Tecnológico de Monterrey

Diseño y Arquitectura de Software

Repaso para CENEVAL

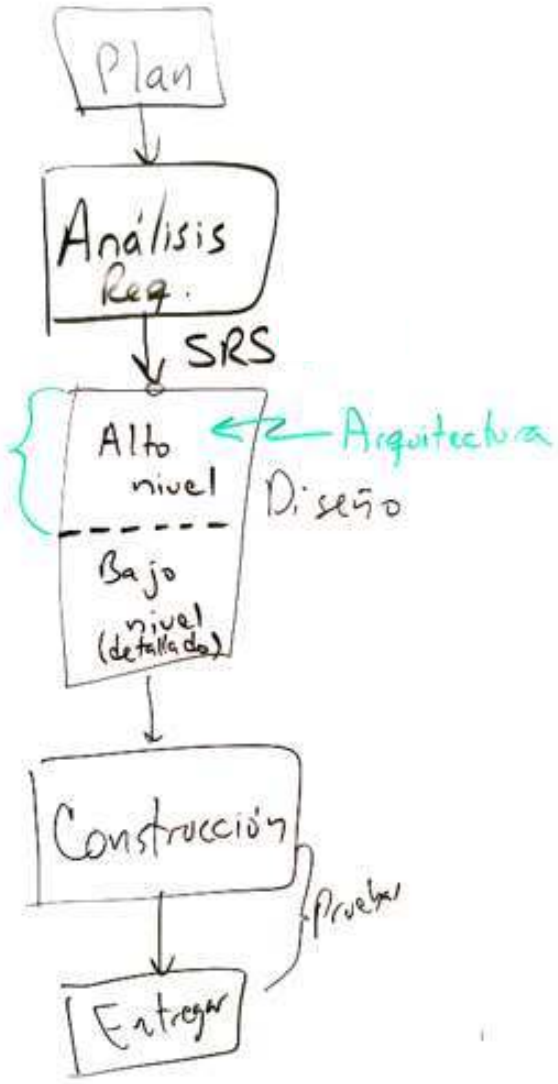
Ing. Rafael Salazar Chávez

Tecnológico de Monterrey

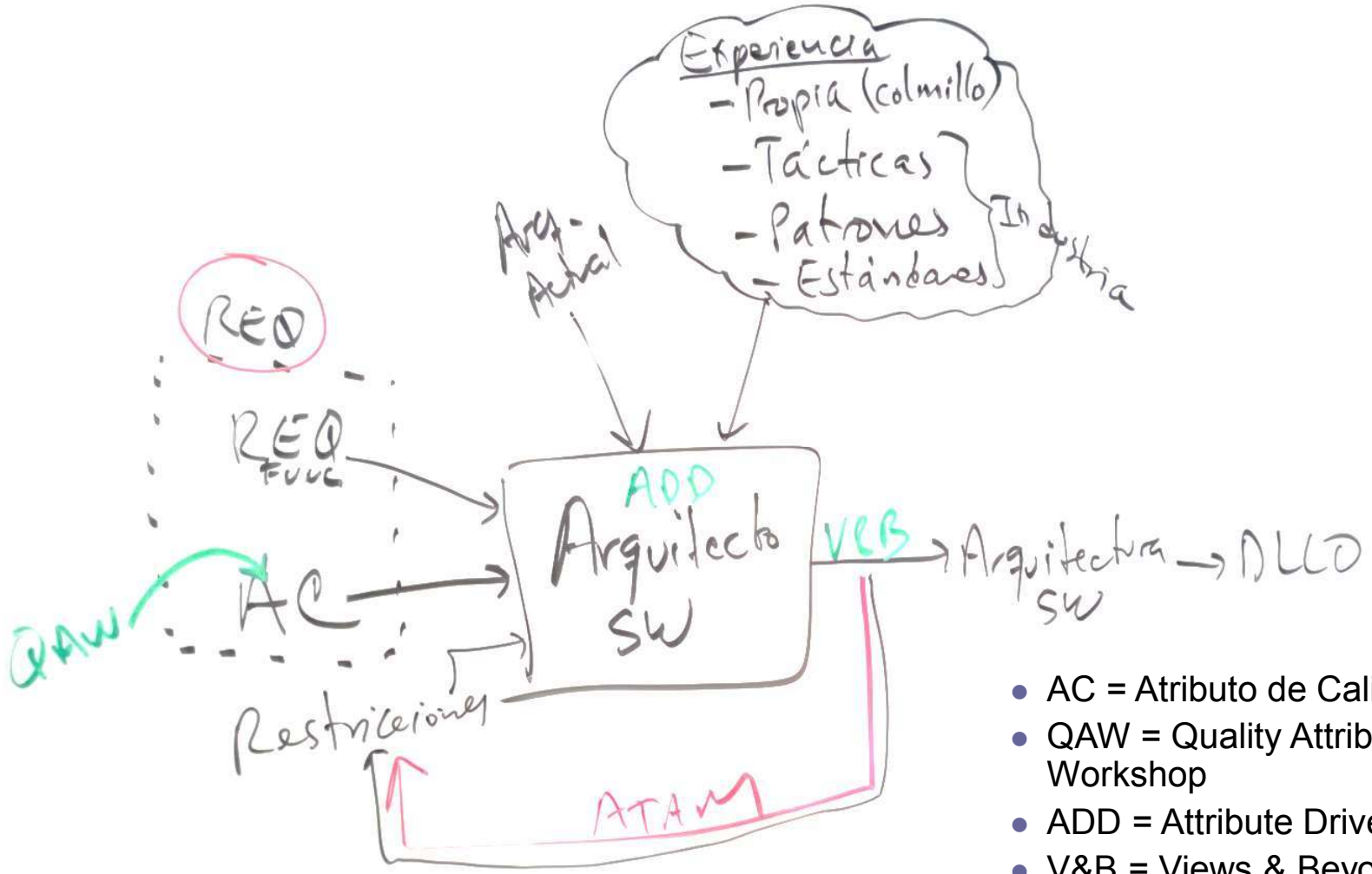
Bibliografía

- Libros:
  - Software Architecture in Practice, 3rd edition
    - Autores: Len Bass; Paul Clements; Rick Kazman
  - Design Patterns
    - Autores: Erich Gamma; Richard Helm; Ralph Johnson; John M. Vlissides; Grady Booch
- Estándares:
  - IEEE 1471
  - IEEE 610.12
  - ISO 12207
- Páginas de internet:
  - http://sourcemaking.com/design_patterns

Ciclo de vida



Diseño de Arquitectura



- AC = Atributo de Calidad
- QAW = Quality Attribute Workshop
- ADD = Attribute Driven Design
- V&B = Views & Beyond
- ATAM = Architecture Tradeoff Analysis Method

