Universidad Rey Juan Carlos

Diseño y Mantenimiento de Servicios (DMS)

Apuntes de la Asignatura

Pablo González Troyano
Grado en Ciencia, Gestión e Ingeniería de Servicios

Primer Cuatrimestre - Curso 2023-24

Índice general

Tema 1: Principios básicos del diseño

1.1. Conceptos básicos

1.1.1. Principios generales de alto nivel

- *Services will be designed against an understanding of purpose, demand and the current capability of the organisation to deliver the service.*
  Los servicios serán diseñados teniendo en cuenta el objetivo, la demanda y la actual capacidad de la organización para ofrecer el servicio.
- *Services will be designed "outside-in" (customer focused), "not in-side out" (internally focused)*
  Los servicios estarán diseñados "de-fuera-a-dentro", no "de-dentro-a-fuera".
- *Services will be designed within the context of the system, not in isolation, e.g. by focusing on the optimisation of the system as a whole, rather than on individual components.*
  Los servicios serán diseñados dentro del contexto del sistema, no en aislamiento. Por ejemplo, centrándose en la optimización del sistema como un todo, en lugar de en componentes individuales.
- *Services will be designed around an understanding of value and efficiency of flow.*
  Los servicios serán diseñados alrededor de una comprensión del valor y la eficiencia del flujo.
- *The design will not treat a special cause of variation as if it was a common cause.*
  El diseño no tratará una causa especial de variación como si fuera una causa común.
- *Services will be co-created with their users.*
  Los servicios serán co-creados con sus usuarios.
- *Service designs will be prototyped.*
  Los diseños de servicios serán prototipados.
- *Against a clear business model and high level design, services will be designed, built and deployed incrementally and iteratively, to deliver value early and to inform the design.*
  A diferencia de un modelo de negocio y un diseño de alto nivel claro, los servicios serán diseñados, construidos y desplegados de forma incremental e iterativa, para aportar valor rápido e informar del diseño.
- *Services will be designed and delivered collaboratively, leveraging maximum benefit from the internal and partner network.*
  Los servicios serán diseñados y entregados de forma colaborativa, aprovechando al máximo los beneficios internos y de la red de colaboradores.

1.1.2. Principios del proceso de diseño de servicios

- *Activities that do not add value for the customer will be minimised.*
  Actividades que no aporten valor al cliente serán minimizadas.

- *Work will be structured around processes, not functions, products or geography.*
  El trabajo será estructurado en torno a procesos, no funciones, productos o ubicación geográfica.

- *The fragmentation of work will be minimised. Where possible, a single individual will have responsibility for a process from start to finish, to minimise hand-offs, reduce errors, delays, rework and administration overhead; and encourage ownership, innovation, creativity and improve control.*
  La fragmentación del trabajo será minimizada. Cuando sea posible, un solo individuo será responsable de un proceso de principio a fin, para minimizar transferencias, reducir errores, retrasos, trabajo repetido y gastos administrativis; y se fomentará la propiedad, innovación, creatividad y mejorará el control.

- *Process complexity will be minimised, by reducing the number of process steps, hand-offs, rules and controls; and by empowering staff to make decisions.*
  La complejidad de los procesos será minimizada, reduciendo el número de pasos en el proceso, transferencias, reglas y controles; y empoderando al personal para tomar decisiones.

- *Multiple versions of a process (tuned to different customer needs, markets, situations or inputs) will be developed where necessary, to ensure that the design of each process is fully aligned to customer demand.*
  Multiples versiones de un proceso (ajustadas a diferentes necesidades de cliente, mercados, situaciones o *inputs*) serán desarrolladas cuando sea necesario, para asegurar que el diseño de cada proceso está completamente alineado con las demandas del cliente.

- *For any given customer demand, process variation will be minimised to maximise reliability and predictability.*
  Para cualquier cliente dado, variaciones en el proceso serán minimizadas para maximizar la fiabilidad y predictibilidad.

- *Linear processing will be minimised, e.g. by avoiding the artificial linear sequencing of tasks and through the use of parallel processing and variable sequencing.*
  El procesamiento lineal será minimizado, por ejemplo: evitando la artificial secuenciación lineal de tareas y mediante el uso de procesamiento paralelo y la secuenciación variable.

- *Excessive process decomposition will be avoided, e.g. by replacing work instructions and callscripts with appropriate staff training and knowledge development.*
  Se evitará la descomposición excesiva de los procesos, por ejemplo sustituyendo las instrucciones de trabajo y los guiones telefónicos por una formación adecuada del personal y el desarrollo de conocimientos.

- *Unnecessary process breaks and delays will be minimised.*
  Se reducirán al mínimo las interrupciones y retrasos innecesarios del proceso.

- *The need for reconciliation will be minimised.*
  La necesidad de conciliación se reducirá al mínimo.

- *The need for controls will be minimised.*
  Se reducirá al mínimo la necesidad de controles.

- *The need for inspection will be minimised.*
  La necesidad de inspección se reducirá al mínimo.

- *We will only measure what matters.*
  Sólo mediremos lo que importa.

1.1.3. Principios de diseño de la organización

- *Work groups will be organised by process and competencies.*
  Los grupos de trabajo se organizarán por procesos y competencias.
- *Staff will be empowered to make decisions.*
  El personal estará capacitado para tomar decisiones.
- *Work will be located where it can be done most efficiently and effectively.*
  El trabajo se ubicará donde pueda realizarse con mayor eficiencia y eficacia.

1.1.4. Principios de diseño de los datos

- *Data will be normalised across the organisation and its partner network.*
  Los datos se normalizarán en toda la organización y su red de socios.
- *Data will be transferable and re-usable across the organisation and its partner network.*
  Los datos serán transferibles y reutilizables en toda la organización y su red de socios.
- *Data entry will be avoided, by using data lookup, selection and confirmation approaches.*
  Se evitará la introducción de datos, utilizando métodos de búsqueda, selección y confirmación de datos.

1.1.5. Principios del diseño de la tecnología

- *Technology will act as an enabler, rather than as a driver.*
  La tecnología actuará como facilitador, más que como impulsor.
- *Technology will be "pulled" into a design, not "pushed".*
  La tecnología se "introducirá" en el diseño, no se "empujará".
- *The technology design will be flexible and agile, able to be easily modified in response to changing business requirements.*
  El diseño tecnológico será flexible y ágil, capaz de modificarse fácilmente en respuesta a los cambiantes requisitos empresariales.

1.2. Principios de SOA

1. *Standardized service contract: Services adhere to a standard communications agreements, as defined collectively by one or more service-description documents within a given set of services.*
   Contrato de servicio normalizado: los servicios se adhieren a unos acuerdos de comunicación normalizados, definidos colectivamente por uno o varios documentos de descripción de servicios dentro de un conjunto determinado de servicios.
2. *Service reference autonomy (an aspect of loose coupling): The relationship between services is minimized to the level that they are only aware of their existence.*
   Autonomía de **referencia de los servicios** (un aspecto del **acoplamiento débil**): La relación entre servicios se minimiza al nivel de que sólo son conscientes de su existencia.
3. *Service location transparency (an aspect of loose coupling): Services can be called from anywhere within the network that it is located no matter where it is present.*

Transparencia en la ubicación de los servicios (un aspecto del **acoplamiento débil**): Los servicios se pueden llamar desde cualquier punto de la red en el que se encuentren, independientemente de dónde estén presentes.

4. *Service abstraction: The services act as black boxes, that is their inner logic is hidden from the consumers.*
   Abstracción de servicios: Los servicios actúan como cajas negras, es decir, su lógica interna se oculta a los consumidores.

5. *Service autonomy: Services are independent and control the functionality they encapsulate, from a Design-time and a run-time perspective.*
   Autonomía de los servicios: Los servicios son independientes y controlan la funcionalidad que encapsulan, desde una perspectiva de diseño y de ejecución.

6. *Service statelessness: Services are stateless that is either return the requested value or a give an exception hence minimizing resource use.*
   Servicio sin estado: Los servicios no tienen estado, es decir, devuelven el valor solicitado o emiten una excepción, minimizando así el uso de recursos.

7. *Service granularity: A principle to ensure services have an adequate size and scope. The functionality provided by the service to the user must be relevant.*
   Granularidad de los servicios: Un principio para garantizar que los servicios tengan un tamaño y un alcance adecuados. La funcionalidad proporcionada por el servicio al usuario debe ser relevante.

8. *Service normalization: Services are decomposed or consolidated (normalized) to minimize redundancy. In some, this may not be done. These are the cases where performance, access, and aggregation are required.*
   Normalización de servicios: Los servicios se descomponen o consolidan (normalizan) para minimizar la redundancia. En algunos casos, esto puede no hacerse. Estos son los casos en los que se requiere rendimiento, acceso y agregación.

9. *Service composability: Services can be used to compose other services.*
   Capacidad de composición de servicios: Los servicios pueden utilizarse para componer otros servicios.

10. *Service discovery: Services are supplemented with communicative meta data by which they can be effectively discovered and interpreted.*
    Descubrimiento de servicios: Los servicios se complementan con metadatos comunicativos que permiten descubrirlos e interpretarlos eficazmente.

11. ***Service re-usability**: Logic is divided into various services, to promote reuse of code.*
    Reutilización de servicios: La lógica se divide en varios servicios, para favorecer la reutilización del código.

12. *Service encapsulation: Many services which were not initially planned under SOA, may get encapsulated or become a part of SOA.*
    Encapsulación de servicios: Muchos servicios que no se planearon inicialmente bajo SOA, pueden encapsularse o convertirse en parte de SOA.

