Resumen RATS

Arquitectura de software (Universidad ORT Uruguay)

RAT #1

Capítulo 18 - Documenting Software Architectures

Hasta la mejor arquitectura es inútil si las personas que necesitan usarla no saben que es, no la comprenden lo necesario como para usarla, modificarla o no entenderla y aplicarla de manera incorrecta. Crear la arquitectura no es suficiente, tiene que ser comunicada de manera que los stakeholders la usen correctamente.

La documentación habla por la arquitectura

- Habla por la arquitectura hoy: para que el arquitecto no esté contestando cientos de preguntas
- Habla por la arquitectura mañana: cuando alguien nuevo está encargado del mantenimiento y evolución de la misma.

Hoy en día la documentación esta vista como un requerimiento y suele realizarse al final únicamente para cumplir esta obligación, pero una buena documentación

- es esencial para producir un producto de alta calidad
- puede hacer el proceso de diseño mucho más sistemático y smoothly (limpio, suave).

18.1 - Uses and Audiences for Architecture Documentation

La documentación debe

- Ser transparente y accesible para que sea rápidamente entendida por nuevos empleados
- Ser concreta para servir como modelo para la construcción
- Tener suficiente información para servir como base para el análisis.

Además es, dependiendo de quien la esté utilizando

- prescriptiva, prescribe lo que debe ser cierto.
- descriptiva, describe lo que es cierto.

Fundamentalmente, la documentación de la arquitectura tiene 3 usos:

- Medio de educación: el uso educacional consiste en introducir personas (nuevos miembros del equipo, analistas externos o un nuevo arquitecto) al sistema.
- Vehículo principal para la comunicación entre los stakeholders.
- Guía para la implementación del sistema: La arquitectura le dice a los desarrolladores que hay que implementar. Cada módulo tiene interfaces que deben ser provistas y usa interfaces de otros módulos. No solo define las interfaces sino que también define que equipos se deben comunicar entre sí

Durante el desarrollo, una arquitectura puede ser muy compleja, con muchos problemas por resolver. La documentación puede servir como receptáculo para registrar y comunicar estos problemas.

La documentación debe contener la información necesaria para evaluar atributos tales como seguridad, performance, usabilidad, disponibilidad y modificabilidad.

18.2 - NotationsforArchitectureDocumentation

Hay 3 categorías de notaciones:

- Notaciones informales. Las vistas se representan usando herramientas de diagramado de propósito general. Se utiliza lenguaje natural y no pueden ser formalmente analizadas. Ejemplo Power Point.
- Notaciones semiformales. Las vistas están expresadas en una notación estandarizada que prescribe elementos gráficos y reglas de construcción pero no provee una completa semántica del significado de esos elementos. Se puede hacer una análisis rudimentario. Ejemplo UML.
- Notaciones formales. Las vistas se describen en una notación que tiene una semántica precisa (generalmente basado en matemática). Tanto el análisis sintáctico como semántico es posible. Ejemplo ArchitectureDescriptionLanguage, que puede ser automatizado para ayudar al análisis de la arquitectura y la generación de código.

Una notación formal requiere un mayor tiempo y esfuerzo para crear y entender pero reduce la ambigüedad y aumenta las oportunidades de análisis. Análogamente, una notación más informal llevan menos tiempo y son más fáciles de hacer pero no brindan oportunidades de análisis.

18.3 - Views

Arquitectura de software: es una entidad compleja que no puede ser descrita de una manera sencilla unidimensional.

Vista: es una representación de un conjunto de elementos del sistema y relaciones entre ellos (no todos los elementos del sistema sino los de un tipo particular). Nos permiten dividir la entidad multidimensional que es la arquitectura en un número manejable de representaciones del sistema. Diferentes vistas exponen diferentes atributos de calidad a diferentes niveles y resaltan diferentes elementos del sistema y sus relaciones.

- Por ejemplo :una vista de capas nos muestra la descomposición del sistema en capas y las relaciones entre ellas y no nos mostraría por ejemplo el modelo de datos, los servicios del sistema, etc. Permite razonar sobre la portabilidad del sistema mientras que una vista de deployment le permite razonar sobre la performance del sistema.

Principio fundamental de la documentación de la arquitectura de software:
Documentar una arquitectura es una cuestión de documentar las vistas relevantes y luego añadir documentación que se aplica a más de una vista.

Module Views

Table 18.1. Summary of the Module Views

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

Módulo: es una unidad de implementación que proporciona un conjunto coherente de responsabilidades.

- Tiene una colección de propiedades asignadas a él que expresan información importante asociada al módulo, así como restricciones en el módulo.
    - Nombre: es el principal medio para referirse al módulo. Muchas veces sugiere algo sobre el rol del módulo en el sistema. A su vez puede reflejar su posición en una

descomposición jerárquica (por ejemplo A.B.C donde C es submódulo de B y B es submódulo de A).
    - Responsabilidades: es una forma de identificar el rol de un módulo en el sistema. Debe especificar lo que un módulo hace.
    - Visibilidad de interfaces: cuando un módulo tiene submódulos, algunas interfaces del submódulo son públicas y otras son privadas, lo que significa que algunas se usan solo dentro del submódulo.
    - Información de implementación: se debe incluir el mapeo a las unidades de código fuente, información de test e implementación de restricciones.
- Existen diferentes relaciones entre módulos: “es una parte de”, “depende de” y “es un”.

Usos: como plan para la construcción del código, para análisis del impacto de cambios, para planear desarrollo incremental, para comunicar la funcionalidad del sistema y la estructura de su código base.
Estas vistas son particiones estáticas de las funciones del software, por lo tanto no se usan para análisis de performance u otro tipo de atributos de runtime.
Tipos

- Descomposición: muestra que un módulo está compuesto por otros, se utiliza para estructurar el código por responsabilidades o por features según corresponda
- Usos: muestra las dependencias entre los módulos, de forma de poder razonar el impacto de los cambios y tomar decisiones de mantenibilidad.
    - Layered: además de mostrar las dependencias marca las capas del sistema de forma de saber de qué es responsable cada módulo
- Clases: muestra detalles de la implementacion

Las modules views generalmente se mapean a las vistas de componentes y conectores. Las implementationunits mostradas en los módulos tienen un mapeo a componentes que ejecutan en tiempo de ejecución.

Component and ConnectorViews

Table 18.2. Summary of Component-and-Connector Views

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.

Las vistas de componentes y conectores muestran

- Componentes: elementos que tienen alguna presencia en tiempo de ejecución, tales como procesos, objetos, clientes, servidores y bases de datos.
    - Los componentes tienen interfaces llamadas puertos, que definen un potencial punto de interacción de un componente con su entorno. Un puerto tiene un tipo, que define la clase de comportamiento que puede adoptar en ese punto de interacción. Un componente puede tener muchos puertos del mismo tipo.
    - Un componente en las vistas C&C puede representar un subsistema complejo, que a su vez puede ser descrito como una subarquitectura C&C.
- Conectores: las vías de comunicación, como por ejemplo links y protocolos de comunicación, flujo de información y acceso a almacenamiento compartido.
    - Los conectores muchas veces representan una forma compleja de interacción tal como una transacción entre una base de datos en un servidor y un cliente.
    - Los conectores tienen roles, que son sus interfaces, que definen las maneras en los cuales el conector debe ser usado por los componentes para llevar a cabo la interacción. Al igual que los puertos, los roles pueden ser replicados indicando cuantos componentes pueden involucrarse en la interacción. Diferencia entre las interfaces de los módulos y los puertos/roles: las interfaces de los módulos no pueden ser replicadas.
    - Los conectores encarnan protocolos de interacción. Cuando dos o más componentes interactúan, deben obedecer órdenes de interacción. El protocolo debe ser documentado.