Preparación Ingeniería de Software CENEVAL (Diseño y Arquitectura de Software)

Juan Carlos Lavariega Jarquín
Departamento de Ciencias Computacionales
CETEC 4to Piso Torre Sur
lavariega@itesm.mx

CONTENIDO

- Diseño y Arquitectura de Software
  - Conceptos fundamentales
  - Arquitectura de Software
  - Estilos Arquitectónicos
  - Patrones de Diseño

CONCEPTOS FUNDAMENTALES

Fundamental Concepts

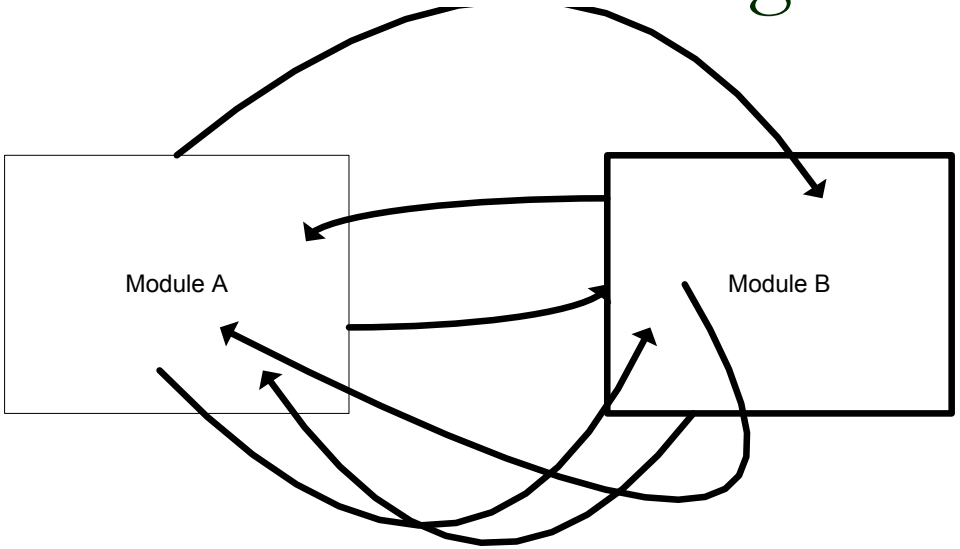
- abstraction—data, procedure, control
- refinement—elaboration of detail for all abstractions
- modularity—compartmentalization of data and function
- architecture—overall structure of the software
  - Structural properties
  - Extra-structural properties
  - Styles and patterns
- procedure—the algorithms that achieve function
- hiding—controlled interfaces