Tema 2: Patrones y estilos arquitectónicos de Servicios

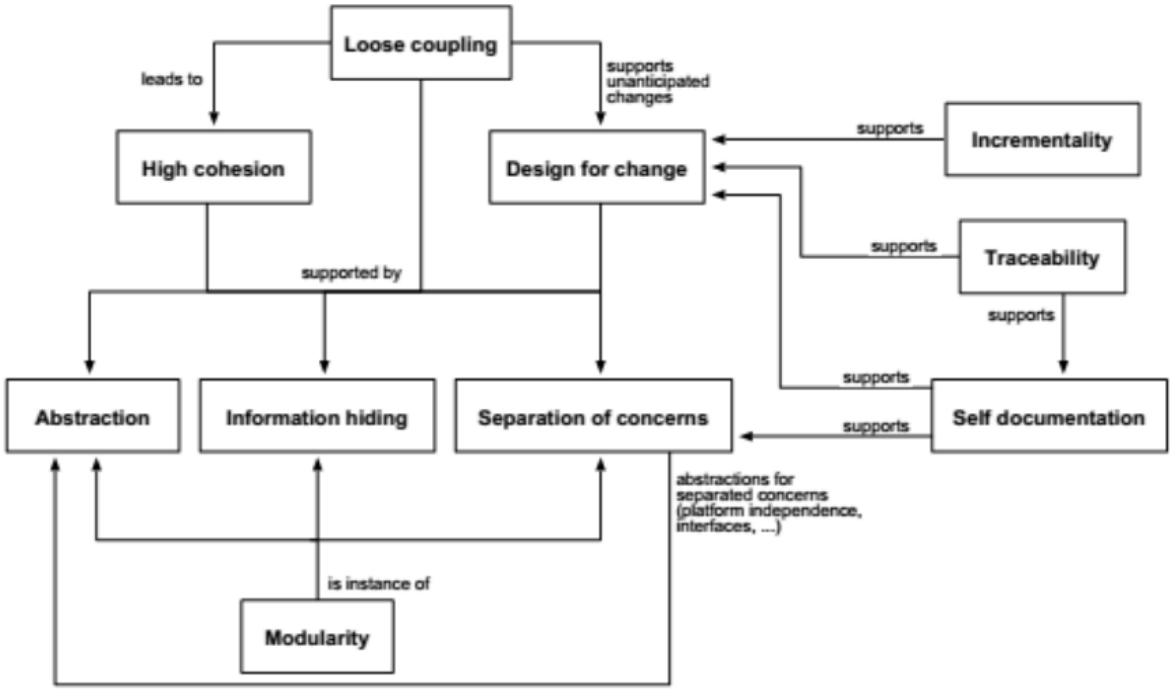
2.1. Arquitectura *software*

El término *arquitectura software* tiene muchas definiciones. Por ejemplo:

- Garlan y Perry: *la estructura de componentes que conforman un programa o sistema, sus relaciones, interrelaciones, principios y directrices que rigen su diseño y evolución en el tiempo*
- U. Zdun: *software architecture is a metaphor that helps us to better cope with the challenges we see in today's software system*.
- Bass, Clements y Kazman (1997): *estructuras de un sistema, en la que abarca componentes software, propiedades externas visibles de estos componentes, en el sentido de qué estos pueden hacer, como es, la gestión de fallos, el uso compartido de recursos, servicios y sus relaciones entre ellos*.
- SO/IEC/IEEE 42010 Systems and software engineering — Architecture description (2011): *fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution*
- Retos del diseño de arquitecturas software: cambio continuo y complejidad incremental. [1]
- Necesidad de documentar, crear y mantener la arquitectura, asegurando la calidad de la arquitectura.
  - Atributos básicos que dan respuesta a los requisitos del software/negocio:
    - Atributos de calidad del **sistema**: disponibilidad, fiabilidad, mantenibilidad, entendibilidad, modificabilidad, evolución, facilidad de prueba, portabilidad, eficiencia, escalabilidad, seguridad, facilidad de integración, reusabilidad...
    - Atributos de calidad del **negocio**: *time-to-market*, costes, expectativas de durabilidad del producto, mercado objetivo, integración de sistemas legados...
    - Atributos de la calidad de la **arquitectura**: integridad conceptual, corrección, completud, facilidad de construcción...
  - Principios arquitectónicos del software que guían su diseño.
- Atributos de calidad arquitectónica:
  - U. Zdun: *Reliability, Changeability, Availability, Reusabilitu, Maintainability, Time to Market, Costs, Performance, Resource consumption*
  - Object Management Group: *Cost, Performance, Fault Tolerance, Functcionality, Maintainability, Fail Safe, Interoperability, Availability, Capacity, Scalability, Throughput*

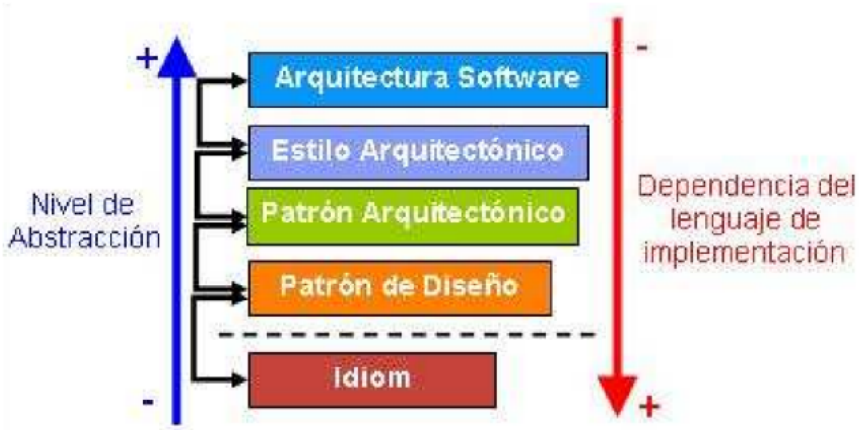
[1] «Laws of Software Evolution» (Lehman and Belady, 1980)

- Principios arquitectónicos del software:



2.2. Organización del diseño del *software*

Idea:



- **Estilos arquitectónicos**: definen los componentes y los conectores. Qué.
- **Patrones arquitectónicos**: definen estrategias de implementación de esos componentes y conectores. Nivel abstracto.
- **Patrones de diseño**: establecen soluciones estándar a problemas de implementación conocidos. Nivel de programación.

2.2.1. Estilos arquitectónicos

Definiciones

- *Una especialización de tipos de elementos, relaciones y un conjunto de restricciones sobre cómo pueden ser utilizados.* — Software Eng. Institute.
- *Una clase de arquitecturas o piezas importantes de la arquitectura, que se encuentran repetidamente en la práctica y poseen propiedades conocidas que permiten su reutilización.* — Kazman y Clements

Características

- Colección (con nombre) de un conjunto de decisiones arquitectónicas de diseño que:
  - Son aplicables en un contexto de desarrollo determinado.
  - Restringen las decisiones de diseño arquitectónico que son específicas para un sistema particular dentro de un contexto.
  - Derivan ciertas características de calidad en el sistema resultante.
- Derivan ciertas características de calidad en el sistema resultante.
- Son útiles a la hora de determinar todo: desde la estructura de una subrutina a la estructura de más alto nivel del sistema.

Elementos

Vocabulario de componentes, tipos de conectores y un conjunto de restricciones sobre los que estos pueden estar combinados.

Un vocabulario de elementos de diseño: tipos de componentes, conectores, elementos de datos...

Reglas de configuración: restricciones topológicas que determinan las composiciones permitidas.

Interpretación semántica.

Beneficios

- Reutilización de diseño y código.
- Entendimiento de la organización de un sistema.
- Interoperabilidad. Estandarización de los estilos.
- Especifidad en los estilos, fácilmente aplicables.

2.2.2. Patrones arquitectónicos

Taylor, Medvidovic y Dashofy:

> Una colección de decisiones de diseño arquitectónico, que se aplican sobre problemas recurrentes, teniendo en cuenta los diferentes contextos del desarrollo software en el que el problema surge. Los patrones arquitectónicos proponen una solución demostrada para un problema específico, mediante la descripción de los componentes y subcomponentes que la constituyen, la colaboración de estos y sus responsabilidades particulares.

Buschmann:

> El nivel más alto de patrón de nuestro sistema de patrones. Ayudan a especificar la estructura fundamental de la aplicación. Cada patrón arquitectónico facilita la especificación de las propiedades globales del sistema, además de adaptar las diferentes interfaces de comunicación.

De estructuración, sistemas distribuidos, sistemas interactivos, sistemas adaptables.

2.2.3. Patrones de estilo

Buschmann:

> Un esquema para refinar subsistemas o componentes de un sistema software y las relaciones existentes entre ellos.

Gamma et al.:

> Una descripción desde un nivel de abstracción, que ayuda a identificar los aspectos clave de una estructura de diseño común, lo que los hace útiles para crear un diseño reutilizable.

De creación, estructurales, de comportamiento.

2.2.4. Idioms

Buschmann:

> Describen cómo implementar aspectos particulares de los componentes, las relaciones y características entre ellos

2.2.5. Comparativa

Tabla 13 Comparativa Estilo arquitectónico - Patrón arquitectónico - Patrón de diseño

Estilo arquitectónico	Patrón arquitectónico	Patrón de diseño
Flujo de datos	Pipe & Filter	Forwarder-Receiver (B) Client-Dispatcher-Server (B) Bridge (G)
Datos centralizados	Pizarra	Proxy (B) (G) Mediator (G)
Máquinas virtuales	Intérpretes	Command-Procesor (B) View-Handler (B) Interpreter (G) Command (G) Memento (G)
	Sistemas basados en reglas	
Componentes independientes	Broker	Client-Dispatcher-Server (B) Publisher-Subscriber (B) Whole-Part (B) Observer (G)
	Invocación implícita	
Llamada y retorno	Presentation-Abstraction Control	Master-Slave (B) Facade (G)
	Model-View-Controller	
	Capas	

2.2.6. Arquitecturas heterogéneas

La mayoría de los sistemas utilizan una combinación de estilos. En una arquitectura por capas, un componente en una puede desarrollarse con un estilo *pipe&filter*.