Un elemento tiene asociadas varias propiedades. Todos deben tener un nombre y un tipo(componente o conector). Otras propiedades dependen del tipo. Algunas propiedades: confiabilidad, performance, funcionalidad, seguridad, concurrencia, modificabilidad, etc.

Usos: para mostrar a los desarrolladores y stakeholders cómo funciona el sistema, para razonar acerca de los atributos de calidad de tiempo de ejecución (runtimesystemqualityattributes) y guiar el desarrollo especificando la estructura y comportamiento de elementos de tiempo de ejecución (runtimeelements).

Propiedades

- Reliability: Cual es la probabilidad de fallo de un componente o un conector_
- Performance: Que tiempos de respuesta va a tener el componentes y bajo qué cargas?
- Requerimiento de recursos: Cuales son las capacidades de procesamiento y almacenamiento que necesita un componente o un conector?
- Funcionalidad: Que funciones cumple un elemento?
- Concurrencia: Este componente ejecuta como un proceso un hilo separado?

Relación: “attachment”/adjuntamiento indica qué conectores están adjuntos a qué componentes, definiendo un grafo de componentes y conectores. Se especifica asociando un puerto de un componente a un rol del conector. El puerto y el rol deben ser compatibles (generalmente la compatibilidad se define en términos de tipo y protocolo).

Notationsfor C&C Views

- Los componentes de UML se corresponden muy bien semánticamente con los componentes de C&C porque permiten una documentación intuitiva de información importante como interfaces o propiedades. Además UML distingue entre tipos de componentes e instancias de componentes.
- Los puertos de UML también se corresponden muy bien dado que en UML pueden ser decorados con multiplicidad.
- Con los conectores no sucede lo mismo. Mientras que en C&C los conectores son semánticamente ricos, en UML los conectores no pueden tener subestructuras o atributos. Para representar un simple conector, UML es apropiado (se puede usar un estereotipo para determinar el tipo del conector). Los roles no pueden ser representados en UML dado que los conectores UML no permiten la inclusión de interfaces, lo que se puede hacer es usar labels.
  Para representar un conector “rico” hay dos opciones:

- usar un componente UML
- usar un conector UML y usar una documentación auxiliar que explique el concepto de conector complejo.

AllocationViews

Table 18.3. Summary of the Characteristics of Allocation Views

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.

Las vistas de asignación describen el mapeo de las unidades de software con los elementos de un entorno en el cual el software es desarrollado o en el cual sea ejecutado. El entorno puede ser el hardware, los file system de apoyo al desarrollo o el desarrollo de la organización.
Los elementos de las allocationviews son

- Elementos de software: provienen de la vista de módulos o la vista de C&C. Tienen propiedades que se requieren del entorno.
- Elementos del entorno: tienen propiedades que se proveen al software. Por ejemplo procesador, disco, etc.

Usos: para razonar acerca de la performance, disponibilidad y seguridad, para razonar acerca del acceso concurrente a las versiones de software, para razonar acerca de la forma y mecanismos de la instalación del sistema.
El objetivo más común es para comparar las propiedades requeridas por los elementos de software con las propiedades ofrecidas por los elementos de entorno para determinar si la asignaciones va a ser satisfactoria o no. Por ejemplo, para asegurar el tiempo de respuesta requerido por un componente, debe ejecutar en (debe ser asignado a) un procesador que provea suficiente poder de procesamiento.
Relación: "asignado a". Un único elemento de software puede ser asignado múltiples elementos del entorno y múltiples elementos de software pueden ser asignados a un único elemento del entorno. Si estas asignaciones cambian se dice que la arquitectura es dinámica respecto a las asignaciones.

QualityViews

Las vistas de módulos, C&C y asignaciones son todas vistas estructurales. Principalmente muestran las estructuras que el arquitecto ha diseñado para satisfacer requerimientos funcionales y atributos de calidad. Estas vistas son excelentes para guiar y limitar a los desarrolladores cuyo trabajo es implementar las estructuras diseñadas. Sin embargo, en sistemas en los cuales ciertos atributos de calidad (o diferentes preocupaciones de los stakeholders) son particularmente importantes, las vistas estructurales no son la mejor manera para presentar la solución arquitectónica. La razón es que la solución se puede propagar a través de múltiples estructuras que son inconvenientes para combinar (por ejemplo porque los tipos de los elementos que muestran son diferentes).

Usos: pueden ser adaptadas para los stakeholders específicos o para abordar preocupaciones específicas. Estas vistas se forman extrayendo las piezas relevantes de las vistas estructurales y poniéndolas juntas.
Ejemplos:

- Security view: puede mostrar todas las medidas arquitectónicas que se tomaron para proveer seguridad. Muestra los componentes que tienen un rol en la seguridad y como esos componentes se comunican. La información del contexto puede mostrar medidas de seguridad en el entorno del sistema. Muestra también como el sistema respondería ante amenazas.
- Communicationview: es de ayuda para sistemas heterogéneos y globalmente dispersos. Se mostrarían todos los canales componente - a - componente, la red de canales, áreas de concurrencia. La parte del comportamiento puede mostrar como el ancho de banda de la red es dinámicamente asignado.
- Exceptionor error handlingview: ayuda a iluminar y diseñar un reporte de errores y mecanismos de resolución de errores. Una vista muestra como un componente detecta, reporta y resuelve fallas.

18.4 - ChoosingtheViews

Para el momento en el cual se debe liberar la documentación, es muy probable que se cuente con una buena colección de vistas bien trabajadas. Se deberá decidir cuáles tomar, con cuantos detalles y cuales incluir en el documento. También se deberá decidir cuales pueden combinarse útilmente para reducir el número total de vistas y resaltar las relaciones importantes entre ellas.
Como mínimo se espera contar con por lo menos una vista de módulos y una de C&C. Para sistemas más grandes por lo menos una allocationview.
Existe un método de 3 pasos para elegir la vistas que se incluirán en la documentación:

1. Construir una tabla de stakeholders/vistas: enumerar los stakeholders para la documentación de la arquitectura del software del proyecto en las filas. Para las columnas, enumerar las vistas que aplican al sistema. Una vez armada la tabla, rellenar en la celdas cuánta información requiere el stakeholder para la vista. La lista de vistas candidatas consiste de aquellas vistas que tienen alguna importancia para algún stakeholder.
2. Combinar vistas: es probable que la lista de vistas candidatas cuente con una gran cantidad de vistas. Este paso dejará la lista con un largo manejable. Se debe combinar cada vista marginal (aquella que sirve para unos pocos stakeholder) con alguna vista significativa.
3. Priorizar: después del paso 2 se debe tener un conjunto mínimo de vistas necesarias para todos los stakeholders. En este punto se debe decidir cuál hacer primero.

