Resumen Arquitectura de Software

Arquitectura de software (Universidad ORT Uruguay)

Arquitectura de software

Documenting software architectures:

Ninguna arquitectura sirve para nada si la gente no sabe cual es, como usarla, modificarla y el porque de ella. La arquitectura debe ser modificada de manera que las parte interesadas puedan hacer su trabajo.

La documentación de la arquitectura será utilizada por el mismo, por todos aquellos que necesiten información del sistema, o hasta por algún otro arquitecto en el futuro. Por esto es tan importante.

Usos y audiencias:

- Debe ser suficientemente transparente y accesible para ser rápidamente entendible por nuevos empleados
- Debe poder ser usada como un "plano de construcción" para el programa
- Debe tener suficiente información para ser utilizada

La documentación puede ser prescriptiva (indica cómo el sistema debería ser,limitando decisiones a tomar) o descriptiva (muestra como es el sistema, explicando decisiones tomadas).

Existen 3 usos para la documentación de una arquitectura:

- **Educación**

Se utiliza para introducir gente al sistema. Estos pueden ser nuevos programadores, analistas o un nuevo arquitecto.

- **Comunicación entre partes interesadas**

Sirve para comunicar el porqué se tomaron determinadas decisiones o como se atacaron distintas dificultades del sistema. Uno de los interesados puede ser el mismo arquitecto en el futuro.

- **Analisis y construccion del sistema**

La arquitectura le indica a los programadores que implementar, ya que indica responsabilidades e interfaces necesarias. A su vez, la documentación debe permitir evaluar al sistema. Esta evaluación es función de determinados atributos, como seguridad, performance, usabilidad, etc.

Notaciones:

Existen 3 tipos de notaciones para documentar vistas, las cuales varían en su formalidad:

- **Notaciones informales:**

Se muestran gráficamente utilizando diagramas genéricos. La semántica generalmente es mediante lenguaje natural y no pueden ser analizadas formalmente.

- **Notaciones semiformales:**

Se expresan mediante una notación estandarizada con elementos gráficos y reglas, pero no se hace un trato semántico completo. Se pueden hacer análisis simples. UML es un ejemplo.

- **Notaciones formales:**

Se describen mediante una notación precisa con bases matemáticas. Se puede hacer análisis formales tanto de la semántica como de la sintaxis. Son poco utilizadas en la práctica

La decisión de que notacion utilizar depende completamente de lo necesario para el sistema. Las notaciones formales son mucho más costosas(tiempo y esfuerzo) pero más precisas.

Vistas:

Una vista es una representación de un conjunto de elementos del sistema y las relaciones entre ellos (no todos los elementos del sistema, solo algunos de un tipo).

Las vistas existen debido a que no es fácil explicar un sistema complejo en una sola dimensión. Por ejemplo, una vista de layers muestra elementos de tipo layer y las relaciones entre las layers. No muestra, por ejemplo, modelos de datos o servidores.

Documentar una arquitectura consiste en documenta todas las vistas relevantes y la información que se aplica a más de una de ellas.

La elección de que vistas son relevantes para el sistema depende de los objetivos. Cada vista expone distintos atributos de calidad en diferentes niveles. Por ende, se debe priorizar las vistas que exponen más los atributos de calidad que son importantes para el sistema.

Vistas de módulos

Un módulo es una unidad de implementación que provee un conjunto coherente de responsabilidades. Un módulo puede ser una clase, un conjunto de clases, una capa o cualquier tipo de descomposición. Cada módulo cuenta con un conjunto de propiedades importantes.

Son importantes ya que indican como como dividir el sistema en componentes manejables. Se determina que se puede asumir de otros componentes, como se juntan en componentes más grandes y como se descompone el código.

Raramente una documentación no cuenta con por lo menos una vista de módulos, ya que estas muestran la modificabilidad, portabilidad y el reuso de un sistema.

Elements	Modules, which are implementation units of software that provide a coherent set of responsibilities.
Relations	▪ *Is part of*, which defines a part/whole relationship between the submodule—the part—and the aggregate module—the whole. ▪ *Depends on*, which defines a dependency relationship between two modules. Specific module views elaborate what dependency is meant. ▪ *Is a*, which defines a generalization/specialization relationship between a more specific module—the child—and a more general module—the parent.
Constraints	Different module views may impose specific topological constraints, such as limitations on the visibility between modules.
Usage	▪ Blueprint for construction of the code ▪ Change-impact analysis ▪ Planning incremental development ▪ Requirements traceability analysis ▪ Communicating the functionality of a system and the structure of its code base ▪ Supporting the definition of work assignments, implementation schedules, and budget information ▪ Showing the structure of information that the system needs to manage

No es fácil hacer inferencias de atributos que correspondan al runtime del sistema (como performance o reliability), para eso existen las vistas de componentes y conectores.

Componentes y conectores

Las vistas de componentes y conectores muestran elementos que tienen presencia en tiempo de ejecución del sistema, como proceso, objetos, clientes, servidores, etc. Estos elementos se llaman **componentes**. A su vez, se incluyen las interacciones entre ellos, como conexiones de comunicación, protocolos, accesos a datos en común, etc. Estos son llamados **conectores**.

Puertos: Los puertos son los puntos de interacción (interfaces) del componente con el entorno. El puerto tiene un tipo, una cardinalidad y puede aparecer muchas veces (mismo tipo).

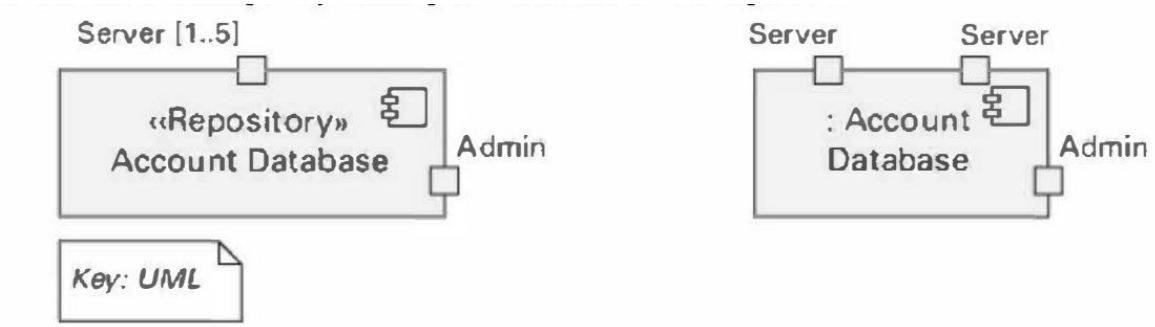
Roles: Son interfaces que definen como puede ser utilizado un conector para hacer una interacción. Como los puertos, los roles pueden ser replicados, indicando cuántos pueden estar involucrados en la interacción.