Un componente puede estar asociado a conectores elaborados con diferentes estilos. Un componente puede acceder a un repositorio a través de invocación explícita y comunicarse con otros componentes a través de un *pipe*.

2.3. SOA

Colección de servicios que son capaces de comunicarse unos con otros.
Cada servicio es el final de una conexión, que puede ser utilizada para acceder a dicho servicio y/o para interconectar con otros servicios.

Propiedades:

- Interfaces públicas e invocables.
- Interacciones independientes. Sin estado.
- Independiente de plataforma y protocolo de comunicación
- SOA es un concepto arquitectónico, no una tecnología.

¿Qué es SOA?

Tabla 39 Resumen comparativo SOA

Comparativas	Características a cumplir	Conclusión
SOA como Estilo Arquitectónico	- Ofrece un estructura general del sistema: **SÍ** - Es independiente a otros estilos: **SÍ** - Especifica componentes, conectores y restricciones: **SÍ**	**Si**, puede considerarse como Estilo Arquitectónico
SOA como Patrón Arquitectónico	- Define una estructura básica del sistema: **SÍ** - Provee un subconjunto de subsistemas predefinidos, incluyendo reglas y pautas: **SÍ** - Puede estar contenido en otros patrones: **NO** - Puede usarse como una plantilla de construcción: **NO**	**No** puede considerarse como Patrón Arquitectónico
SOA como Patrón de Diseño	- Posee un conjunto de soluciones de diseño comunes a problemas particulares: **SÍ**	**Sí**, existe un catalogo y una especialización de Patrones de Diseño

Un buen **conjunto de patrones** SOA está disponible en el **libro de Thomas Erl**:

- *Foundational Inventory Patterns*
- *Logical Inventory Layer Patterns*
- *Inventory Centralization Patterns*
- *Inventory Implementation Patterns*
- *Inventory Governance Patterns*
- *Foundational Service Patterns*
- *Service Implementation Patterns*
- *Service Security Patterns*
- *Service Contract Design Patterns*
- *Legacy Encapsulation Patterns*
- *Service Governance Patterns*
- *Capability Composition Patterns*
- *Service Messaging Patterns*
- *Composition Implementation Patterns*
- *Service Interaction Security Patterns*
- *Transformation Patterns*

***Mashup* de servicios** Las funcionalidades de los servicios pueden ser mezclados para proveer nuevos servicios. Hay patrones como de recolección, mejora, ensamblaje, gestión y test.

2.3.1. Principios de buen diseño en soluciones SOA

- Bajo acoplamiento
- Nivel de abstracción adecuado / conceptualización
- Reutilización
- Autonomía / nivel de granularidad / cohesión
- Carencia (o no) de estado
- "Composicionalidad"
- Capacidades de (auto)descubrimiento / reconfiguración
- Estandarización de contratos

Ver sección 1.2

2.3.2. Estilos arquitectónicos más comunes

- Tradicionales (influenciados por el lenguaje) > Programa principal/subrutinas y estilo orientado a objetos.

- Estratificados > Máquinas virtuales y estilos por capas (cliente-servidor).
- Estilos de flujo de datos > Batch secuencial y tuberías y filtros.
- Memoria compartida > Repositorio/pizza y basados en reglas.
- Intérprete > Interprete y código móvil.
- Invocación implícita > Basados en eventos y publicación-suscripción.
- Peer-to-peer y estilos derivados: c2/corba.

Pipes & Filters

Flujo: Lee flujos de datos de sus entradas y genera flujos de datos a sus salidas y transforma los datos incrementalmente(se comienza a generar la salida antes de acabar de leer toda la entrada).

Tubería: Transmite el flujo de datos.

Invariantes (Restricciones):

- Los filtros son independientes (no comparten estado con otros filtros).
- No saben a qué otros filtros están conectados.

Especializaciones del estilo:

- Pipelines: La topología es una secuencia lineal de filtros).
- *Pipes* con capacidad limitada.
- *Pipes* con tipos (los datos pasados entre dos filtros tienen un tipo bien definido; eg: *char*).

Ejemplo: programas escritos en Shell de Unix.

Ventajas:

- Descomposición del problema en pasos independientes.
- Soporta la reutilización de los filtros.
- Sistemas fáciles de mantener y mejorar.
- Permiten análisis.
- Soportan la ejecución concurrente de filtros.
- Escalables. Inserción de nuevos filtros.

Desventajas:

- No adecuados para el procesamiento interactivo.
- Problemas de rendimiento, ya que los datos se transmiten en forma completa entre filtros.

Estilo Orientado a Objetos (Componentes)

- Los componentes son objetos, con datos y operaciones asociadas. **ENCAPSULACIÓN**.
- Los conectores son mensajes para invocar métodos.
  RPC: *Remote Procedure Call*.

Invocación implícita (eventos)

Se anuncian **EVENTOS** en lugar de invocar métodos.

- Los componentes registran su interés en ciertos eventos, y para cada uno de ellos asocian un método.
- Cuando otro componente anuncie el evento (modo *broadcast*), el sistema invoca de manera implícita todos los métodos asociales a dicho evento.

Invariantes (restricciones):

- Los componentes que anuncian esos eventos no conocen qué componentes se verán afectados.
- Ninguna suposición en cuanto al procesamiento realizado como respuesta a un evento.
- Evitan aplicaciones que preguntan si hay cambio de datos.

Ejemplo:
Uso frecuente en: gestores de bases de datos e interfaces de usuarios.

Ventajas:

- Alta reutilización de componentes.
- Sistema que evolucionan con facilidad.
- Alta versatilidad al poder registrarse a eventos dinámicamente.
- Acoplamiento inexistente entre componentes.

Desventajas:

- Los componentes renuncian al control sobre el sistema.
- No hay conocimiento sobre qué componentes responderán al evento.
- No hay conocimiento sobre el orden en que se activarán los métodos asociados a un evento.
- Si los datos asociados con un evento no pueden pasarse con el propio evento, hace falta una estructura global. Ello implica problemas de rendimiento y gestión del repositorio compartido.
- Dificultad para razonar acerca de la corrección, puesto que ya no existen precondiciones y postcondiciones.

Estilo por capas

Arquitectura software que organiza al software en capas, donde cada capa se construye sobre otra.
Una capa se define como un conjunto de (sub)sistemas con el mismo grado de generalidad.

- Organización jerárquica del sistema:
  - Cliente-servidor multinivel.
  - Cada capa soporta una API (servicios) para que lo utilice la capa superior.
  - Cada capa actúa como productor de las capas superiores.
  - Cada capa actúa como consumidor de las capas inferiores.
- Los componentes son cada una de las capas.
- Los conectores son protocolos de interacción entre capas.

- Restricciones topológicas: interacción solo entre capas adyacentes.

Ventajas:

- Facilita la descomposición del problema en varios niveles de abstracción.
- Soporta la evolución (mejora). El cambio de una capa solo afecta a las adyacentes.
- Soporta la reutilización.
- Se pueden cambiar las implementaciones respetando las interfaces con capas adyacentes.

Inconvenientes:

- Ejecución, debido a los problemas de coordinación entre capas.
- Determinar el nivel de abstracción correcto. La separación entre capas.
- No todos los sistemas pueden construirse en capas.

Cliente-Servidor

Objetivo: proveer arquitectura escalable.
Cada ordenador o proceso en la red es cliente o servidor.

Servidor:

- Pasivo (esclavo).
- Espera peticiones.
- Cuando recibe una petición, la procesa y envía respuesta.
- Los servidores no conocen ni el número ni las identidades de los clientes.

Cliente:

- Activo (maestro).
- Los clientes conocen las identidades de los servidores.
- Envía peticiones.
- Espera hasta que llega la respuesta.

Repositorio

- Basada en una estructura central de datos.
- Conjunto de componentes que operan en función del almacén de datos.
- Las interacciones entre el repositorio y los demás componentes es la variable:
  - Repositorio. La entrada de los datos es seleccionada por los componentes.
    $APP \rightleftarrows BBDD$
  - Pizarra. El estado de los datos del repositorio selecciona el proceso a ejecutar.
    $APP \leftarrow BBDD$

Ventajas:

- Es escalable. Posibilita la integración de nuevos módulos.
- Adecuado para la resolución de problemas no deterministas.

- Se puede resumir el estado de conocimiento en cada momento del proceso.

Desventajas:

- Estructura de datos común a todos los agentes.
- Problemas de carga a la hora de chequear y vigilar el estado de la pizarra.
- Cualquier cambio en la estructura de datos afecta a los módulos/agentes.

Máquina virtual (intérprete)

4 componentes:

1. Un motor de simulación o interpretación.
2. Una memoria que contiene el código a interpretar.
3. Una representación del estado de la interpretación.
4. Una representación del estado del programa que se está simulando.

Ventajas: solución *software* a problemas *hardware*.

Desventajas: no siempre es aplicable; reducido a lenguajes de programación concretos.

2.4. Arquitectos *software*

Deben trabajar siguiendo algún método bien especificado y reutilizar soluciones que han sido exitosas en el pasado. Está sometido a una fuerte responsabilidad. Debe ser:

- **Diseñador *software***: debe poder reconocer, reutilizar o inventar diseños eficientes y efectivos y aplicarlos correctamente y saber utilizar los patrones y estilos de diseño. Un arquitecto de *software* debería poder justificar todas sus decisiones de diseño y poder comunicar sus decisiones al grupo de desarrollo.
- **Experto en el dominio**: debe conocer el dominio del sistema de información que está construyendo y saber qué arquitecturas debería utilizar en cada uno.
- **Especialista en la tecnología *software***: debe asegurar que sus ideas podrán ser implementadas sobre una tecnología existente en el mercado. Un diseño es inútil si no puede ser implementado.
- **Experto en un estándar**: conocer y seguir un estándar de desarrollo puede facilitar tareas como arquitecto.
- **Buen planificador**: debe saber que el diseño de la arquitectura está restringido al presupuesto del que dispone, a la tecnología existente, al tiempo del proyecto...

