienza con la estructura de caja (especificación) de nivel más alto y avanza hacia el detalle y el código de diseño.
  - El primer nivel de la verificación de exactitud ocurre al aplicar un conjunto de “preguntas de exactitud” sobre las cajas claras (diseño procedural). Si esto no demuestra que la especificación es correcta, se usan métodos mas formales (matemáticos) para la verificación.
- **Generación, inspección y verificación de código (CG, CI).** Las especificaciones de estructura de caja, representadas en un lenguaje especializado, **se traducen al lenguaje de programación adecuado**.
  - Las **revisiones técnicas** se usan entonces para asegurar la conformidad semántica del código y las estructuras de código, así como la exactitud sintáctica del código.
  - Después la verificación de exactitud para el código fuente.
- **Planificación de la prueba de uso estadístico (TP).**
  - Se planifica y diseña una suite de casos de prueba que ejercitan la distribución de probabilidad de uso.
  - Esta actividad de cuarto limpio se realiza en paralelo con la especificación, la verificación y la generación de código.
- **Prueba de uso estadístico (SUT).** La prueba exhaustiva del software es imposible. Siempre es necesario diseñar un numero finito de casos de prueba.
  - Las **técnicas de uso estadístico** ejecutan una serie de pruebas derivadas de una muestra estadística (la distribución de probabilidad) de todas las posibles ejecuciones de programa efectuadas por todos los usuarios de una población objetivo.
- **Certificación (C).** Una vez completadas la fases **CG, CI y SUT** (y corregidos todos los errores), el incremento se certifica como listo para su integración.

### 3. Estrategia de Cuarto Limpio

El enfoque de modelado en la ingeniería del software de cuarto limpio usa un método llamado **especificación de estructura de caja**.

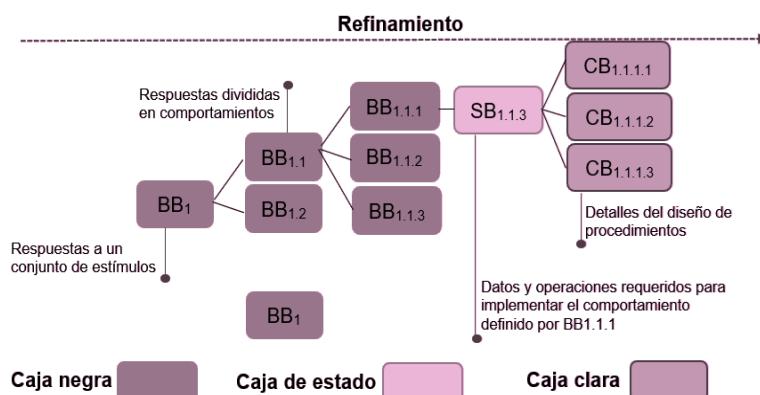
- Una “caja” encapsula el sistema (o algún aspecto del mismo) en algún nivel de detalle.
- A través de un proceso de elaboración o refinamiento por pasos, las cajas se refinan en una jerarquía **donde cada caja tiene transparencia referencial** (la funcionalidad encapsulada no depende de ninguna otra caja).

#### Tipos de caja

- **Caja negra:** comportamiento de un sistema o de una parte de un sistema. El sistema (o su parte) responde a estímulos específicos (eventos) al aplicar un conjunto de reglas de transición que mapean el estímulo en una respuesta.
- **Caja de estado.** Define las estructuras de datos y las operaciones de estado en base a entradas (estímulo) y salidas (respuesta). (Ejemplo: Vector.Add(Elemento) // Vector incrementado en 1)

- **Caja clara.** Diseño procedural para la caja de estado. (Ejemplo: Diseño de cómo se implementa el método Add.)

## Refinamiento de estructuras de caja



Conforme ocurre cada uno de estos pasos de refinamiento, también ocurre la verificación de la exactitud.

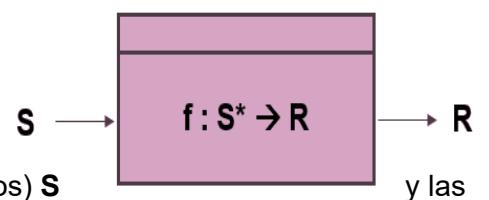
- Las especificaciones de caja de estado se verifican para asegurar que cada una se define con el comportamiento especificado por la caja negra padre.
- Las especificaciones de caja clara se verifican contra la caja de estado padre

## Especificación de Caja Negra

Una especificación de caja negra describe una abstracción, estímulos y respuesta, usando la siguiente notación:

La función **f** se aplica a una secuencia **S\*** de entradas (estímulos) **S** y las transforma en una salida (respuesta) **R**.

- Para componentes de software simples, **f** puede ser una función matemática, pero, en general, **f** se describe en lenguaje natural o un lenguaje de especificación formal.
- Pueden haber **jerarquías de uso**: cajas de nivel inferior heredan las propiedades de las cajas superiores en la jerarquía de cajas

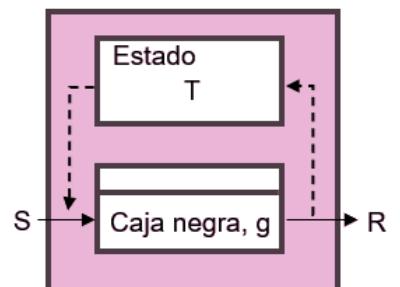


## Especificación de Caja de Estado

**“Simple generalización de una máquina de estado”**

**Estado = modo observable de comportamiento del sistema**

- Durante el procesamiento, un sistema responde a los eventos (estímulos), haciendo una transición desde el estado actual hasta algún estado nuevo.
- Conforme se realiza la transición, puede ocurrir una acción.



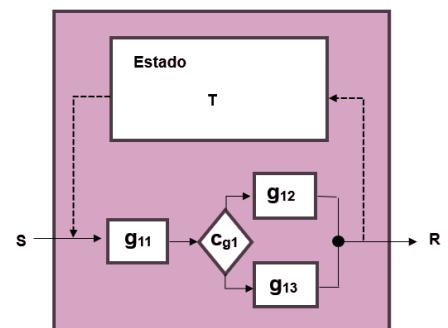
La caja de estado usa una **abstracción de datos** para determinar la transición al siguiente estado y la acción (respuesta) que ocurrirá como consecuencia de la transición.

La caja de estado **incorpora una caja negra g**.

El estímulo **S** que es entrada a la caja negra llega desde alguna fuente externa y desde un conjunto de estados internos del sistema **T**.

## Especificación de Caja Clara

La caja negra **g** dentro de la caja de estado se sustituye con los constructos de programación estructurada que implementan **g** (en el ejemplo de la imagen se trata de un constructo secuencia que incorpora un condicional)



## 4. Prueba de Exactitud Formal

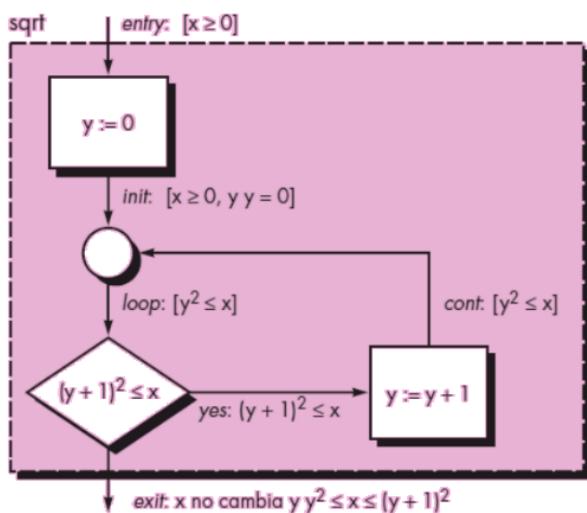
Cada especificación de **Caja Clara** representa el diseño de un procedimiento (subfunción) requerido para lograr una transición de caja de estado.

- Dentro de la caja clara se usan constructos de programación estructurada y refinamiento por pasos para representar detalles procedurales.
- Una función de programa **f** se refina en una **secuencia** de subfunciones **g** y **h**.
- Se refinan en **constructos condicionales** (por ejemplo, if-then-else y do-while).
- El refinamiento continúa hasta que hay suficiente detalle procedural para crear el componente en cuestión (salto al código casi inmediato).

### Preguntas de Exactitud

- A los constructos de programación se añaden **condiciones de exactitud genéricas**.
- Si una función **f** se expande en una secuencia **g** y **h** (**f: h → g**), la condición de exactitud para toda entrada a **f** es:
  1. ¿**g** seguida de **h** hace **f**?  
→ **1 Condición a verificar**
- Cuando una función **p** se refina en una condicional de la forma “**p: if c then q, else r**”, la condición de exactitud para toda entrada a **p** es:
  1. Siempre que la condición **c** es verdadera, ¿**q** hace **p**?
  2. Siempre que **c** es falsa, ¿**r** hace **p**?  
→ **2 condiciones a verificar**
- La función **m** se refina como un bucle (**m: while (c) {n}**), las condiciones de exactitud para toda entrada a **m** son:
  1. ¿Está garantizada la finalización? Es decir, ¿el bucle es finito?
  2. Siempre que **c** es verdadera, ¿**n** seguida por **m** hace **m**?
  3. Siempre que **c** es falsa, ¿saltar el bucle todavía hace **m**?  
→ **3 condiciones a verificar**

## EJEMPLOS



Para probar que **el diseño es correcto**, es necesario probar que las condiciones **init**, **loop**, **cont**, **yes** y **exit**, son verdaderas en todos los casos:

1. condición para las secuencias
2. condiciones para if-then-else
3. para bucles

La condición **init** : [ $x \geq 0$  e  $y = 0$ ]:

- Como no tiene sentido un valor negativo para la raíz cuadrada, se satisface  $x \geq 0$
- En el diagrama de flujo, el enunciado inmediatamente anterior a la condición **init** establece  $y := 0 \rightarrow$  OK.  
→ **init** es verdadera

Para la condición **loop**: [ $y^2 \leq x$ ]:

- Directamente de **init** → OK
- Por medio del flujo de control:  
(condición **cont**) → OK  
→ **loop** es verdadera

Para la condición **cont**: [ $y^2 \leq x$ ]:

- Se encuentra cuando el valor de  $y$  se aumenta en 1
- La ruta de flujo de control que conduce a **cont** puede invocarse solo si la condición **yes** es también es verdadera  
Si  $(y + 1)^2 \leq x \rightarrow y^2 \leq x$   
→ **cont** es verdadera

**1<sup>a</sup> condición del exit:**  $x$  no cambia

- Como no hay instrucciones/funciones que modifiquen  $x$   
→ 1<sup>a</sup> condición del **exit** es verdadera

**2<sup>a</sup> condición del exit:** [ $y^2 \leq x \leq (y + 1)^2$ ]

- Como la prueba  $(y + 1)^2 \leq x$  debe fallar para llegar a **exit** →  $(y + 1)^2 > x$ .
- La condición **loop** debe incluso ser verdadera →  $y^2 \leq x$   
→  $(y + 1)^2 > x \& y^2 \leq x \rightarrow$  **exit** es verdadera

La condición **yes**: [ $(y + 1)^2 \leq x$ ]:

- Se prueba en la lógica condicional y cuando se recorre el arco de izquierda a derecha  
→ condición **yes** es verdadera

Adicionalmente, debe asegurarse de que el **bucle es finito**.

