Arquitectura de Software

Resumen

Primer Parcial

(Cap. 1-2-4-5 libro)

Matthias Rauhut

157107

Índice

Introducción

El Arquitecto de Software

Influencias sobre los Arquitectos

El arquitecto es influenciado por:

1. **Los Stakeholders**
   Existen 5 stakeholders básicos de un producto de software, cada uno con sus **intereses particulares, muchos de los cuales pueden ser contradictorios**. El arquitecto **debe priorizar y tratar de satisfacer a todos.** Los stakeholders son**:**
   a. El Cliente
   b. El Usuario final
   c. Los Desarrolladores
   d. El Gerente de Proyecto
   e. Los mantenedores
2. **La Organización Desarrolladora**
   Cada organización tiene sus empleados que pueden tener diferentes:
   a. Conocimientos
   b. Presupuestos
   c. Horarios

   Sin embargo, existen 3 clases de influencia de las Organizaciones:

   a. **Negocio inmediato**: La organización puede tener arquitecturas existentes y productos basadas en ellas que desean desarrollar.
   b. **Negocio a largo plazo:** La organización puede querer aprovechar la oportunidad para desarrollar el Software bajo una infraestructura mantenida por la misma, a fin de perseguir objetivos estratégicos.
   c. **Estructura Organizacional:** La organización puede tener una estructura de empleados que incluye subcontratar gente para hacer subsistemas ya que determinados grupos poseen habilidades y conocimientos especiales.
3. **Experiencia del Arquitecto**
   Si un arquitecto recibió entrenamiento especial o ha tenido mejores experiencias con determinadas herramientas, metodologías o patrones, es más propenso a querer volver a implementarlos.
4. **Entorno Técnico**
   Referente a lo que “se usa en la actualidad”. Nadie en su sano juicio va a utilizar tecnología en desuso u obsoleta.

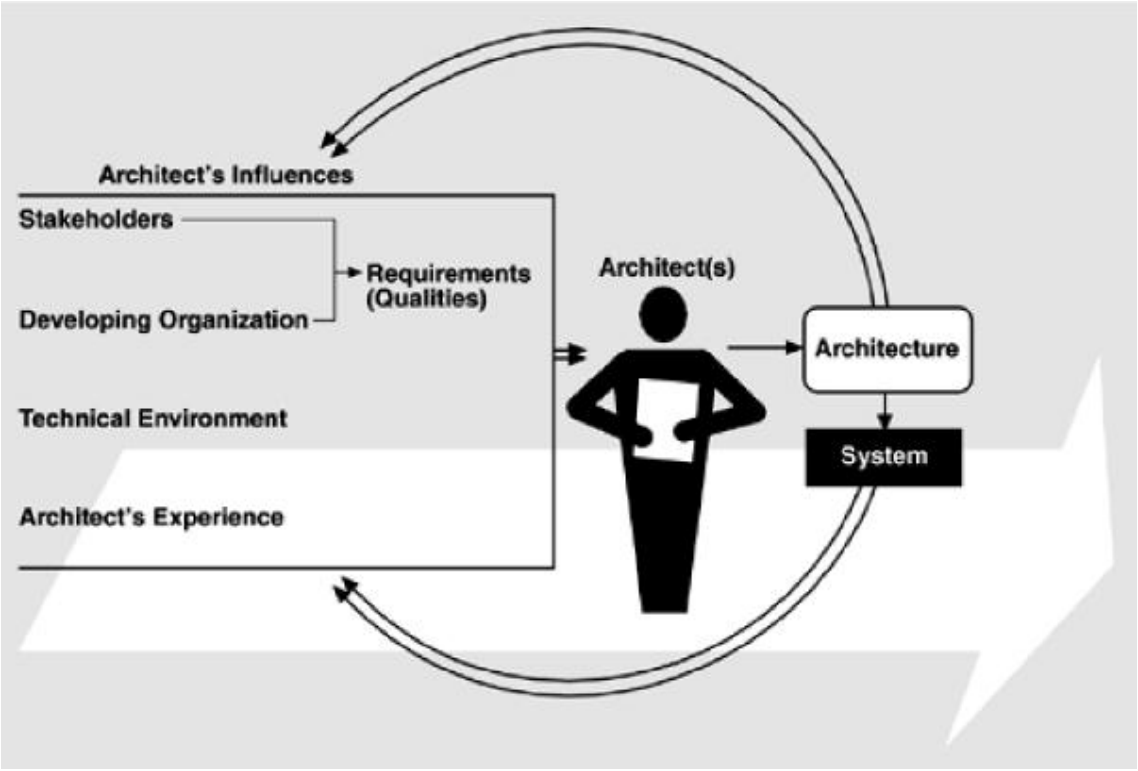
Ramificaciones en las influencias sobre el Arquitecto

Casi nunca los requerimientos de todos los stakeholders y la organización son contemplados ni entendidos por completo. Es trabajo del arquitecto:

- Entender los requisitos de la tarea
- Gestionar las expectativas
- Negociar prioridades
- Negociar

Para esto, deben **interactuar con todos los stakeholders lo más tempranamente posible**, ya que cambios tardíos pueden tener un impacto considerable, tanto en tiempo como costos y viabilidad. Aquí se obtienen los Requerimientos o Cualidades del producto.

El Ciclo de Influencias (La Arquitectura influye en las influencias)



1. **La Arquitectura afecta la estructura organizacional**
   El arquitecto define unidades de software a desarrollar. Muchas veces estas son llevadas a cabo individualmente por diferentes grupos de desarrollo. De esta forma la arquitectura influye sobre la estructura organizativa.
2. **La Arquitectura afecta los objetivos de la organización**
   La arquitectura puede abrir las puertas para la producción eficiente y depolyment de sistemas similares. La Organización puede sacar ventaja de la experiencia de un sistema exitoso.

3. **La Arquitectura afecta a los requerimientos del cliente**
   Al ofrecerle la oportunidad de recibir un sistema basado en la misma arquitectura de un sistema exitoso, el cliente se puede ver tentado a modificar sus requerimientos a cambio de un sistema más confiable, económico y de rápido desarrollo, a que si se hiciera a medida desde cero.
4. **La Arquitectura afecta a la experiencia del Arquitecto**
   Sumando experiencias positivas o negativas en torno a determinadas arquitecturas.
5. **La Arquitectura afecta al entorno técnico**
   Algunos sistemas basados en arquitecturas parecidas, si son exitosos, pueden cambiar la cultura de la Ingeniería de Software, o sea, el entorno técnico en el que se desarrollan, operan y aprenden, los desarrolladores de sistemas.

El Proceso de Software y el ciclo del Negocio de Arquitectura

Se le llama **Proceso de Software** a la organización, sistematización y mantenimiento de actividades de desarrollo de software.

Las actividades incluyen (se verán en detalle):

- Crear un caso de negocio para el sistema
- Comprender los requerimientos
- Crear o seleccionar la arquitectura
- Documentar y comunicar la arquitectura
- Analizar o evaluar la arquitectura
- Implementar el sistema basándose en la arquitectura
- Asegurar que la implementación es conforme a la arquitectura

Crear un caso de negocio para el sistema

Responde a las siguientes preguntas:

- ¿Cuánto costará?
- ¿Cuál es el mercado objetivo?
- ¿Cuál es el tiempo estimado para el lanzamiento al mercado?
- ¿Tendrá interferencia con otros sistemas?
- ¿Hay limitaciones de sistema sobre las cuales trabajará?

Comprender los requerimientos

Existen varias técnicas para validar los requerimientos:

- Creación de Casos de Uso
- Creación de modelos de máquinas de estado (sistemas de riesgo)
- Creación de Prototipos
- Etc.

Independientemente de la técnica usada, las cualidades del sistema a ser construido, determina la forma de la arquitectura.

Creación o Selección de la Arquitectura

Según "The Mythical Man-Month", de Fred Brooks:

> La **integridad conceptual es la clave** para cualquier diseño de software. Esta integridad solo puede ser **alcanzada y mantenida por un reducido número de mentes** confluentes en el diseño de la arquitectura del sistema.

Comunicación de la Arquitectura

Para que la arquitectura sea un **backbone del diseño del proyecto efectivo**:

- Debe ser **comunicada claramente y sin ambigüedades** a todos los stakeholders.
- La **documentación debe ser informativa, no ambigua y legible** por personas de diferentes áreas y contextos.