Tema3a: Patrones y estilos arquitectónicos — SOA, básicamente

3.1. Conceptos

- SOA: Architectural style for building systems based on interactions of loosely coupled, coarse-grained, and autonomous components called services.
- Cada servicio expone: procesos y comportamiento, a través de *contratos*.
- Los contratos están compuestos de mensajes, en direcciones "descubribles" llamadas *endpoints*.
- El comportamiento de los servicios está gobernado por políticas, que son externas al servicio.
- Contratos y mensajes son utilizados por componentes externos, llamados consumidores de servicios.

3.2. Mitos de SOA

- *SOA es una forma de alinear los equipos de IT y negocio.*
  → No es el objetivo de SOA.
- *SOA es una aplicación que tiene un servidor web como interfaz.*
  → SOA es una arquitectura.
- *SOA ES UN CONJUNTO DE TECNOLOGÍAS (SOAP, REST, WS, ETC).*
  → SOA es independiente de la tecnología.
- *SOA es una estrategia de resusabilidad.*
  → Depende de la granularidad.
- *SOA es una solución lista para usar.*
  → SOA no es un producto.

3.3. Patrones SOA

Proveen una solución probada a un problema común, individualmente documentado en un formato consistente y, normalmente, como parte de una colección mayor.

3.3.1. Beneficios

- Representan soluciones probadas en el mundo real, a problemas comunes de diseño.
- Organizados en un formato estandarizado y de fácil referencia.

- Generalmente repetibles por la mayoría de los profesionales de IT.
- Pueden ser usados para asegurar la consistencia en cómo los sistemas son diseñados y construidos.
- Pueden ser los pilares para estándares de diseño.
- Normalmente, son flexibles y opcionales; así como documentado el impacto de su aplicación.
- Pueden ser usados como ayudas educativas a la hora de documentar aspectos específicos del diseño de un sistema.
- Pueden, a veces, ser aplicados antes y después de la aplicación de un sistema.
- Pueden estar soportados en la aplicación de otros patrones de diseño que son parte de la misma colección.
- Enriquecen el vocabulario de un campo IT, porque cada patrón tiene un nombre significativo.

3.3.2. Aplicabilidad

- Patrones SOA: aplican los principios de diseño de la orientación a servicios.
- Patrones Compuestos (*Compound*): aplican diferentes patrones a la vez.
- Patrones para *Big Data*: utilización de servicios para extracción de conocimiento en redes de datos empresariales.
- Patrones para *Cloud*: utilización de servicios en sistemas y modelos basados en la nube.
- Patrones para *Grid*: utilización de servicios para implementación de sistemas de computación distribuida.

Tema3b: Implementación de patrones SOA e integración de servicios

4.1. Integración y composición de servicios

- Orquestación (centralizado) y coreografía (comunicación independiente).
- Existen tecnologías *middleware* para facilitar la integración: ESB.
- Existem metodologías y modelos de integración: ITIL y SIAM.

Coordinación basada en **orquestación** (composición):

- Lógica de ejecución de aplicaciones basadas en servicios Web, mediante la definición de sus flujos de control y establecimiento de reglas para gestionar los datos de negocio no observables.
- Ejemplo: BPEL (*Business Process Execution Language*).

Coordinación basada en **coreografías** (colaboración):

- Describen colaboraciones de servicios web (entre empresas), mediante la definición del comportamiento común observable.
- Los intercambios de información se producen a través de los puntos de contacto compartidos cuando las reglas de ordenación de mensajes se cumplen.
- Ejemplo: WS-CDL (*Web Services Choreography Description Language*).

Coordinación basada en **coordinación/transacción**:

- Conseguir interacciones transaccionales entre servicios.
- *WS-Coordination* y *WS-Transaction*. Complementan BPEL para ofrecer mecanismos que definan protocolos específicos para sistemas de procesamiento de transacciones o sistemas de flujos de trabajo.

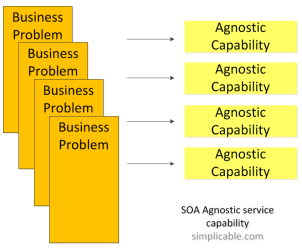
Estándares de **valor añadido**:

- Soportan interacciones de negocio complejas. Embebido en SOAP.
- Mecanismos de seguridad y autenticación, autorización, confianza, privacidad, conversaciones seguridas, gestión de contratos...
- Ejemplos: *WS-Security*, *WS-Policy* y *WS-Management*.

4.2. Implementación de patrones SOA - 10 patrones básicos

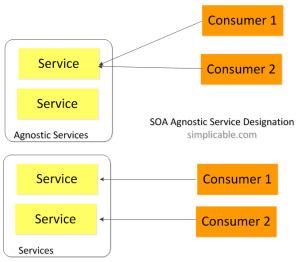
4.2.1. Servicios agnósticos

- Implementación de funcionalidades (*capabilities*) que son comunes a varios problemas de negocio.
- Ventajas: reutilización y facilidad de composición.



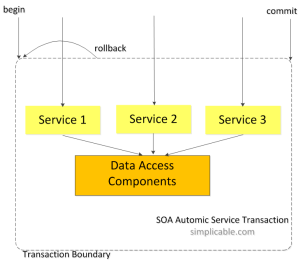
4.2.2. Declaración de servicios agnósticos

- Los servicios agnósticos deberían declarar explícitamente que son agnósticos (soluciones genéricas) a través de sus interfaces.
- Ventajas: reusabilidad en futuros desarrollos y facilidad de composición.



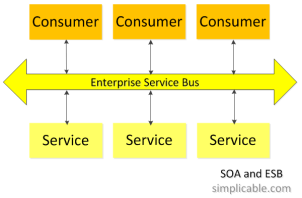
4.2.3. Transacciones en servicios atómicos

- Los servicios deberían ser capaces de implementar operaciones y acciones *reversibles* o ser capaces de participar en composiciones transaccionales.
- Ha de implementarse en un componente/servicio de una capa superior.
- Ventajas: *statelessness*/gestión de estado entre invocaciones.



4.2.4. *Enterprise Service Bus* (ESB)

- Un ESB es un componente tecnológico que actúa de gestor de mensajes: transformación, enrutamiento e interconexión; a través de la implementación de protocolos de comunicación.
- Ha de implementarse en un componente/servicio de una capa superior.
- Ventajas: bajo acoplamiento, interoperabilidad, abstracción de puntos de conexión.

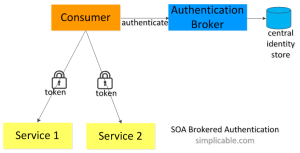


4.2.5. *Service Façade*

- Implementación de un servicio intermedio que se sitúa entre el contrato de servicio que firma el consumidor y el servicio que implementa dicho contrato.
- Sirve para eliminar o reducir el acoplamiento entre el servicio y su contrato.
- La idea es ser capaz de reducir el impacto de un cambio en el contrato o en el servicio.
- Un mismo servicio puede tener varias *façades* para dar soporte a varios tipos de contratos.
- Ventajas: bajo acoplamiento.

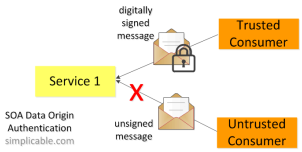
4.2.6. Gestor de autenticación

- Única responsabilidad es actuar de autenticador de consumidores que acceden a un servicio.
- Normalmente, mediante el uso de *tokens*.
- Ventajas: facilidad de composición.



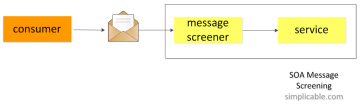
4.2.7. Autenticación de mensajes de origen

- Utilización de certificados para la autenticación de clientes.
- Ventajas: facilidad de composición.



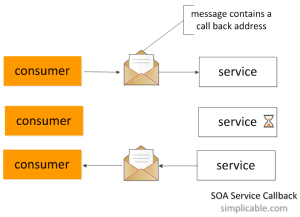
4.2.8. Análisis de mensajes

- Implementación de un servicio de análisis de mensajes enviados a un servicio para evitar mensajes malintencionados.
- Ventajas: reutilización y facilidad de composición.



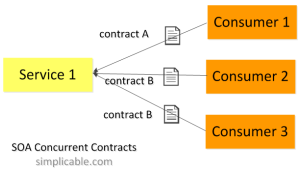
4.2.9. *Service Callback*

- El servicio requiere que sus consumidores le consulten de forma asíncrona.
- El servicio notifica al consumidor cuando ha terminado o en cierto punto.
- Ventajas: Liberación de recursos para procesos de larga duración.



4.2.10. Varios contratos de servicio

- Un servicio soporta varios contratos a la vez.
- Puede usarse para mantener el soporte a versiones antiguas o para facilitar la reutilización (diferentes vistas para diferentes objetivos).
- Ventajas: Facilita la reutilización.