Functional Independence

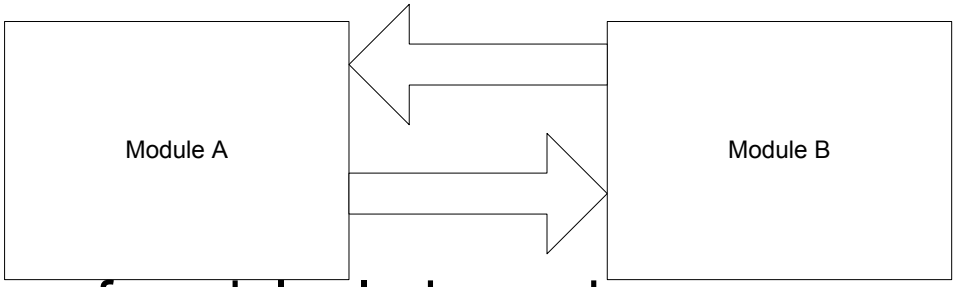
- COHESION
  - The degree to which a model performs one and only one function

- COUPLING
  - The degree to which a model is connected to other modules in the system

Functional-Oriented Design



Highly Coupled

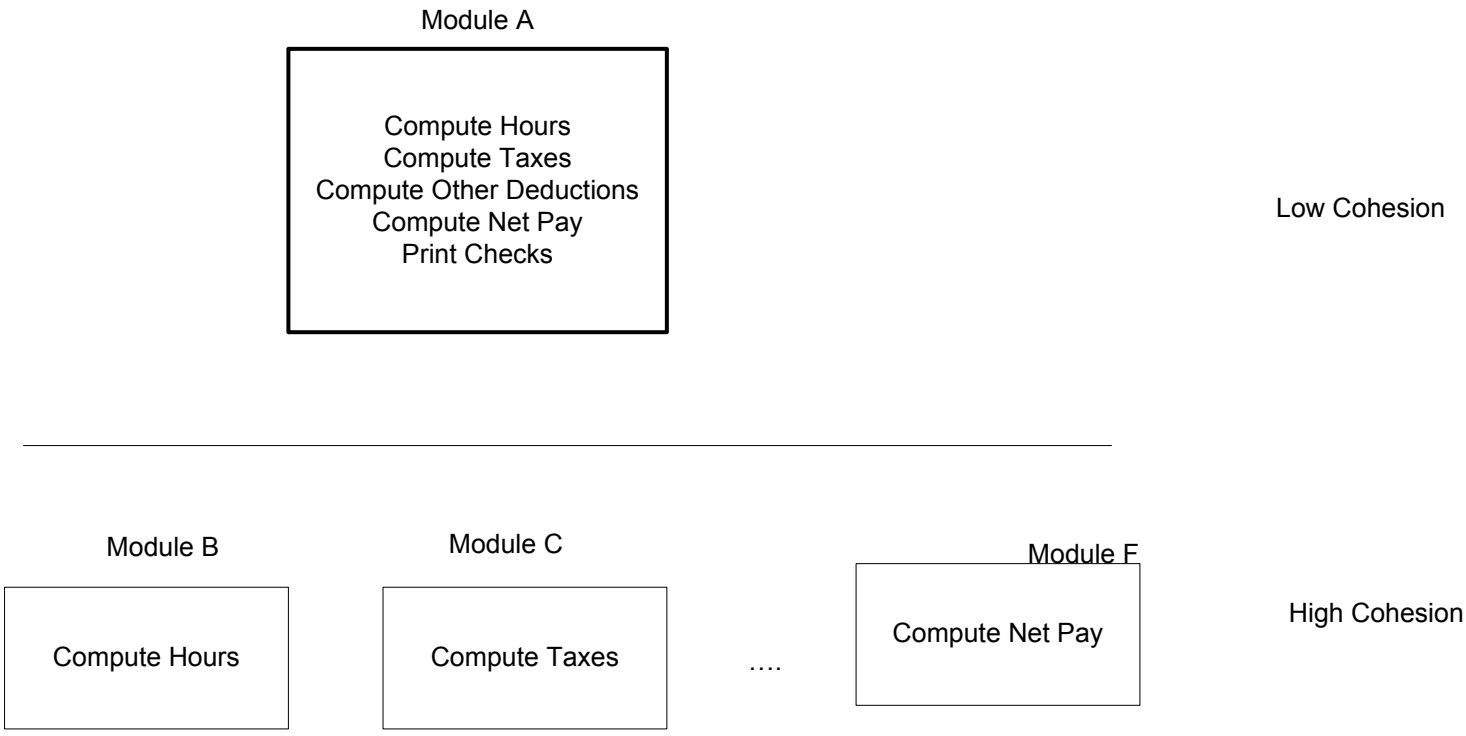


Loosely Coupled

- Degree of modular Independence
- Goal: loosely coupled modules
- Reduces complexity of Integration testing
- Reduces complexity of design and maintenance

Functional-Oriented Design

- Cohesion: A measure of functional strength within a module
- Goal: Modules with high cohesion
- Reduces complexity of design and maintenance



ARUITECTURAS DE SOFTWARE

Arquitectura de Software

- La Arquitectura del Software es el diseño de más alto nivel de la estructura de un sistema.

- Una Arquitectura de Software, es un marco de referencia que muestra los componentes principales de un sistema y la comunicación entre ellos.
- La Arquitectura de Software se puede reutilizar para sistemas con características similares

Definition of Software Architecture

- The software architecture of a program or computing system is the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them.

Bass, Clements, Kazman, Software Architecture in Practice. Reading, MA, Addison- Wesley, 1998

Definition 2

- The fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution.
- REF: ANSI/IEEE Std 1471-2000
- IEEE Std 610.12:1990, IEEE Standard Glossary of Software Engineering Terminology.
- IEEE/EIA Std 12207.0:1996, IEEE/EIA Standard—Industry Implementation of software
- ISO/IEC 12207: 1995, Information Technology—Software life cycle processes

Conceptos involucrados

- Vistas (Código, Módulo, Ejecución, Conceptual)
- Procesos de desarrollo (RUP, cascada, prototipo, espiral)
- OOP
- Patrones de software (arquitecturales, de diseño)
- Frameworks
- Estilos arquitectónicos (MVC, Cliente/Servidor, P2P, Thin client).
- Lenguajes de descripción de arquitecturas (UML)