Attachment: Es la principal relación. Indica que conectores están conectados a qué componentes. Esto define al sistema como un grafo de componentes y conectores. Se define como una asociación

entre un puerto de un componente y un rol de un conector. Se debe validar que el puerto de compatible con el rol, validando con las restricciones impuestas.

Elements	▪ *Components.* Principal processing units and data stores. A component has a set of *ports* through which it interacts with other components (via connectors). • *Connectors.* Pathways of interaction between components. Connectors have a set of roles (interfaces) that indicate how components may use a connector in interactions.
Relations	• *Attachments.* Component ports are associated with connector roles to yield a graph of components and connectors. ▪ *Interface delegation.* In some situations component ports are associated with one or more ports in an "internal" subarchitecture. The case is similar for the roles of a connector.
Constraints	• Components can only be attached to connectors, not directly to other components. ▪ Connectors can only be attached to components, not directly to other connectors. • Attachments can only be made between compatible ports and roles. ▪ Interface delegation can only be defined between two compatible ports (or two compatible roles). ▪ Connectors cannot appear in isolation; a connector must be attached to a component.
Usage	▪ Show how the system works. ▪ Guide development by specifying structure and behavior of runtime elements. ▪ Help reason about runtime system quality attributes, such as performance and availability.

Notaciones:



Allocation Views

Las vistas de allocation describe el mapeo de los elementos del software a los elementos del ambiente en el que el sistema será ejecutado. El ambiente puede ser el hardware, el sistema operativo, el file system, o la organización que desarrolló el software.

Consiste de dos tipos de elementos: software y de ambiente. Ejemplo de elementos de ambiente son discos duros, carpetas, grupos de desarrolladores. Los elementos de software vienen de una vista de componentes y conectores o de una de modulos.

La relación es "alocado a". Esto se puede expresa en el sentido de asignar un elemento de software a un elemento del ambiente, aunque también se puede utilizar la inversa. Por ejemplo, asignar determinados desarrolladores a un módulo.

El objetivo es comparar los recursos requeridos por los elementos de software con los provistos por los elementos del ambiente, de manera de identificar discrepancias que puedan traer complicaciones.

Elements	▪ *Software element*. A software element has properties that are *required* of the environment. ▪ *Environmental element*. An environmental element has properties that are *provided* to the software.
Relations	*Allocated to*. A software element is mapped (allocated to) an environmental element. Properties are dependent on the particular view.
Constraints	Varies by view
Usage	▪ For reasoning about performance, availability, security, and safety. ▪ For reasoning about distributed development and allocation of work to teams. ▪ For reasoning about concurrent access to software versions. ▪ For reasoning about the form and mechanisms of system installation.

Quality views

Las vistas previamente mostradas son estructurales, muestran las estructuras del software. Las vistas de calidad sin embargo, buscan enfatizar en determinados atributos de calidad que no son efectivamente especificados con estas vistas.

Las vistas de calidad son creadas con un objetivo específico y a veces para interesados específicos. Se forman extrayendo piezas relevantes de las vistas estructurales y juntandolas. Las siguientes son ejemplo de ellas:

- **Vista de seguridad:**

Se muestran todas las medidas arquitectónicas tomadas para proveer seguridad. Estas pueden ser roles que tienen determinados componentes, como se comunican y autentican, etc.

- **Vista de comunicación:**

Describe las metodologías de comunicación entre componentes. Sirven para analizar determinados tipos de performance y confiabilidad.

- **Vista de manejo de errores**

Hace énfasis en el manejo de errores, particularmente de la resolución y reporte de ellos.

- **Vista de confianza**

Se muestra los mecanismos utilizados para la confianza del sistema, como replicación.

- **Vista de performance**

Elección de vistas:

Previo a la liberación del documento de arquitectura, se debe decidir qué vistas son necesarias de definir con exactitud y con cuánto detalle. También se debe decidir cuáles usar en conjunto para reducir el número total y mostrar las relaciones.

La decisión de esto puede ser en base a varios factores: estándares, personal disponible, presupuesto, tiempo, que información es importante, etc.

El proceso para decidir qué vistas son pertinentes es el siguiente:

- Step 1: Crear una tabla de interesados / vistas

Se genera una tabla con las filas de interesados y en las columnas las vistas del sistema. En cada celda, se describe cuanta informacion necesiten los interesados de esa vista (ninguna, poca, bastante, mucha). A partir de esto se toman en cuenta solo las vistas que los interesados tengan bastante interés.

- Step 2: Combinar vistas

Las vistas obtenidas del paso previo probablemente sean bastantes. Se buscan vistas marginales (que tienen poco interés o solo sirven a pocos interesados) y se combinan con vistas que tengan más importancia

- Step 3: Priorizar

Se debe priorizar qué vistas se deben hacer primero que otras. Depende mucho del proyecto, pero hay consideraciones a hacer:

- La vista de descomposición (módulos) es de muchas ayuda al principio ya que da un pantallazo grande del sistema y es fácil de hacer
- Tomar en cuenta que no toda la información es necesaria siempre. Con el 80% es suficiente muchas veces.
- Se pueden hacer muchas vistas a la vez, no es necesario esperar

Combinar vistas:

Puede ser de utilidad combinar vistas para demostrar las asociaciones entre distintos aspectos del sistema. Una forma de hacer esto es combinarlas, mostrando los elementos y conexiones de ambas vistas en un mismo diagrama. La manera más fácil de hacer esto es un overlay entre ambas.

Armado de la documentación

Documentar una vista:

Consiste de las siguientes partes:

Sección 1: Representación primaria

Muestra los elementos y relaciones de la vista. Contiene la información del sistema que se quiere mostrar en el vocabulario de la vista. No tiene que mostrar todo, puede mostrar solo el flujo normal.

Generalmente es una representación gráfica, como un diagrama (formal o informal). Puede llegar a ser textual, como una tabla o una lista, aunque sea un caso más anormal.

