4. Entendiendo los atributos de calidad:

A quality attribute (QA) is a measurable or testable property of a system that is used to

indicate how well the system satisfies the needs of its stakeholders.

veremos como:

* como expresar la calidad que queremos en nuestra arquitectura
* como cumplirlas / lograrlas
* como determinar las decissiones de diseno que respeten estos atributos de calidad

4.1 Arquitectura y requerimientos

* Functional requirements:

Estos requerimientos dicen lo que el sistema debe hacer.

* Quality attribute requirements:

Estos requerimientos son calificaciones sobre el producto entero o sobre un requerimiento funcional. Dice que tan rapido debe hacer tan requerimiento o si es para el producto que tan rapido se debe hacer el deploy o una limitacion sobre costos.

* Constraints

Es una decision de diseno especifica que ya fue tomada. Tiene que cumplirse. No da lugar a una interpretacion distinta.

La respuesta sobre cada una de estas cosas sobre arquitectura es:

1. Functional requirements are satisfied by assigning an appropriate sequence of responsibilities throughout the design. As we will see later in this chapter, assigning responsibilities to architectural elements is a fundamental architectural design decision.
2. Quality attribute requirements are satisfied by the various structures designed into the architecture, and the behaviors and interactions of the elements that populate those structures.
3. Constraints are satisfied by accepting the design decision and reconciling it with other affected design decisions.

4.2. Funcionalidad

Funcionalidad es la habilidad que tiene el sistema de hacer segun lo intencionado/pactado.

La funcionalidad no determina la arquitectura. Sino que varias funcionalidades la determinan.

Si la funcionalidad fuese lo unico que importara se haria un solo componente sin estructura interna. Pero lo que se hace es agregar modules, layers, classes, services, databases, apps, threads, peers, tiers para que se entienda y para otros propositos.

La funcionalidad es independiente de cualquier estructura particular, la funcionalidad es lograda asignando responsabilidades a elementos de arquitectura.

4.3 consideraciones de un atributo de calidad

a performance QA annotation might describe how quickly the dialog will appear; an availability QA annotation might describe how often this function will fail, and how quickly it will be repaired; a usability QA annotation might describe how easy it is to learn this function.

Desde una vista de una arquitectura, hay tres problemas con discusiones previas sobre sistemas de atributos de calidad:

* la descripcion provista para un atributo de calidad no son testeables. no quiere decir nada que un sistema va a ser modificable. todos los sistemas deberian ser modificables.
* A veces cuando hay un ataque a la aplicacion es dificil de atribuir la responsabilidad a un atributo de calidad. si fue de performance, seguridad, o usabilidad
* Otro discusion sale de que cada departamento le da diferentes definiciones a las cosas. Ejemplo el departamento de seguridad le llama ataques, el departamento de performance lo llama eventos, el departamento de availability lo llama “failures” de un sistema. Todo refiere a una misma cosa pero son descriptas con diferentes terminos.

4.4 especificando los atributos de calidad

Los requerimientos de un atributo de calidad deberian ser testeables y sin ambiguedad.

Se describe un formulario para la expresion de un atributo de calidad

* Stimulus

usamos la palabra estimulo para describir un evento que llega al sistema

* Stimulus source

el estimulo debe tener una fuente

* Response

como deberia responder el sistema frente a este estimulo

* Response measure

determina si un requisito se satisface tambien si la respuesta es satisfactoria.

When the response occurs, it should be measurable in some fashion so that the requirement can be tested.

Estas cuatro caracteristicas son las mas importantes a la hora de describir un atributo de calidad pero podrian agregarse dos mas

* Environment.

set de circunstancias en la cual aplica el escenario

* Artifact

el artifact es la porcion del sistema donde aplica el requerimiento

collection of systems, the whole system,

4.5. Achieving Quality Attributes through Tactics (tacticas para lograr los atributos de calidad

Disponibilidad

Introducción

La disponibilidad se refiere a la capacidad de un sistema para enmascarar o reparar fallos de tal manera que el período acumulado de interrupción del servicio no exceda un valor requerido en un intervalo de tiempo especificado.