Analizar o Evaluar la Arquitectura

Mayormente, siempre habrá varios diseños candidatos que analizar y evaluar. Elegir entre dichos candidatos de manera racional es uno de los mayores retos para los arquitectos.

Uno de os más generales y eficaces acercamientos para evaluar la arquitectura es mediante escenarios.

Implementar basándose en la Arquitectura

Se trata de mantener a los desarrolladores fieles a las estructuras y protocolos de integración contenidos en la arquitectura.

Asegurar que la implementación es conforme a la arquitectura

Se logra mediante constante vigilancia para asegurar que la arquitectura implementada y la representación se mantienen fieles entre ellas durante esta fase.

¿Qué hace a una buena arquitectura?

NO existe algo así como una inherentemente **buena o mala arquitectura**. **Las Arquitecturas son mejores o peores para determinado propósito.**

Sin embargo, existen algunos **indicadores** que pueden ser síntoma de una arquitectura deficiente. Estos se dividen en 2 partes:

1. Recomendaciones de proceso
2. Recomendaciones estructurales o "de producto"

Recomendaciones de Proceso

1. **La arquitectura** debe ser **producto de un solo arquitecto o un grupo reducido de arquitectos.**
2. **La arquitectura** debe contener los **requerimientos funcionales** y **una lista priorizada de atributos de calidad**.
3. **La arquitectura** debe estar **bien documentada con al menos una vista dinámica y una estática** (en una nomenclatura y representación acordada por los stakeholders).
4. **La arquitectura** debe ser circulada por los **stakeholders, involucrados activamente en su revisión.**
5. **La arquitectura** debe ser analizada en lo que refiere a **atributos cuantitativa y cualitativamente medibles.**
6. **La arquitectura** debe prestarse para ser **implementada de manera incremental.** Esto es**, mediante la creación de un "Skeletal-System".**
7. **La arquitectura** debe resultar en un pequeño pero especifico s**et de contención de recursos**. Este debe ser debidamente especificado, circulado y mantenido.

Recomendaciones Estructurales o "de producto"

1. **La arquitectura** debe **disponer de módulos bien definidos** y **basados en el principio del Information-Hiding y Separación de Responsabilidades.**

2. **Cada módulo** debe tener **Interfaces bien definidas que encapsulen o escondan aspectos modificables** (como implementación o elección de estructuras de datos).

3. **Atributos de calidad** debe ser alcanzados mediante el empleo de **tácticas de arquitectura bien conocidas para cada atributo.**

4. Dentro de lo posible, **la arquitectura no debe depender de una versión particular de un producto o herramienta comercial.** En los casos donde este sea el caso, se debe prever un mecanismo de cambio viable y barato.

5. **Módulos que producen datos** deben **encontrarse separados** de **los que los consumen**. Esto tiende a mejorar la modificabilidad.

6. Para arquitecturas de procesamiento paralelo, se debe definir bien los procesos o tareas que no necesariamente espejan la estructura de descomposición de módulos.

7. Cada **proceso o tarea** debe ser escrita de tal manera que **su asignación a un procesador específico puede ser cambiada con facilidad**.

8. **La arquitectura** debe tener un **set pequeño y simple de patrones de interacción**.

La Arquitectura de Software

Introducción

La Arquitectura de Software de un programa o sistema computacional es la **estructura de estructuras del sistema**, que **contienen los elementos del software**, **las propiedades externamente visibles** de dichos elementos y **las relaciones entre los mismos**.

De esta definición se desprende:

1. La arquitectura es una **abstracción del sistema** que **ignora** los **elementos que no tienen incidencia** en:
    - **Como son usados** los elementos
    - **Por quien** son usados
    - Se **relacionan**
    - **Interactúan**

2. **La arquitectura es una estructura de estructuras**. Razón por la cual ninguna estructura puede hacerse llamar "la arquitectura".
    - **Cada estructura hace referencia a un aspecto diferente de la arquitectura del sistema.**

3. **Todo sistema computacional tiene una arquitectura de software.** Esto no significa que la misma se desprenda del sistema. Desgraciadamente, la arquitectura puede existir independiente de su descripción o especificación, lo que resalta la importancia de la documentación de arquitectura.

4. **El comportamiento de cada elemento es parte de la arquitectura.** Además, este comportamiento puede ser visto analizado del punto de vista de otro elemento. Este comportamiento es el que permite la interacción, que claramente es parte de la arquitectura.

5. **La definición de arquitectura es indiferente de si la arquitectura es buena o mala para determinado sistema.** Esto implica que independiente de la elección de arquitectura para un sistema, sigue siendo una arquitectura para dicho sistema y la mejor manera de elegirla es mediante ensayo y error, apoyado en diferentes factores. Esto remarca la importancia de los métodos de evaluación de arquitecturas.

Otros puntos de vista

A continuación, se detallan otras características de la Arquitectura de Software:

1. **La Arquitectura es diseño de alto nivel**. Sin embargo, no vale el recíproco. Esto se debe a que otras tareas asociados con el diseño no son arquitectónicas, sino de implementación (la interfaz de dichos elementos son de perspectiva arquitectónica, sin embargo su elección e implementación no lo son).

2. **La Arquitectura es la estructura general del sistema.** No significa que la arquitectura sea una estructura, sino que la estructura de estructuras que representa el sistema es de lo que se compone la arquitectura.

3. **La Arquitectura es la estructura de componentes de un programa o sistema, sus relaciones internas y los principios y guías que dictan su diseño y evolución a lo largo del tiempo.**

4. **La Arquitectura es componentes y conectores.**

Patrones de Arquitectura, Modelos de Referencia y Arquitecturas de Referencia

Entre cada modelo, existen etapas intermedias que reflejan decisiones de arquitectura. Estas decisiones se toman en base a ciertas estructuras arquitectónicas. Definiremos 3 de ellas:

1. **Patrón de Arquitectura**

   Es una descripción de elementos y tipos de relación junto con un set de restricciones acerca de cómo pueden ser usados.

   Un ejemplo de patrón muy usado es Cliente – Servidor.

   Importante: Un patrón no es una arquitectura, sino que converge en una imagen útil del sistema, imponiendo restricciones útiles en la arquitectura, y por ende, en el sistema.

   Otro término para denotar los patrones de arquitectura, es **estilos de arquitectura**. Cada patrón tiene asociado su impacto en atributos de calidad conocidos.

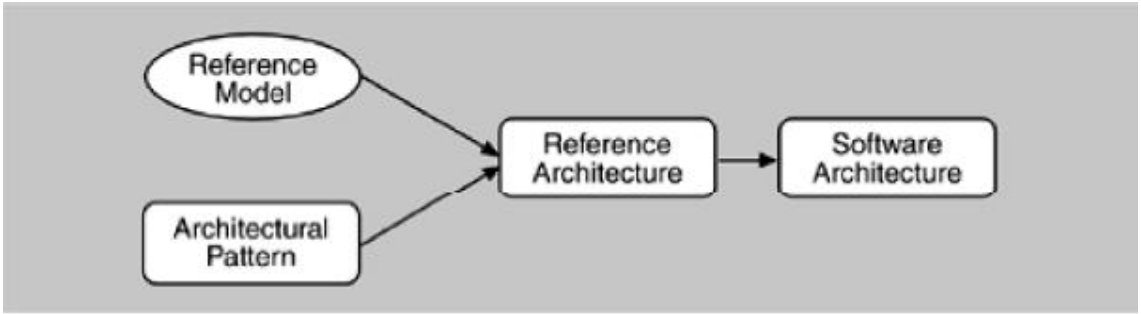
2. **Modelo de Referencia**

   Es una división de funcionalidad junto con el data flow entre las partes.

   El modelo de referencia es una descomposición estándar de un problema conocido en partes que cooperativamente solucionan el problema.

3. **Arquitectura de Referencia**

   Una arquitectura de referencia es un Modelo de Referencia mapeado con elementos del software y el data flow entre ellos.