Sección 2: Catálogo de elementos

Detalla los elementos representados en la representación primaria. Explica qué son y cuál es su funcionalidad. En el caso de haber omitido determinados elementos en la representación primaria, se detallan aquí. Las partes pueden ser las siguientes:

- Elementos y sus propiedades:

Nombra cada elemento y lista sus propiedades, las cuales son definidas por cada vista

- Relaciones y sus propiedades:

Cada vista tiene determinados tipos de relaciones que hay entre elementos de la vista.
Generalmente se especifican las relaciones que no son mostradas en la vista primaria

- Interfaces de elementos
- Comportamiento de los elementos

Sección 3: Diagrama de contexto

Muestra cómo el sistema o la porción del sistema se relacionan con el ambiente. El objetivo es determinar el alcance de la vista. Es decir, con que parte del sistema interactúa.

Sección 4: Guia de variabilidad

Muestra cómo pueden ocurrir determinadas variaciones que son parte de la arquitectura mostrada en la vista

Sección 5: Decisión de diseño

Es explica el porqué del diseño que se ve en la vista. El objetivo de esta sección es explicar porqué el diseño es como es y proveer un argumento convincente de su razón de ser. Las elecciones de patrones, aplicaciones de tácticas, etc deben ser explicadas aquí

Documentando información extra-vistas

Se puede dividir en dos secciones:

- Vista general de la documentación

Se explica el layout de la documentación y su organización, para poder encontrar las cosas fácilmente

- Información sobre la arquitectura

Se incluye la información general del sistema, tal como el propósito del sistema y cómo se relacionan las vistas y la justificación del diseño general de todo el sistema.

Secciones:

Document control information
Información tal como la versión, la organización emisora, fecha de publicación y tal.

Documentation Roadmap
Indica qué información hay en el documento y donde encontrarla. Consiste de 4 subsecciones:

- Alcance y resumen

Explica el objetivo del documento y explica cuál es el alcance.

- Cómo está organizado

Da una pequeña sinopsis sobre cada sección del documento y su orden.

- Resumen de las vistas

Explica que vistas fueron incluidas y porque

- Cómo pueden usar los interesados

Esta sección incluye como pueden utilizar los interesados la documentación para sus intereses.

How a view is documented
Se explica la organización que se utiliza para documentar las vistas. Explica cómo encontrar la información dentro de una vista.

System overview
Un texto pequeño que describa la funcionalidad del sistema, y cualquier limitación o información importante.

Mapping between views

Como todas las vistas describen el mismo sistema, es normal que dos vistas tengan mucho en común. En esta sección se describen estas relaciones.

Las relaciones entre elementos de las vistas pueden ser de muchas a muchas. Las relaciones de vista a vista generalmente se expresan como tablas.

Rationale
Documenta decisiones arquitectónicas que se aplican a más de una vista. Generalmente se incluyen decisiones de todo el sistema que surgieron a partir de una limitación previa de la organización. Las decisiones sobre porque usar los patrones arquitectónicos fundamentales se incluyen aquí.

Directory:
Es un conjunto de material de referencia que ayuda al lector, incluyendo glosarios, índices, lista de acrónimos, etc.

Documenting behaviour

Muchas veces se requiere documentación de comportamiento que complemente las vistas estructurales al describir cómo los elementos de la arquitectura interactúan uno con el otro. Esto permite razonar sobre características particulares.

Existen dos tipos de notaciones para documentar comportamiento:

Trace-oriented languages
Un trace es un conjunto de actividades o interacciones que describen la respuesta del sistema frente a un estímulo cuando está bajo un determinado estado. El trace describe la secuencia de actividades o interacciones entre los elementos estructurales del sistema. Existen 4 maneras de documentarlos:

- Casos de uso

Describen cómo los actores pueden usar el sistema para cumplir sus objetivos. Generalmente utilizado para obtener los requerimientos funcionales de un sistema.

- Diagrama de secuencia

Muestra una secuencia de interacciones entre instancias de elementos que son parte de la documentación estructural. Muestra solo las instancias que participan del escenario. El diagrama tienes 2 dimensiones, vertical (representa el tiempo) y horizontal (las instancias)

- Diagramas de comunicación

Muestra un grafo de los elementos interactuando y anota qué interacción con número que denota orden.

- Diagramas de actividad

Son similares a los flowcharts. Muestran una secuencia de pasos y agrega una notación especial para demostrar la separación de los flujos condicionalmente y la concurrencia, así como mostrar envios de eventos.

Comprehensive model

Estas muestran el comportamiento completo de los elementos estructurales. Dado este tipo de documentación, es posible inferir todos los pasos posibles desde el estado inicial al estado final.La notacion de una maquina de estado representa el comportamiento de la arquitectura ya que se puede preveer todas las posibles situaciones.

Atributos de calidad

La calidad de un sistema va más allá de la funcionalidad. Aunque se relacionen mucho, la funcionalidad generalmente se pone en primer plano y se ignora el resto, lo cual puede traer consecuencias negativas. El no cumplimiento de otros atributos de calidad es generalmente el principal motivo para un rediseño.

Un **atributo de calidad** es una propiedad del sistema que es medible o testeable que indica que tan bien se satisface un necesidad de un interesado.

Arquitectura y los requerimientos

Las categorías de los requerimientos son las siguientes:

- **Requerimientos funcionales**

Definen qué es lo que el sistema debe hacer y cómo se debe comportar

- **Requerimientos no funcionales (de calidad)**

Son calificaciones de los requerimientos funcionales o del producto entero. Expresan atributos que deben ser cumplidos.

- **Restricciones**

Es una decisión ya tomada en la cual no se puede tomar libertad alguna. Son determinadas por factores externos generalmente

Como se trata cada uno?

- Los requerimientos funcionales se cumplen asignando responsabilidad a través del diseño, del tipo de este módulo hace tal funcionalidad
- Los no funcionales se satisfacen mediante las estructuras del diseño del sistema, y el comportamiento e interacciones de los elementos.
- Las restricciones se satisfacen aceptandolas e introduciendolas al diseño.

Funcionalidad

La funcionalidad es la habilidad del sistema de hacer el trabajo para el que se hizo. Esta no determina la arquitectura en absoluto. El arquitecto debe interesarse en cómo cumplir la funcionalidad mientras a su vez se cumplen los otros requerimientos

Consideraciones

Existen 3 problemas con las discusiones sobre atributos de calidad:

- Muchas veces las definiciones no son testeables. Es decir, no es fácil de comprobarlo
- La discusión se concentra en a qué atributo de calidad corresponde un hecho. Por ejemplo, X es performance, usabilidad, etc.
- Existen vocabularios muy distintos entre las distintas comunidades. Por ejemplo, "eventos", "ataques", "input", todo para lo mismo

La solución a estos problema es el uso de escenarios de atributos de calidad. Esto permite traer una estandarización a como probar, que cumplir y donde se aplica.