- Un análisis de la condición **loop** indica que, dado que  $y$  se incrementa y  $x \geq 0$ , el bucle finalmente debe terminar  
→ **El bucle es finito**

## 5. Prueba de Uso Estadístico

- La estrategia y tácticas de las pruebas de cuarto limpio son fundamentalmente **diferentes a las de los enfoques de prueba convencionales**.
  - Los métodos convencionales derivan un conjunto de casos de prueba para descubrir **errores de diseño y codificación** (los requisitos ya están claros).
- La meta de la prueba de cuarto limpio **es validar los requisitos de software** al demostrar que una **muestra estadística** de casos de uso se **ejecuta exitosamente**.
- El comportamiento del programa visible al usuario se activa con entradas y eventos que con **frecuencia** son producidos por el usuario.
- En los sistemas complejos, el posible espectro de entrada y eventos (casos de uso) puede ser extremadamente amplio.
  - **PRUEBAS DE USO ESTADÍSTICO**: subconjunto de casos de uso verificará de manera adecuada el comportamiento del programa.
- Examinar el software de la **forma en la que los usuarios pretenden usarlo**.
- Los equipos de prueba de cuarto limpio (equipos de certificación) deben determinar una **distribución de probabilidad de uso para el software**.
- La especificación (**caja negra**) para cada incremento del software se analiza a fin de **definir un conjunto de estímulos** (entradas o eventos) que hacen que el software cambie su comportamiento

Ejemplo:

Estímulo del programa	Probabilidad	Intervalo
Armar/desarmar (AD)	50%	1-49
Establecer zona (EZ)	15%	50-63
Consulta (C)	15%	64-78
Prueba (P)	15%	79-94
Alarma de pánico	5%	95-99

Se generar una secuencia de casos de prueba de uso que se ajusten a la distribución de probabilidad de uso:

Números aleatorios entre 1 y 99.

En la creación de escenarios de uso y en una comprensión general del dominio de aplicación, a cada estímulo se le asigna una probabilidad de uso con base a **entrevistas con usuarios potenciales**.

Para cada **conjunto de estímulos** se generan **casos de prueba de acuerdo con la distribución de probabilidad de uso**.

13-94-22-24-45-56  
81-19-31-69-45-9  
38-21-52-84-86-4

→ AD-T-AD-AD-AD-ZS  
T-AD-AD-AD-Q-AD-AD  
AD-AD-ZS-T-T-AD

El equipo de prueba los **ejecuta** y verifica el comportamiento del software **contrastándolo con la especificación para el sistema**.

- La **temporización de las pruebas** se registra de modo que puedan determinarse los intervalos de tiempo. Al usar intervalos de tiempo, el equipo de certificación puede calcular el **tiempo medio hasta el fallo (TMHF)**.
- Si una larga secuencia de pruebas se realiza sin fallos, el TMHF es alto y la **confiabilidad del software puede suponerse alta**.

## 6. Certificación

Pasos para la certificación del incremento

- Dentro del contexto del enfoque de ingeniería del software de cuarto limpio, la certificación implica que la **confiabilidad** (medida por el tiempo medio hasta el fallo o TMHF) puede especificarse para cada componente.
- **El enfoque de certificación involucra cinco pasos:**
  1. Se crean escenarios de uso
  2. Se especifica un perfil de uso
  3. Se generan casos de prueba a partir del perfil
  4. Las pruebas se ejecutan y los datos de fallo se registran y analizan
  5. **Se calcula la confiabilidad (TMHF) y se certifica**

Modelos para el cálculo de la confiabilidad (TMHF)

- **Modelo de muestreo.** La prueba de software ejecuta  $m$  casos de prueba aleatorios ( $m$  ha de ser lo suficientemente grande como para dar un valor de confiabilidad válido) y se certifica si no ocurren fallos o un número específico de ellos.
- **Modelo de componente.** Se certifica un sistema compuesto de  $n$  componentes. Probabilidad de que el componente  $i$ -ésimo falle.
- **Modelo de certificación.** La confiabilidad global del sistema se proyecta y se certifica. El equipo de certificación tiene la información requerida para entregar el software que tenga un TMHF certificado

## 7. Métodos Formales

Las facilidades descriptivas de la teoría de conjuntos y la notación lógica permiten un enunciado claro de los requerimientos.

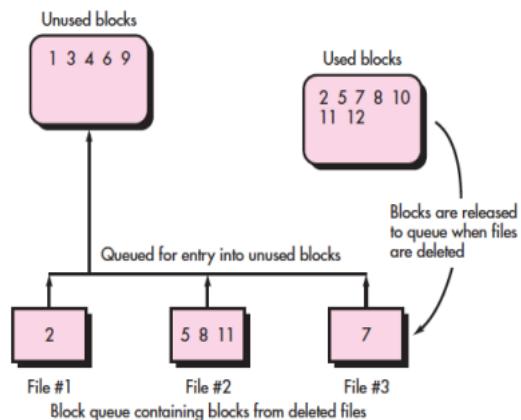
Los métodos formales nos permitirán definir sin ambigüedad:

- **Invariante de datos:** restricciones que siempre tienen que cumplir las estructuras de datos de la aplicación (es decir, el Estado) independientemente de las operaciones que se hagan sobre ellas.
- **Precondición:** circunstancias en las que es válida una operación para poder ejecutarse.
- **Poscondición:** lo que se garantiza que es verdadero tras ejecutar la operación.

### Definición

“Los métodos formales son técnicas con base matemática para describir las propiedades del sistema. Tales métodos formales proporcionan marcos conceptuales dentro de los cuales las personas pueden especificar, desarrollar y verificar los sistemas en forma sistemática”

### Ejemplo 2: Manejador de Bloques en SSFF básico



Un fichero está compuesto de una serie de bloques

Para **crear un fichero**, el manejador:

1. Extrae tantos bloques como sean necesarios de una lista de bloques no usados (Unused blocks).
2. Los añade a una lista de bloques usados por el fichero (Used blocks).

Para **eliminar un fichero**, el manejador: Almacena los bloques del fichero en una cola (Queue) para que, cuando sean eliminados de disco, se almacenen en la lista de bloques no usados (Unused blocks).

**Estado:** colección de bloques libres, la colección de bloques usados y la cola de bloques a reciclar (CBR)

**Invariante de datos:**

- Ningún bloque se marcará como no utilizado y usado al mismo tiempo.
- Todos los bloques de la CBR serán un subconjunto de la colección de los bloques actualmente utilizados.
- Ningún elemento de la CBR tendrá bloques duplicados.
- La colección de bloques utilizados y bloques que no se usan será la colección total de bloques del sistema.

**Operaciones sobre la CBR:** add(bloques), remove(bloques).

• **Precondiciones:**

- **add** (los bloques que se van a agregar a la CBR estarán en la colección de bloques usados);
- **remove** (la CBR debe tener al menos un item);

• **Postcondiciones:**

- **add** (la colección de bloques añadidos se encuentra en la CBR);
- **remove** (los bloques se añaden a la colección de bloques no usados);

### Ejemplo 1: Tabla de Símbolos

↑	<table border="1" style="width: 100%; border-collapse: collapse;"><tr><td>1. Wilson</td></tr><tr><td>2. Simpson</td></tr><tr><td>3. Abel</td></tr><tr><td>4. Fernández</td></tr><tr><td>5.</td></tr><tr><td>6.</td></tr><tr><td>7.</td></tr><tr><td>8.</td></tr><tr><td>9.</td></tr><tr><td>10.</td></tr></table>	1. Wilson	2. Simpson	3. Abel	4. Fernández	5.	6.	7.	8.	9.	10.
1. Wilson											
2. Simpson											
3. Abel											
4. Fernández											
5.											
6.											
7.											
8.											
9.											
10.											

**Estado:** Tabla de Símbolos

**Invariantes de datos:**

1. No puede haber ítems duplicados.
2. Tamaño máximo es MaxIds nombres.

**Operación:** Añadir (Nombre)

- **Precondición:** Para poder añadir un nuevo Nombre a la Tabla, la tabla tendrá menos de MaxIds nombres.
- **Postcondición:** La Tabla de Símbolos aumentará en un nuevo elemento que será el nuevo nombre añadido.

### Ejemplo 3: Ejercicio de clase

En grupos de dos/tres alumnos especificar un marco conceptual basado en metodología formal para una función que describa una operación sobre una estructura de datos que el grupo considere

Definir el estado, invariante de datos, al menos una operación con su precondición y postcondición

## 8. Lenguajes de Especificación Formal

Conjunto de relaciones que definen las reglas que indican qué objetos satisfacen adecuadamente la especificación.

- **Sintaxis:** define la notación específica con la que se representa la especificación.  
Notación estándar de la teoría de conjuntos y cálculo de predicados.
- **Semántica:** ayuda a definir un ‘universo de objetos’ que se usarán para describir el sistema.  
Cómo representa el lenguaje los requerimientos del sistema.