La Importancia de la Arquitectura de Software

Existen 3 principales razones por las cuales la arquitectura importa a nivel técnico (además delas implicancias que vimos en la Introducción, a nivel empresarial). Estos son:

1. **Comunicación entre Stakeholders**
   Una correcta arquitectura es la base para el:
   a. Mutuo entendimiento
   b. Negociación
   c. Consenso
   d. Comunicación

2. **Decisiones de diseño tempranas**
   Es el punto de partida de todo software y también la primera instancia de análisis del sistema, del cual se podrían desprender cambios que a la larga resultarían inviables o demasiado costosos.

3. **Abstracción transferible del sistema**
   Representa un componente altamente reusable.

A continuación veremos estos puntos en detalle.

Comunicación entre Stakeholders

Como vimos, una correcta arquitectura es la base para el:

a. Mutuo entendimiento
b. Negociación
c. Consenso
d. Comunicación

Además, de la arquitectura se desprenden **debates acerca de temas no arquitectónicos, que muchas veces llevan al intercambio entre stakeholders, cuyo consenso es fundamental para el sistema.**
Un ejemplo para esto es la aclaración de requerimientos.

Decisiones de diseño tempranas

La arquitectura de software es el **resultado de las primeras decisiones de diseño**. Normalmente, estas decisiones son las **más difíciles de cambiar a largo plazo** y son las que **conllevan mayor impacto en el proyecto en general**.

Define las restricciones de implementación

Dividen la implementación del sistema en elementos con determinadas restricciones. Estas restricciones permiten la **toma de decisiones de gestión que resultaran en un mejor manejo del personal y capacidad computacional**.

Constructores de elementos del sistema **deben ser expertos en la especificación de su elemento en particular**, pero no en los intercambios a nivel arquitectónico (lo opuesto a los arquitectos).

Define la Estructura Organizacional

Como vimos, **la implementación debe separarse en elementos a ser desarrollados individualmente por diferentes grupos de trabajo**. Como la **arquitectura es la abstracción del sistema de más alto nivel**, es el **punto de partida** para dicha repartición.

En la **etapa de mantenimiento**, esta **repartición también se llevará a cabo** y **cada equipo se encargará de un elemento estructural diferente**.

Inhibe o Habilita ciertos Atributos de Calidad

Ciertas decisiones de diseño afectan la capacidad del sistema de cumplir con determinados requisitos de calidad.

Por ejemplo, modularizar el sistema hace con que se gane en Modificabilidad, sin embargo abre brechas de Seguridad.

Predice los Atributos de Calidad del Sistema

Ingeniería reversa del punto anterior. En base a la Arquitectura es posible, previo a la construcción del sistema, saber cuáles serán sus puntos fuertes.

Permite gestionar mejor el cambio

Como la mayor parte de la vida útil de un sistema corresponde al período luego del lanzamiento, es aquí donde se originan la mayor parte de los costos. Por esta razón la arquitectura ayuda a manejar mejor los cambios en esta etapa.

Los cambios pueden ser de 3 tipos:

- **Locales**
  - Cuando con el cambio de un solo elemento ya se soluciona.
- **No Locales**
  - Cuando requiere el cambio de varios elementos, pero deja la arquitectura intacta.
- **Arquitectónico**
  - Son los más complejos. Afecta las maneras fundamentales por las cuales los elementos interactúan entre sí, el patrón de arquitectura y probablemente requiera de cambios a nivel de todo el sistema.

La arquitectura permite **gestionar mejor los cambios de esta manera**:

- **Decide cuando** los cambios **son necesarios**
- **Determina** qué cambio conlleva el menor **riesgo**
- **Mide las consecuencias** de los cambios propuestos
- **Arbitra la prioridad** de los cambios teniendo en cuenta los **beneficios** en determinados **atributos de calidad**

Ayuda en la prototipación evolutiva

Una vez que se defina la arquitectura, **se puede analizar y prototipar como un "Skeletal-System"**. Esto tiene las siguientes 2 ventajas:

- El **sistema es ejecutable en una etapa más temprana** del desarrollo
  - Mediante la implementación de módulos de baja fidelidad o sustitutos que simulan comportamiento y serán luego suplantados por las versiones finales.
- Se **detectan potenciales problemas de performance en una etapa más temprana** del desarrollo.

Estos beneficios **derivan en la disminución del riesgo del proyecto**.

Ayuda en la estimación de tiempo-costo

Esto se debe a 2 aspectos fundamentales:

1. Es **más fácil estimar en base a los diferentes componentes o módulos** del sistema, **que en base a un sistema completo**.
2. La definición de una arquitectura conlle**va que los requisitos fueron relevados y validados**. Cuanto **más conocimiento se tenga** acerca del sistema final, **mejor será la estimación**.

Arquitectura como una abstracción transferible del sistema

Cuanto **más temprano en el proceso de desarrollo se incorpore el reuso, mayores son los beneficios**. Como la arquitectura es la abstracción de más alto nivel, hay mucho que ganar en este aspecto.

No solamente el código puede ser reusado, sino:

- **Los requerimientos** que llevaron a determinada arquitectura
- **La experiencia** de construir la arquitectura
- **Las decisiones de arquitectura**

Líneas de Productos de Software comparten una Arquitectura común

Una familia o productos es un set de sistemas de software gestionados con un set de atributos que satisfacen las necesidades de determinado sector del mercado. Son desarrollados desde una perspectiva común para los activos más importantes.

Se determinan los aspectos que son variables para cada producto y cuáles no, logrando así maximizar el reúso para todos los productos.

Sistemas pueden ser construidos usando elementos desarrollados externamente

Mientras que antes los paradigmas de Software se focalizaban en la programación y el progreso se media en líneas de código, los desarrollos basados en arquitectura se focalizan en:

- **Componer o ensamblar elementos que probablemente se hayan desarrollado separadamente y hasta independientemente**.

El aspecto clave a tomar en cuenta es la organización de la estructura de dichos elementos, las interfaces y los conceptos de operación. El concepto clave es: **elementos intercambiables**.

Menos es más en términos de alternativas de diseño

Al elegir pocos mecanismos diferentes de intercambio entre los elementos del sistema (que pueden ser combinados infinitamente), hay mucho que ganar. Algunas ventajas:

- Ventajas en **reúso**
- **Diseños simplificados** y estandarizados
    - Mejor comunicación y comprensión
- **Mejor análisis**
- **Menor tiempo de selección** y deliberación
- **Mayor interoperabilidad**

Arquitectura de Sistemas vs. Arquitectura de Software

- Pensar en desarrollar un software sin tener en cuenta el sistema sobre el que correrá, hace difícil hacer previsiones acerca de las características y bondades ya que solamente una parte del sistema está especificada.
- Sin embargo se habla primariamente del software y no del sistema, ya que la mayor parte de las libertades del arquitecto se centrarán en decisiones de Software y no de Hardware. Esto se debe a disposiciones de infraestructura de la empresa en la que se trabaja, presupuesto, normas y estándares, entre otros, además de ser un aspecto que probablemente cambiará con el tiempo.

La Arquitectura permite un desarrollo basado en plantillas (templates)

La Arquitectura puede ser la base para el entrenamiento

Parte de familiarizarse con el entorno en un grupo de trabajo es conocer como los elementos interactúan para llevar a cabo determinado comportamiento. La arquitectura puede servir de base para introducir nuevos miembros a un proyecto.

Estructuras Arquitectónicas y Vistas

Una Vista es una **representación de un set coherente de elementos de arquitectura y sus relaciones**, descritos por y destinados a los stakeholders.

A **dichos sets les llamamos estructuras arquitectónicas**. O sea:

Una **estructura arquitectónica** es un **set coherente de elementos de arquitectura**.

Una **vista** es la **representación de una estructura arquitectónica** y **las relaciones** entre sus elementos.