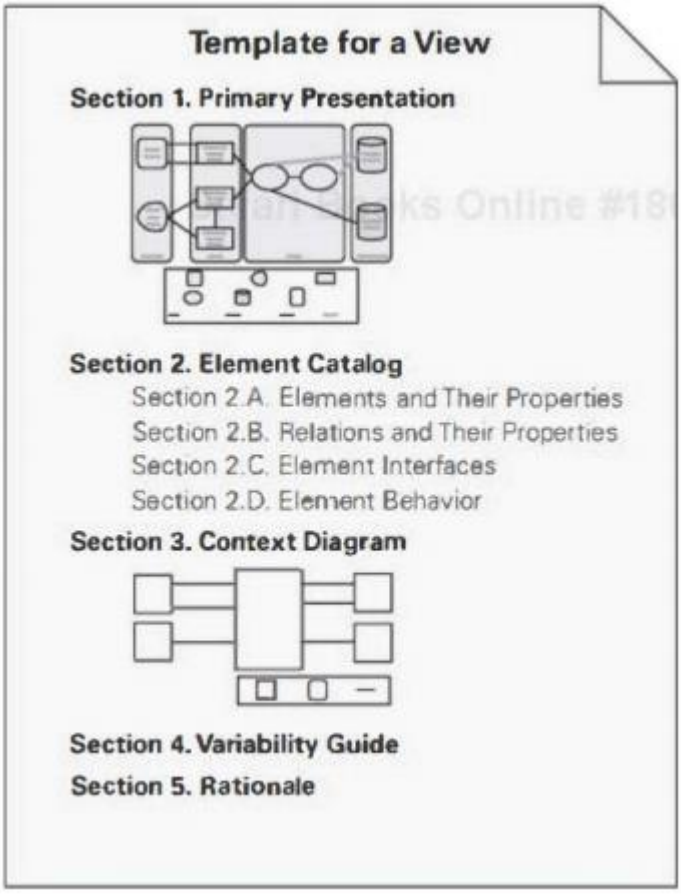
18.5 - CombiningViews

Vista combinada: es una vista que contiene elementos y relaciones que provienen de dos o más vistas diferentes.
Dado que todas las vistas en un arquitectura son parte de la misma arquitectura y existen para lograr un objetivo común, muchas de ellas están fuertemente asociadas. Muchas veces, la forma más conveniente de mostrar una asociación entre dos vistas es combinándolas en una única vista creando una superposición que combine información que de otra manera estaría en dos vistas separadas.
Vistas que se combinan naturalmente:

- Varias vistas C&C: dado que todas las vistas C&C muestran runtimerelations entre componentes y conectores de varios tipos, tienden a combinarse bien. Diferentes vistas C&C tienden a mostrar diferentes partes del sistema.
- Vistas de deployment (allocation) con vistas SOA o communicating-process: una vista SOA muestra servicios y una de comunicación de procesos muestra procesos, en ambos casos son componentes que se despliegan en procesadores.
- Vistas de descomposición con vistas de asignación de trabajo, implementación, uso o capas

18.6 - BuildingtheDocumentationPackage

Documentar una vista



- Sección 1: Theprimarypresentation
  La presentación primaria muestra los elementos y las relaciones de la vista. Generalmente es una muestra gráfica (diagrama). Debe contener una key que explique la notación.
- Sección 2: Theelementcatalog
  El catálogo de elementos detalla los elementos mostrados en la sección 1. Por ejemplo si el diagrama muestra un elemento A, el catálogo debe explicar qué es A.
  En el catálogo se incluye:
  - Elementos y sus propiedades
  - Relaciones y sus propiedades
  - Interfaces de elementos
  - Comportamiento de elementos
- Sección 3: Contextdiagram
  El diagrama de contexto muestra como el sistema o parte del sistema representado en la vista se relaciona con el entorno. El objetivo del diagrama de contexto es describir el alcance de la vista.
- Sección 4: Variablityguide
- Sección 5: Rationale
  La razón fundamental explica por qué el diseño es como es.

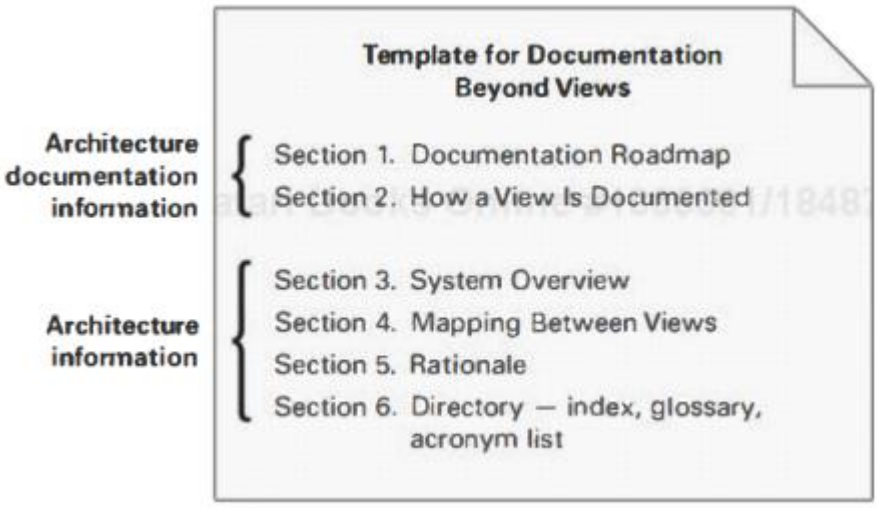
Documentar información más allá de las vistas

La documentación más allá de las vistas puede ser dividida en dos partes:

- **Resumen de la documentación de arquitectura:** explica como la documentación está organizada para que un stakeholder pueda encontrar la información fácilmente

- **Información acerca de la arquitectura:** la información que queda por explicar más allá de las vistas, es un resumen corto del sistema para informar a cualquier lector el propósito del sistema y cómo las vistas se relacionan entre ellas.

Template para la documentación más allá de las vistas:



- Document control information: Se muestra el número de versión, fecha y estado de los issues, etc.
- Sección 1: DocumentationroadmapLe muestra al lector qué información hay en el documento y dónde encontrarla. Consiste de 4 secciones:
  - Objetivo y resumen: objetivo del documento, que está cubierto y que no.
  - Cómo está organizado el documento: para cada sección hay una breve información sobre lo que se puede encontrar ahí.
  - View overview: describe las vistas que se incluyeron.
  - Cómo los stakeholders pueden usar la documentación
- Sección 2: How a viewisdocumented: Referencia al template para documentar las vistas.
- Sección 3: Systemoverview: Breve descripción de las funciones del sistema, el objetivo del sistema, los usuarios y las restricciones.
- Sección 4: Mappingbetweenviews: Para entender las asociaciones entre las vistas.
- Sección 5: Rationale: En esta sección se documentan las decisiones arquitectónicas que se aplican a más de una vista, decisiones sobre el uso de patrones.
- Sección 6: Directory: Conjunto de referencia a materiales, glosario.