4.3. Tecnologías *middleware* para integración - 5 principios básicos de la integración de servicios

1. **Orquestación**: componer diferentes servicios de grano fino existentes para conseguir un servicio compuesto de alto nivel.
   - Conseguir la granularidad apropiada.
   - Promover la reusabilidad y la facilidad de gestión de los componentes compuestos.
2. **Transformación**: los formatos de los datos intercambiados se transforman cuando son procesados por elementos de gestión de información intermedios (ESB).
   - P. ej.: CSV, SOAP/XML, JSON, etc.
   - La solución pasa por utilizar formatos estándar o "canónicos".
3. **Transporte**: Es importante conocer los protocolos de negociación y transporte de información y el impacto que estos tienen en la implementación de la solución integrada.
   - HTTP, JMS, JDBC.
   - Convertir los gestores de protocolos en servicios facilita la integración.
4. **Mediación**: proveer múltiples interfaces para conseguir a) soportar múltiples versiones de un servicio para tener compatibilidad hacia atrás; o, b) permitir múltiples canales de comunicación con el mismo servicio de base.
   - Interfaces para un mismo componente: interfaz nativa e interfaz compatible con el estándar.
5. **Consistencia no funcional**: administrar aspectos no funcionales en la integración, tales como seguridad o rendimiento y mantener e implementar políticas de monitorización.
   - Otros aspectos importantes: escalabilidad y disponibilidad (redundancia para tolerancia a fallos).

4.4. ESB: *Enterprise Service Bus*

4.4.1. Definiciones

- Estilo de arquitectura/producto software/conjunto de productos software que proporciona servicios fundamentales para arquitecturas de servicios complejas a través de un sistema de mensajes (el bus) basado en normas y que responde a eventos.

- Plataforma para la realización de una arquitectura orientada a servicios.
- "Bus de servicios de empresa": infraestructura que facilita dicha arquitectura.
- Bus: ¿SOAP+WS-Addressing?

4.4.2. Funciones

Categoría	Funciones
Invocación	soporte para protocolos de transporte síncrono y asíncrono, mapeo de servicios (localización y emparejamiento)
Enrutamiento(*Routing*)	*addressability*, encaminamiento estático/determinista, encaminamiento basado en contenidos, encaminamiento basado en normas, encaminamiento basado en políticas
Mediación	adaptadores, transformación de protocolos, mapeo de servicios
Transmisión de mensajes	procesamiento de mensajes, transformación de mensajes y mejora de mensajes
Coreografía de procesos	implementación de procesos de empresa complejos
Orquestación de servicios	coordinación de múltiples servicios de implementación presentados como un único servicio agregado
Procesamiento de eventos complejo	interpretación de eventos, correlación, emparejamiento de patrones
Otros servicios de calidad	seguridad (cifrado y firma), entrega confiable, administración de transaciones
Administración	monitorización, auditoría, registro, medición, consola de administración, BAM ("Monitorización de la actividad empresarial").

4.4.3. Características

- Agnosticismo general respecto a sistemas operativos y lenguajes de programación; por ejemplo, debería proporcionar interoperatividad entre aplicaciones Java y .NET.
- Uso general de XML como lenguaje de comunicación normalizado.
- Soporte para estándares de servicios web.
- Soporte para varios MEP (Patrones de intercambio de mensajes). Por ejemplo: petición/respuesta asíncronas, petición/respuesta síncronas, send-and-forget, publicación/suscripción.
- Adaptadores para permitir la integración con sistemas de compatibilidad, posiblemente basados en normas como la (Java-EE-Connector-Architecture/JCA).
- Un modelo de seguridad normalizado para autorizar, autenticar y auditar el uso del ESB.

4.4.4. Ventajas

- Acomodación de sistemas existentes más rápida y barata.
- Mayor flexibilidad: más fácil de cambiar si hay nuevos requisitos.
- Basado en normas.
- Posibilidad de escalar desde soluciones puntuales hasta implementaciones de empresa (bus distribuido).
- Tipos de servicio listos-para-funcionar (*ready-to-use*) predefinidos.
- Mayor configuración en vez de tener que codificar la integración.

- Sin motor de normas central, sin divisor central.
- Parches incrementales con tiempo de apagado instantáneo; la empresa se hace "refactorizable".

4.4.5. Desventajas

- Normalmente requiere un Modelo de Mensajes de Empresa, lo cual exige una administración adicional.
- Requiere una administración constante de versiones de mensajes para asegurar el pretendido beneficio de un emparejamiento flexible.
- Normalmente precisa más hardware que para un simple sistema de mensajes de punto-a-punto
- Se precisan conocimientos en el análisis de *middleware* para configurar, administrar y operar un ESB.
- Mayor latencia general causada por los mensajes que atraviesan la capa extra del ESB, especialmente si se compara con las comunicaciones punto-a-punto.
- La mayor latencia también se origina por un procesamiento de XML extra (el ESB normalmente utiliza XML como lenguaje de comunicación).
- El ESB se convierte en un elemento único de fallo.
- Aunque los sistemas de ESB pueden requerir un esfuerzo significativo para ser implementados, no producen un valor comercial sin el consiguiente desarrollo de servicios AOS para el ESB.

4.4.6. Soluciones comerciales

- MuleESB
- OpenESB (Java)
- Oracle ESB Oracle Service Bus (BEA AquaLogic Service Bus)
- Microsoft BizTalk Server
- Windows Azure Service Bus
- IBM WebSphere ESB
- JBoss ESB
- Spring Integration
- Phoenix Service Bus (C#)

4.5. SIAM: *Service Integration and Management*

- SIAM es una adaptación de ITIL que se centra en la gestión de la prestación de servicios proporcionados por múltiples proveedores. SIAM no es un proceso.
- SIAM es una capacidad de servicio y un conjunto de prácticas en un modelo y que se basan, elaboran y complementan cada parte de las prácticas ITIL.
- El objetivo principal de SIAM es proporcionar la gobernanza, la garantía y la gestión coherentes y necesarias de estos servicios múltiples. de estos múltiples proveedores y servicios, ya sean proveedores y servicios, ya sean externos, internos o una combinación de estos.

Tema 4/5: El proceso de mantenimiento de los servicios

5.1. El mantenimiento como parte de un proceso de ingeniería

Mantenimiento:

> Conjunto de técnicas destinadas a conservar equipos e instalaciones en servicio, durante el mayor tiempo posible y buscando la más alta disponibilidad, con el máximo rendimiento.

Tipos de mantenimiento:

- Adaptativo
- Preventivo
- Predictivo
- Correctivo
- Mejorativo
- De emergencia
- Legal
- ¿SAT?

5.1.1. Mantenibilidad de los servicios

> Facilidad de mantenimiento del servicio.
> Medida cualitativa de la facilidad de comprender, corregir, adaptar y/o mejorar un servicio.

Factores que influyen en la mantenibilidad:

- Falta de cuidado en la fase de diseño, codificación o prueba.
- Pobre configuración del producto software.
- Adecuada cualificación del equipo de desarrolladores del software
- Estructura del software fácil de comprender.
- Facilidad de uso del sistema.
- Empleo de lenguajes de programación y sistemas operativos estandarizados.
- Estructura estandarizada de la documentación.
- Documentación disponible de los casos de prueba.
- Incorporación en el sistema de facilidades de depuración.
- Disponibilidad del equipo hardware para realizar el mantenimiento.
- Disponibilidad de la persona o grupo que desarrolló originalmente el software.
- Planificación del mantenimiento.

5.1.2. Clasificación de mantenimiento (*software*)

- **Adaptativo**: modificar el sistema para hacer frente a cambios en el ambiente del software (DBMS, OS).
- **Perfectivo**: implementar nuevos, o cambiar requerimientos de usuario referentes a mejoras funcionales para el software.
- **Correctivo**:diagnosticar y corregir errores, posiblemente los encontraron por los usuarios.
- **Preventivo**: aumentar la capacidad de mantenimiento de software o fiabilidad para evitar problemas en el futuro.

5.1.3. Mantenimiento del Sistema de Información

Actividades:

1. Registro de la petición.
2. Análisis de la petición.
3. Preparación de la implementación de la modificación.
4. Seguimiento y evaluación de los cambios hasta la aceptación.

Entradas de información: del IAS, plan de mantenimiento y ANS; entradas externas: peticiones de mantenimiento y producto software en producción.

Salidas: Catálogo de peticiones, propuesta de solución, análisis de impacto de los cambios, plan de acción, plan de pruebas de regresión, evaluación del cambio, resultado de las pruebas.

MS1: Registro de la petición

- Objetivo
  Establecer un sistema estandarizado de registro de información para las peticiones de mantenimiento, con el fin de controlar y canalizar los cambios propuestos.
- Importante
  Asignar responsabilidades.
  Se determina el tipo de mantenimiento, los sistemas de información a los que puede afectar y se comprueba su viabilidad (cobertura según plan de mantenimiento y servicios).
- Tareas:

Tarea		Productos	Técnicas y Prácticas	Participantes
MSI 1.1	- Registro de la Petición	- Catálogo de Peticiones	- Catalogación	- Responsable de Mantenimiento
MSI 1.2	- Asignación de la Petición	- Catálogo de Peticiones: o Aceptación / Rechazo de la Petición o Asignación de Responsable	- Catalogación	- Responsable de Mantenimiento

MS2: Análisis de la petición

- Objetivo
  Diagnóstico y análisis del cambio para dar respuesta a las peticiones de mantenimiento aceptadas.

- Importante
  Alcance de la modificación del sistema. Correctivo (reproducir problema y estudiar criticidad) y evolutivo (estudiar factibilidad, impacto, política de versiones, etc.)

- Tareas:

Tarea		Productos	Técnicas y Prácticas	Participantes
MSI 2.1	Verificación y Estudio de la Petición	- Catálogo de Peticiones: - Verificación de la Petición ○ Resultado del Estudio de la Petición	- Sesiones de trabajo - Catalogación	- Equipo de Mantenimiento

MSI 2.2	Estudio de la Propuesta de Solución	- Propuesta de Solución
		- Catálogo de Peticiones:
		  ○ Estudio del Impacto
		  ○ Aceptación / Rechazo de la solución
			- Sesiones de trabajo
			- Catalogación
				- Responsable de Mantenimiento
				- Equipo de Mantenimiento

MS3: Preparación de la implementación de la modificación

- Objetivo
  Determinar qué parte del sistema de información se ve afectada, y en qué medida. Qué componentes hay que modificar (sw y hw).