## 9. El lenguaje OCL

Notación formal desarrollada de modo que los usuarios de UML puedan agregar más precisión a sus especificaciones

**Diagramas UML:** clase, estado o actividad.

- Se agregan **expresiones OCL y hechos de estado** acerca de elementos de los diagramas: invariantes de datos, precondiciones y postcondiciones.
- Cualquier implementación derivada del modelo debe asegurar que cada una de las restricciones siempre sigue siendo verdadera.

### Notación OCL

x.y	Obtiene la propiedad y del objeto x. Una propiedad puede ser un atributo, el conjunto de objetos al final de una asociación, el resultado de evaluar una operación y otras cosas, dependiendo del tipo de diagrama UML. Si x es un Conjunto, entonces y se aplica a cada elemento de x; los resultados se recopilan en un nuevo Conjunto.
C->f()	Aplica la operación f interna de OCL a la Colección c en sí (en oposición a cada uno de los objetos en c). A continuación, se mencionan ejemplos de operaciones internas.

<b>And, or, =, &lt;&gt;</b>	And lógica, or lógica, igual, no es igual.
<b>p implica q</b>	Verdadero si q es verdadero o p es falso
<b>Muestra de operaciones sobre colecciones (incluidos conjuntos y secuencias)</b>	
<b>C-&gt;size()</b>	El número de elementos en la Colección c.
<b>C-&gt;isEmpty()</b>	Verdadero si c no tiene elementos, falso de otro modo
<b>c1—&gt;includesAll(c2)</b>	Verdadero si cada elemento de c2 se encuentra en c1.
<b>c1—&gt;excludesAll(c2)</b>	Verdadero si ningún elemento de c2 se encuentra en c1.
<b>C-&gt;forAll(elem   boolexpr)</b>	Verdadero si boolexpr es verdadera cuando se aplica a cada elemento de c. Conforme se evalúa un elemento, se enlaza a la variable elem, que puede usarse en boolexpr. Esto implementa cuantificación universal, que se estudió anteriormente.
<b>C-&gt;forAll(elem1, elem2   boolexpr)</b>	Igual que el anterior, excepto que boolexpr se evalúa para cada posible par de elementos tomados de c, incluidos casos donde el par tiene el mismo elemento.
<b>C-&gt;isUnique(elem   expr)</b>	Verdadero si expr evalua un valor diferente cuando se aplica a cada elemento de c
<b>Muestra de operaciones específicas para conjuntos</b>	
<b>s1 -&gt; intersection(s2)</b>	El conjunto de aquellos elementos que se encuentran en s1 y también en s2
<b>s1 -&gt; unión(s2)</b>	El conjunto de aquellos elementos que se encuentran en s1 o en s2
<b>s1 -&gt; excluding(x)</b>	El conjunto s1 con la omisión del objeto x
<b>Muestra de operación específica a secuencias</b>	
<b>Seq -&gt; first()</b>	El objeto que es el primer elemento en la secuencia seq.

- Una expresión OCL involucra **operadores** que operan sobre **objetos**.
- Los objetos pueden ser **instancias de la clase Collection OCL**:
  - **Set** (conjunto) y **Sequence** (secuencia) son dos subclases.
- El **resultado** de una **expresión completa** siempre debe ser booleana (V o F).
- El objeto **self** es el elemento del diagrama UML en cuyo contexto se evaluará la expresión OCL.
- El símbolo “.” del objeto self permite obtener otros objetos:
  - Si **self** es la clase C, con atributo a, entonces **self.a** evalúa al objeto almacenado en a.
  - Si C tiene una asociación uno a muchos llamada assoc con otra clase D, entonces **self.assoc** evalúa un Set cuyos elementos son del tipo D.
  - Si D tiene el atributo b, entonces la expresión **self.assoc.b** evalúa al conjunto de todos los b que pertenecen a todos los D.

## Ejemplo: Manejador de Bloques en SSFF básico

Ningún bloque se marcará como no utilizado y usado al mismo tiempo

```
context BlockHandler inv:  
    (self.used->intersection(self.free)) -> isEmpty()
```

Palabra clave **context**

- Elemento del diagrama UML que restringe la expresión

Palabra clave **self**

- Refiere a la instancia de BlockHandler
- Se puede omitir

Todos los conjuntos de bloques que se están en la cola (blockQueue) serán subconjuntos de la colección de los bloques actualmente utilizados

```
context BlockHandler inv:  
    blockQueue->forAll(aBlockSet | used->includesAll(aBlockSet ))
```

Ningún elemento de la cola (blockQueue) contendrá el mismo número de bloque

```
context BlockHandler inv:  
    blockQueue->forAll(blockSet1, blockSet2 |  
        blockSet1 <> blockSet2 implies  
        blockSet1.elements.number->excludesAll(blockSet2.elements.number))
```

La colección de bloques utilizados y bloques que no se utilizan será la colección total de bloques que constituyen los archivos

```
context BlockHandler inv:  
    allBlocks = used->union(free)
```

La colección de bloques no utilizados no tendrá números de bloque duplicados

```
context BlockHandler inv:  
    free->isUnique(aBlock | aBlock.number)
```

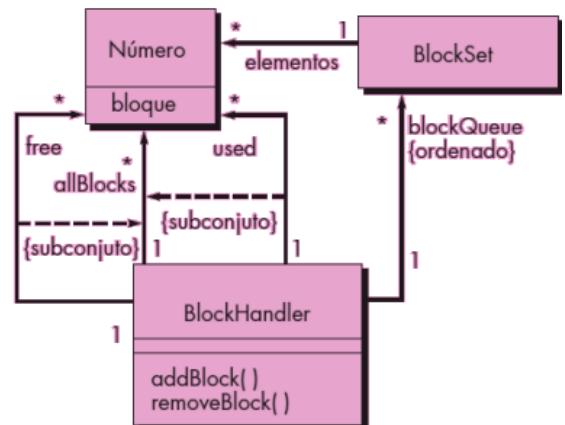
La colección de bloques utilizados no tendrá números de bloque duplicados

```
context BlockHandler inv:  
    used->isUnique(aBlock | aBlock.number)
```

**Operación removeBlocks():**

- **Precondición:** La blockQueue no estará vacía.
- **Postcondición:** eliminar el bloque a eliminar del conjunto de usados, añadirlo al de bloques libres y eliminarlo de la blockQueue

```
context BlockHandler::removeBlocks()  
    pre: blockQueue->size() > 0  
    post: used = used@pre-blockQueue@pre->first() and  
          free = free@pre->union(blockQueue@pre->first()) and  
          blockQueue = blockQueue@pre->excluding(blockQueue@pre->first())
```



## Ejemplo: Aplicación de compra de vehículos

El propietario de un vehículo tiene que tener al menos 18 años

- **context Vehicle inv:**  
**self.owner.age >= 18**

Ninguna persona tiene más de tres vehículos

- **context Person inv:**  
**self.fleet → size() <= 3**

Todos los coches de las personas son negros

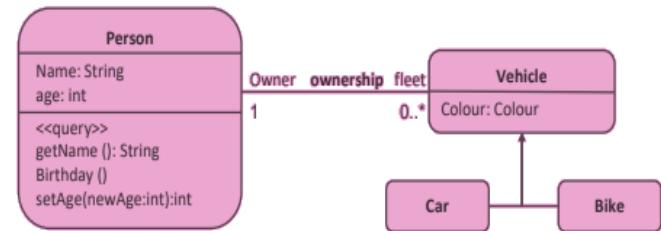
- **context Person inv:**  
**self.fleet → forAll(v | v.colour = black)**

Para operación setAge, precondición: la edad tiene que ser mayor que cero

- **context Person::setAge(newAge:int)**  
**pre: newAge > 0**

Para operación birthday, postcondición: la edad se incrementa en uno

- **context Person::birthday()**  
**post: self.age = self.age@pre + 1**





# Tema 6. Gestión de la Configuración del SW

## 1. La Gestión de la Configuración del Software

- El desarrollo del software implica que haya cambios que se tienen que llevar a cabo
  - Los cambios descontrolados pueden comprometer la calidad del SW

### Sirve Para

- Coordinar el desarrollo de SW para minimizar la confusión entre los miembros de un equipo de SW.  
Situaciones que fomentan la confusión:
  - Los cambios no se analizan antes de que se realicen
  - No se registran los cambios antes de que se implanten
  - No se informan a quienes tienen necesidad de conocerlos
  - No se controlan para mejorar la calidad y reducir el error
- **La GCS permite determinar qué falló y quién hizo el cambio**

### Objetivos

- **Identificar** los productos de trabajo que pueden cambiar
- Definir mecanismos para administrar distintas **versiones** de los productos de trabajo y **controlar** los cambios
- Garantizar que el cambio se **implementó** de manera adecuada
- **Auditar** los cambios e informar a todos el personal de SW que pueda estar interesado en cada cambio

### Algunas fuentes de cambio del SW

- Nuevas reglas empresariales o de mercado que cambien los requisitos del producto, modificación de los datos a producir, funcionalidad ofrecida o servicios ofrecidos ú
- La reorganización o crecimiento/reducción de la empresa produce cambios en las prioridades proyectadas o en la estructura del equipo de ingeniería de software
- Restricciones presupuestales o de calendario causan una redefinición del sistema o del producto

## 2. El Escenario GCS

### Tareas del GCS vs Roles del equipo de Software

- **GERENTE DE PROYECTO** que está a cargo de un grupo de software. Garantiza que el producto se desarrolle dentro de cierto marco temporal
  - Monitoriza el progreso del desarrollo e identifica y reacciona ante los problemas
  - Genera y analiza informes acerca del estado del sistema de software y al realizar revisiones al sistema
- **La GCS es un mecanismo de auditoría**
- **GERENTE DEL CAMBIO** que es responsable de los procedimientos y políticas de gestión de la configuración (cambio)
  - Introduce mecanismos para realizar peticiones oficiales de cambios, evaluarlos y autorizarlos. Ejemplo: Herramienta GLPi
  - Difunde la lista de tareas de cambio para los ingenieros de SW
  - Recopila estadísticas acerca de los componentes que hay en el sistema de software para determinar componentes del sistema problemáticos
- **La GCS es un mecanismo de control, rastreo y generación de políticas**

- **INGENIEROS DE SOFTWARE** encargados de desarrollar y mantener el producto de software
  - No deben interferir unos con otros en la creación y prueba del código y deben intentar comunicarse y coordinarse de manera eficiente
  - Se conserva una historia de la evolución de todos los componentes del sistema
  - Una bitácora con las razones de los cambios y un registro de los cambios
  - En cierto punto, el código se convierte en una línea de referencia (a.k.a., baseline) desde la cual continúan mayores desarrollos

**La GCS es un mecanismo de control de cambio, construcción y acceso**

- **CLIENTE** que usa el producto de software
  - Sigue procedimientos formales para solicitar cambios e indicar errores en el producto. ü
  - Ejemplo: soporte.ucam.edu

**La GCS es un camino para garantizar la calidad**

### 3. Terminología de la GCS

#### Ítem de Configuración

- Información que se crea como parte del proceso de ingeniería del software
- Un Ítem de Configuración (ICS) es todo o parte de un producto de trabajo:
  - Una sola sección de una gran especificación, un caso de prueba de una suite de pruebas.
  - Un documento, una suite de casos de prueba o un componente de programa.
- También las **herramientas de SW** son un ICS:
  - Versiones específicas de editores, compiladores, navegadores y otras herramientas automatizadas.