Documentación online, Hipertexto y Wikis

Un documento puede ser estructurado en forma de páginas web enlazadas. Documentos orientados a la web generalmente consisten de páginas cortas con una estructura más profunda. Una página contiene un resumen de la información y uno o más links a información más detallada.
Es posible crear un documento compartido para que los stakeholders puedan contribuir. Se le puede dar permiso a los stakeholders de agregar información para aclarar conceptos pero no se le da el permiso de cambiar la arquitectura. Un documento ideal para esto es el wiki.

Documentar patrones

Los arquitectos usan patrones como punto de partida para el diseño. Estos patrones pueden estar publicados en catálogos existentes o un repositorio de la organización. En cualquier de los casos, proveen una solución genérica que el arquitecto debe refinar e instanciar. Usar un patrón significa tomar decisiones de diseño sucesivas que resultan en una arquitectura. El arquitecto puede documentar cada paso para resolver más fácilmente el problema en un futuro.

18.7 - DocumentingBehavior

Documentar una arquitectura requiere documentar comportamiento que complemente las vistas estructurales, describiendo como los elementos interactúan entre ellos.

Hay dos tipos de notaciones para documentar comportamiento

- **Trace-orientedlanguage**: Traces son secuencias de actividades o interacciones que describen cómo el sistema responde a estímulos específicos cuando el sistema está en un estado específico. No tienen como intención mostrar todo el comportamiento de los elementos.
  4 notaciones para trace-orientedlanguage:
  - Casos de uso: describen como un actor puede usar el sistema para lograr sus objetivos. Se usan para capturar los requerimientos funcionales.
  - Diagramas de secuencia: muestran una secuencia de interacciones entre instancias de elementos. Las instancias interactúan enviando mensajes. Un mensaje puede ser un método, la llamada a una función, etc.
  - Diagramas de comunicación: muestran un grafo de elementos que interactúan y se marca cada interacción con un número que denota un orden.
  - Diagramas de actividad: son similares a los diagramas de flujo. Muestran un proceso de negocio como una secuencia de pasos e incluyen notación para expresar concurrencia, condicionales, mensajes enviados y mensajes recibidos.
- **Comprehensivelanguage:** Los comprehensivemodels muestran el completo comportamiento de los elementos. Es posible inferir todos los caminos posibles desde un estado inicial a un estado final, por ejemplo las máquinas de estado.

RAT #2

4. Understanding Quality Attributes

Many factors determine the qualities that must be provided for in system's architecture. These qualities go beyond functionality, which is the basic statement of the system's capabilities, services, and behavior. Although functionalityand other qualities are closely related, as you will see, functionality often takes the front seat in the development scheme. Systems are frequently redesigned not because they are functionally different but because they are difficult tomaintain.

Quality Attribute: A quality attribute is a measurable or testable property of a system that is used to indicate how wellthe system satisfies the needs of its stakeholders. The "goodness" of a product along some dimension of interest to a stakeholder.

4.1. Architecture and Requirements

Tipos de requerimientos

- **Functional requirements:** These requirements state what the system must do, and how it must behave or react to runtime stimuli. These are satisfied by assigning an appropriate sequence of responsibilities through the design.
- **QA requirements**: These are qualifications of the functional requirements or of the overall product. A qualification of a functional requirement is an item such as how fast the function must be performed. These are satisfied by the various structures designed into the architecture and the behaviour and interactions of the elements that populate those structures.
- **Constraints (Restricciones):** A constraint is a design decision with zero degrees of freedom. These are satisfied by accepting the design decision and its consequences.

4.2. Functionality

Functionality is the ability of a system to do the work for which it was intended. Of all of the requirements, functionality has the strangest relationship to architecture. It does not determines architecture, but, functionality is achieved by assigning the responsibilities to architectural elements, resulting in one of the most basic of architectural structures.

4.3. Quality Attribute Considerations

Just as a system's functions do not stand on their own without due consideration of other QA, neither do QA stand on their own, they pertain to the functions of the system.

From an architect's perspective, there are three problems with previous discussions of system quality attributes:

1. The definitions provided for an attribute are not testable. It is meaningless to say that a system will be modifiable.
2. Discussion often focuses on which QA a particular concern belongs to but this is very hard to identify this correctly, a concern probably belongs to more than one QA.
3. Each attribute community has developed its own vocabulary.

A solution to the first two is to use QA scenarios as means of characterizing QA's and a solution to the third problem is to provide a discussion to each attribute.

There are two categories of QA to focus on:

- Those that describe some property of the system at runtime
- Those that describe some property of the development of the system.

Within complex systems, QA can never be achieved in isolation. The achievement on any will have an effect on the rest

4.4. Specifying Quality Attribute Requirements

A quality attribute requirement should be unambiguous and testable. We'll use a common form to specify all QA's which is as six-part scenarios:

- **Stimulus source**: The entity that generated the stimulus

- **Stimulus:** a condition that requires a response when it arrives at a system.
- **Environment:** The set of circumstances in which the scenario takes place.
- **Artefact:** The artefact is the portion of the system to which the requirement applies.
- **Response**: the activity undertaken as the result of the arrival of the stimulus.
- **Response measure**: When the response occurs, it should be measurable in some fashion so that the requirement can be tested.

Two types of QA scenarios

- General: are system independent and can, potentially, pertain to any system
- Concrete: are specific to the particular system under consideration.

Example of a general scenario for availability

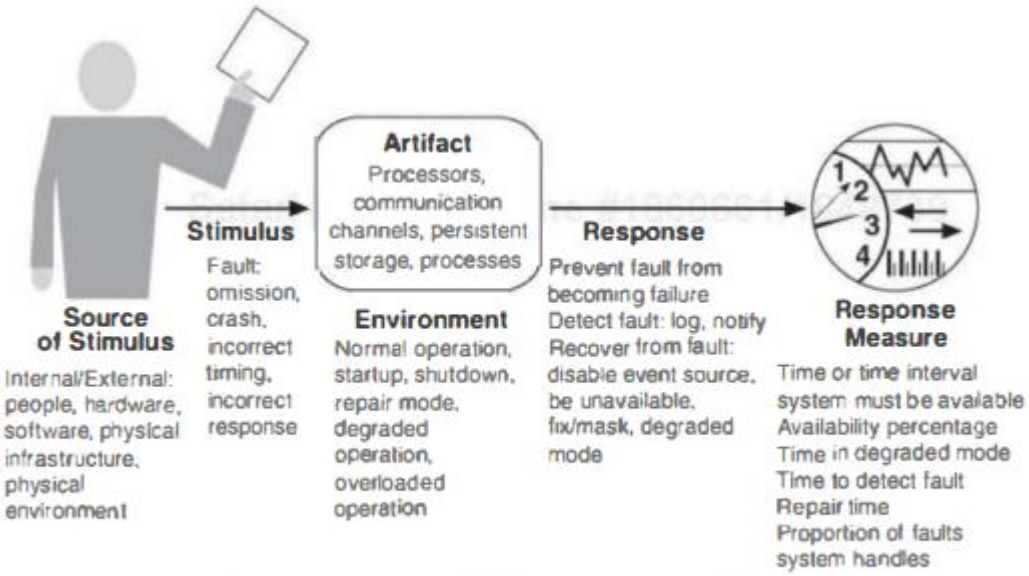


Figure 4.2. A general scenario for availability

4.5. Achieving Quality Attributes through Tactics

Architectural tactics: the techniques an architect can use to achieve the required quality attributes, is a design decision that influences the achievement of a quality attribute response.