Tipos de atributos de calidad:

- **De tiempo de ejecución**: performance, usabilidad
- **De desarrollo del sistema**: modificabilidad, testeabilidad

Especificación de requerimientos de atributos de calidad

La forma estándar de especificar un requerimiento de atributo de calidad es la siguiente:

- **Fuente de estímulo**

Entidad que genera el estimulo (persona, computadora)

- **Estímulo**

Es la condición que requiere una respuesta al llegar al sistema. Es un evento, como por ejemplo, un ataque, una solicitud de modificación, etc.

- **Ambiente**

Son las condiciones bajo las que el estímulo puede ocurrir. Que debe pasar para pueda ocurrir.

- **Artefacto**

A qué porción del sistema es que el requerimiento aplica. Que parte es la que recibe el estímulo. Puede ser un pedazo o todo el sistema

- **Respuesta**

La respuesta del sistema es la actividad realizada a partir de la llegada del estímulo

- **Medida de respuesta**

Manera de medir la respuesta para saber si fue exitosa o no. Permite que el requerimiento sea testeado luego.

Tácticas para los atributos de calidad

Una táctica es una decisión de diseño que influencia en el éxito de una respuesta en un atributo de calidad. Una táctica afecta directamente a la respuesta del sistema frente a un estímulo.

El objetivo de una táctica es un solo atributo de calidad. Dentro de la táctica no se considera ningún tradeoff.

Porque es que existen las tácticas?

- Los patrones de diseño son complejos, consisten generalmente de un conjunto de decision de diseño. Esto genera que sean muchas veces difíciles de aplicar, por lo que terminan siendo modificados y adaptados. Con las tácticas, el arquitecto puede fácilmente modificar los patrones y así aumentar sus opciones para cumplir con determinados atributos de calidad
- Si no existe un patrón para lo que se está buscando, las tácticas permiten construir el diseño fácilmente a partir de los principios fundamentales.
- Al catalogar las tácticas, se facilita que el diseño sea mas sistematico.

Disponibilidad

Se refiere a la propiedad del software de estar disponible y preparado para ejecutar una acción cuando sea necesario.

Refiere a la habilidad del sistema de enmascarar y repara falla tal que el tiempo que el sistema no esté disponible no exceda el máximo deseado.

Existen dos conceptos, falla y defecto:

Falla: Implica que fue visible para el sistema o persona que está observando. Es una desviación visible de la especificación del sistema.

Defecto: Es la causa de la falla. Puede ser interna o externa. Los estados intermedios entre la ocurrencia de un defecto y la ocurrencia de una falla se llaman errores. Los defectos pueden ser prevenidos, tolerados, removidos, etc.

Si se ejecuta codigo que contiene un defecto pero el sistema es capaz de recuperarse de él sin ninguna desviación observable de la especificación del sistema, entonces no hay ninguna falla.

La disponibilidad de un sistema puede ser calculado con la siguiente fórmula:

$$\frac{MTBF}{(MTBF + MTTR)}$$

Donde MTBF es el tiempo promedio entre fallas y MTTR es el tiempo promedio en que se reparan. A partir de esta fórmula se calculan probabilidades tales como "99.9%" de disponibilidad, que luego son usados en los SLA (Service level agreements).

En los sistemas, los defectos son detectados previos a ser reportas y reparados. Se categorizan en función a la severidad y al impacto, de manera de dar la mayor cantidad de información posible para seguir una estrategia de recuperación.

Tácticas para la disponibilidad

Estas tácticas tienen como objetivo hacer que el sistema se recupere de los defectos para que el servicio ofrecido se mantenga dentro de su especificación. Existen 3 tipos de categorías de tácticas de disponibilidad

Detectar defectos:

- **Ping/echo**

Refiere a el intercambio de solicitud/respuesta. Consiste en enviar un mensaje pequeño entre nodos para demostrar el correcto alcance, funcionamiento y tiempos de comunicación.

- **Monitor**

Es un componente que está encargado de chequear el estado de varios otros componentes del sistema. Utilizar otras tácticas a su vez para cumplir su función, por ejemplo heartbeats, o self-tests

- **Heartbeat**

Consiste en enviar mensajes periódicos hacia un monitor del sistema para avisar que sigue en funcionamiento. Por temas de performance se puede enviar en conjunto de otros datos. La diferencia con el ping es quien es el encargado de arrancar la comunicación (el monitor o el elemento)

- **Time stamp**

Mediante la utilización de una marca de tiempo, se puede verificar el orden de una secuencia de eventos

- **Sanity checking**

Chequea la validez de determinadas operaciones del sistema, verificando su output

- **Condition monitoring**

Consiste en el checkeo de las condiciones de un proceso o dispositivo. Los checksums son un caso de esto

- **Voting**

Se utilizan varios componentes (impares) que hacen lo mismo y reciben el mismo input. Luego un elemento lleva el proceso de votación. En el caso de encontrar inconsistencias entre las 3 salidas, se reporta un defecto y se usa como salida la 'más salida' es decir, la mayoría simple.

- **Replicacion**

Es una implementación de voting donde todos los elementos son clones entre sí. No protegen frente a problemas de implementación

- **Functional redundancy**

Es una implementación de voting donde siempre se debería tener la misma salida pero se tiene distinta implementación

- **Analytic redundancy**

Es una implementación de voting en donde se permite diversidad de implementación y de salidas.

- **Exception detection**

Refiere a la detección de una condición del sistema que altera el flujo normal de ejecución.

- **Self-test**

Consiste en que los componentes corran procesos que testeen a sí mismos su correctitud. Pueden ser iniciados por ellos mismos o por un monitor.

Recuperación de defectos

Se subdividen en dos categorías. Tácticas para reparar y reparar y tácticas para reintroducir.

Preparacion y reparacion:

- **Redundancia activa**

Refiere a una configuración donde todos los nodos del sistema (activos y redudantes) reciben y procesan la misma información, de manera que los nodos redundantes siempre están sincronizados con los activos. En caso de de que se caiga uno de los nodos activos, el redundante puede tomar su lugar rápidamente

- **Redundancia pasiva**

Similar a la anterior, solo que los nodos activos periódicamente le transmiten su información a los nodos redundantes. Provee un balance entre performance y gasto de cómputo y disponibilidad

- **Cold spare**

Similar a las anteriores, pero los nodos redundantes no contienen informacion, sino que en el caso de un defecto y tengan que entrar al sistema, la performance es muy baja.