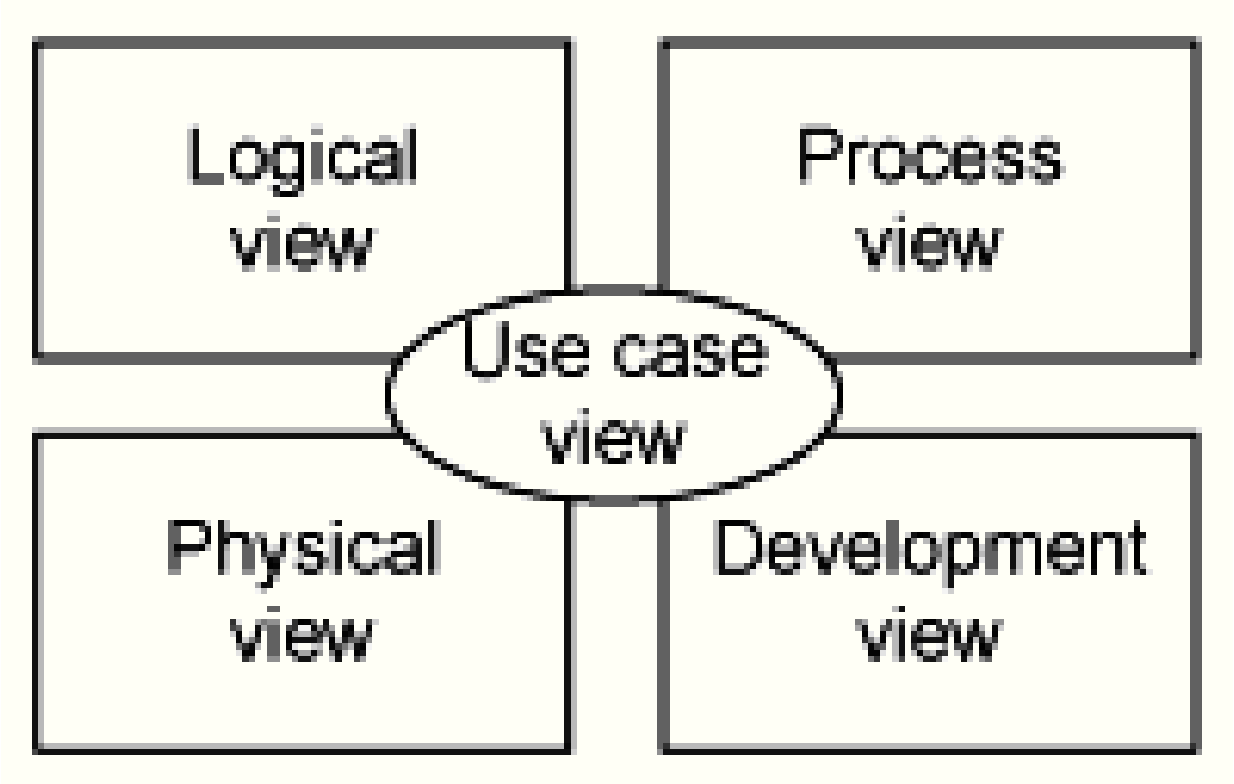
Vistas arquitecturales

- Modelo 4 + 1
  - Introducido en 1995 por Philippe Kruchten
  - Examina en partes la arquitectura del sistema
  - Reduce complejidad la vista general del sistema
  - Los arquitectos lo usan para entender y documentar las múltiples capas de una aplicación de forma sistemática y estandarizada.
  - Los documentos creados usando el proceso 4+1 son fácilmente utilizados por todos los miembros del equipo de desarrollo.

Modelo de vistas 4 + 1

- 4 vistas escenciales
  - Lógica
  - Procesos
  - Física
  - Desarrollo
- +1
  - Vista de Casos de Uso

Diagrama del modelo 4+1

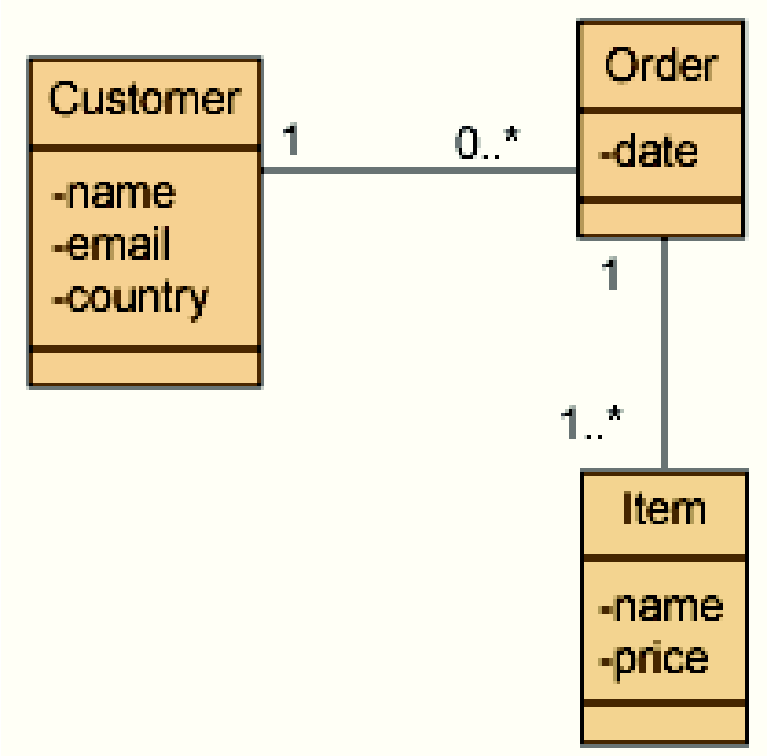


Vista Lógica (1)

- Soporta requerimientos de comportamiento y muestra como el sistema es descompuesto en un conjunto de abstracciones.
- Clases y objetos son los principales elementos estudiados en esta vista.
- Se pueden usar diagramas de clases, diagramas de colaboración y diagramas de secuencia, entre otros, para mostrar las relaciones de esos elementos desde una vista lógica.