The focus of a tactic is on a single QA response. Within a tactic, there is no consideration of trade-offs. Trade-offs must be explicitly considered and controlled by the designer. In this respect, tactics differ from architectural patterns where trade-offs are built into the patter.

The tactics can and should be refined and the application of a tactic depends on the context.

A system design consists of a collection of decisions. Some of these decisions help control the quality attribute responses; others ensure achievement of system functionality.

The tactics, like design patterns, are design techniques that architects have been using for years. So, why do we capture them?

- By understanding the role of tactics, an architect can more easily assess the options for augmenting an existing pattern to achieve a quality attribute goal.
- Tactics allow the architect to construct a design fragment from "first principles." Tactics give the architect insight into the properties of the resulting design fragment.
- By cataloguing tactics, we provide a way of making design more systematic within some limitations.

5. Availability

Availability: refers to a property of software that it is there and ready to carry out its task when you need it to be. It builds upon the concept of reliability by adding the notion of recovery, that is, when the system breaks. Fundamentally, availability is about minimizing service outage time by mitigating faults.

Dependability: the ability to avoid failures that are more frequent and more severe than is acceptable.

Failure: is the deviation of the system from its specification, where the deviation is externally visible.

Fault: what causes a failure, can be either internal or external. It can be prevented, tolerated, removed, or forecast making the system "resilient" to faults.

Error: intermediate states between the occurrence of a fault and the occurrence of a failure.

- Fault != Failure: if code containing a fault is executed but the system is able to recover from the fault without any deviation from specified behaviour being observable, there is no failure.

Because a system failure is observable by users, the time to repair is the time until the failure is no longer observable (the notion of "observability" can be a tricky one) and we are often concerned with the level of capability that remains when a failure has occurred a degraded operating mode.

When thinking about availability, you should think about:

- What will make your system fail
- How likely that is to occur
- Time required to repair it

Service-level agreement (SLA) specifies the availability level that is guaranteed and, usually, the penalties that the computer system or hosting service will suffer if the SLA is violated.

5.2. Tactics for Availability

Goal: Availability tactics are designed to enable a system to endure system faults so that a service being delivered by the system remains compliant with its specification.

- Keep faults from becoming failure
- Bound the effects of the fault and make repair possible

Tactics

Detect faults

- Ping/echo: using an asynchronous request/response message pair exchanged between nodes to check if the pinged component is alive and responding correctly. It requires to set a time threshold = how long you wait until considering it a time out.
- Monitor: component that is used to monitor the state of health of various other parts of the system, it can detect failure or congestion in the network or other shared resources.
- Heartbeat: is a fault detection mechanism that employs a periodic message exchange between a system monitor and a process being monitored.**The big difference between heartbeat and ping/echo is who holds the responsibility for initiating the health check.**
- Time stamp: used to detect incorrect sequences of events
- Sanity checking: checks the validity or reasonableness of specific operations or outputs of a component.
- Conditions monitoring: involves checking conditions in a process or device, or validating assumptions made during the design. Prevents a system from producing faulty behaviour.
- Voting: it depends critically on the voting logic (ex: majority rule)
    - Replication: having multiple copies of identical components, it can protect from random failures but not against design or implementation errors.
    - Functional redundancy: components must always give the same output given the same input, but they are diversely designed and diversely implemented.

    - Analytic redundancy: different components and different inputs/outputs.
- Exception detection: detection of a system condition that alters the normal flow of execution
    - System exceptions
    - The parameter fence tactic incorporates an a priori data pattern after any variable-length parameters of an object to detect overwriting
    - Parameter typing employs a base class that defines functions that add, find, and iterate over type-length-value (TLV) formatted message parameters.
    - Timeout
- Self-test: run procedures to test themselves, initiated by itself or by a system monitor.

Recover from faults

- Preparation-and-repair
    - Active redundancy (hot spare): all of the nodes receive and process identical inputs in parallel, allowing the redundant spare(s) to maintain synchronous state with the active node(s). In case of failure the redundant can take over instantly.
    - Passive redundancy (warm spare): provide the redundant spare(s) with periodic state updates
    - Spare (cold spare): the redundant spares of a protection group remain out of service until a fail-over occurs
    - Exception handling: the mechanism employed for exception handling depends largely on the programming environment employed
    - Rollback: revert to a previous known good state, "rollback line", upon the detection of a failure
    - Software upgrade: achieve in-service upgrades to executable code images in a non-service-affecting manner. As a patch, class-patch (fix bugs) or hitless in-service software upgrade (new features).
    - Retry: assumes that the fault that caused a failure is transient and retrying the operation may lead to success
    - Ignore faulty behaviour
    - Degradation: maintains the most critical system functions in the presence of component failures
    - Reconfiguration: reassigning responsibilities while maintaining as much functionality as possible
- Reintroduction: where a failed component is reintroduced after it has been corrected
    - Shadow: operating in "shadow mode", in order to monitor it, prior to reverting it to an active role
    - State resynchronization: used when working with the active redundancy and passive redundancy preparation-and-repair tactics
    - Escalating restart: recover from faults by varying the granularity of the component(s) restarted and minimizing the level of service affected
    - Non-stop forwarding: based on routers, functionality is split into two parts: supervisory and data plane. If a router experiences the failure of an active supervisor, it can continue forwarding packets along known routes.

Prevent faults

- Removal from service/software rejuvenation: temporarily placing a system component in an out-of-service state for the purpose of mitigating potential system failures.
- Transactions: prevents race conditions caused by two processes attempting to update the same data item.
- Predictive model: is employed to monitor the state of health of a system process and to take corrective action when conditions are detected that are predictive of likely future faults.
- Exception prevention: techniques employed for the purpose of preventing system exceptions from occurring
- Increase competence set: designing it to handle more cases faults as part of its normal operation.

7. Modifiability

Modifiability is about change, and our interest in it centres on the cost and risk of making changes.
To plan for modifiability, an architect has to consider four questions:

- What can change?
- What is the likelihood of the change?
- When is the change made and who makes it?
- What is the cost of the change?
    - The cost of introducing the mechanism(s) to make the system more modifiable
    - The cost of making the modification using the mechanism(s)

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

7.2. Tactics for Modifiability

Goal: controlling the complexity of making changes, as well as the time and cost to make changes

Cohesion: measures how strongly the responsibilities of a module are related. The higher the cohesion, the lower the probability that a given change will affect multiple responsibilities. High cohesion is good.