- **Exception handling**

Cuando se detecta una exception, se maneja de alguna manera. Existen muchas maneras, como códigos de error o el uso de clases de excepciones. El sistema puede a partir de esto detectar la causa de la falla, corregirla y reintentar la operación

- **Rollback**

Permite ir al sistema ir "para atrás" a un estado del sistema previo que es sabido que es correcto. Se deben mantener copias del estado del sistema regularmente, ya sea en un intervalo de tiempo fijo o previo a una operación compleja

- **Software upgrade**

Consiste en entregar una nueva versión del software o parches, que corrigen determinados defectos sin afectar al sistema

- **Retry**

Asume que la causa del defecto fue transitoria, por lo que intenta realizar de vuelta la operación. Se usa mucho en operaciones a través de la red

Reintroduccion:

- **Shadow**

Consiste en tener un nodo que había fallado en un modo "shadow" antes de volver a estar activo, de manera de verificar que fue corregido correctamente

- **Sincronización del estado**

Se sincroniza el estado de los nodos activos con el nodo que se quiere reingresar. Esto puede ser más fácil o más difícil dependiendo de otras tácticas utilizadas (active redundancy => más fácil).

- **Escalating restart**

Se varía la granularidad de los componentes a reingresar a partir de distintos niveles, comenzando por lo componentes más granulares.

- **Non-stop forwarding**

Prevenir defectos

- **Removal from service**

Consiste en retirar un componente temporalmente cuando se detecta posible fallas del sistema. Por ejemplo, se remueve un componente y se resetea para evitar posibles memory leaks.

- **Transactions**

Consiste en el uso de transacciones que cumplan con los principios ACID para poder evitar posibles errores, como dos procesos modificando el mismo conjunto de datos

- **Predictive model**

Se busca obtener modelos que describan comportamientos que pueden llevar a posibles fallas a futuro, "predicen" que un error ocurrirá dadas determinadas circunstancias.

- **Prevención de excepciones**

Refiere a las tácticas empleadas para prevenir las excepciones. El uso de clases de excepciones, punteros inteligentes, son unos de los ejemplos

- **Aumentar el conjunto de estados de competencia**

Los estados de competencia son estados del sistema en donde está operable. Consiste en agrandar los casos en donde el sistema se encuentra operable aunque haya ocurrido un defecto.

Modificabilidad

El mayor costo de un sistema de software ocurre luego de que liberado por primera vez. El cambio en un software es inevitable. Se agregan, remueven y modifican requerimientos continuamente, se arreglan bugs, etc.

Las 4 preguntas más importantes a la hora de evaluar modificabilidad son las siguientes:

- **Que puede cambiar?**

Un cambio puede surgir para cualquier aspecto del sistema (performance, plataforma, capacidad)

- **Cual es la probabilidad de que cambie?**

No se puede planear todos los cambios, es tarea del arquitecto priorizar qué cambios son los más probables e importantes de tener controlados.

- **Cuando ocurren y quien los hace?**

Se pueden hacer cambios de implementación, en compilación, en configuración, o en ejecución.

- Cual es el costo?

Se debe evaluar la diferencia entre el costo de introducir el mecanismo para hacer el sistema más modificable y cuando cuesta hacer la modificación utilizando el mecanismo. La siguiente fórmula se utiliza:

$$N \times \text{Cost of making the change without the mechanism} \leq \text{Cost of installing the mechanism} + (N \times \text{Cost of making the change using the mechanism}).$$

Tácticas para la modificabilidad

Coupling (Acoplamiento)

El acoplamiento indica que tanto cambian juntos dos elementos en un módulo. Es decir, mide que en el caso de ocurrir un cambio, en cuántos módulos se debe hacer ese cambio. Se busca que el acoplamiento sea bajo, ya que siempre se debe desear tener que realizar un cambio en la menor cantidad de módulos posibles. Dos módulos están altamente acoplados si al hacer un cambio en uno, se debe hacer un cambio en el otro también.

Cohesion

La cohesión indica que tan relacionadas están las responsabilidades de un módulo.

La cohesión de un módulo es la probabilidad que un cambio que afecta una responsabilidad afecte otra responsabilidad del módulo también. Se debe buscar que la cohesión sea alta, ya que esto lleva a que los módulos tengan una única responsabilidad y sea más simple modificarlos.

Existen 4 tipos de tácticas:

Reducing size of a module:

- **Dividir un módulo**

Si un módulo a ser modificado incluye una gran cantidad de responsabilidades, el costo de modificarlo será alto. Refinar el módulo en varios módulos más pequeños achicara este costo

Aumentar la cohesión

Si un módulo A tiene baja cohesión, se puede mejorar removiendo responsabilidad que no son afectadas por los cambios. Estas tácticas consisten en mover responsabilidades de un módulo a otro.

- **Increase semantic coherence**

Si las responsabilidades A y B de un módulo no tienen el mismo objetivo, entonces deben estar en distintos módulos.

Reducir el acoplamiento

Reducir el acoplamiento entre un modulo A y un modulo B disminuye el costo esperado de una modificación en A.

- **Encapsular**

Consiste en introducir una interfaz explícita al módulo (API). Toda comunicación de los módulos externos a este módulo deben ser mediante la interfaz. Si la API es suficientemente abstracta respecto a los detalles del módulo, en caso de querer realizar un cambio en este módulo, este no se propagara a los que lo utilicen.

- **Usar un intermediario**

Consiste en romper una dependencia. Si existe una dependencia entre A y B, se rompe utilizando un intermediario. De esta manera, un cambio en B es menos costoso. El tipo del intermediario depende del tipo de la dependencia.

- **Restrict dependencies**

Se restringe que módulos pueden interactuar entre sí. Esto se logra restringiendo la visibilidad de los módulos y mediante la autorización. Esto se puede ver en arquitectura basadas en layers (no se permite comunicarse con una layer no adyacente)

- **Refactor**

Cuando dos módulos son afectadas por un mismo cambio porque son muy similares, se extraen a un nuevo módulo que tenga esa responsabilidad.

- **Abstract common services**

Cuando dos módulos proveen servicios similares, puede ser efectivo implementar estos servicios en un módulo más abstracto. A partir de esto cualquier modificación al servicio en común debe ocurrir en un solo lugares, reduciendo el costo.

Diferir el tiempo de enlace

Si se puede diferir el momento en el que se acopla a un valor o elementos específico un programa, es mejor. Esto permite que realizar cambios sea menos costoso en más etapas del software. Es tan simple como pensar en términos de parametrización. Siempre es mejor utilizar elemento parametrizados que fijos.