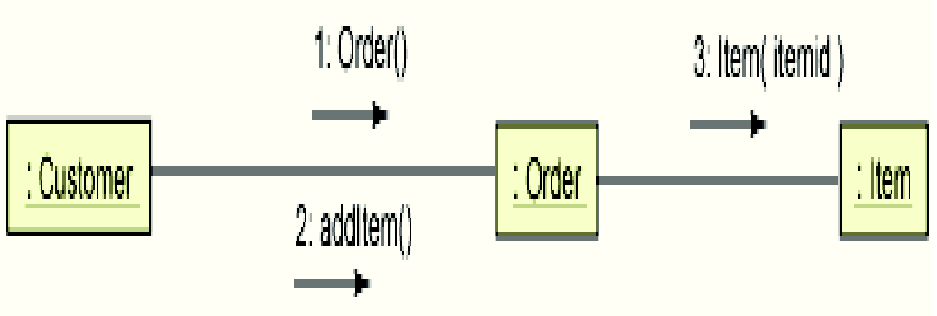
Vista Lógica (2)

- Diagramas de Clases
  - Proporcionan una vista completa del sistema.
  - Son diagramas estáticos



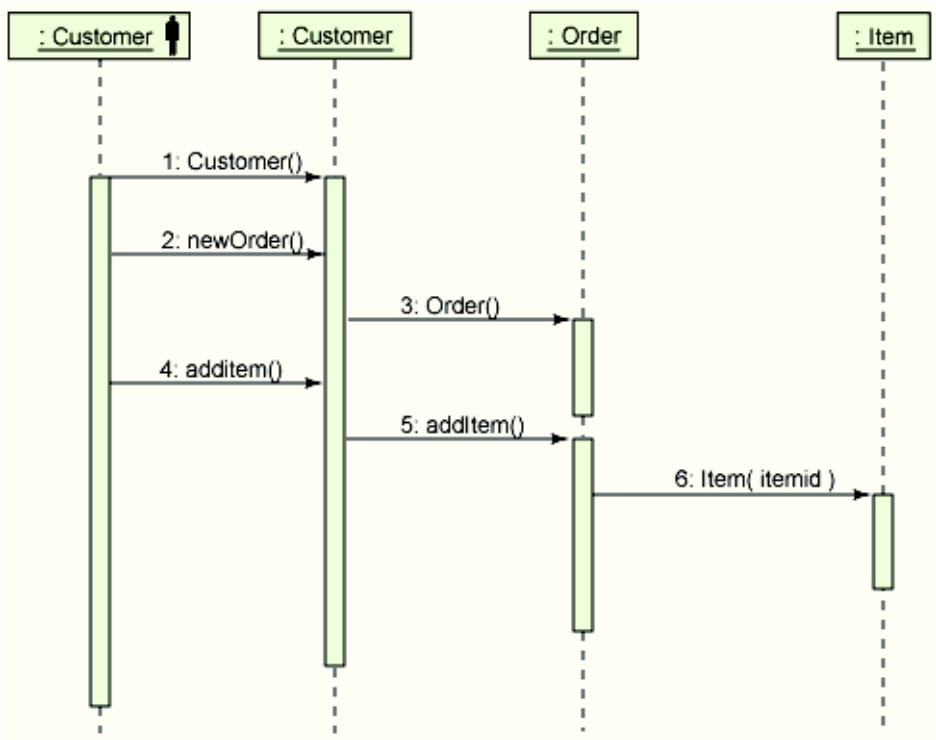
Vista Lógica (3)

- Diagramas de colaboración
  - Forma en la que se pasan los mensajes y llamadas entre los objetos del sistema.