- Importante
  - Establecer un plan de acción → Cumplir un plazo de entrega.
  - Establecer pruebas de regresión → Estudiar el impacto en el resto del sistema.

- Tareas:

Tarea		Productos	Técnicas y Prácticas	Participantes
MSI 3.1	- Identificación de Elementos Afectados	- Catálogo de Peticiones: ○ Elementos Afectados - Análisis de Impacto de los Cambios	- Catalogación - Análisis de Impacto	- Equipo de Mantenimiento - Jefe de Proyecto
MSI 3.2	- Establecimiento del Plan de Acción	- Catálogo de Peticiones: ○ Actividades y Tareas de los Procesos de Desarrollo a Realizar - Plan de Acción para la Modificación	- Planificación - Catalogación	- Responsable de Mantenimiento - Equipo de Mantenimiento - Jefe de Proyecto
MSI 3.3	- Especificación del Plan de Pruebas de Regresión	- Plan de Pruebas de Regresión		- Equipo de Mantenimiento - Jefe de Proyecto

MS4: Seguimiento y evaluación de los cambios hasta la aceptación

- Objetivo
  Comprobar que solo se han modificado los elementos que se ven afectados por el cambio.
  Ejecutar pruebas de regresión: de integración y de sistema.

- Tareas:

Tarea		Productos	Técnicas y Prácticas	Participantes
MSI 4.1	Seguimiento de los Cambios	- Evaluación del Cambio		- Equipo de Mantenimiento - Responsable de Mantenimiento - Jefe de Proyecto
MSI 4.2	Realización de las Pruebas de Regresión	- Resultado de las Pruebas de Regresión - Evaluación del Resultado de las Pruebas de Regresión	- Pruebas de Regresión	- Responsable de Mantenimiento - Equipo de Mantenimiento - Jefe de Proyecto
MSI 4.3	Aprobación y Cierre de la Petición	- Catálogo de Peticiones: o Nueva Versión y Aprobación	- Catalogación	- Directores de los Usuarios - Responsable de Mantenimiento

5.1.4. Participantes en el MSI

MANTENIMIENTO DE SISTEMAS DE INFORMACION	ACTIVIDADES			
	MSI 1	MSI 2	MSI 3	MSI 4
Directores Usuarios				x
Equipo de Mantenimiento		x	x	x
Jefe de Proyecto			x	x
Responsable de Mantenimiento	x	x	x	x

5.1.5. Técnicas y prácticas en el MSI

MANTENIMIENTO DE SISTEMAS DE INFORMACION	ACTIVIDADES			
	MSI 1	MSI 2	MSI 3	MSI 4
Análisis de Impacto			x	x
Catalogación	x	x	x	x
Planificación			x	
Pruebas de Regresión				x
Sesiones de Trabajo		x		

5.2. Aseguramiento de la calidad *software*

- Un *software* es **fiable** cuando las operaciones que ejecuta (en un entorno y tiempo determinado) están libres de fallo.
- Las pruebas contribuyen a mejorar la fiabilidad, pero no la garantizan.
- Son técnicas estáticas que se aplican en varios momentos del desarrollo del software y sirven para detectar defectos y ser corregidos.
- Diferentes pruebas:

- Prueba de validación: se introducen datos de prueba que sabemos que son erróneos.
- Comparación de los datos: comparar la salida del sistema a partir de unos parámetros con la salida esperada.
- Prueba de tensión: se usa el *software* de forma exhaustiva para comprobar que es capaz de soportar niveles altos de carga.
- Prueba de utilidad: los usuarios utilizan el software durante un tiempo y encuentran fallos y dificultades.

5.3. Validación y verificación: pruebas de software

5.3.1. Introducción, conceptos

Frases célebres:

- "As a rule, software systems do not work well until they have been used, and have failed repeatedly, in real applications".
- "The software isn't finished until the last user is dead".
- "Poor management can increase software costs more rapidly than any other factor".
- "The best programmers write only easy programs".
- "The trouble with programmers is that you can never tell what a programmer is doing until it's too late".
- "Fast, good, cheap: pick any two".
- "Q: How many QA testers does it take to change a lightbulb? A: QA testers don't change anything. They just report that it's dark".

Conceptos previos:

- Defecto
  Definición de datos incorrecta. Paso de procesamiento incorrecto en el programa.
- Fallo
  Incapacidad de un sistema para realizar las funciones requeridas dentro de los requisitos de rendimiento especificados.
- Error
  Resultado incorrecto, acción humana que lleva a un resultado incorrecto, la diferencia entre un valor calculado y el verdadero.

  Un *error* del programador introduce un *defecto* y este defecto produce un *fallo*.
- Verificar el *software*
  Determinar si los productos de una fase dada satisfacen las condiciones impuestas al inicio de la fase.
  ¿Estamos construyendo el producto correctamente?
- Validar el *software*
  Evaluar un sistema o uno de sus componentes, durante o al final de su desarrollo, para determinar si

satisface los requisitos.
Estamos construyendo el producto correcto

- La importancia del software y los costes asociados a un fallo motivan la creación de pruebas minuciosas y bien planificadas.
- Las organizaciones de desarrollo de software emplean entre el 30 % y el 40 % del esfuerzo total de un proyecto a las pruebas.
- Las pruebas presentan una anomalía para el ingeniero de software: durante el desarrollo se intenta construir, pero con las pruebas se intenta destruir el *software*.
- Las pruebas requieren descartar ideas preconcebidas y superar cualquier conflicto de intereses que aparezcan cuando se descubran errores.

Caso de prueba: conjunto de entradas, condiciones de ejecución y resultados esperados. Llamado **escenario de prueba**.

5.3.2. Características

- La prueba es el proceso de ejecución de un programa con el fin de descubrir un error, con poca cantidad de tiempo y esfuerzo.
- Un buen caso de prueba es aquel que tiene una alta probabilidad de encontrar un error no descubierto hasta entonces.
- Una prueba tiene éxito si descubre un error no detectado con anterioridad.

5.3.3. Consecuencias

- Hay que desechar la idea de que una prueba tiene éxito si no descubre errores.
- Las pruebas pueden usarse como una indicación de la fiabilidad del software y, de alguna manera, indican su calidad.
- Las pruebas no garantizan la ausencia de defectos.

5.3.4. Principios

1. A todas las pruebas se les debería poder hacer un seguimiento hasta los requisitos del cliente.
2. Las pruebas deberían planificarse mucho antes de que empiecen (Plan de prueba).
3. El principio de Pareto (80 % resultado/20 % esfuerzo) es aplicable.
4. Se deben incluir tanto entradas correctas como incorrectas.
5. Las pruebas deberían empezar por "lo pequeño" y progresar a "lo grande".
6. Imposibilidad de hacer unas pruebas exhaustivas.
7. Gran efectividad → realización de pruebas por equipos independientes.

8. Una buena prueba no debe ser redundante.
9. Una buena prueba no debería ser ni demasiado sencilla ni demasiado compleja.
10. Las pruebas se deben documentar detalladamente.

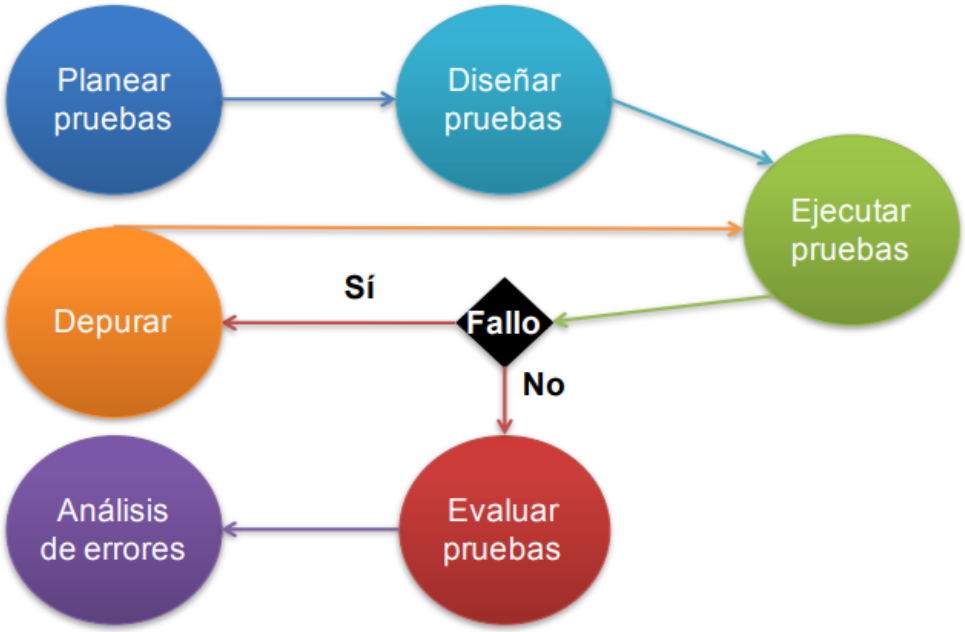
5.3.5. Facilidad de prueba

Características de un software "fácil de probar":

- Operatividad
  Cuanto mejor funcione, más eficiente se puede probar.
  - El sistema tiene pocos errores.
  - Ningún error bloquea la ejecución de las pruebas.
  - El producto evoluciona en fases funcionales.
- Observabilidad
  Lo que ves es lo que pruebas.
  - Se genera una salida distinta para cada entrada.
  - Los estados y variables del sistema están visibles.
  - Todos los factores que afectan a los resultados están visibles.
  - Un resultado incorrecto se identifica fácilmente.
  - Se informa automáticamente de los errores internos.
  - El código fuente es accesible.
- Controlabilidad
  Cuanto mejor podemos controlar el software, más se puede automatizar y optimizar.
  - Se pueden controlar directamente los estados y las variables.
  - Los formatos de entradas y resultados son consistentes y estructurados.
  - Capacidad de descomposición.
- Capacidad de descomposición
  Controlando el ámbito de las pruebas, podemos aislar más rápidamente los problemas y llevar a cabo mejores pruebas de regresión.
  - El sistema software está construido con módulos independientes.
  - Los módulos del software se pueden probar independientemente (cohesión y acoplamiento).
- Simplicidad
  Cuanto menos haya que probar, más rápidamente podremos probarlo.
  - Simplicidad funcional.
  - Simplicidad estructural.
  - Simplicidad del código.