Estas estructuras se pueden separar en 3 grupos:

1. **Estructuras de Modulo**

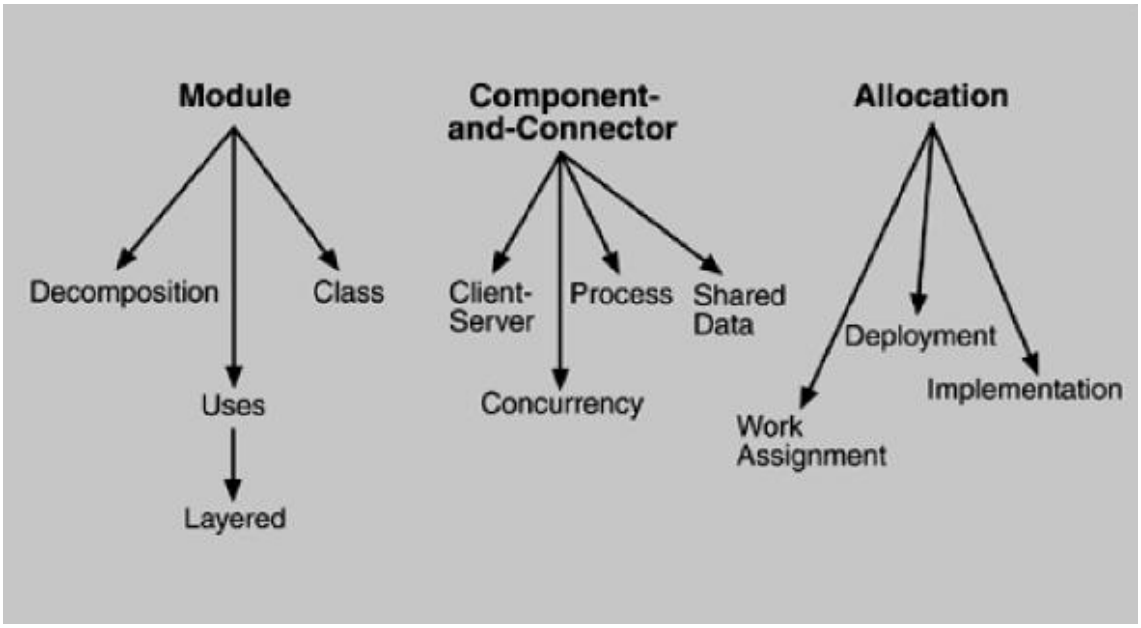
   Es la estructura cuyos **elementos son módulos**, que son las **unidades de implementación**. A estos se les asigna áreas de responsabilidad funcional.

2. **Estructuras de Componentes-Conector**

   Es la estructura cuyos **elementos son los componentes de ejecución y** sus **componentes**.

3. **Estructuras de asignación**

   Es la estructura que muestra la **relación entre elementos del software** y **elementos en uno o más ambientes externos en los cuales el software es creado y/o ejecutado**.



Estructuras de Modulo

Descomposición

Muestra los módulos con una relación de "es un submódulo de". Ayuda a comprender módulos complejos descomponiéndolos en módulos más simples y con menos responsabilidades funcionales.

Usos

Muestra los módulos con una relación de "usa". Ayuda a comprender como los módulos se relacionan entre sí desde el punto de vista de que la operación de uno depende de la presencia de un módulo correcto del cual este depende.

En Capas

Cuando las relaciones de "usa" son controladas de manera particular, se desprende un sistema de capas, en donde se agrupa un set coherente de funcionalidad.

Clase o Generalización

Muestra las clases, con sus herencias. En base a la estructura de clases, se puede considerar el reúso y la agregación incremental de funcionalidad.

Estructuras de Componente-Conector

Son **ortogonales a las estructuras de módulos**. Tienen como **relación el "attachment"**.

Proceso

Muestra los procesos o threads que se conectan entre sí para comunicación, sincronización y/o exclusión.

Concurrencia

Muestra los "threads lógicos" mediante una estructura que permite tempranamente detectar las ocurrencias de eventos asociados a la ejecución concurrente. Además permite planificar la ejecución paralela e identificar las ocasiones en las que habrá que manejar la contención.

Datos compartidos o Repositorio

Se ocupa de ilustrar como se crea, almacena y accede a datos persistentes. Es particularmente útil cuando se estructura el sistema de manera de usar varios repositorios. Muestra también como los elementos del software de tiempo de ejecución consumen y producen los datos, para así manejar la performance e integridad.

Cliente-Servidor

Los componentes de esta estructura son clientes y servidores y los conectores son los protocolos y mensajes necesarios para que el sistema funcione. Especialmente útil para planificar la distribución física y balanceo de cargas.

Estructuras de Asignación

Deployment

Muestra como el software es asignado a los elementos de hardware que los procesan y comunican. Los elementos son software, entidades de hardware y vías de comunicación.

Implementación

Muestra como los elementos de software (usualmente módulos) son mapeados a la estructura de archivos en el sistema en el entorno de desarrollo, integración o configuración.

Asignación de trabajo

Muestra como se le asigna la responsabilidad de implementar e integrar los módulos al equipo de desarrollo apropiado.

Tabla comparativa

Software Structure	Relations	Useful for
Decomposition	Is a submodule of; shares secret with	Resource allocation and project structuring and planning; information hiding, encapsulation; configuration control
Uses	Requires the correct presence of	Engineering subsets; engineering extensions
Layered	Requires the correct presence of; uses the services of; provides abstraction to	Incremental development; implementing systems on top of "virtual machines" portability
Class	Is an instance of; shares access methods of	In object-oriented design systems, producing rapid almost-alike implementations from a common template
Client-Server	Communicates with; depends on	Distributed operation; separation of concerns; performance analysis; load balancing
Process	Runs concurrently with; may run concurrently with; excludes; precedes; etc.	Scheduling analysis; performance analysis
Concurrency	Runs on the same logical thread	Identifying locations where resource contention exists, where threads may fork, join, be created or be killed
Shared Data	Produces data; consumes data	Performance; data integrity; modifiability
Deployment	Allocated to; migrates to	Performance, availability, security analysis
Implementation	Stored in	Configuration control, integration, test activities
Work Assignment	Assigned to	Project management, best use of expertise, management of commonality

¿Cuáles estructuras usar?

Seguramente no todas, por más que todas estén presentes en el sistema que se desea desarrollar. La recomendación que se hace es elegir los aspectos más relevantes y específicos del sistema que se está desarrollando, documentando aquellos aspectos del sistema que sean de vital importancia para entender el desarrollo y funcionamiento del mismo.

Se introduce también la aproximación 4+1, que contiene las siguientes vistas:

- **Lógica** (Módulos)
  Los elementos son las abstracciones claves, que son manifestados en el mundo orientado a objetos como objetos o clases de objetos.

- **Procesos** (Componente-Conector)
  Muestra la concurrencia y distribución de funcionalidad.

- **Desarrollo** (Asignación)
  Muestra la organización de módulos de software, librerías, subsistemas y unidades de desarrollo. Mapea el software al ambiente de desarrollo.

- **Física** (Asignación)
  Mapea demás elementos en nodos de proceso y comunicación. Es llamada también vista de Deploy.

Atributos de Calidad

Los atributos de calidad y la funcionalidad son ortogonales.

De no ser así, la funcionalidad requerida para un sistema dictaría el nivel de performance, seguridad o usabilidad que el sistema debería tener. Siendo ortogonales, funcionalidad y atributos son independientes.

La funcionalidad es lo que habilita el sistema a hacer el trabajo para el cual está pensado. Esto puede ser logrado a partir de cualquier tipo de estructura. De esto resulta que si la funcionalidad fuera el único requerimiento, el sistema podría existir como un módulo monolítico y único con ninguna estructura interna.

La descomposición y estructuración del sistema cumple propósitos de los atributos de calidad, haciéndolo entendible y sirviendo otros propósitos, ajenos a la funcionalidad.

La Arquitectura y Atributos de Calidad

Un **atributo de calidad** bien logrado **tiene que ser tomado en cuenta en toda etapa del desarrollo**, ya sea en el diseño, implementación o deploy. No existe un atributo enteramente logrado en una etapa. El éxito se basa en conciliar el nivel de abstracción más alto (arquitectura) con el más bajo (implementación).

Por ejemplo:

- Usabilidad. Sabemos que depende de la arquitectura, pero también de la implementación (posición de botones, etc.)
- Modificabilidad. Depende de la arquitectura, pero también de técnicas de implementación, como patrones de diseño.
- Performance. Depende de cuanta comunicación sea necesaria entre los componentes (arquitectura), como de los algoritmos que se usen (implementación), etc.

En suma:

1. **La arquitectura es vital** para la **satisfacción de los atributos de calidad**. Estos deben ser **previstos en la etapa de diseño de la arquitectura**.
2. **La arquitectura** por sí **sola no puede satisfacer enteramente ningún atributo de calidad**.

Además, la **obtención de un atributo de calidad puede favorecer o desfavorecer otro atributo de calidad**.