Tactics

Reduce the Size of a Module

- Split module: If the module being modified includes a great deal of capability, the modification costs will likely be high. Refining the module into several smaller modules should reduce the average cost of future changes.

Increase Cohesion

- Increase semantic coherence: If the responsibilities A and B in a module do not serve the same purpose, they should be placed in different modules. This will reduce the likelihood of side effects affecting other responsibilities.

Reduce Coupling (acoplamiento)

- Encapsulate: introduce an explicit interface, abstract with respect to the details of the module, to a module. This reduces the probability that a change to one module propagates to other modules.
- Use an intermediary: this will break the dependency, the type of intermediary depends on the type of dependency.
- Restrict dependencies: restrict the modules that a given module interacts with or depends on by restricting a module's visibility (when developers cannot see an interface, they cannot employ it) and by authorization (restricting access to only authorized modules).
- Refactor: common responsibilities are "factored out" of the modules where they exist and co-located.
- Abstract common services: implement the services just once in a more general form, that way any modification will need to be done to only one module.

Defer Binding

An architecture that is suitably equipped to accommodate modifications late in the life cycle will, on average, cost less than an architecture that forces the same modification to be made earlier.
If we design artefacts with built-in flexibility, then exercising that flexibility is usually cheaper than hand-coding a specific change.
The later in the life cycle we can bind values, the better. However, putting the mechanisms in place to facilitate that late binding tends to be more expensive

Bind values at compile or build time:

- Component replacement (for example, in a build script or make file)
- Compile-time parameterization
- Aspects

Bind values at deployment time:

- Configuration-time binding

Bind values at startup or initialization time:

- Resource files

Bind values at runtime:

• Runtime registration • Dynamic lookup (e. g., for services) • Interpret parameters • Startup time binding • Name servers	• Plug-ins • Publish-subscribe • Shared repositories • Polymorphism

RAT #3

8. Performance

Performance is about time and the software system's ability to meet timing requirements. When events occur, the system, or some element of the system, must respond to them in time.

Performance has been the driving factor in system architecture. As such, it has frequently compromised the achievement of all other qualities. All systems have performance requirements even if they are not expressed. Performance continues to be a fundamentally important quality attribute for all software.

8.2. Tactics for Performance

Goal: generate a response to an event arriving at the system within some time-based constraint.

Contributors to the response time

- Processing time: Processing consumes resources, which takes time, events are handled by the execution of one or more components, whose time expended is a resource.
- Blocked time: A computation can be blocked because of contention for some needed resource
  - Contention for resources: many resources can only be used by a single client at the time. The more contention for a resource, the more likelihood of latency being introduced.
  - Availability of resources: Computation cannot proceed if a resource is unavailable. You must identify places where resource unavailability might cause a significant contribution to overall latency.
  - Dependency on other computation: If a component calls another component and must wait for that component to respond, the time can be significant if the called component is at the other end of a network.

Tactics

Control resource demand

Carefully manage the demand for resources.

- Manage sampling rate: by reducing the sampling frequency at which a stream of environmental data is captured, the demand can be reduced. Typically this implies some loss of fidelity.
- Limit event response: choose to process events only up to a set maximum rate. If it is unacceptable to lose any events, then you must ensure you can queue the rest.
- Prioritize events: Impose a priority scheme that ranks events according to how important it is to service them.
- Reduce overhead: The use of intermediaries increases the resources consumed in processing an event stream, and so, removing them improves latency. A strategy for reducing computational overhead is to co-locate resources. Also, performing periodic cleanup of resources that have become inefficient may enhance performance.
- Bound execution times: Place a limit on how much execution time is used to respond to an event. The cost is usually less accurate computation.
- Increase resource efficiency: Improving the algorithms used in critical areas will decrease latency

Manage resources

- Increase resources: Cost is usually a consideration in the choice of resources. Increasing resources is definitely a tactic to reduce latency and in many cases, the cheapest way to get improvement
- Introduce concurrency: Concurrency can be introduces by processing different streams of events on different threads or by creating additional threads to process different sets of activities.

- Maintain multiple copies of computations: Reduce the contention that would occur if all computations took place on a single server.
- Maintain multiple copies of data: Caching is a tactic that involves keeping copies of data on storage with different access speeds.
- Bound queue sizes: Maximum number of queued arrivals and consequently the resources used to process the arrivals.
- Schedule resources: Whenever there is contention for a resource, the resource must be scheduled.

VA??????????????????????

Scheduling Policies

A scheduling policy conceptually has two parts: a priority assignment and dispatching. Competing criteria for scheduling include optimal resource usage, request importance, minimizing the number of resources used, minimizing latency, etc.
Common scheduling policies are:

- First-in/First-out (FIFO)
- Fixed priority Scheduling
    - Semantic importance: Each stream is assigned a priority statically according to some domain characteristic of the task that generates it.
    - Deadline monotonic: a static priority assignment that assigns higher priority to streams with shorter deadlines.
    - Rate monotonic: assignment for periodic streams that assigns higher priority to streams with shorter periods.
- Dynamic priority scheduling
    - Round-Robin
    - Earliest deadline first.
    - Least slack first: Slack time = execution time remaining - deadline.

??????????????????????????????

8.4 Summary

Performance is about the management of system resources in the face of particular types of demand to achieve acceptable timing behaviour.

- It can be measured in terms of throughput and latency for both interactive and embedded real-time systems
    - Throughput is usually more important in interactive systems
    - Latency is more important in embedded systems.
- It can be improved by:
    - Reducing demand, this will have the side effect of reducing fidelity or refusing to service some requests.
    - Managing resources more appropriately through scheduling, replication, or just increasing the resources available.

9. Security

Security is a measure of the system's ability to protect data and information from unauthorized access while still providing access to people and systems that are authorized.

Characteristics (CIA+3)

- Confidentiality: is the property that data or services are protected from unauthorized access.
- Integrity: Property that data or services are not subject to unauthorized manipulation
- Availability: is the property that the system will be available for legitimate use.
- Authentication: verifies the identities of the parties to a transaction and checks if they are truly who they claim to be.
- Nonrepudiation: Guarantees that the sender of a message cannot later deny having sent the message and the same for the recipient.
- Authorization: grants a user the privileges to perform a task.

9.2. Tactics for Security

Detect attacks

- Detect intrusion: is the comparison of network traffic or services request patterns within a system to a set of signatures or known patterns of malicious malware behaviour stored in a database.
- Detect service denial: the comparison of the pattern or signature of network traffic coming into a system to historic profiles of known denial of service attacks.
- Verify message integrity: employs techniques such as checksums or hash values to verify the integrity of messages, resource files, etc.
- Detect message delay: intended to detect potential man in the middle attacks where a malicious party is intercepting messages.

Resist attacks

- Identify actors: is really about identifying the source of any external input to the system.
- Authenticate actors: Authentication means ensuring that an actor is actually who or what it purports to be.
- Authorize actors: Ensuring that authenticated an actor has the rights to access and modify either data or services.
- Limit access: The limit exposure tactic minimizes the attack surface of a system. This tactic focuses on reducing the probability of and minimizing the effects of damage caused by a hostile action.
- Encrypt data.
- Separate entities: Separating different entities within the system can be done through physical separation on different servers that are attached to different networks.
- Change default settings.