- Estabilidad
  Cuanto menos cambios, menos interrupciones a las pruebas.
  - Los cambios del software son infrecuentes.
  - Los cambios del software están controlados.
  - Los cambios del software no invalidan las pruebas existentes.
  - El software se recupera bien de los fallos.
- Facilidad de comprensión
  Cuanta más información tengamos, más inteligentes serán las pruebas.
  - La documentación técnica está bien organizada.
  - La documentación técnica es específica y detallada.
  - La documentación técnica es exacta.

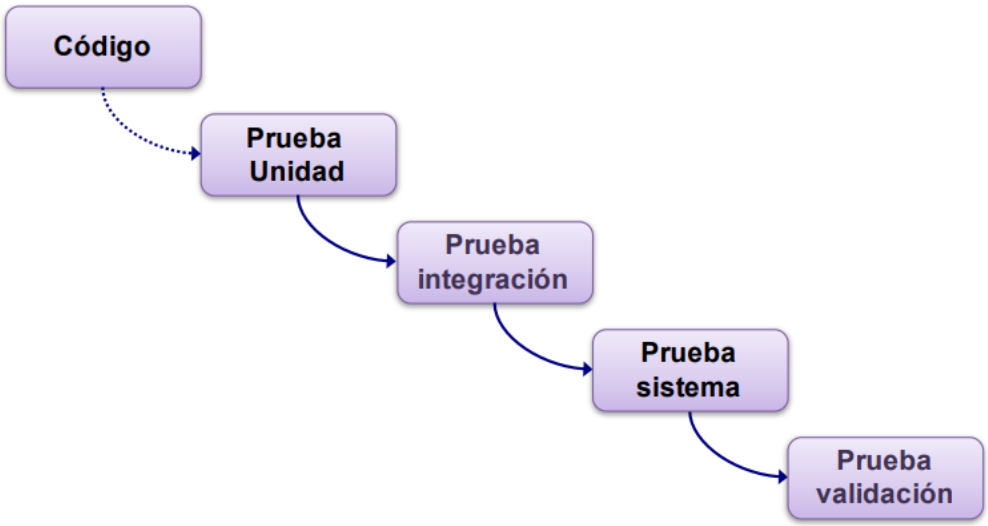
5.3.6. Proceso de pruebas de software



Diseño de pruebas:

- Probar exhaustivamente un *software* es imposible.
- Necesitamos técnicas que nos ayuden a confeccionar casos de prueba que me aseguren la evaluación eficiente del software.
- La mejor técnica es aquella que con menos casos de prueba me lo certifique.

5.3.7. Ciclo de pruebas



Pruebas de unidad

Pruebas de unidad > Enfoque de Caja Blanca

- Es un método de diseño de casos de prueba que usa la estructura de control del diseño procedimental para obtener los casos de prueba.
- Casos de prueba que:
  - Garanticen que se ejercita por lo menos una vez todos los caminos independientes de cada módulo.
  - Se ejerciten todas las decisiones lógicas.
  - Ejecuten todos los bucles en sus límites.
  - Ejerciten las estructuras internas de datos para asegurar la validez
- Procedimiento de diseño de prueba:
  1. Dibujar el grafo de flujo.
  2. Calcular la complejidad ciclomática.
  3. Determinar el conjunto de caminos independientes.
  4. Preparar el caso de prueba para cada camino.

Pruebas de unidad > Enfoque de Caja Blanca >Prueba del Camino básico

- Permite obtener una medida de la complejidad lógica procedimental.
- Se debe generar un caso de prueba para cada camino de ejecución.
- Los casos de prueba garantizan que se ejecuta por lo menos una vez cada sentencia del programa.
- Elementos:

- Notación de grafo de flujo:
  - Representa el flujo de control lógica mediante la notación ilustrada.
  - Un solo nodo puede corresponder a una secuencia de **cuadros** de proceso y a un **rombo** de decisión.
  - Las flechas del grafo de flujo, denominadas **aristas** o enlaces, representan flujo de control y son análogas a las flechas del diagrama de flujo.
  - Los nodos se agrupan en **regiones**.
- Cálculo de la complejidad ciclomática:
  - Es una métrica del *software* que proporciona una medición cuantitativa de la complejidad lógica de un programa.
  - Su valor define el número de caminos independientes del conjunto básico de un programa.
  - Límite superior para el número de pruebas que se deben realizar para asegurar que se ejecuta cada sentencia al menos una vez.
  - El camino independiente es cualquier camino del programa que introduce, por lo menos, un nuevo conjunto de sentencias de proceso o una nueva condición.
  - Es una métrica útil para predecir los módulos que son más propensos a error. Puede ser usada tanto para planificar pruebas como para diseñar casos de prueba
- CAMINO: Por donde el flujo de ejecución debe pasar. Se descubren mediante el diagrama de flujo.

Pruebas de unidad > Enfoque de Caja Blanca >Prueba de la estructura de control

- Aunque la prueba del camino básico es sencilla y altamente efectiva, no es suficiente por sí sola, por lo que las pruebas de estructura de control.
- Amplían la cobertura de la prueba y mejoran la calidad de la prueba de caja blanca.
- Tipos de pruebas de estructura de control, usando como criterio el grado de cobertura (por qué sentencias se debe pasar para considerar que se ha pasado la prueba):
  - Prueba de condición:
    - Es un método de diseño de casos de prueba que ejercita las condiciones lógicas contenidas en el módulo de un programa.
    - Una condición simple es una variable lógica o una expresión relacional.
    - Si una condición es incorrecta, entonces es incorrecto al menos un componente de la condición.
    - Tipos de errores de una condición: de operador lógico, en variable lógica, en paréntesis lógico, en operador relacional, en expresión aritmética.
  - Prueba de flujo de datos: selecciona caminos de prueba de un programa de acuerdo con la ubicación de definiciones y los usos de las variables del programa.
  - Prueba de bucles: Es una técnica de prueba de caja blanca que se centra exclusivamente en la validez de las construcciones de bucles.

Se pueden definir cuatro clases diferentes de bucles: simples, concatenados, anidados, no estructurados.

Pruebas de unidad > Enfoque de Caja Negra

Pruebas de unidad > Enfoque de Caja Negra > Principios

- También denominada prueba de comportamiento.
- Se centran en los requisitos funcionales del software.
- Permite obtener conjuntos de condiciones de entrada que ejerciten completamente todos los requisitos funcionales de un programa.
- NO es alternativa a las pruebas de caja blanca. Se trata de un enfoque complementario.
- Errores que intenta encontrar:
  - Funciones incorrectas o ausentes.
  - Errores de interfaz.
  - Errores en estructuras de datos o en accesos a bases de datos externas.
  - Errores de rendimiento.
  - Errores de inicialización y de terminación.
- Tiene a aplicarse durante fases posteriores.
- Ignora intencionadamente la estructura de control.
- Se obtiene un conjunto de casos de prueba que satisfacen los siguientes criterios:
  - Casos de prueba que reducen el número de casos de prueba adicionales.
  - Obtiene info sobre la presencia o ausencia de clases de errores.

Pruebas de unidad > Enfoque de Caja Negra > Técnicas

- Métodos de prueba basados en grafos
- Partición en clases de equivalencia
  - DEF. Clase de equivalencia: representa un conjunto de estados válidos o no válidos para condiciones de entrada.
  - Divide el campo de entrada de un programa en clases de datos de los que se pueden derivar casos de prueba.
  - Puede descubrir de forma inmediata una clase de errores (por ejemplo, proceso incorrecto de todos los datos de carácter) que, de otro modo, requerirían la ejecución de muchos casos antes de detectar el error genérico.
  - El diseño de casos de prueba se basa en una evaluación de las clases de equivalencia para una condición de entrada.

    - Una clase de equivalencia representa un conjunto de estados válidos o no válidos para condiciones de entrada
- Análisis de valores límite
    - Técnica complementaria a la partición en clases de equivalencia.
    - Los errores tienden a darse más en los límites del campo, que en valores centrales.
    - Explora las condiciones límites de un programa.
    - No solo se fija el dominio de entrada sino también el dominio de salida.
- Pruebas de comparación
- Prueba de tabla ortogonal
- Conjetura de errores: enumerar una lista de errores comunes, en función elaborar casos de prueba.
- Enfoque aleatorio: se crean datos que sigan frecuencia y secuencia que tendrían en la práctica. Se simula la entrada de datos. Se usan generadores de pruebas.