Existen 3 tipos de atributos de calidad:

1. **Calidad del Sistema**
2. **Calidad del Negocio**
3. **Calidad de Arquitectura**

Calidad del Sistema

Existen 3 problemas con la definición anterior de atributos de calidad:

1. **Las definiciones provistas para un atributo no son operacionales** (No tiene sentido decir que un sistema es modificable. Todos lo son hasta cierto punto y determinado propósito).
2. **El foco de discusión normalmente se centra en torno a que elemento de calidad pertenece determinado aspecto** (Una falla del sistema es un aspecto de disponibilidad, seguridad o usabilidad?).
3. **Cada comunidad de atributos desarrolló su propio vocabulario** (Performance adoptó "evento", seguridad "ataques" y usabilidad "user input", etc.).

La **solución para los primeros dos** fue el uso de **escenarios de atributos de calidad**.
La **solución para el tercero** fue una **breve discusión para cada atributo** y sus implicancias.

Escenarios de Atributos de Calidad

Consiste de 6 partes:

- **Fuente de estímulo**
  Alguna entidad, sea humana o artificial, que genera el estímulo.

- **Estímulo**
  La condición a ser considerada por el sistema.

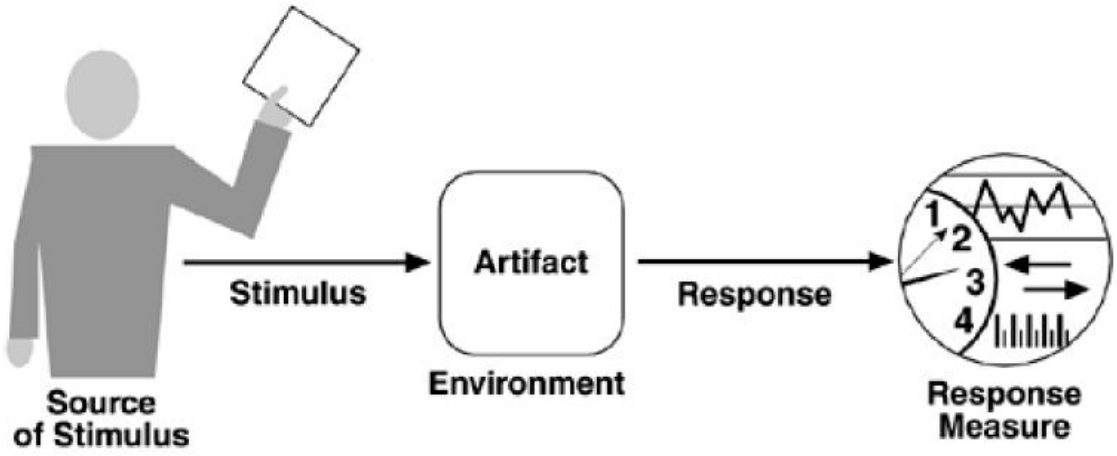
- **Ambiente**
  La condición en la que se encuentra el artefacto cuando el estímulo ingresa.

- **Artefacto**
  El sistema o subsistema que debe responder al estímulo.

- **Respuesta**
  La actividad que lleva a cabo el sistema al recibir el estímulo.

- **Medida**
  La respuesta debe ser cuantificable o medible. De esta forma se puede considerar un éxito o un intento fallido de satisfacer el atributo de calidad.

Escenario General



Disponibilidad

Un "**failure**" ocurre cuando el sistema **no responde adecuadamente de acuerdo a su especificación**. Este **comportamiento es observable por el usuario**.

Algunas de las áreas de la Disponibilidad:

- Como se detectan los failures
- Cuan seguido pueden ocurrir
- Que pasa cuando ocurren
- Cuanto tiempo queda el sistema fuera de operación
- Cuando pueden ocurrir seguramente
- Como puede prevenirse
- Que tipos de notificación deben ocurrir

Existen también los "**faults**". Estos son **promitentes failures**, **si no son manejados adecuadamente**. Un fault puede ser enmascarado de manera que no sea observable para el usuario. Esto es, si el sistema encuentra un fault, pero se puede recuperar, **esto no constituye un failure**.

La disponibilidad normalmente se calcula de esta manera:

$$\alpha = \frac{\text{mean time to failure}}{\text{mean time to failure} + \text{mean time to repair}}$$

Portion of Scenario	Possible Values
Source	Internal to the system; external to the system
Stimulus	Fault: omission, crash, timing, response
Artifact	System's processors, communication channels, persistent storage, processes
Environment	Normal operation; degraded mode (i.e., fewer features, a fall back solution)
Response	System should detect event and do one or more of the following: record it notify appropriate parties, including the user and other systems disable sources of events that cause fault or failure according to defined rules be unavailable for a prespecified interval, where interval depends on criticality of system continue to operate in normal or degraded mode
Response Measure	Time interval when the system must be available Availability time Time interval in which system can be in degraded mode Repair time

Modificabilidad

Es el **costo del cambio**. Se define en base a dos preocupaciones principales:

1. ¿Qué puede cambiar? (artefacto)
2. ¿Cuándo sucede y quien lo realiza?

Portion of Scenario	Possible Values
Source	End user, developer, system administrator
Stimulus	Wishes to add/delete/modify/vary functionality, quality attribute, capacity
Artifact	System user interface, platform, environment; system that interoperates with target system
Environment	At runtime, compile time, build time, design time
Response	Locates places in architecture to be modified; makes modification without affecting other functionality; tests modification; deploys modification
Response Measure	Cost in terms of number of elements affected, effort, money; extent to which this affects other functions or quality attributes

Performance

Es el **tiempo en el que el sistema responde a los eventos.**

Lo que hace gestionar esto complicado es:

- El **número de fuentes** de eventos (usuarios, otros sistemas o componentes del sistema)
- **Patrones de arribo** de eventos.

Acerca de los **patrones de arribo** de eventos, estos se pueden basar en **modelos**:

- **Periódicos**
- **Probabilísticos**

Pero esto no quita que puedan suceder **esporádicamente**.

Algunas características medibles acerca de la performance son:

- **Latencia**. (tiempo entre que la fuente envía el pedido y recibe el retorno)
- **Deadline** del proceso. (tiempo límite para responder)
- **Throughput del sistema**. (número de pedidos que debe atender en determinado tiempo)
- **Jitter de respuesta**. (la variación de latencia)
- **Numero de eventos rechazados**, por sobrecarga del sistema.
- **Datos perdidos** por sobrecarga del sistema.

Portion of Scenario	Possible Values
Source	One of a number of independent sources, possibly from within system
Stimulus	Periodic events arrive; sporadic events arrive; stochastic events arrive
Artifact	System
Environment	Normal mode; overload mode
Response	Processes stimuli; changes level of service
Response Measure	Latency, deadline, throughput, jitter, miss rate, data loss

Seguridad

Es la medida que indica la **habilidad del sistema de resistir uso no autorizado mientras provee servicio a usuarios legítimos**.

Un **intento de uso no autorizado** del sistema se llama "**ataque**".

Un sistema seguro cumple con 6 atributos:

- **No-repudiación**
  Una transacción que una vez es realizada, no puede ser negada por ninguna de las partes.
- **Confidencialidad**
  Tanto los datos como los servicios deben estar protegidos de acceso no autorizado.
- **Integridad**
  Tanto los datos como los servicios están siendo entregados como corresponde.
- **Aseguramiento**
  En una transacción las partes deben ser quien alegan ser.
- **Disponibilidad**
  El sistema tiene que estar disponible para los usos legítimos.
- **Auditoría**
  El sistema debe mantener registros de la actividad para posterior reconstrucción.

Portion of Scenario	Possible Values
Source	Individual or system that is
	correctly identified, identified incorrectly, of unknown identity
	who is
	internal/external, authorized/not authorized
	with access to
	limited resources, vast resources
Stimulus	Tries to
	display data, change/delete data, access system services, reduce availability to system services
Artifact	System services; data within system
Environment	Either
	online or offline, connected or disconnected, firewalled or open

Portion of Scenario	Possible Values
Response	Authenticates user; hides identity of the user; blocks access to data and/or services; allows access to data and/or services; grants or withdraws permission to access data and/or services; records access/modifications or attempts to access/modify data/services by identity; stores data in an unreadable format; recognizes an unexplainable high demand for services, and informs a user or another system, and restricts availability of services
Response Measure	Time/effort/resources required to circumvent security measures with probability of success; probability of detecting attack; probability of identifying individual responsible for attack or access/modification of data and/or services; percentage of services still available under denial-of-services attack; restore data/services; extent to which data/services damaged and/or legitimate access denied

Testeabilidad

Es la medida que indica la **facilidad** con la que se puede **demostrar** que **el sistema carece de fallas mediante testeo.**

También se puede ver como la **probabilidad**, asumiendo que el sistema contiene al menos una falla, de que **falle en la próxima ejecución de una prueba**. Para que un sistema sea testeable:

Debe ser posible el **control** del **estado interno de cada componente y** luego **observar sus salidas**.