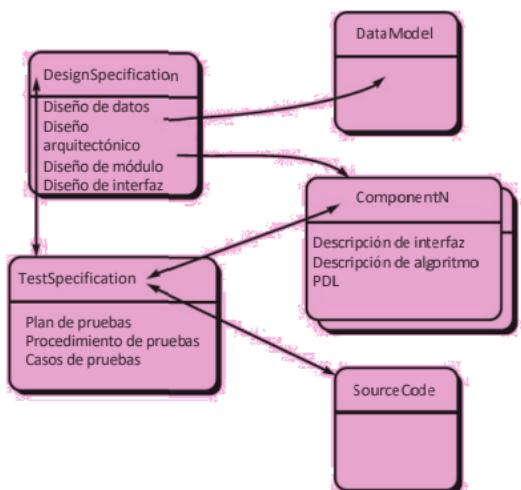
#### Línea de Referencia o Baseline

- Una **Línea de referencia (LR)** se marca al entregar uno o más ítems de Configuración (ICS) aprobados tras una revisión técnica
- Ejemplo de los pasos para declarar un modelo de diseño como LR:
  1. Los elementos de un modelo de diseño se documentaron y revisaron
  2. Se encontraron y corrigieron errores
  3. Una vez que todas las partes del modelo se revisaron, corrigieron y luego aprobaron, el modelo de diseño se convierte en línea de referencia
  4. Los cambios adicionales a la arquitectura del programa (documentada en el modelo de diseño) pueden realizarse solo después de que cada uno se evalúa y aprueba



- **Progresión de eventos que conducen a una línea de referencia:** desde el espacio de trabajo del Ingeniero de SW al Repositorio GCS.
- **Modificación a un ICS que es una línea de referencia:** desde Repositorio GCS a espacio de trabajo del Ingeniero de SW siguiendo controles GCS.

## Objeto de Configuración



- Los ICS se organizan para formar OCS que puedan catalogarse con un solo nombre en el repositorio GCS.
- Un OCS tiene un nombre y atributos, y está 'conectado' con otros objetos mediante relaciones



**Relación de composición** = Son parte de.

**Interrelación**: Si se modifica un OC los otros OCS interrelacionados se verán afectados.

## 4. El Repositorio GCS

Inicialmente la gestión de los ICS era ineficiente porque se almacenaban sin soporte informatizado (papel o en la mente de los programadores).

- Dificultad para encontrar un IC cuando se necesitaba.
- Difícil determinar qué ICs cambiaban, cuándo y por quién.
- Describir relaciones detalladas y complejas entre los ítems de configuración era extremadamente tedioso.

**Hoy en día los ICS se mantienen en una base de datos del proyecto, o repositorio GCS.**

El repositorio GCS es el conjunto de mecanismos y estructuras de datos que permiten a un equipo de software administrar el cambio de manera efectiva

- Proporciona las funciones de una BBDD: integridad y posibilidad de compartir datos
- Centro para la integración de herramientas de software

El repositorio se define como un **metamodelo**

- Determina cómo se almacena la información en el repositorio
- Cómo pueden acceder las herramientas a los datos y cómo pueden verlos los ingenieros de software
- Cómo puede mantenerse la seguridad y la integridad de los datos
- Cómo puede extenderse el modelo existente para alojar nuevas necesidades

## Funcionalidad del repositorio GCS

- **RASTREO DE DEPENDENCIA Y GESTIÓN DEL CAMBIO**: El repositorio administra una amplia variedad de relaciones entre los elementos de datos almacenados en él.
  - Relaciones entre entidades y procesos empresariales, entre las partes de un diseño de aplicación, entre componentes de diseño y la arquitectura de información de la empresa, entre elementos de diseño y entregables, etc.
  - La capacidad de seguir la pista de todas estas relaciones es vital para la integridad de la información almacenada en el repositorio y para la generación de entregables con base en él.
  - Si un diagrama de clase UML se modifica, el repositorio puede detectar si clases relacionadas, descripciones de interfaz y componentes de código también requieren modificación y si pueden llevar los ICS afectados a la atención del desarrollador

- **VERSIONES:** El repositorio debe guardar todas las versiones de los productos de trabajo resultantes durante el proceso de software.
  - Permite la administración efectiva de los productos terminados y poder regresar a versiones anteriores durante las pruebas y la depuración.
  - Debe controlar una amplia variedad de tipos de objeto, incluidos texto, gráficos, mapas de bits, documentos complejos y objetos únicos, como definiciones de pantalla y reportes, archivos objeto, datos de prueba y resultados.
- **RASTREO DE REQUISITOS:**
  - Capacidad de rastrear todos los productos de trabajo que resulten de una especificación de requisitos determinada (rastreo hacia adelante).
  - También la capacidad de identificar qué requisito genera algún producto de trabajo determinado (rastreo hacia atrás).
- **ENsayos de auditoría:**
  - Información adicional acerca de cuándo, por qué y quién realiza los cambios.
  - Siempre que se modifique un producto de trabajo, se mostrará al desarrollador (o a la herramienta que se utilice), la entrada de la información de auditoría para conocer la razón para el cambio.

## 5. Herramientas para l GCS



### **Microsoft Project:**

- Diagrama de actividades del proyecto
- Informe de cambios
- Auditoría del desarrollo del proyecto

### **Control de Versiones:** Subversion/Git/Mercurial...

**Otras:** SmartBear, Spectrum SCM, Plastic SCM, IBM Rational ClearCase, AccuRev SCM, Microsoft Team Foundation, Server, Microsoft Visual Studio.

# Tema 7. Métricas de Producto

## 1. Métricas de Producto

Las métricas de producto ayudan a los ingenieros de software a obtener comprensión acerca del diseño y la construcción del software que elaboran.

1. Enfocarse en atributos mensurables específicos de los productos de trabajo de la ingeniería del software.

Las métricas de producto proporcionan una base desde donde el análisis, el diseño, la **codificación** y las **pruebas** pueden realizarse de manera más objetiva y valorarse de modo más cuantitativo.

¿Quién la hace? Los **ingenieros de software** usan métricas de proyecto para auxiliarse en la construcción de software de mayor calidad.

Pasos para las métricas de producto:

1. Derivar las mediciones y métricas del software que sean adecuadas para la presentación del software que se está construyendo
2. Recolectan los datos requeridos para derivar las métricas formuladas
3. Las métricas adecuadas se analizan con base en políticas y reglas preestablecidas y en datos anteriores
4. Los resultados del análisis se interpretan para obtener comprensión acerca de la calidad del software
5. Los resultados de la interpretación conducen a modificación de requerimientos y modelos de diseño, código fuente o casos de prueba

Terminología

- **Medida:** indicio cuantitativo de la extensión, cantidad, dimensión, capacidad o tamaño de algún atributo de un producto. Medición: acto de determinar una medida.
  - Ej.: Número de errores dentro de un diagrama UML, Esfuerzo de Reparación, Esfuerzo de Revisión, TPT, etc
- **Métrica:** ‘una medida cuantitativa del grado en el que un sistema, componente o proceso posee un atributo determinado’
  - Relaciona en alguna forma las medidas individuales:
  - Ej.: Errores/ Revisión; Tasa de Fallos ( $TF = \text{Fallos}/\text{Ejecuciones}$ ), etc.
- **Indicador:** métrica o combinación de métricas que proporcionan comprensión acerca del proceso de software, el proyecto de software o el producto en sí.
  - Si la TF de las pruebas > 0.2 se revisará el producto SW

Principios de Medición

1. **Formulación.** La derivación de medidas y métricas de software apropiadas para la representación del software que se está construyendo.
2. **Recolección.** Mecanismo que se usa para acumular datos requeridos para derivar las métricas formuladas.
3. **Análisis.** El cálculo de métricas y la aplicación de herramientas matemáticas.
4. **Interpretación.** Evaluación de las métricas resultantes para comprender la calidad de la representación.
5. **Retroalimentación.** Recomendaciones derivadas de la interpretación de las métricas del producto, transmitidas al equipo de software.

- Una métrica debe tener **propiedades matemáticas deseables**: el valor de la métrica debe estar en un rango significativo.
  - Ej.: de 0 a 1, donde 0 realmente significa ausencia, 1 indica el valor máximo y 0.5 representa el ‘punto medio’.
- Cuando una métrica representa una característica de software que aumenta cuando ocurren rasgos positivos o que disminuye cuando se encuentran rasgos indeseables, el valor de la métrica debe aumentar o disminuir en la misma forma.
  - Ej.: Confiabilidad.
- Cada métrica debe **validarse de manera empírica** en una gran variedad de contextos antes de publicarse o utilizarse para tomar decisiones.
  - Una métrica debe medir el factor de interés, independientemente de otros factores.
  - Debe ‘escalar’ a sistemas más grandes y funcionar en varios lenguajes de programación y dominios de sistema.