Pruebas de unidad > Consideraciones finales

- Sobre la evaluación de la manipulación de errores:
    - Un buen diseño exige que las condiciones de error sean previstas de antemano.
    - Existe cierta tendencia a incorporar manipulación de errores, pero nunca probarlo.
    - Errores potenciales de la manipulación de errores:
        - Descripción ininteligible del error.
        - El error señalado no es el error encontrado.
        - El procesamiento de la condición excepcional es incorrecto.
        - La descripción del error no proporciona información suficiente para ayudar en la localización de la causa del error.
- Lo que es imprescindible saber de las pruebas de unidad:
    - Centran el proceso de verificación en la menor unidad del diseño del software: el componente software o módulo.
    - Se prueba la interfaz del módulo para asegurar que la información fluye de forma adecuada hacia y desde la unidad de programa.
    - Se examinan las estructuras de datos para asegurar que los datos conservan su integridad
    - Se ejercitan todos los caminos independientes (caminos básicos)
    - Se prueban las condiciones límite. La prueba de límites es la última (y probablemente, la más importante) tarea del paso de la prueba de unidad.
    - Finalmente, se prueban todos los caminos de manejo de errores.
    - Errores detectables con pruebas de unidad:

    - Comparaciones ente tipos de datos distintos.
    - Operadores lógicos o de precedencia incorrectos.
    - Igualdad esperada cuando los errores de precisión la hacen poco probable.
    - Variables o comparaciones incorrectas.
    - Terminación de bucles inapropiada o inexistente.
    - Fallo de salida cuando se encuentra una iteración divergente.
    - Variables de bucles modificadas de forma inapropiada.
- Errores más comunes durante la prueba de unidad:
    - Precedencia aritmética incorrecta o malinterpretada.
    - Límites de valores no contemplados correctamente.
    - Inicializaciones incorrectas.
    - Falta de precisión.

Pruebas de integración

- Técnica sistemática para construir la estructura del programa.
- El objetivo es coger los módulos probados mediante la prueba de unidad y construir una estructura de programa que esté de acuerdo con lo que dicta el diseño.
- Debe ser conducida incrementalmente: el programa se construye y se prueba en pequeños segmentos, así los errores son más fáciles de aislar y corregir.
- Enfoque: descendente o ascendente.
- Pruebas: regresión y humo.

Pruebas de integración > Enfoque de Integración descendente

- Es un planteamiento incrementa para la construcción de la estructura de programas.
- Se integran los módulos moviéndose hacia abajo por la jerarquía de control, comenzando por el módulo de control principal y añadiendo los módulos subordinados en algún orden concreto.
- Estrategia *top-down*: integra todos los módulos de un camino de control principal de la estructura. La selección del camino principal es arbitraria y dependerá de las características específicas de la aplicación

Pruebas de integración > Enfoque de Integración ascendente

- Es otro planteamiento incremental a la construcción de la estructura de programas.
- Comienza la construcción y la prueba con los módulos atómicos.
- El proceso requerido de los módulos subordinados siempre está disponible y se elimina la necesidad de resguardos.

Pruebas de integración > Pruebas de regresión

- Definiciones:
  - Cualquier tipo de pruebas de software que intentan descubrir nuevos errores inducidos por cambios recientemente realizados en partes de la aplicación que anteriormente al citado cambio no eran propensas a este tipo de error.
  - Volver a ejecutar un subconjunto de pruebas que se han llevado a cabo anteriormente para asegurarse de que los cambios no han propagado efectos colaterales no deseados.
  - La prueba de regresión es la actividad que ayuda a asegurar que los cambios no introducen un comportamiento no deseado o errores adicionales
- Cada vez que se añade un nuevo módulo como parte de una prueba de integración, el software cambia.
- Estos cambios pueden causar problemas con funciones que antes trabajaban perfectamente.
- La prueba de regresión es una estrategia importante para reducir efectos colaterales. Se deben ejecutar pruebas de regresión cada vez que se realice un cambio importante en el software (incluyendo la integración de nuevos módulos).
- Tres clases, según qué se prueba: todas las funciones, las funciones afectadas por el cambio, los componentes que han cambiado.
- Tipos de regresión según el cambio: Local (errores en un módulo concreto), Desenmascarada (revelan errores previos) y Remota (los cambios vinculan algunas partes del programa (módulo) e introducen errores en ella).
- A medida que progresa la prueba de integración, el número de pruebas de regresión puede crecer.

Pruebas de integración > Pruebas de humo

- Método de prueba de integración.
- Comúnmente utilizada para software «empaquetado».
- Es diseñado como un mecanismo para proyectos críticos por tiempo.
- Diseñado para cubrir una amplia porción de la funcionalidad del sistema.
  Las pruebas así diseñadas dan prioridad al porcentaje del total del sistema que se pone a prueba, sacrificando la precisión de estas de ser necesario.
- Actividades:
  - Los componentes software que han sido traducidos a código se integran en una «construcción» (ficheros de datos, librerías... Todo lo necesario para implementar una o más funciones del producto).
  - Se diseña una serie de pruebas para descubrir errores que impiden a la construcción realizar su unción adecuadamente. Objetivo: descubrir errores «bloqueantes».
  - Es habitual en la prueba de humo que la construcción se integre con otras construcciones y que se aplica una prueba de humo al producto completo.

- La prueba de humo ejercita el sistema entero de principio a fin. No ha de ser exhaustiva, pero será capaz de descubrir importantes problemas.
- Beneficios:
  - Se minimizan los riesgos de integración.
  - Se perfecciona la calidad del producto final.
  - Se simplifican el diagnóstico y la corrección de errores
  - El progreso es fácil de observar

Pruebas de integración > Conclusiones

- Ventajas y desventajas:
  - Las ventajas de una estrategia tienden a convertirse en desventajas para la otra estrategia
  - La principal desventaja del enfoque descendente es la necesidad de resguardos y las dificultades de prueba que pueden estar asociados con ellos
  - La principal desventaja de la integración ascendente es que el programa como entidad no existe hasta que se ha añadido el último módulo.
- La selección de una estrategia depende de las características del software y de la planificación del proyecto
- En general, es recomendable usar un enfoque combinado (prueba sandwich). Descendente para niveles superiores; ascendente para niveles subordinados.
- Identificación de módulos críticos: dirigido a varios requisitos del software, mayor nivel de control (estructura alta), complejo o propenso a errores, tiene unos requisitos de rendimiento muy definidos.
- Los módulos críticos deben probarse lo antes posible.
- Las pruebas de regresión deben centrarse en funcionamiento de los módulos críticos.

Pruebas de sistema

Al final el *software* es incorporado a otros elementos del sistema: Hardware, Información, Software de base, Personas, Procesos no sistematizados...

Se realizan una serie de pruebas de integración del sistema y de validación:

- Probar toda la información procedente de otros elementos del sistema.
- Pruebas que simulen la presencia de datos en mal estado o posibles errores en la interfaz.
- Los resultados son evidencias para errores en sistemas externos.

Pruebas de sistema > Prueba de recuperación

- Recuperación ante fallos y continuar el proceso en un tiempo previamente especificado

- MTTR (*Mean/Minimum/Maximum Time to Recovery/ Repair/Respond/Restore*).
- La prueba de recuperación es una prueba del sistema que fuerza el fallo del software de muchas ormas y verifica que la recuperación se lleva a cabo apropiadamente.

Pruebas de sistema > Prueba de seguridad

- Entradas impropias o ilegales al sistema.
- Incluye un amplio rango de actividades: piratas informáticos, empleados disgustados, entre otros.
- Intenta verificar que los mecanismos de protección incorporados protegerán el sistema.
- Durante la prueba, todo vale. Es necesario probar todas las opciones.
- Es imposible la seguridad total. Se intenta que el coste de la entrada ilegal sea menor que la información que se pueda obtener. De este modo, no merece la pena para el agresor.

Pruebas de sistema > Prueba de resistencia (*stress*)

- Las pruebas de resistencia están diseñadas para enfrentar a los programas con situaciones anormales.
- Se busca responder a la pregunta: *¿A qué potencia puedo ponerlo a funcionar antes de que falle?*

Pruebas de sistema > Prueba de rendimiento

- La prueba de rendimiento está diseñada para probar el rendimiento del software en tiempo de ejecución dentro del contexto de un sistema integrado.
- Se da durante todos los pasos del proceso de la prueba (incluso al nivel de unidad).
- Hasta que no están completamente integrados no se puede asegurar realmente el rendimiento del sistema.
- A menudo, van emparejadas con las pruebas de resistencia.

Pruebas de validación

Principios de las pruebas de validación:

- La validación se consigue cuando el software funciona de acuerdo con las expectativas razonables del cliente.
- Las expectativas razonables están definidas en la Especificación de Requisitos del Software.
- La especificación debería contener una sección denominada «Criterios de validación».
- Como en otras etapas de la prueba, la validación permite descubrir errores, pero su enfoque está en el nivel de requisitos — sobre cosas que son necesarias para el usuario final.

- La validación del software se consigue mediante una serie de pruebas de caja negra que demuestran la conformidad con los requisitos.
- Tanto el plan como el procedimiento estarán diseñados para asegurar: Que se satisfacen todos los requisitos funcionales, que se alcanzan todos los requisitos de rendimiento, que la documentación es correcta y entendible, que se alcanzan otros requisitos (ej. portabilidad).

Pruebas de validación > Técnica: Revisión de la configuración

- Asegurarse de que todos los elementos de la configuración del software se han desarrollado apropiadamente.
- Se han catalogado y están suficientemente detallados para soportar la fase de mantenimiento durante el ciclo de vida del software.
- La revisión de la configuración a veces se denomina auditoría.

Pruebas de validación > Técnica: Pruebas alfa y beta

- Es virtualmente imposible que un desarrollador de software pueda prever cómo utilizará el usuario realmente el programa.
- Pruebas alfa:
  - Se lleva a cabo, por un cliente, en el lugar de desarrollo.
  - Se usa el software deforma natural con el desarrollador como observador del usuario (registrando los errores y problemas de uso).
  - Las pruebas alfa se llevan a cabo en un entorno controlado.
- Pruebas beta:
  - Se lleva a cabo por los usuarios finales del software en los lugares de trabajo de los clientes.
  - A diferencia de la prueba alfa, el desarrollador no está presente.
  - Es una aplicación «en vivo» del software en un entorno que no puede ser controlado por el desarrollador.

Depuración

- Ocurre como consecuencia de una prueba efectiva cuando un caso de prueba descubre un error.
- La depuración no es una prueba, pero ocurre siempre como consecuencia de la prueba.
- El objetivo se consigue mediante una combinación de una evaluación sistemática, de intuición y de suerte