React to attacks

- Revoke access: The access can be severely limited to sensitive resources even for normal users if the admin thinks an attack is in place.
- Lock computer: Repeated failed login attempts may indicate a potential attack.
- Inform actors: Ongoing attacks may require actions by operators. Such subjects must be notified

Recover from attack

- Restore: using the tactics for recovering assessed on Availability
  - Preparation-and-repair: redundancy, software upgrade, etc.
  - Reintroduction
- Audit: keep a record of user and system actions and their effects.

9.4. Summary

Attacks against a system can be characterized as attacks against the confidentiality, integrity or availability of a system or its data.

The emphasis of distinguishing various classes of actors in the characterization leads to many of the tactics used to achieve security. Identifying, authenticating and authorizing actors are tactics intended to determine which users or systems are entitled to what kind of access to a system.

An assumption is made that no security tactic is fool proof and that systems will be compromised. Hence, tactics exist to detect an attack, limit the spread of any attack and to rack and recover from an attack.

Recovering from an attack involves returning the system to a consistent state prior to any attack.

10. Testability

Software testability refers to the ease with which software can be made to demonstrate its faults through testing. Testability refers to the probability, assuming that software has at least one fault, that it will fail on its next execution.
For a system to be properly testable, it must be possible to control each component's inputs and then observe their outputs.
Testing is carried out by various developers, users or QA personnel. Portions of the system or the entire system may be tested.
Testing of code is a special case of validation, which is making sure that an engineered artefact meets the needs of its stakeholders or is suitable for use.

10.2. Tactics for Testability

Goal: allow for easier testing when an increment of software development is completed

Tactics

Control and Observe System State

Control and observation are so central to testability that some authors even define testability in those terms, it is a way of adding capability or abstraction to the software that otherwise would not be there.

- Specialized interfaces: having specialized testing interfaces allows you to control or capture variable values for a component either through a test harness or through normal execution. Specialized testing interfaces and methods must be clearly identified or kept separate from the access methods and interfaces for required functionality.
- Record / playback: Recording the state whet it crosses an interface allows that state to be used to play the system back and re-create the fault.
- Localized state storage: To start a system it is most convenient if that state is stored in a single place.
- Abstract data sources: easily controlling its input data makes it easier to test and substitute the data more easily.
- Sandbox: refers to isolating an instance of the system for the real world to enable experimentation that is unconstrained.
- Executable assertions: This are used to check that data values satisfy specified constraints.

Limit complexity

- Limit structural complexity: Avoid or resolving cyclic dependencies between components, isolating and encapsulating dependencies and reducing dependencies in general.
- Limit no determinism: The counterpart to limiting structural complexity is limiting behavioural complexity. Nondeterministic systems are header to test than deterministic systems.

10.4. Summary

Ensuring that a system is easily testable has payoffs both in terms of the cost of testing and the reliability of the system.
Common ways of testing

- Test harnesses: software systems that encapsulates test resources so that it is easy to reapply test across iterations and it is easy to apply test infrastructure to no increments of the systems.
- Creating test cases prior to the development: this way developers know which test their components must pass.

Testability tactics

- Controlling and observing: by, for instance, providing the ability to do fault injection, to record system state at key portions, to isolate the system from its environment, and to abstract various resources.
- Keeping the system simple: complex systems are difficult to test because of the large state space in which their computation take place, and because of the larger number of interconnections among the elements of the system.

RAT #4

13. Architectural Tactics and Patterns

Architectural pattern

- is a package of design decisions that is found repeatedly in practice (therefore, it isn't invented but discovered)
- has known properties that permit reuse
- describes a class of architectures

Patterns package tactics

Tactics use just a single structure or computational mechanism, and they are meant to address a single architectural force.

Most patterns are constructed from several different tactics.

13.1. Architectural Patterns

An architectural pattern establishes a relationship between:

- A context: a recurring, common situation in the world that gives rise to a problem.
- A problem: the problem, appropriately generalized, that arises in the given context. The pattern description outlines the problem and its variants, describes any complementary or opposing forces and often includes quality attributes that must be met.
- A solution: a successful architectural resolution to the problem, appropriately abstracted. The solution describes the architectural structures that solve the problem, including how to balance the many forces at work.

  The solution for a pattern is determined and described by:
  - A set of element types (for example, data repositories, processes, and objects)
  - A set of interaction mechanisms or connectors (for example, method calls, events, or message bus)
  - A topological layout of the components
  - A set of semantic constraints covering topology, element behaviour, and interaction mechanisms

13.2. Overview of the Patterns Catalo

Applying a pattern is not an all-or-nothing proposition, in practice architects may choose to violate them in small ways when there is a good design trade-off to be had.

Layered Pattern (Module Pattern)

Context: All complex systems experience the need to develop and evolve portions of the system independently. For this reason the developers of the system need a clear and well-documented separation of concerns, so that modules of the system may be independently developed and maintained.

Problem: The software needs to be segmented in such a way that the modules can be developed and evolved separately with little interaction among the parts, supporting portability, modifiability, and reuse.

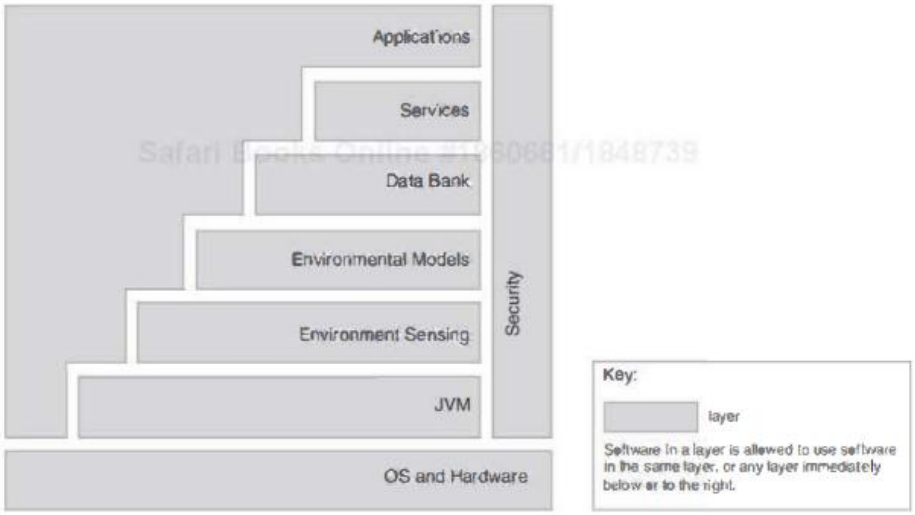
Solution

Table 13.1. Layered Pattern Solution

Overview	The layered pattern defines layers (groupings of modules that offer a cohesive set of services) and a unidirectional *allowed-to-use* relation among the layers. The pattern is usually shown graphically by stacking boxes representing layers on top of each other.
Elements	*Layer*, a kind of module. The description of a layer should define what modules the layer contains and a characterization of the cohesive set of services that the layer provides.
Relations	*Allowed to use*, which is a specialization of a more generic *depends-on* relation. The design should define what the layer usage rules are (e.g., "a layer is allowed to use any lower layer" or "a layer is allowed to use only the layer immediately below it") and any allowable exceptions.
Constraints	▪ Every piece of software is allocated to exactly one layer. ▪ There are at least two layers (but usually there are three or more). ▪ The *allowed-to-use* relations should not be circular (i.e., a lower layer cannot use a layer above).
Weaknesses	▪ The addition of layers adds up-front cost and complexity to a system. ▪ Layers contribute a performance penalty.