### Atributos deseables para las métricas

- **Simple y calculable**. Debe ser relativamente fácil aprender como derivar la métrica y su cálculo no debe demandar esfuerzo o tiempo excesivo.
- **Empírica e intuitivamente convincente**. Debe satisfacer las nociones intuitivas del ingeniero acerca del atributo de producto que se elabora (por ejemplo, una métrica que mide la cohesión del módulo debe aumentar en valor conforme aumenta el nivel de cohesión).
- **Congruente y objetiva**. Siempre debe producir resultados que no tengan ambigüedades. Una tercera parte independiente debe poder derivar el mismo valor de métrica usando la misma información acerca del software.
- **Constante en su uso de unidades y dimensiones**. El cálculo matemático de la métrica debe usar medidas que no conduzcan a combinaciones extrañas de unidades.
- **Independiente del lenguaje de programación**. Debe basarse en el modelo de requerimientos, el modelo de diseño o la estructura del programa en sí. No debe depender de los caprichos de la sintaxis o de la semántica del lenguaje de programación. Un mecanismo efectivo para retroalimentación de alta calidad. Debe proporcionar información que pueda conducir a un producto final de mayor calidad.

## 2. Métricas para el Modelo de Requisitos

### Métrica de Puntos de Función

Medio para medir la funcionalidad que entra a un sistema utilizando datos históricos (mediciones de productos SW ya desarrollados).

Sirve para:

- **Estimar el costo o esfuerzo** requerido para diseñar, codificar y probar el software
- **Predecir el número de errores** que se encontrarán durante las pruebas
- **Prever el número de componentes y/o de líneas** fuente proyectadas en el sistema implementado

Valor de dominio de información	Conteo	Factor ponderado			=
		Simple	Promedio	Complejo	
Entradas externas (EE)	x	3	4	6	=
Salidas externas (SE)	x	4	5	7	=
Consultas externas (CE)	x	3	4	6	=
Archivos lógicos internos (ALI)	x	7	10	15	=
Archivos de interfaz externos (AIE)	x	5	7	10	=
Conteo total					
$PF = \text{conteo total} \times [0.65 + 0.01 \times \sum(F_i)]$					

Los puntos de función se derivan usando una relación empírica basada en medidas contables del dominio de información del software y en valoraciones cualitativas de la complejidad del software.

### Valores de dominio de información:

- **Número de entradas externas (EE).** Cada entrada externa se origina por un usuario o se transmite desde otra aplicación, y proporciona distintos datos orientados a aplicación o información de control.
- **Número de salidas externas (SE).** Cada salida externa es un dato o conjunto de datos derivados dentro de la aplicación que ofrecen información al usuario. En este contexto, salida externa se refiere a reportes, pantallas, mensajes de error, etc.
- **Número de consultas externas (CE).** Una consulta externa se define como una entrada que da como resultado la generación de alguna respuesta de software inmediata.
- **Número de archivos lógicos internos (ALI).** Cada archivo lógico interno es un agrupamiento lógico de datos que reside dentro de la frontera de la aplicación y se mantiene mediante entradas externas.
- **Número de archivos de interfaz externos (AIE).** Cada archivo de interfaz externo es un agrupamiento lógico de datos que reside fuera de la aplicación, pero que proporciona información que puede usar la aplicación.

### Ejemplo para cálculo de la métrica de puntos de función



- **3 entradas externas** (contraseña, botón de pánico y activar/desactivar)
- **2 salidas externas** (mensajes y estado de sensor)
- **2 consultas externas** (consulta de zona y consulta de sensor)
- **1 ALI** (subsistema monitoreo y respuesta)
- **4 AIE** (sensor de prueba, establec. de zona, activar/desactivar y alerta de alarma)

$$PF = 50 \times [0.65 + 0.01 \times 46] = 56$$

Valor de dominio de información	Conteo	Simple	Promedio	Complejo	
Entradas externas (EE)	x	3	4	6	= 9
Salidas externas (SE)	x	4	5	7	= 8
Consultas externas (CE)	x	3	4	6	= 6
Archivos lógicos internos (ALI)	x	7	10	15	= 7
Archivos de interfaz externos (AIE)	x	5	7	10	= 20
Conteo total					50

La empresa de SW podrá utilizar su histórico de proyectos realizados para conocer a qué equivale cada punto de función (pf):

- Líneas de código por cada pf = 20
- Horas por cada pf = 2,5h
- pfs al mes = 20

Teniendo 56 puntos de función, se pueden realizar estimaciones:

- Duración\_Estimada:  $56/20 = 2,8$  meses de trabajo
- Líneas\_de\_Código:  $20 \times 56 = 1120$
- Tiempo\_Total(horas) =  $2,5 \times 56 = 140$  horas

### 3. Métricas para Código Fuente

Leyes cuantitativas al desarrollo de software usando un conjunto de medidas primitivas que pueden derivarse después de generar el código o de que el diseño este completo.

$n_1$ = número de operadores distintos que aparecen en un programa

$n_2$ = número de operandos distintos que aparecen en un programa

$N_1$ = número total de ocurrencias de operador

$N_2$ = número total de ocurrencias de operando

**Longitud de programa:**  $N = n_1, \log_2 n_1 + n_2 \log_2 n_2$

**Volumen de programa (bits):**  $V = N \log_2 (n_1 + n_2)$

## 4. Métricas para el Modelo de Diseño OO

Conforme un modelo de diseño OO crece en tamaño y complejidad, una visión más objetiva de las características del diseño puede beneficiar tanto al diseñador experimentado (quien adquiere comprensión adicional) como al principiante (quien obtiene un indicio de la calidad que de otro modo no tendría disponible).

### Características medibles en un diseño OO

- **TAMAÑO:**

- **Población:** se mide al realizar un conteo estático de entidades OO, tales como clases u operaciones.
- **Volumen:** son idénticas a las medidas de población, pero se recolectan de manera dinámica: en un instante de tiempo determinado. Ej.: Clases/Día; Operaciones/Semana.
- **Longitud:** es una medida de una cadena de elementos de diseño interconectados (ej.: profundidad de un árbol de herencia es una medida de longitud).
- **Funcionalidad:** valor entregado al cliente por la aplicación OO.

- **COMPLEJIDAD:** Características estructurales al examinar cómo se relacionan mutuamente las clases de un diseño OO.
- **ACOPLAMIENTO.** Las conexiones físicas entre elementos del diseño OO (el número de colaboraciones entre clases o el de mensajes que pasan entre los objetos) representan el acoplamiento dentro de un sistema OO.
- **SUFICIENCIA.** Un componente de diseño (ej.: clase) es suficiente si refleja por completo todas las propiedades del objeto de dominio de aplicación que se modela.
- **COMPLETITUD.** Más general que la suficiencia que solo se centra en la aplicación actual que se desarrolla.
- **COHESIÓN.** Un componente OO debe diseñarse de manera que tenga todas las operaciones funcionando en conjunto para lograr un solo propósito bien definido.
- **PRIMITIVISMO.** El grado en el que una operación es atómica. La operación no puede construirse a partir de una secuencia de otras operaciones contenidas dentro de una clase.
- **SIMILITUD.** El grado en el que dos o más clases son similares en su estructura, función, comportamiento o propósito se indica mediante esta medida.
- **VOLATILIDAD.** De un componente de diseño OO mide la probabilidad de que ocurrirá un cambio.

## 5. Métricas para Mantenimiento

IEEE Std. 982.1-1988 [IEE93] sugiere un índice de madurez de software (IMS) que proporcione un indicio de la estabilidad de un producto de software (con base en cambios que ocurran para cada liberación del producto).

$MT$ = número de módulos en la liberación actual

$F_c$ = número de módulos en la liberación actual que cambiaron

$F_a$ = número de módulos en la liberación actual que se agregaron

$F_d$ = número de módulos en la liberación anterior que se borraron en la liberación actual

$$IMS = \frac{M_T - (F_a + F_c + F_d)}{M_T}$$

## 6. Métricas para Pruebas OO

Las métricas pueden ayudar a dirigir los recursos de prueba en hebras, escenarios y paquetes de clases que son 'sospechosas' a partir de características medidas tales como:

- **Falta de cohesión en métodos (FCOM).** Mientras más alto sea el valor de la FCOM más estados deben ponerse a prueba para garantizar que los métodos no generan efectos colaterales.
- **Porcentaje público y protegido (PPP).** Indica el porcentaje de los atributos de clase que son públicos o protegidos.
  - Valores altos de PPP aumentan la probabilidad de efectos colaterales entre las clases porque los atributos públicos y protegidos conducen a alto potencial para acoplamiento.
  - Las pruebas deben diseñarse para garantizar el descubrimiento de tales efectos colaterales.
- **Acceso público a miembros de datos (APD).** Número de clases (o métodos) que pueden acceder a otros atributos de clase (violación de la encapsulación). Valores altos de APD conducen al potencial de efectos colaterales entre clases.
  - Las pruebas deben diseñarse para garantizar el descubrimiento de tales efectos colaterales.
- **Número de clases raíz (NCR).** Conteo de las distintas jerarquías de clase que se describen en el modelo de diseño. Deben desarrollarse las suites de prueba para cada clase raíz y la correspondiente jerarquía de clase.
  - Conforme el NCR aumenta, también aumenta el esfuerzo de prueba.
- **Fan-in (FIN).** Cuando se usa en el contexto OO, el fan-in (abanico de entrada) en la jerarquía de herencia es un indicio de herencia múltiple.
  - FIN > 1 indica que una clase hereda sus atributos y operaciones de más de una clase raíz.
  - FIN > 1 debe evitarse cuando sea posible.
- **Número de hijos (NDH) y profundidad del árbol de herencia (PAH).** Los métodos de superclase tendrán que volverse a probar para cada subclase.