Existen maneras de aplazar este acoplamiento y generar que el software sea más parametrizado. Dependiendo del momento, existen las siguiente tácticas:

Tiempo de compilación:

- Component replacement (build script o makefile)
- Compila-time parametrization
- Aspects

Tiempo de deploy

- Configuration-time binding

En tiempo de inicialización:

- Resource files

En tiempo de ejecución:

- Runtime registration
- Dynamic lookup
- Plugins
- Polymorphism
- Publish-subscribe

Performance

La performance es la habilidad del software de cumplir con los requisitos de tiempo. Cuando ocurre un evento, el sistema debe responder dentro un determinado tiempo.

Los eventos pueden ser variados, pero siempre se pueden hacer una caracterización para a partir de esto, tomar decisiones de performance. Todos los sistemas tienen requerimientos de performance, ya sean explicitos o implicitos.

Tácticas para la performance:

El objetivo de las tácticas de performance es el de generar una respuesta a un evento entrando dentro de un marco de tiempo determinado. Estas controlan el tiempo que demora la respuesta en ser generada.

Mientras el sistema está procesando la respuesta, puede estar en dos situaciones, o trabajando en la respuesta o bloqueado por alguna razón. De aquí surgen los dos conceptos principales, processing time y blocked time.

Processing time:
El procesamiento consume recursos, los cuales llevan tiempo. Los eventos son manejados por la ejecución de uno o más componentes, cuyos tiempo empleado es un recurso. Los recursos pueden ser hardware (CPU, network, etc) o software (entidades, buffers).

Distintos recursos se comportan distinto frente a la saturación, ya que pueden bajar bruscamente(memoria ram) o proporcionalmente(cpu).

Blocked time:
Se puede estar bloqueado por 3 razones:

- Contention of resources: Algunos recursos solo pueden ser utilizados por uno a la vez, por lo que los restantes deben esperar
- Availability of resources: Los recursos pueden encontrarse momentaneamente no disponibles, por alguna falla o mantenimiento por ejemplo.
- Dependencia de otra computacion: Cuando se deben sincronizar resultados entre operaciones o se necesita el resultado, se debe esperar

De esto surgen los dos tipos de tácticas: Control resource demand and Manage resources

Control resource demand

Estas tácticas buscan controlar el acceso a los recursos.

- **Manage sampling rate**

Busca reducir la frecuencia con que los datos son recibidos, de manera de reducir la demanda. Generalmente se pierde un poco de fidelidad de los datos. Se utiliza cuando se prioriza la baja latencia frente a la fidelidad de los datos. Es muy utilizado en sistemas de streaming, por ejemplo

- **Limit event response**

Cuando llegan eventos al sistema demasiado rápido para ser procesados, estos deben ser puestos en una cola hasta que puedan ser procesados. También se puede controlar un máximo de elementos encolados para tener un proceso más predecible.

- **Priorizar eventos**

Si no todos los eventos son igual de importantes, se puede imponer un esquema de prioridad antes los eventos entrantes. Se procesaran primero los de mayor prioridad ante un caso de escasos recursos.

- **Reduce overhead**

Determinadas tácticas para otros atributos (como introducir intermedios) incluyen clases y elementos que no son necesarios para el funcionamiento pero que introducen overhead, es decir, introducen elementos innecesarios que pueden disminuir la performance al consumir mayores recursos. Otra manera es la "co-location" que consiste en tener los componentes que se comunican juntos para reducir el tiempo de comunicación

- **Bound execution times**

Se establece un límite en cuanto tiempo de ejecución se usa para responder a un evento. Por ejemplo, se pueden reducir la cantidad de iteraciones en un algoritmo. El resultado puede ser menos preciso. Se basa mucho en el "good enough" y se utiliza con manage sampling rate comúnmente

- **Increase resource efficiency**

Mejora de algoritmos

Manage resources

Aunque no sea posible controles la demanda de los recursos, su manejo si puede ser optimizable.

- **Increase resources**

Agregar poder de cómputo (procesadores, ram, etc) tienen el potencial de reducir la latencia

- **Introducir concurrencia**

Si dos eventos pueden procesarse en paralelo, el tiempo bloqueados ser reduce. Para esto se utilizan streams de eventos distintos en distintos threads.

- **Maintain multiple copies of computations**

Tener muchos servidores es tener réplica de computación. Se reduce la carga que puede tener a llegar en un servidor si toda la computación ocurriera en el. En caso de ser servidores idénticos, se puede utilizar un load balancer.

- **Maintain multiple copies of data**

Caching consiste en tener una copia de determinados datos guardada en un lugar con acceso más rápido. Data replication consiste en tener copias separadas de los datos para reducir la sobrecarga con muchos accesos. Se debe tener especial cuidado de mantener actualizada la información ya que son copias.

- **Bound queue sizes**

Controla el máximo número de llegadas de eventos y en consecuencia los recursos usados para procesarlos.

- **Schedule resources**

Todos los recursos son programados (CPU, buffers, etc). Cada recurso debe ser programado en función de sus características y las características de los que los acceden.

Seguridad

Es la capacidad de un sistema de proteger los datos y la información de acceso no autorizado mientras que se le de correcto acceso a las entidades que sí lo estén. Un acción con el objetivo de hacer daño a un sistema se llama **ataque**

Existen 3 características de la seguridad (CIA)

- **Confidencialidad**

Los datos o servicios deben estar protegidos del acceso no autorizado. Un hacker no puede acceder a tus datos de la cuenta de banco

- **Integridad**

Los datos o servicios no deben poder ser manipulados de una manera no autorizada. Nadie puede cambiar tu nota de facultad excepto tu profesor

- **Disponibilidad**

El sistema siempre debe estar disponible para un uso autorizado. Un ataque a el sistema de bibliotecas no te frena de reservar un libro

A su vez existen 3 características mas que apoyan a las previamente mencionadas:

- Autenticación

Verifica las entidades de las partes en una transacción.

- No-repudio

Garantiza que la entidad que envió un mensaje no pueda negar haberlo enviado posteriormente

- Autorización

Le da privilegio a los usuarios para hacer determinadas tareas

Tácticas para la seguridad:

Se puede hacer una analogía con la seguridad física (de un local por ejemplo). De ahí se detectaron los siguientes 4 tipos de tácticas

Detectar ataques

- **Detectar intrusos**

Se compara el tráfico dentro del sistema con una serie de datos o patrones conocidos como posiblemente maliciosos.