La disponibilidad está estrechamente relacionada con la seguridad. Un ataque de denegación de servicio está explícitamente diseñado para hacer que un sistema falle, es decir, para que no esté disponible. La disponibilidad también está estrechamente relacionada con el rendimiento, porque puede ser difícil saber cuándo un sistema ha fracasado y cuándo simplemente es escandalosamente lento para responder.

Por último, la disponibilidad está estrechamente relacionada con la seguridad, que se ocupa de evitar que el sistema entre en un estado peligroso y de recuperar o limitar el daño cuando lo hace.

Fundamentalmente, la disponibilidad consiste en minimizar el tiempo de interrupción del servicio mediante la mitigación de fallas. El fallo implica visibilidad a un sistema o observador humano en el ambiente.

Es decir, un fracaso es la desviación de un sistema de su especificación, donde la desviación es externamente visible. Una de las tareas más exigentes en la construcción de un sistema de alta disponibilidad y tolerante a fallos es entender la naturaleza de los fallos que pueden surgir durante el funcionamiento (consulte la barra lateral "Planificación de la falla").

Una vez que se entienden, las estrategias de mitigación se pueden diseñar en el software.

La causa de un fallo se denomina falla. Un fallo puede ser interno o externo al sistema considerado.

Los estados intermedios entre la ocurrencia de un fallo y la ocurrencia de una falla se llaman errores. Las fallas pueden prevenirse, tolerarse, eliminarse o preverse. De esta manera un sistema se vuelve "resistente" a fallas.

Debido a que un fallo del sistema es observable por los usuarios, el tiempo para reparar es el tiempo hasta que el fallo ya no es observable.

Esto puede ser un breve retraso en el tiempo de respuesta o puede ser el tiempo que tarda alguien en volar a un lugar remoto en los Andes para reparar un pedazo de maquinaria de minería (como nos lo contó una persona responsable de reparar el tan En un motor de máquina de minería).

La noción de "observabilidad" puede ser complicada: el virus Stuxnet, por ejemplo, pasó desapercibido durante mucho tiempo aunque estaba haciendo daño. Además, a menudo nos preocupa el nivel de capacidad que queda cuando un fallo ha ocurrido en un modo de funcionamiento degradado.

La distinción entre fallas y fallas permite la discusión de estrategias de reparación automática. Es decir, si se ejecuta un código que contiene un fallo pero el sistema es capaz de recuperarse del fallo sin que se observe ninguna desviación del comportamiento especificado, no hay fallo.

La disponibilidad de un sistema se puede calcular como la probabilidad de que proporcionará los servicios especificados dentro de los límites requeridos durante un intervalo de tiempo especificado. Cuando se hace referencia al hardware,

Bien conocida expresión utilizada para derivar estado estacionario disponibilidad:

MTBF se refiere al tiempo medio entre fallas y MTTR se refiere al tiempo medio de reparación. En el mundo actual, esta fórmula debe ser interpretada para significar que al pensar en la disponibilidad, usted debe pensar en lo que hará que su sistema falla, lo probable que se produzca, y que habrá algún tiempo requerido para repararlo.

5.2 Tácticas para Durabilidad

Un fallo ocurre cuando el sistema ya no entrega un servicio que es consistente con su especificación; Este fracaso es observable por los actores del sistema. Un fallo (o combinación de fallos) tiene el potencial de causar un fallo. Las tácticas de disponibilidad, por lo tanto, están diseñadas para permitir que un sistema resista fallas del sistema de modo que un servicio que está siendo entregado por el sistema siga siendo compatible con su especificación. Las tácticas que discutimos en esta sección evitan que las fallas se conviertan en fallas o al menos limiten los efectos de la falla y hagan posible la reparación.

Las tácticas de disponibilidad pueden categorizarse como una de tres categorías: detección de fallos, recuperación de fallas y prevención de fallas. La categorización de las tácticas para la disponibilidad se muestra en la Figura 5.5 (en la página siguiente). Tenga en cuenta que a menudo es el caso de que estas tácticas serán proporcionados por una infraestructura de software, como un paquete de middleware, por lo que su trabajo como arquitecto es a menudo una de elegir y evaluar (en lugar de aplicar) las tácticas de disponibilidad y La combinación correcta de tácticas.