Portion of Scenario	Possible Values
Source	Unit developer
	Increment integrator
	System verifier
	Client acceptance tester
	System user
Stimulus	Analysis, architecture, design, class, subsystem integration completed; system delivered
Artifact	Piece of design, piece of code, complete application
Environment	At design time, at development time, at compile time, at deployment time
Response	Provides access to state values; provides computed values; prepares test environment
Response Measure	Percent executable statements executed
	Probability of failure if fault exists
	Time to perform tests
	Length of longest dependency chain in a test
	Length of time to prepare test environment

Usabilidad

Es la medida que indica la **facilidad** con la que **un usuario puede completar una tarea deseada y el nivel de apoyo que le brinda el sistema.**

Para esto, podemos categorizar las siguientes áreas:

- **Curva de aprendizaje**
- **Eficiencia de uso**
- **Minimizando el impacto de errores**
- **Adaptación del sistema a las necesidades** del usuario
- **Aumento de confianza y satisfacción** al usuario

Portion of Scenario	Possible Values
Source	End user
Stimulus	Wants to learn system features; use system efficiently; minimize impact of errors; adapt system; feel comfortable
Artifact	System
Environment	At runtime or configure time

Portion of Scenario	Possible Values
Response	System provides one or more of the following responses: to support "learn system features" help system is sensitive to context; interface is familiar to user; interface is usable in an unfamiliar context to support "use system efficiently": aggregation of data and/or commands; re-use of already entered data and/or commands; support for efficient navigation within a screen; distinct views with consistent operations; comprehensive searching; multiple simultaneous activities to "minimize impact of errors": undo, cancel, recover from system failure, recognize and correct user error, retrieve forgotten password, verify system resources to "adapt system": customizability; internationalization to "feel comfortable": display system state; work at the user's pace
Response Measure	Task time, number of errors, number of problems solved, user satisfaction, gain of user knowledge, ratio of successful operations to total operations, amount of time/data lost

Calidad del Negocio

Engloba los aspectos del negocio ($) que conlleva la producción de software y puede resumirse en:

1. **Time to Market**
   Como indica, el tiempo de desarrollo hasta que el producto esté disponible para su comercialización y uso.
2. **Costo y Beneficio**
   Si bien existe un presupuesto que no se puede ecceder, diferentes arquitecturas tienen diferentes costos de desarrollo. Por ejemplo, una arquitectura basada en tecnología de terceros será más costosa que una basada en tecnología propia y moldeable. O por ejemplo un producto basado en una arquitectura rígida será más barato que uno con una arquitectura flexible, abierta a la adaptación y modificación.
3. **Ciclo de Vida Proyectado**
   Con un ciclo de vida más largo será importante prever los atributos de calidad asociados con modificabilidad y escalabilidad.
4. **Mercado Objetivo**
   Normalmente el mercado objetivo influye en la plataforma sobre la que correrá el software. Es importante tener en cuenta cual es el mercado objetivo para hacer un producto atractivo para dicho sector.
5. **Rollout Schedule**
   Si un producto es anunciado como base, para ser extendido con nueva funcionalidad en el futuro, es necesario tomar las previdencias necesarias.
6. **Integración con Sistemas Legacy**
   Se deben tomar las previdencias necesarias para admitir adaptación e integración con los demás sistemas.

Calidad de Arquitectura

Con esto nos referimos a la **integridad conceptual**.

Este concepto dicta que:

Es **mejor tener un sistema que omita ciertas características y mejores anómalas** (fuera de lo común o innovadoras) **pero que refleje un set coherente de ideas de diseño**, **a que contenga un montón ideas buenas e independientes pero descoordinadas**.

Esto se debe a que tanto para los usuarios, como los demás stakeholders, tener un sistema conceptualmente íntegro, ayuda a entenderlo mejor, tanto a nivel de arquitectura como a nivel de usabilidad.

Además, el sistema debe ser:

- **Completo**
  Hace lo que debe hacer.
- **Correcto**
  Hace bien las cosas.

Tácticas para cumplir con Atributos de Calidad

Las tácticas en la Arquitectura son usadas para crear el diseño usando **patrones de diseño**, **patrones de arquitectura** o **estrategias de arquitectura**.

Tanto los **patrones** como las **estrategias de arquitectura** implementan una **colección de tácticas**.

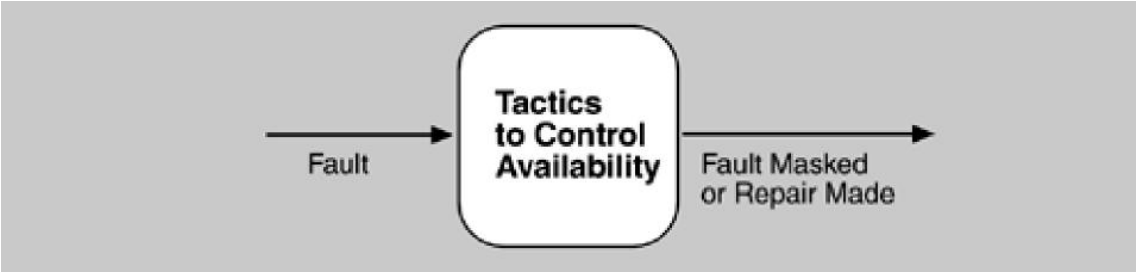
Veremos dos características de las tácticas:

1. **Las tácticas refinan otras tácticas**
   Existe, por así decirlo, una jerarquía de tácticas.
   Ejemplo: Al decidirnos por una táctica, como ser redundancia, podemos refinarla en redundancia de datos o computacional, que son 2 tácticas más específicas.

2. **Los patrones empaquetan tácticas**
   Para soportar un atributo de calidad, un patrón específico puede recurrir a más de una táctica.
   Ejemplo: Un patrón que soporte la disponibilidad, probablemente implemente una táctica de redundancia y sincronización.

Tácticas de Disponibilidad

Teniendo en cuenta la definición de fault y failure, veremos que estos conceptos tienen que ver con la Disponibilidad.



Como receta para toda táctica de Disponibilidad, veremos que se incluye en todo caso:

1. **Redundancia**
2. Un sistema de "salud" o **detección de faults**
3. Un sistema de **recuperación** cuando se detecta una fault

A veces el monitoreo y recuperación es automático, aunque puede ser manual.

Detección de faults

Existen 3 maneras de detectar faults:

- **Ping/Echo**
  Cuando los componentes hacen Ping y esperan la respuesta. Cuando no existe respuesta, se asume que el componente falló.
  Este método opera entre procesos distintos.

- **Heartbeat**
  Cuando los componentes avisan que están disponibles cada x tiempo. Si pasa demasiado tiempo sin emitirlo, se asume que el componente falló.
  Este método opera entre procesos distintos.

- **Exceptions**
  Este método opera en el mismo proceso.

Recuperación de faults

- **Voting**
  Procesos corriendo en procesadores redundantes toman el mismo input pero con algoritmos distintos, desarrollados por equipos distintos, devuelven cada uno un output. Luego se utilizan diversos algoritmos para decidir cuál output tomar o qué hacer con resultados dispares.
  Esto es **caro de implementar**, pero **necesario** para algunos **sistemas críticos**.

- **Redundancia Activa (hot restart)**
  Todos los componentes redundantes contestan a los eventos, y siempre se toma la primer respuesta, descartando las demás. Como todos responden a los eventos, **la sincronización** está asegurada ya que todos los componentes siempre se encuentran en el mismo estado.
  Usualmente se utiliza para arquitecturas Cliente-Servidor o DBMS.