## Calidad del Software - FINAL – ONL- Temas 5, 6 y 7 – 11/07/2020

Nombre: \_\_\_\_\_ DNI: \_\_\_\_\_ Nota: \_\_\_\_\_

(3 puntos) TEST

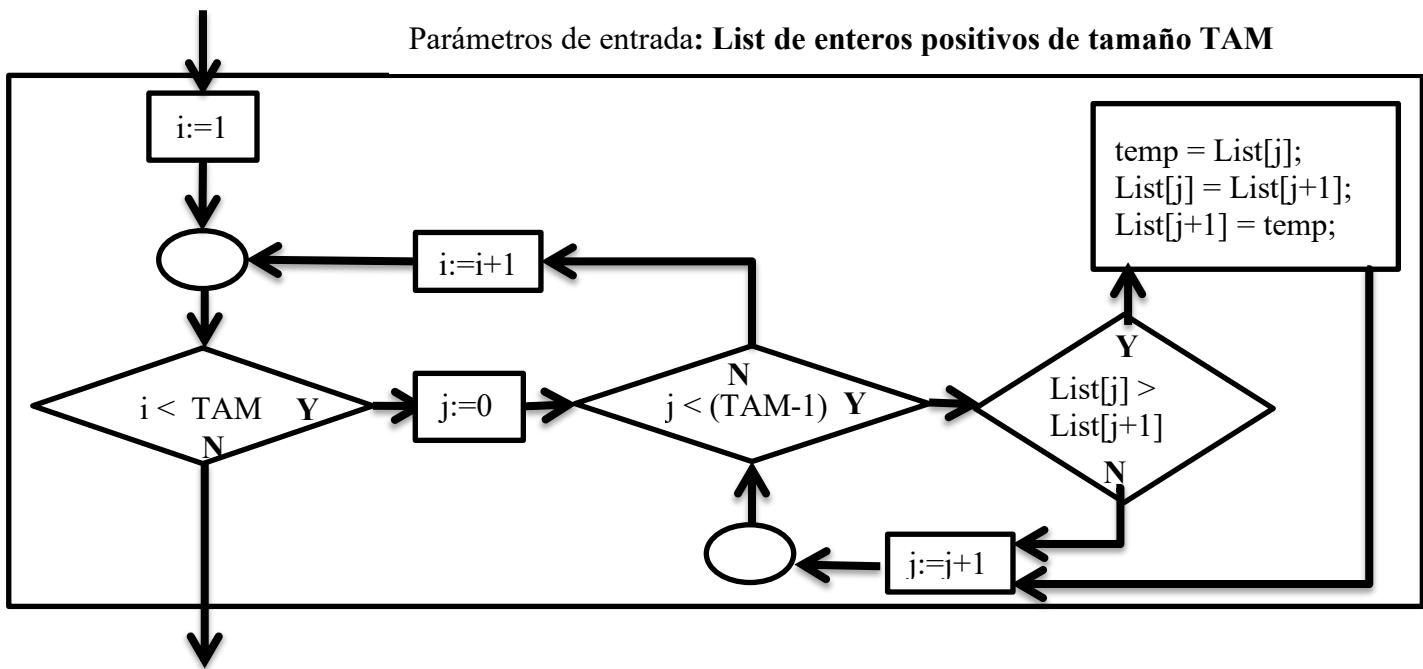
(7 puntos) EJERCICIOS

\_\_\_\_ / 2,5 puntos: Ejercicio 1

\_\_\_\_ / 2,5 puntos: Ejercicio 2

\_\_\_\_ / 2 puntos: Ejercicio 3

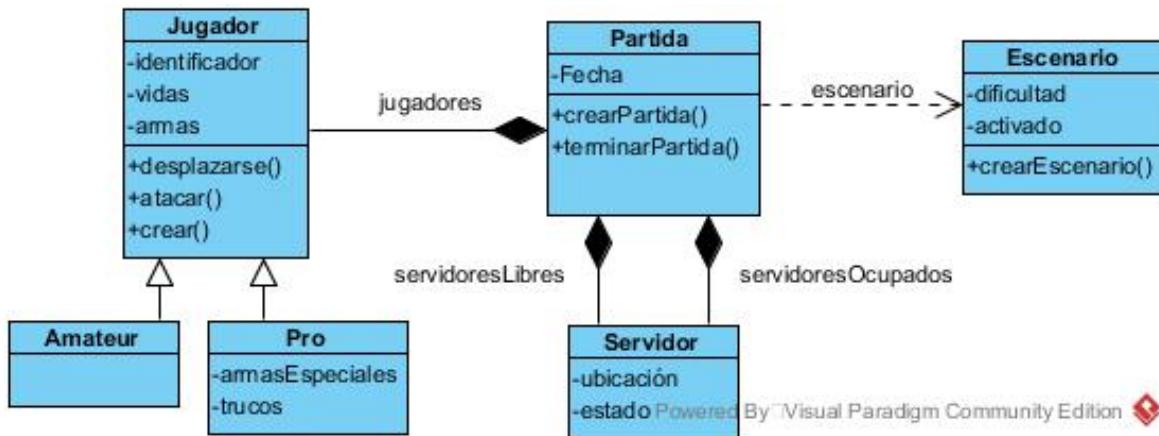
(2,5 puntos) Dado el siguiente diseño procedural del algoritmo de ordenación de la burbuja:



(0,5 puntos).- Escribir 5 condiciones que harían falta comprobar para la verificación formal del diseño anotándolas en el diagrama de flujo donde corresponda.

(1,5 puntos). Para esas condiciones, explicar si el diseño procedural las cumple

(2,5 puntos) C2.- El siguiente diagrama de clases representa un videojuego online:



**(1 punto).- Representar usando OCL los siguientes invariantes de datos:**

- El conjunto de servidores disponibles es igual a la unión de los servidores ocupados y libres.
- El número máximo de jugadores en una partida será de 7.

**(0,75 puntos) Precondiciones en OCL:**

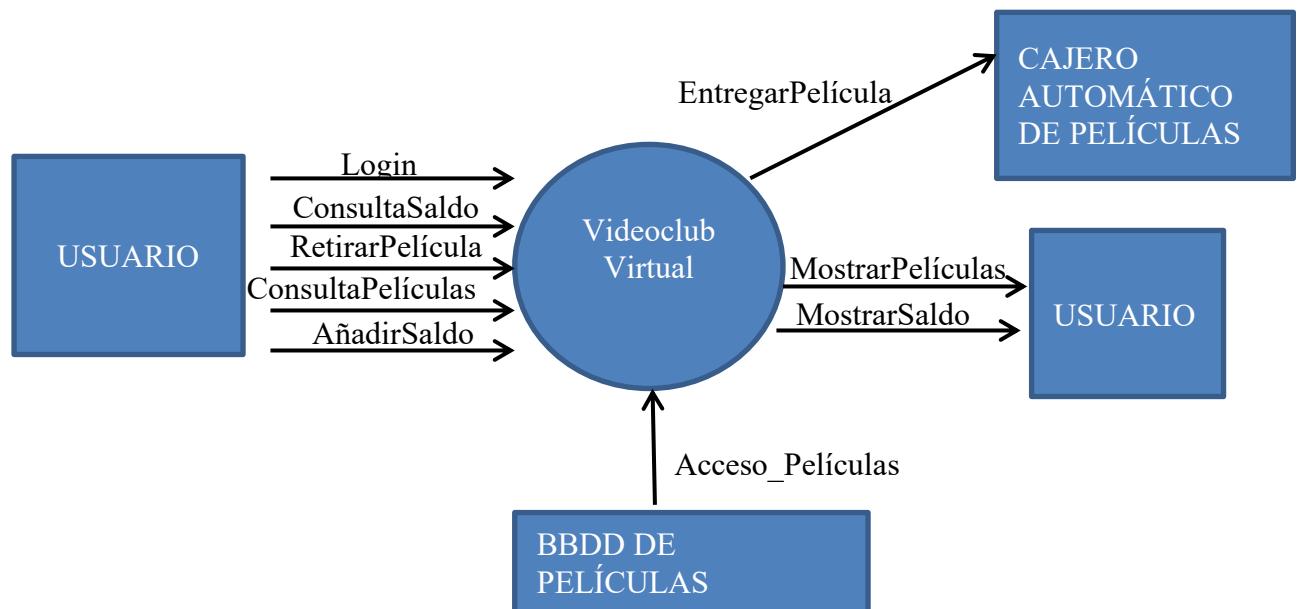
- Para crearPartida, el servidor a enlazar con la partida estará dentro del conjunto de servidores libres y el escenario no estará activado.
- Para terminar una partida, ni el conjunto de jugadores ni el conjunto de servidores ocupados estará vacío.

**(0,75 punto) Postcondiciones en OCL:**

- Tras crear una partida, se almacenará un nuevo servidor al conjunto de servidoresOcupados y se quitará del conjunto de servidoresLibres.
- Cuando un jugador ataca a otro, el número de vidas del jugador que recibe el ataque se reducirá en una unidad.

(2 puntos). C3.- Una empresa de software comienza un nuevo proyecto de software y desea realizar una serie de estimaciones para tener una idea de la envergadura de su proyecto y conocer si van a poder abordarlo en tiempo y recursos económicos y humanos disponibles. Para ello, utiliza la métrica de puntos de función en la fase de análisis del proyecto. Se conoce que el factor ponderado para todos los valores de dominio es 5 y que el resultado de las preguntas para calcular los factores de ajuste de valor es 10.

(1,5 puntos) Calcule la métrica de puntos de función.



Además el histórico de la empresa revela que cada punto de función (pf) equivale a:

- 15 líneas de código
- 20 horas
- 35 pfs al mes.

(0,5 puntos) Calcular la duración estimada del proyecto en horas, en meses de trabajo y el número de líneas de código estimadas.