Modificabilidad

Introducción

La modificabilidad es sobre el cambio, nos centramos en el costo y riesgos de hacer cambios.

Para planear la modificabilidad un arquitecto debe considerar:

1. Que puede cambiar?
2. Todo.
3. Cual es la probabilidad de un cambio?
4. Se debe decidir que cambios son probables, cuales tenemos que soportar y cuales no.
5. Cuando es hecho el cambio y por quien?
6. Cambios se pueden hacer a la implementación, mientras se compila, durante build, durante la config del sistema, o durante la ejecucion. Los pueden realizar un desarrollados, usuario final o el administrador del sistema.
7. Cual es el costo del cambio?
8. Hacer un sistema mas modificable involucra dos tipos de costos:
   1. El costo de introducir el mecanismo(s) para hacer el sistema mas modificable.
   2. El costo de hacer de hacer la modificación usar los mecanismo(s).

Para un numero N de modificaciones similares la justification para el cambio es:

N es una predicción.

7.2 Tácticas para Modificabilidad

Para entender modificabilidad, arrancamos con acoplamiento y cohesion.

Los módulos tienen responsabilidades, cuando un cambio causa que un modulo se modifique sus responsabilidades cambiar en alguna forma.

Alto acoplamiento es el enemigo de Modificabilidad.

Cohesion mide en un modulo que tan fuerte están relacionadas las responsabilidades.

Informalmente Cohesion mide “unity of purpose” de un modulo.

Unity of Purpose puede medirse por el cambio de escenarios que afecta el modulo.

La Cohesion de un modulo es la probabilidad de que al cambiar de escenario afecte alguna responsabilidad que pueda afectar mas que a ella misma.

Cuanto mas alta Cohesion menos probabilidad de que un cambio afecte multiples responsabilidades. -> Es bueno.

Parámetros para las tácticas:

* Tamaño de un modulo.
* Las tácticas que dividen módulos reducirán el costo de realizar una modificación al módulo que se está dividiendo, siempre que la división se elija para reflejar el tipo de cambio que es probable que se realice.
* Acoplamiento.
* Reducir la fuerza del acoplamiento entre dos módulos A y B disminuirá el coste esperado de cualquier modificación que afecte a A. Las tácticas que reducen el acoplamiento son aquellas que colocan intermediarios de varios tipos entre los módulos A y B.
* Cohesion.
* Si el módulo A tiene una cohesión baja, entonces se puede mejorar la cohesión eliminando las responsabilidades no afectadas por los cambios anticipados.
* Binding Time of Modification:
* Una arquitectura que esté adecuadamente equipada para adaptarse a las modificaciones tardías del ciclo de vida tendrá, en promedio, un coste menor que una arquitectura que obligue a la misma modificación a hacerse antes. La preparación del sistema significa que algunos costos serán nulos o muy bajos para las modificaciones tardías del ciclo de vida.
* Reducir el tamaño de un modulo:
  + Separar un Modulo.
* Aumentar Cohesion:
  + Incrementar la coherencia de la semantica.
* Reducir Acoplamiento:
  + Encapsular.
  + Usar un intermediado que rompa la dependencia.
  + Restringir dependencias.
  + Refactor.
  + Servicios comunes abstractos.
* Defer Binding:
  + Tácticas para enlazar valores en tiempo de compilación o build time:
    - Remplazo de componentes.
    - Compile-time parametizacion.
    - Aspectos
  + Tácticas para enlazar valores en tiempo de despliegue:
    - Configuracion-time binding.
  + Tácticas para enlazar los valores en el inicio o en tiempo de inicialización:
    - Resource Files.
  + Tácticas para enlazar valores en tiempo de ejecución:
    - Registraron Runtime.
    - Dynamic lookup.
    - Interpret parameters.
    - Startup time binding.
    - Name Servers
    - Plug-ins
    - Publish-suscribe.
    - Repositorios compartidos.
    - Polimorfismo.