- **Redundancia Pasiva (warm restart)**
  El componente primario responde a los eventos y luego notifica a los componentes redundantes. Si el primario falla, primero hay que fijarse si los de respaldo se encuentran en condiciones suficientemente actualizadas para atender al evento. **La sincronización** está a cargo del componente primario.

- **Repuesto**
  Se mantiene un componente de repuesto para substituir al que falle.
  Al ocurrir un fallo, este debe ser sincronizado con la última versión consistente del componente primario con sus configuraciones y estados.

Existen además tácticas que dependen de la reintroducción de componentes. Cuando un componente redundante falla, puede ser reintroducido cuando es corregido. Veremos 3:

- **Shadow Operation**
  Un componente que falla opera en “modo sombra” por un breve período de manera que imite el comportamiento de un componente funcionando hasta que este vuelva al estado de funcionamiento normal.

- **Resincronización de estados**
  Cuando se usa tácticas de redundancia se pueden adoptar diferentes mecanismos para volver a sincronizar los componentes redundantes con el componente primario fallado. Este depende de características varias, como el tiempo que puede permanecer bajo el servicio, el peso de la actualización o cantidad de mensajes para actualizar.

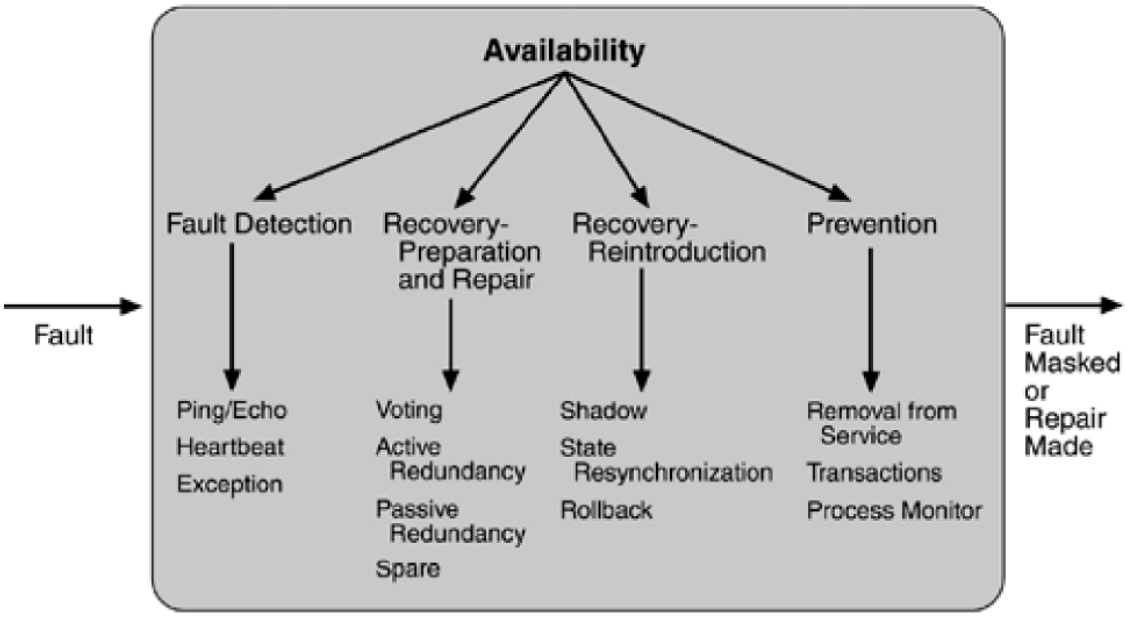
- **Checkpoint/Rollback**
  Se crean checkpoints a lo largo de la ejecución para poder hacer rollbacks de los sistemas cuando fallan.

Prevención de faults

- **Removal from service**
  Se retira anticipadamente el componente de funcionamiento para prevenir faults. Si este cambio es automático, la arquitectura debe soportarlo, cuando es manual, el sistema debe poder soportarlo.

- **Transacciones**
  Se trata de empaquetar una serie de pasos secuenciales de manera que todo el conjunto pueda ser deshecho de una vez.

- **Process Monitor**
  Cuando se detecta un fault, el monitor de procesos mata el proceso defectuoso y lo reinicializa en un estado que pueda mantener al sistema funcional.



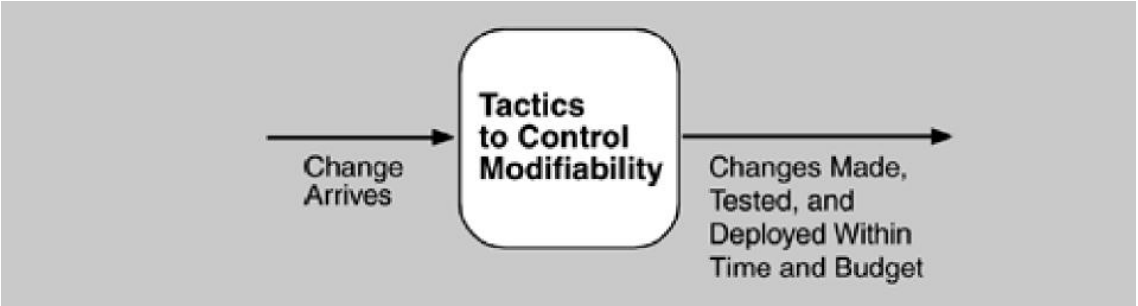
Tácticas de Modificabilidad

Para mejorar la modificabilidad, contamos con 3 tipos de tácticas:

- **Localizar Modificaciones**
  Se fijan en reducir el número de módulos directamente afectados por el cambio.

- **Prevenir el "Ripple-effect"**
  Se fijan en limitar la modificación a los módulos que semánticamente debe cambiar.

- **Diferir tiempo de bindeo**
  Se fijan en controlar el tiempo y costo del deploy de la nueva versión.



Localizar Modificaciones

Se centran en **definir las responsabilidades de los módulos en el diseño**, **anticipando cambios** para que sean localizados a dichos módulos.

- **Mantener coherencia semántica**
  Se trata de mantener las relaciones dentro del módulo, minimizando las dependencias hacia otros módulos para tareas similares o de mismo significado semántico. Una sub-táctica es abstraer los servicios comunes.

- **Anticipar cambios esperados**
  La diferencia entre esta táctica y la anterior es que la anterior asume que los cambios serán semánticamente coherentes. Esta táctica se centra específicamente en los cambios esperados y la minimización de módulos a tocar, muchas veces armando módulos en torno a la funcionalidad y no a la semántica.

- **Generalizar el módulo**
  Hacer que el componente reaccione a los inputs de una manera similar, de manera que los cambios se reflejen solo en la modificación de esos inputs, y no en el módulo general.

- **Limitar posibles opciones**
  Se trata de limitar los cambios posibles. Por ejemplo, limitando los cambios de plataforma a una línea de plataformas en particular.

Prevenir el "Ripple-effect"

Se centran en **limitar las modificaciones a los módulos que no dependen directamente del cambio.**

Esto ocurre cuando se desea modificar el módulo A, pero hay ciertas relaciones con B a tomar en cuenta:

- **Sintaxis** de
  - **Datos**: El módulo B consume los datos producidos por A, que tienen determinada sintaxis.
  - **Servicios**: El módulo B consume los servicios de A, que tienen determinada sintaxis.
- **Semántica** de
  - **Datos**: El módulo B consume los datos producidos por A, que tienen determinada semántica.
  - **Servicios**: El módulo B consume los servicios de A, que tienen determinada semántica.
- **Secuencia** de
  - **Datos**: El módulo B consume los datos producidos por A en determinada secuencia.
  - **Control**: El módulo B ejecuta luego de A, que debe haber ejecutado antes con ciertas restricciones de tiempo.
- Las **interfaces de A** deben ser consistentes con lo que B asume.
- La **localización**, en tiempo de ejecución, **de A** debe ser consistente con lo que B asume.
- La **calidad del servicio de A** debe ser consistente con lo que B asume.
- La **existencia de A**
- Los **recursos utilizados** por ambos.

Para esto se han creado las tácticas que veremos a continuación.

- **Information Hiding**
  Se centra en ocultar la información de los demás módulos, logrando menor acomplamiento.

- **Mantener las interfaces**
  Diseñar interfaces que perduren independientemente de los cambios que se le pueda llegar a hacer a los módulos que las contienen.