Notation

Layers are almost always drawn as a stack of boxes. The allowed-to-use relation is denoted by geometric adjacency and is read from the top down.



The most important point about layering is that a layer isn't allowed to use any layer above it. A module "uses" another module when it depends on the answer it gets back. But a layer is allowed to make upward calls, as long as it isn't expecting an answer.

Publish-Subscribe Pattern (C&C Pattern)

Context: There are a number of independent producers and consumers of data that must interact. The precise number and nature of the data producers and consumers are not predetermined or fixed, nor is the data that they share.

Problem: How can we create integration mechanisms that support the ability to transmit messages among the producers and consumers in such a way that they are unaware of each other's identity, or potentially even their existence?

Solution

Table 13.8. Publish-Subscribe Pattern Solution

Overview	Components publish and subscribe to events. When an event is announced by a component, the connector infrastructure dispatches the event to all registered subscribers.
Elements	*Any C&C component* with at least one publish or subscribe port. Concerns include which events are published and subscribed to, and the granularity of events. *The publish-subscribe connector*, which will have *announce* and *listen* roles for components that wish to publish and subscribe to events.
Relations	The *attachment* relation associates components with the publish-subscribe connector by prescribing which components announce events and which components are registered to receive events.
Constraints	All components are connected to an event distributor that may be viewed as either a bus—connector—or a component. Publish ports are attached to announce roles and subscribe ports are attached to listen roles. Constraints may restrict which components can listen to which events, whether a component can listen to its own events, and how many publish-subscribe connectors can exist within a system. A component may be both a publisher and a subscriber, by having ports of both types.
Weaknesses	Typically increases latency and has a negative effect on scalability and predictability of message delivery time. Less control over ordering of messages, and delivery of messages is not guaranteed.

Publish-subscribe adds a layer of indirection between senders and receivers, it provides less control over ordering of messages, and delivery of messages is not guaranteed.

The publish-subscribe pattern is used to send events and messages to an unknown set of recipients, this results in easy modification but has cost at runtime performance.

Examples

Graphical user interfaces, MVC-based apps, social networks (friends are notified), mailing lists.

Types

- List-based publish-subscribe: every publisher maintains a subscription list a list of subscribers that have registered an interest in receiving the event. It does not provide as much modifiability, but it can be quite efficient in terms of runtime overhead. Also, if the components are distributed, there is no single point of failure.
- Broadcast-based publish-subscribe: publishers simply publish events, which are then broadcast, and subscribers examine each event as it arrives and determine whether the published event is of interest. This version has the potential to be very inefficient if there are lots of messages and most messages are not of interest to a particular subscriber.
- Content-based publish-subscribe: each event is associated with a set of attributes and is delivered to a subscriber only if those attributes match subscriber-defined patterns (instead of being "topic-based" like the first two).

Multi-tier pattern (Allocation)

The multi-tier pattern is a C&C pattern or an allocation pattern, depending on the criteria used to define the tiers.

- Tiers: logical group components of similar functionality, in an enterprise system it will not be running on the computer that hosts the database (that's why it is allocation instead of C&C).

Context: In a distributed deployment, there is often a need to distribute a system's infrastructure into distinct subsets. This may be for operational or business reasons (for example, different parts of the infrastructure may belong to different organizations).

Problem: How can we split the system into a number of computationally independent execution structures groups of software and hardware connected by some communications media? This is done to provide specific server environments optimized for operational requirements and resource usage.

Solution

Table 13.11. Multi-tier Pattern Solution

Overview	The execution structures of many systems are organized as a set of logical groupings of components. Each grouping is termed a *tier*. The grouping of components into tiers may be based on a variety of criteria, such as the type of component, sharing the same execution environment, or having the same runtime purpose.
Elements	*Tier*, which is a logical grouping of software components. Tiers may be formed on the basis of common computing platforms, in which case those platforms are also elements of the pattern.
Relations	*Is part of*, to group components into tiers. *Communicates with*, to show how tiers and the components they contain interact with each other. *Allocated to*, in the case that tiers map to computing platforms.
Constraints	A software component belongs to exactly one tier.
Weaknesses	Substantial up-front cost and complexity.

Uses

In practice it is most often used in the context of client-server patterns. Tiers induce topological constraints , connectors may exist only between components in the same tier or residing in adjacent tiers (may be constrained).

Found in many Java EE and Microsoft .NET applications.

Disadvantage

Cost and complexity, for simple systems it might not justify its up-front and ongoing costs.

Layering pattern != Multi-tier

Layering is a pattern of modules (a unit of implementation), while tiers applies only to runtime entities.

- Layering = implementation
- Tiers = runtime

13.3. Relationships between Tactics and Patterns

Patterns Comprise Tactics

Tactics are atoms and patterns are molecules. Most patterns consist of (are constructed from) several different tactics.

For example: The layered pattern can be seen as the amalgam of several tactics increase semantic coherence, abstract common services, encapsulate, restrict communication paths, and use an intermediary.

Using Tactics to Augment Patterns

A documented pattern is underspecified with respect to applying it in a specific situation because it is described for a general context and applied to a specific one.

Make a pattern work in a given architectural context

- We need to compare the QA they promote (and the ones they diminish) with our needs.
- Other quality attributes that the pattern isn't directly concerned with, but which it nevertheless affects, and which are important in our application.

BrokerPatternexample??? No es de los patterns que nos tocan

We can use tactics to plug the gaps between the "out of the box" pattern and a version of it that will let us meet the requirements of our system. Of course, each of these tactics brings a trade-off.

13.5. Summary

Because patterns are (by definition) found repeatedly in practice, one does not invent them; one discovers them.

Tactics are simpler than patterns. For this reason they give more precise control to an architect when making design decisions than patterns. Tactics are the "building blocks" of design.

An architectural pattern establishes a relationship between:

- A context. A recurring, common situation in the world that gives rise to a problem.
- A problem. The problem, appropriately generalized, that arises in the given context.
- A solution. A successful architectural resolution to the problem, appropriately abstracted.

Patterns can be categorized by the dominant type of elements that they show

- Module patterns
- Component-and-connector patterns, most published patterns are C&C patterns, but there are module patterns and allocation patterns as well
- Allocation patterns show a combination of software elements (modules, components, connectors) and non-software elements.

A pattern is described as a solution to a class of problems in a general context. When a pattern is chosen and applied, the context of its application becomes very specific. A documented pattern is therefore underspecified with respect to applying it in a specific situation. We can make a pattern more specific to our problem by augmenting it with tactics