## Calidad del Software - FINAL – PRE- Temas 5, 6 y 7 – 08/07/2020

Nombre: \_\_\_\_\_ DNI \_\_\_\_\_ Nota: \_\_\_\_\_

(3 puntos) TEST

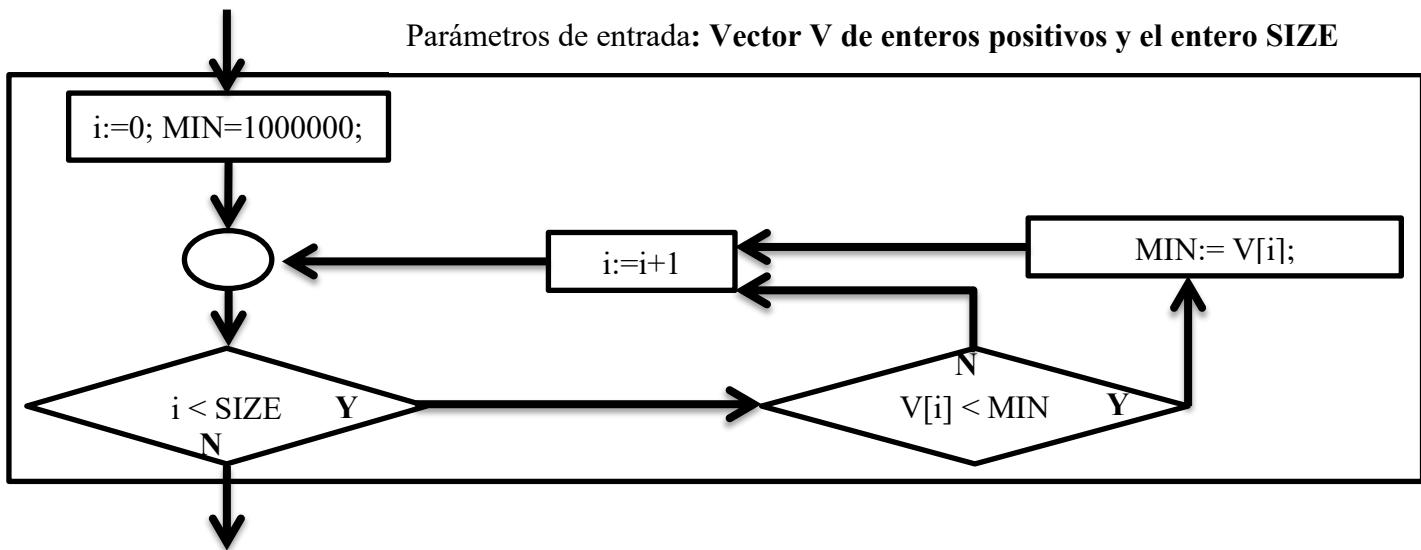
(7 puntos) EJERCICIOS

\_\_\_\_ / 2,5 puntos: Ejercicio 1

\_\_\_\_ / 2,5 puntos: Ejercicio 2

\_\_\_\_ / 2 puntos: Ejercicio 3

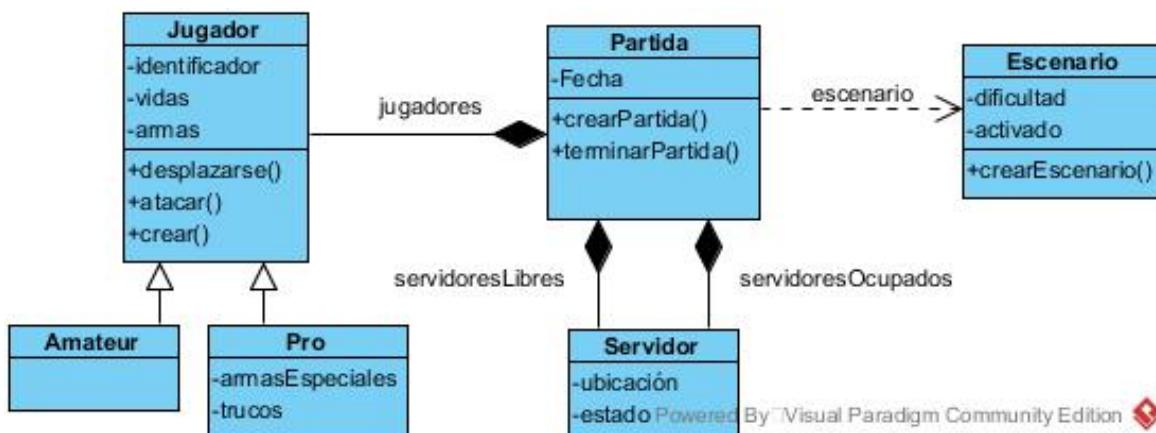
(2,5 puntos) Ejercicio 1. Dado el siguiente diseño procedural se pide:



(1 puntos). – Escribir 5 condiciones que harían falta comprobar para la verificación formal del diseño anotándolas en el diagrama de flujo donde corresponda.

(1,5 puntos). - Explicar si el diseño procedural cumple todas las condiciones encontradas.

(2,5 puntos) C2.- El siguiente diagrama de clases representa un Sistema de Ficheros EXT3:



**(1 punto).- Representar usando OCL los siguientes invariantes de datos:**

- El conjunto de servidores disponibles es igual a la unión de los servidores ocupados y libres.
- El número máximo de jugadores en una partida será de 7.

**(0,75 puntos) Precondiciones en OCL:**

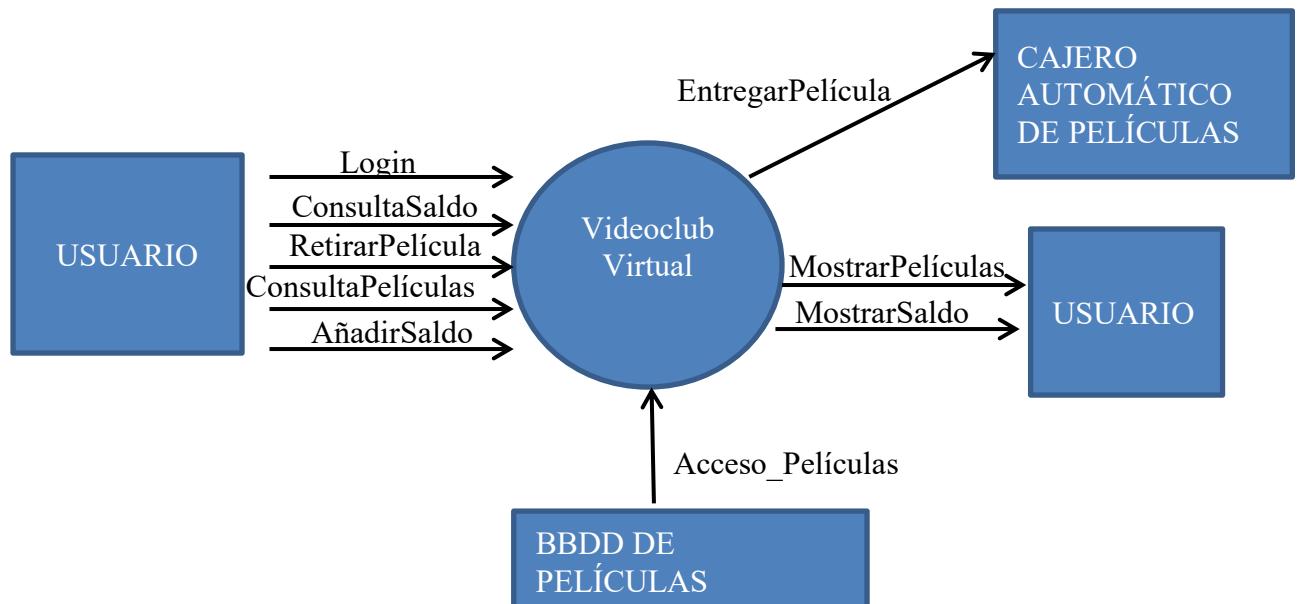
- Para crearPartida, el servidor a enlazar con la partida estará dentro del conjunto de servidores libres y el escenario no estará activado.
- Para terminar una partida, ni el conjunto de jugadores ni el conjunto de servidores ocupados estará vacío.

**(0,75 punto) Postcondiciones en OCL:**

- Tras crear una partida, se almacenará un nuevo servidor al conjunto de servidoresOcupados y se quitará del conjunto de servidoresLibres.
- Cuando un jugador ataca a otro, el número de vidas del jugador que recibe el ataque se reducirá en una unidad.

**(1,5 puntos). C3.- Una empresa de software comienza un nuevo proyecto de software y desea realizar una serie de estimaciones para tener una idea de la envergadura de su proyecto y conocer si van a poder abordarlo en tiempo y recursos económicos y humanos disponibles. Para ello, utiliza la métrica de puntos de función en la fase de análisis del proyecto. Se conoce que el factor ponderado para todos los valores de dominio es 3 y que el resultado de las preguntas para calcular los factores de ajuste de valor es 20.**

**(1,5 puntos) Calcule la métrica de puntos de función.**



Además el histórico de la empresa revela que cada punto de función (pf) equivale a:

- 10 líneas de código
- 2 horas
- 15 pfs al mes.