Vista Lógica (4)

- Diagramas de secuencia
  - Agregan detalle a los diagramas de colaboración

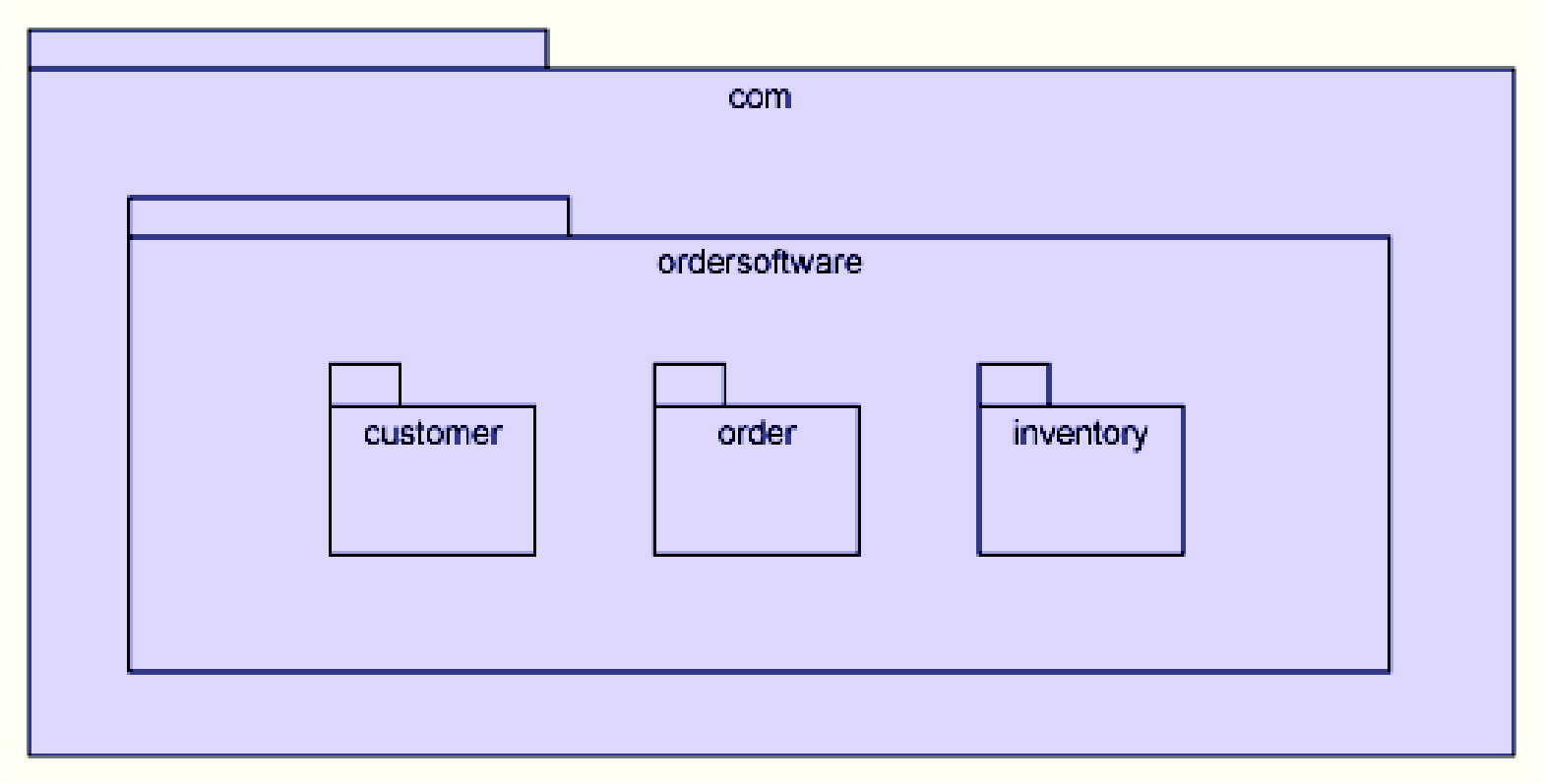


Vista de desarrollo (1)

- Usada para describir los módulos del sistema.
- Módulos: Bloques más grandes que las clases y objetos. Varían según el ambiente de desarrollo.
- Paquetes, subsistemas y librerías de clases son considerados módulos.