- **Detectar negación de servicio**

Es la comparación del tráfico entrante con datos o patrones conocidos de denegación de servicio

- **Verifica la integridad del mensaje**

Utiliza herramientas como hashes o checksums para verificar que el mensaje sea efectivamente el que fue enviado. Cualquier cambio en el mensaje genera que el hash o el checksum sean distintos

- **Detectar demora del mensaje**

Detecta posibles ataque de man-in-the-middle. Si hay un intermediario viendo las respuestas, probablemente la latencia sea mayor.

Resistir ataques

- **Identificar actores**

Identifica quién o qué es que está introduciendo el input al sistema. Puede ser un id, IP, protocolos, etc

- **Autenticación de actores**

Verifica que el actor es efectivamente quien dice ser. Se hace mediante passwords, certificados, biometrics, etc

- **Autorizar actores**

Verifica que el autor tenga los permisos necesarios para acceder o modificar los datos o servicios deseados.

- **Limitar el acceso**

Involucra controlar quien y que puede acceder a que partes del sistema. Se puede limitar acceso a recursos (CPU, memoria, etc) o a servicios. Un firewall es un ejemplo de esto

- **Limitar la exposición**

"Security by obscurity". Consiste en limitar lo más posible los puntos de entrada al sistema. Esto reduce la posibilidad de un ataque y el posible daño. También se pueden dividir las responsabilidades en muchos recursos para no tener un único punto de falla (ataque)

- **Encriptar datos**

Se provee encriptación a todos los datos para que aquellos que tengan acceso no autorizado a ellos no puedan sacar información útil. Hay casos donde pueden no llegar a verse (bases de datos) pero hay casos donde es público (requests a través de internet) donde la encriptación es la única manera de protegerlos.

- **Separar entidades**

Se separan las entidades físicamente de manera de reducir la posibilidad de que un ataque se propague. Por ejemplo, se puede dividir los datos delicados de los no delicados.

Reaccionar a ataques

- **Revocar acceso**

Si se detecta que hubo un acceso no autorizado, se limita el acceso a datos sensibles hasta que se pueda frenar. Puede ser a todos los usuarios o sistemas o solo al que fue detectado como no autorizado

- **Bloque de computadora**

Consiste en bloquear la computador en caso de detectar un acceso erróneo varias veces

- **Informar actores**

Los ataques muchas veces necesitan una respuesta física de operadores, tales como devops o managers. Se debe notificar a estos de la manera mas rapida posible de que está habiendo un ataque

Recuperación de ataques

- **Auditoria**

Mantener un registro de quien ejecuta cada acción y los resultados puede ser efectivo para buscar culpables y elaborar nuevas defensas

Testeabilidad

Refiere a la facilidad que tiene un sistema para mostrar sus defectos a través del testing. Específicamente, refiere a la probabilidad de que si el software cuenta con un defecto, este se vea al ejecutar los tests. Esto es importante ya que los defectos se desean encontrar lo más rápido posible (en la siguiente corrida de tests).

Para lograr esto, debe ser posible controlar el input de los componentes y observar el output. El testing puede ser hecho por developers, software, usuarios, etc.

La manera de medir la testeabilidad consiste que que tan efectivo los tests son en descubrir defectos y cuando demoran en correrse para lograr un cubrimiento mínimo.

Tacticas para la testeabilidad

El objetivo de estas tácticas es permitir que el testing sea más simple al finalizar una porción del desarrollo. Existen dos categorías: Control and observe systema state y limiting complexity

Control and observe system state

Estas tácticas tienen como objetivo que los componentes puedan mantener estado de la información, permitir a los testers acceder a la información y modificarla, de manera de luego poder observar los resultados. Estas tácticas introducen abstracciones al software que de no ser necesario el testing no existirían.

- **Interfaces especializadas**

Tener interfaces especializadas para testing permiten controlar y acceder a datos que no es fácil obtener durante un flujo normal, algunas de estas son

- Getters y setters de variables importantes
- Método del objeto llamado *reporte* que retorne el estado completo del objeto.
- Método del objeto llamado *reset* que retorne el estado del objeto a uno inicial
- Métodos para output verbose, logging, etc

Es importante tenerlas correctamente separadas de las interfaces usadas para producción (funcionamiento normal)

- **Grabar y playback**

Es difícil a veces recrear un estado donde se encontró un defecto. Si se va grabando el estado en cada paso, luego esta grabación puede ser utilizada para recrear este estado.

- **Localizar el guardado del estado**

Para poder inicializar el sistema o una parte de él con un determinado estado, es importante que este sea guardado en un único lugar. De esta manera, es mas facil de cambiarlo y mockearlo después. Si el estado está distribuido por todos los componentes y no está abstraído, puede ser muy difícil esto.

- **Abstraer fuentes de datos**

Similar a la anterior, hace más fácil controlar los datos que utiliza el sistema. Hace que sea más fácil cambiar los datos por datos para testing. Un caso claro es el de las bases de datos de test.

- **Sandboxing**

Consiste en aislar una porción del sistema del resto del mundo real para poder experimentar sin preocuparse de las consecuencias en el resto. Testing se basa mucho en poder operar sin tener consecuencias en el mundo real. Una manera muy común es la de virtualizar los recursos.

- **Assertions ejecutables**

Son pedazos de código que validan que el programa no esté en determinado estado erróneo. En caso de que no se cumpla, el sistema lo indica.

- **Reemplazar componentes**

Consiste en cambiar la implementación de componentes con una implementación distinta para facilitar el testing

- **Preprocessor macros**

Mediante parametros al compilador, se compila codigo que permita testear mas facilmente solo en ambientes de testing

Limitar complejidad

El software complejo de difícil de testear. Estas tácticas buscan evitar esto

- **Limitar la complejidad estructural**

Consiste en hacer la estructura más simple. Esto puede ser resolviendo dependencias cíclicas entre componentes, encapsulando dependencias externas, o limitando las dependencias entre componentes en general. Otras tácticas consisten en limitar la estructura de jerarquía de clases (OOP). Todas las tácticas de modificabilidad ayudan a su vez a limitar la complejidad.

- **Limitar no determinismo**

Los sistemas no deterministas son más difíciles de testear. Se debe buscar que esto ocurra lo mínimo posible. En determinados sistemas, es imposible, pero en esos casos se usan otras tacticas.

Estilos arquitectonicos

Un patrón arquitectónico es un paquete de decisiones de diseño que se encuentran regularmente en la práctica, que tiene propiedades que permiten su reuso y que describen una clase de arquitecturas. Un patrón es descubierto en la práctica, no se inventa.

Las tácticas son los "bloques de construcción" del diseño, a partir de los cuales se crean los patrones. Cada táctica se concentra en una única estructura para resolver un solo problema sin tomar en cuenta los restantes.