- **Restringir vías de comunicación**
  Se centra en restringir los módulos que comparten información (reducir módulos que producen y consumen datos).

- **Usar Intermediarios**
  Se centra en asignar intermediarios que centralicen la interacción entre dos módulos.
    - **Sintaxis de Datos**: **Repositorios** que convierten la sintaxis de los datos.
    - **Sintaxis de Servicios**: **Facade, Bridge, mediator, strategy y proxy** son patrones que convierten la sintaxis de un servicio.
    - **Identidad de Interface**: **Broker** es un patrón que enmascara los cambios de identidad de las interfaces.
    - **Localización**: Un servidor de nombres enmascara dicha complejidad.
    - **Existencia**: **El patrón Factory** asegura la creación de instancias.

Diferir Tiempo de Bindeo

Se centran en **minimizar el tiempo y costo del cambio y habilitar a no desarrolladores a hacer los cambios.**

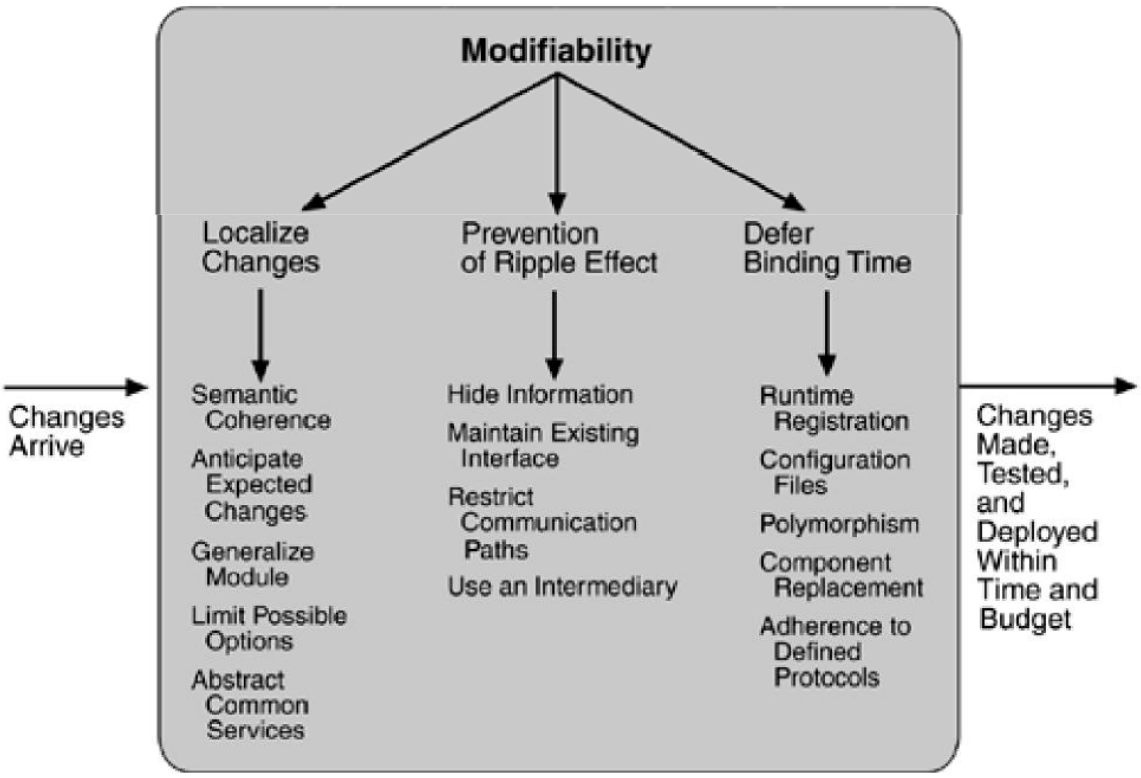
- **Registro de Tiempo de Ejecución**
  Se centra en crear operaciones plug-and-play como subscribirse a servicios.

- **Archivos de Configuración**
  Se centra en crear archivos con parámetros de startup para el programa.

- **Polimorfismo**
  Se centra en utilizar esta característica para aprovechar el late binding.

- **Reemplazo de Componentes**
  Se centra en crear componentes intercambiables.

- **Adherencia a protocolos definidos**
  Se centra en permitir el binding de procesos en tiempo de ejecución.

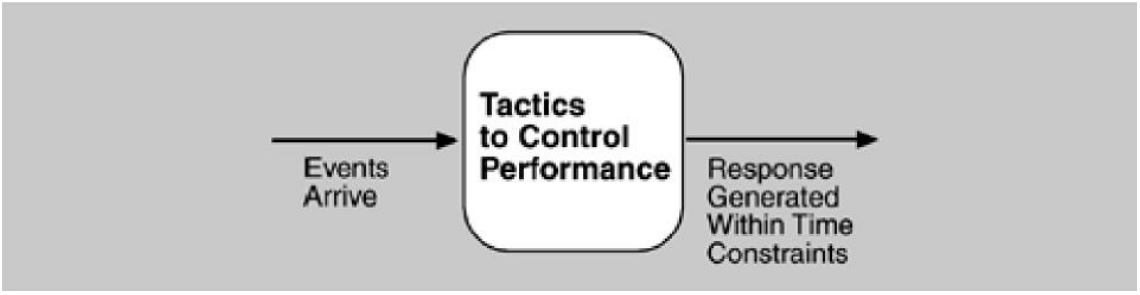

Modifiability
Localize Changes
Prevention of Ripple Effect
Defer Binding Time
Changes Arrive
Semantic Coherence
Anticipate Expected Changes
Generalize Module
Limit Possible Options
Abstract Common Services
Hide Information
Maintain Existing Interface
Restrict Communication Paths
Use an Intermediary
Runtime Registration
Configuration Files
Polymorphism
Component Replacement
Adherence to Defined Protocols
Changes Made, Tested, and Deployed Within Time and Budget

Tácticas de Performance

Para mejorar la performance, contamos con tácticas que cumplen 2 objetivos:

- **Consumo de Recursos**
  Se fijan en distribuir y arbitrar el uso de recursos para que todo el sistema opere de manera estable y bajo restricciones de tiempo.

- **Tiempo de Bloqueo**
  Como los recursos son limitados, pero no así los eventos, algunos recursos tendrán que ser compartidos y no pueden ser usados simultáneamente para atender varios eventos o la entrada de un proceso es la salida de otro .
  Se fijan entonces en:
  - **Arbitrar el acceso a los recursos**
  - **Asegurarse que los recursos estén disponibles** para manejar eventos.
  - **Manejar la dependencia entre procesos**



Demanda de Recursos

1. Para bajar la latencia, una táctica se basa en **reducir los recursos que hace falta para procesar un evento**. De 2 maneras:

   - **Aumentando la eficiencia computacional**
     - Cambiando un recurso por otro
     - Cambiando el algoritmo que procesa el evento
   - **Reduciendo el Overhead**
     - Eliminando intermediarios
     - Eliminando procesos de control

2. Otra táctica es **reduciendo el número de eventos procesados**. De las siguientes 2 maneras:

   - **Manejando el rate de llegada de eventos**
     - Reduciendo el polling de variables
   - **Manejando la frecuencia de muestreo**
     - Sin embargo esto puede llevar a perdida de datos

3. Otra táctica es **controlando el uso de recursos**. De las siguientes 2 maneras:

   - **Manejando el tiempo de dedicación a un evento**
   - **Manejando el tamaño de las colas de eventos**

Manejo de Recursos

- **Introducción de Concurrencia**
  Se centra en crear operaciones que se procesen en paralelo.

- **Mantenimiento de copias de datos**
  Se centra en crear copias de los datos para que la concurrencia no bloquee procesos.

- **Aumento de los recursos disponibles**
  Se centra en mejorar los recursos (más y mejor procesador, más memoria…).

Arbitraje de Recursos

- **FIFO**
  First in – First Out.

- **Prioridad fija**
  Se puede fijar en base a:
  - Importancia semántica
  - Por Deadline (menor deadline va antes)
  - Por Rate de aparición (menor aparición va antes)

- **Asignación dinámica**
  Se puede fijar en base a modelos como:
  - Round Robin
  - Deadline más próximo va antes

- **Asignación estática**
  Se fijan las prioridades cuando el sistema está bajo.