**(0,5 puntos) Calcular la duración estimada del proyecto en horas, en meses de trabajo y el número de líneas de código estimadas.**







UCAM

UNIVERSIDAD CATÓLICA  
SAN ANTONIO

# Escuela Universitaria Politécnica

## Grado en Ingeniería Informática



### Calidad del Software - PARTE 2 – PRE – Temas 5, 6 y 7 – 08/02/2021

Nombre: Juan José López López DNI \_\_\_\_\_ Nota: \_\_\_\_\_

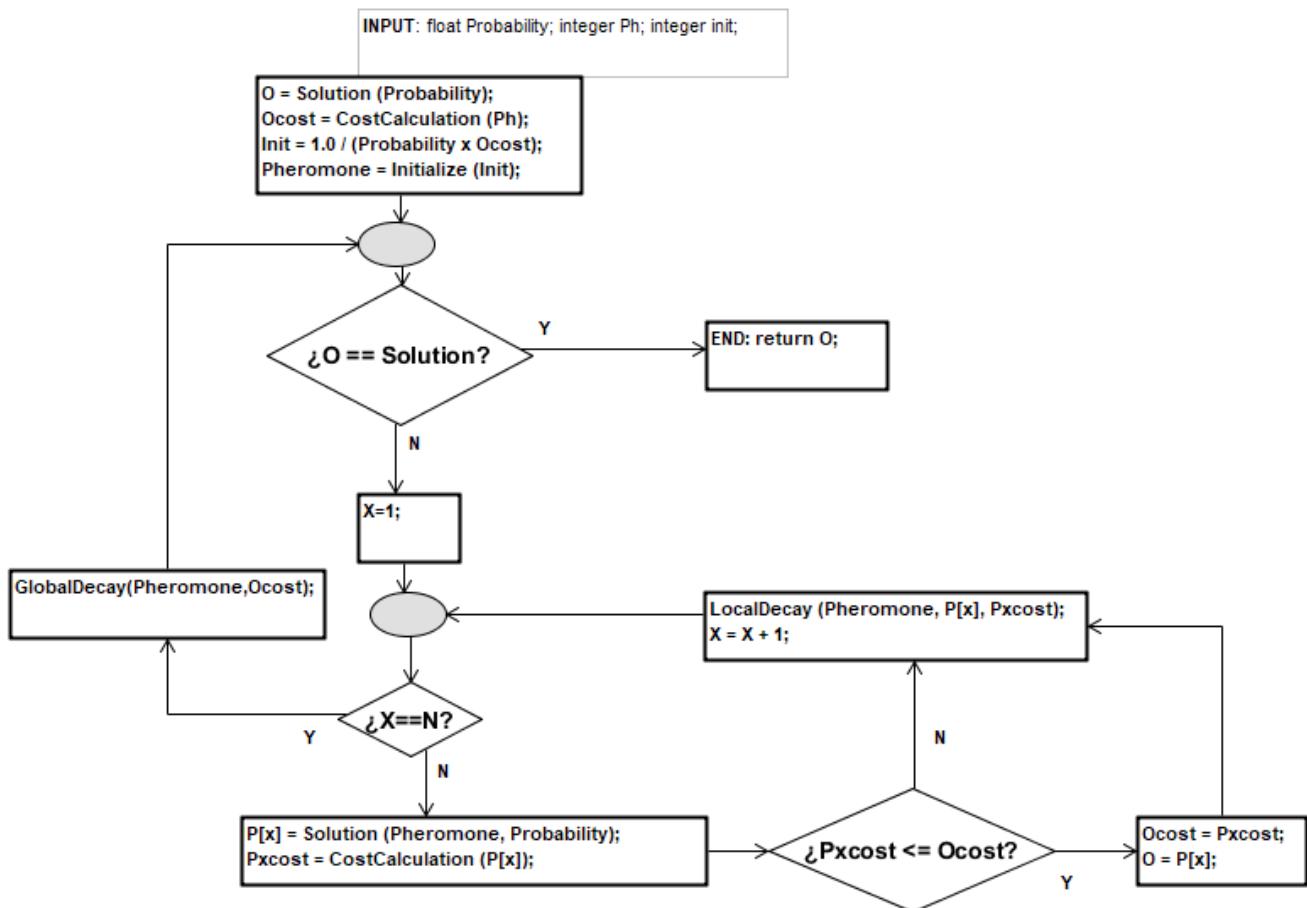
(3 puntos) TEST

(7 puntos) EJERCICIOS:

\_\_\_\_\_ / 1,5 puntos: Ejercicio 1; \_\_\_\_\_ / 3,5 puntos: Ejercicio 2; \_\_\_\_\_ / 1,5 puntos: Ejercicios 3

### EJERCICIOS (7 puntos)

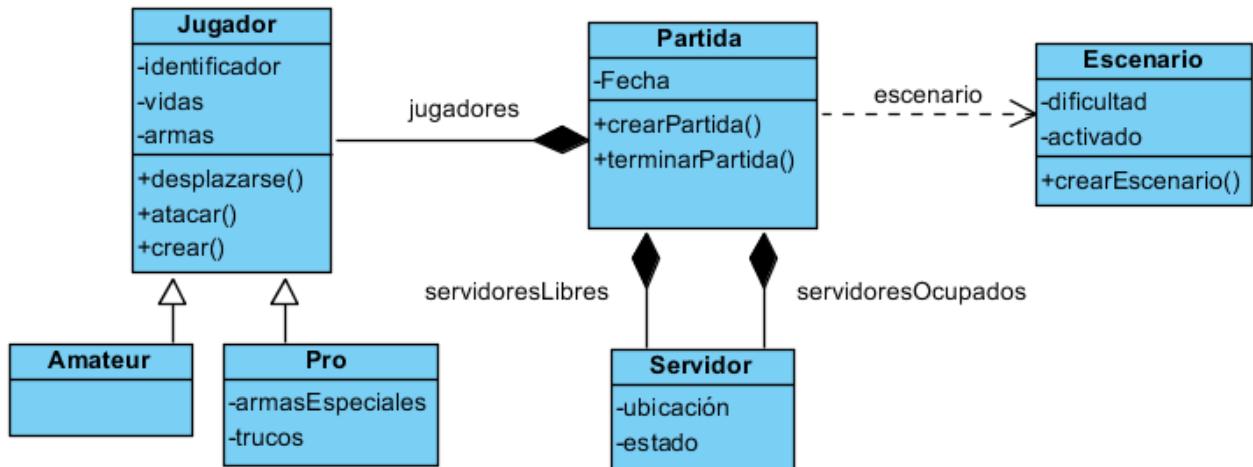
(1.5 puntos) C1. Dado el siguiente diseño procedural del algoritmo de la Colonia de Hormigas:



(0,5 puntos).- Escribir TODAS las condiciones que harían falta comprobar para la verificación formal del diseño anotándolas en el diagrama de flujo donde corresponda.

(1,5 puntos). Para 6 de esas condiciones (haya variabilidad), explicar si el diseño procedural las cumple

**(3.5 puntos) C2.- El siguiente diagrama de clases representa un Sistema de Ficheros EXT3:**



**(1.5 puntos).- Representar usando OCL los siguientes invariantes de datos:**

- a) Cada jugador tiene un Identificador único.
- b) El número de armas de un jugador está comprendido entre 1 y 5.
- c) El conjunto de servidores disponibles es igual a la unión de los servidores ocupados y libres.
- d) El número máximo de jugadores en una partida será de 7.

**(1 punto) Precondiciones en OCL:**

- a) Para crearPartida, el servidor a enlazar con la partida estará dentro del conjunto de servidores libres y el escenario no estará activado.
- b) Para terminar una partida, ni el conjunto de jugadores ni el conjunto de servidores ocupados estará vacío.

**(1 punto) Postcondiciones en OCL:**

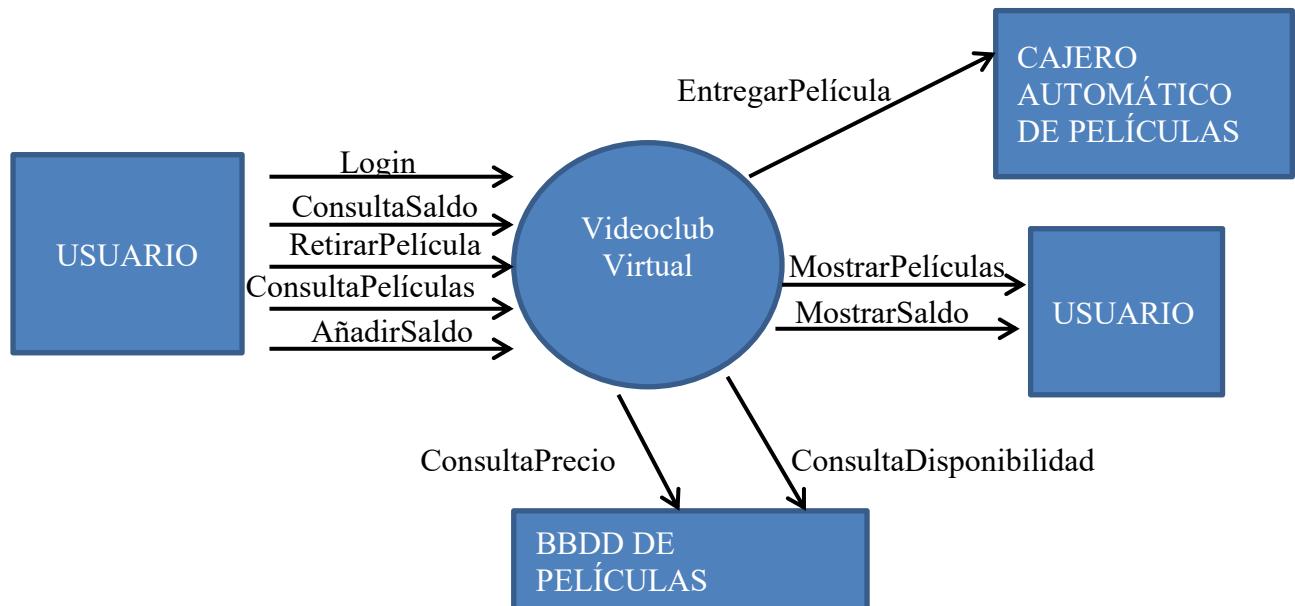
- a) Tras crear una partida, se almacenará un nuevo servidor al conjunto de servidoresOcupados y se quitará del conjunto de servidoresLibres.
- b) Cuando un jugador ataca a otro, el número de vidas del jugador que recibe el ataque se reducirá en una unidad.

(1.5 puntos). C3.. Una empresa de software comienza un nuevo proyecto de software y desea realizar una serie de estimaciones para tener una idea de la envergadura de su proyecto y conocer si van a poder abordarlo en tiempo y recursos económicos y humanos disponibles. Para ello, utiliza la métrica de puntos de función en la fase de análisis del proyecto.

El resultado de las preguntas para calcular los factores de ajuste de valor es 98.

Además el histórico de la empresa revela que cada punto de función (pf) equivale a:

- 10 líneas de código
- 12 horas
- Y que se es capaz de realizar 2 pfs al mes.



(1 punto) Calcule la métrica de puntos de función para un caso Optimista y además para otro Pesimista de los valores de los factores ponderados. Invente los valores de los factores ponderados que estime oportunos.

(0.25) Calcular la duración estimada del proyecto en horas y en meses de trabajo.

(0.25) Calcular el número de líneas de código estimadas.