Las tácticas son los átomos y los patrones son las moléculas.

Estructura de un patrón

Un patrón arquitectónico establece una relación entre los siguiente elementos. Estos forman el template para la documentación del patrón.

- **Contexto**

Una situación recurrente del mundo real que genera un problema

- **Problema**

Problema que ocurre en el contexto.

- **Solución**

Un solucion arquitectonica correctamente abstraída. Se describe que estructura arquitectónicas lo resuelven, incluyendo cómo balancear las distintos fuerzas involucradas. Se describen responsabilidad, relaciones, comportamiento en runtime, etc. Se describe con:

- Tipos de elementos
- Mecanismos de interacción

Se debe dejar claro que atributos de calidad son afectados, tanto positiva como negativamente.

Patrones:

Es importante aclarar que implementar un patrón no es "all-or-nothing", si no que en la práctica se pueden hacer pequeñas violaciones a los patrones mientras sean justificables. Los patrones se categorizan en función del tipo dominantes de los elementos que se usan en el (módulos, componentes y conectores, etc)

Patrones de módulos:

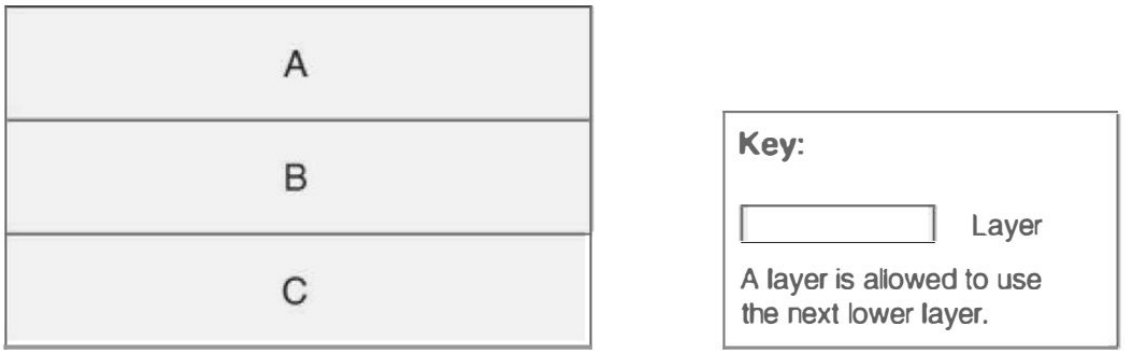
Patrón de capas

Contexto:
Es necesario que distintas porciones del sistema evoluciones y sean desarrollados por separado, independientemente. Por esta razón, los desarrolladores del sistema necesitan una clara separación de responsabilidades, para que se puedan desarrollar independientemente

Problema:
El sistema debe ser segmentado de una manera que los módulos puedan ser implementados por separado con poca interacción entre las partes, soportando portabilidad, modificabilidad y reuso

Solución:
Se divide el software en capas. Cada capa es un conjunto de módulo que ofrece un conjunto cohesivo de servicios. Las relaciones entre capas debe ser unidireccional y sólo adyacente en un orden estricto. La interacción entre capas es mediante una interfaz pública definida.

El diseño de las capas puede agregar complejidad y costo al desarrollo del sistema y agregar penalidad en la performance, ya que se introducen muchos intermediarios.



Patrón de publish-subscribe

Contexto:
Existe una cantidad de productores y consumidores de información independientes que deben interactuar. No se conoce la cantidad exacta de cada uno ni los datos.

Problema:
Como se crea un mecanismo de integración para que interactúen sin que conozcan la entidad del uno y el otro ni su existencia?

Solución:
Los componentes interactúa mediante mensajes o eventos. Los componentes se pueden subscribir a un conjunto de eventos. Es el trabajo de la infraestructura de asegurarse que cada evento publicado sea enviado a cada subscriptor.

Se agrega una capa de indirección entre productores y consumidores. Puede tener un efecto en la latencia y en la potencial escalabilidad dependiendo de la implementación. No se da ninguna garantía que el mensaje fue enviado (el productor no sabe quien esta suscrito). No se debe utilizar cuando esta informacion es crítica.

Permite facilidad de modificabilidad ya que no hay acoplamiento entre los productores y consumidores (no se conocen).

Ejemplos: Mailing lists, redes sociales donde se indica que ocurrió a un conjunto de amigos, GUI donde se avisa a los handlers de que un botón fue presionado.

Implementaciones:

- Basadas en listas: Cada productor tiene una lista de consumidores. Más acoplada, pero más performante
- Broadcast-based: Se publican eventos y todos los consumidores las reciben. Ellos deciden si es de interés o no. Puede ser muy ineficiente si hay muchos eventos

- Content-based: Un componente se subscribe a un contenido con tales características, y le llegan los eventos que cumplan con ellas.

En la práctica, generalmente se utiliza un middleware, que maneja las conexiones y la información. Funciona como un broker.

Multi-tier pattern:

Este puede ser considerado de allocation o de C&C, dependiendo de los elementos utilizados.

Contexto:
Cuando se debe desplegar un sistema distribuidamente, es necesario dividir la infraestructura del sistema en distintos conjuntos.

Problema:
Como se puede dividir el sistema en un número de estructuras computacionales independientes (software y hardware) que se comuniquen entre sí?

Solución:
Se divide en un conjunto de agrupaciones lógicas llamadas *tiers.* Esta division se puede hacer en funcion de muchos criterios, tales como tipos de componentes, ambientes de ejecución, mismo propósito en tiempo de ejecución.

Relación entre tácticas y patrones

Un patrón puede usar muchas tácticas que busquen priorizar distintos atributos de calidad, no tiene porque ser el mismo.

Por ejemplo, el patrón de capas utiliza muchas tácticas: Aumentar la coherencia semántica, abstraer servicios en común, encapsular, restringir dependencias y uso de intermediarios. Si una de estas tácticas no es utilizada, el patrón pierde muchas de sus ventajas, haciendo que no sea efectivo.

Uso de tácticas para aumentar los patrones

Un patrón es descrito como una solución a una clase de problemas en un contexto. Cuando es aplicado, este contexto se hace muy específico. Para hacer que funcione en este contexto, se debe analizar desde dos perspectivas:

- Los trade offs que genera el patrón. Tanto como un patrón beneficia determinados atributos de calidad, afecta a otros. Se debe comparar esto con los atributos que son más importantes en nuestro contexto

- Otros atributos de calidad que no son tomados en cuenta por el patrón pero que sí afecta y nos importan.