Vista de Desarrollo (2)

- Diagrama de Paquetes

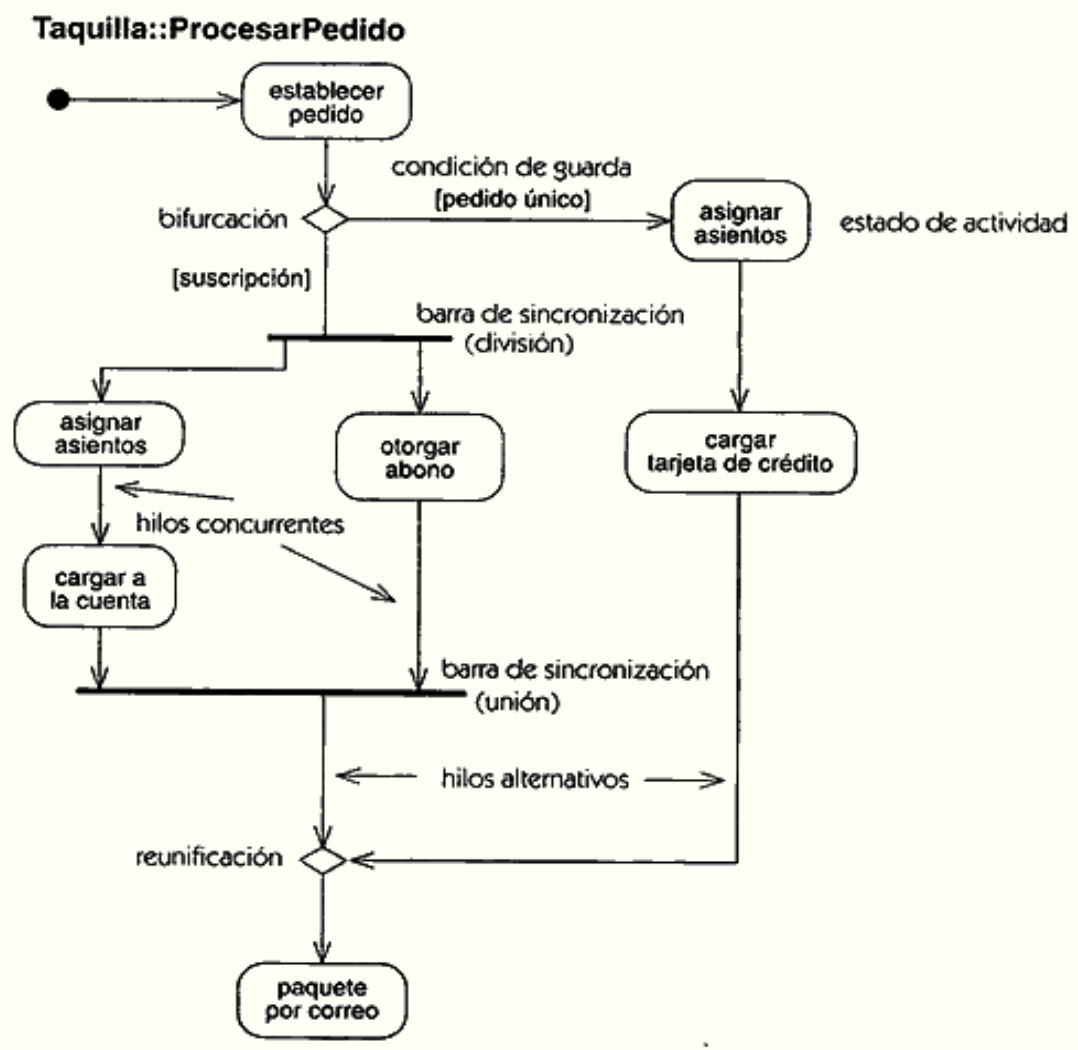


Vista de Procesos (1)

- Permite describir y estudiar los procesos de sistema y la forma en la que se comunican.
- Es muy útil cuando se tienen procesos múltiples y simultáneos o hilos en su software.
- Toma en cuenta muchos requerimientos no funcionales o requerimientos de calidad, tales como desempeño, disponibilidad, etc.
- Los diagramas de actividad son ampliamente usados para describir esta vista.

Vista de Procesos (2)

- Diagrama de Actividades

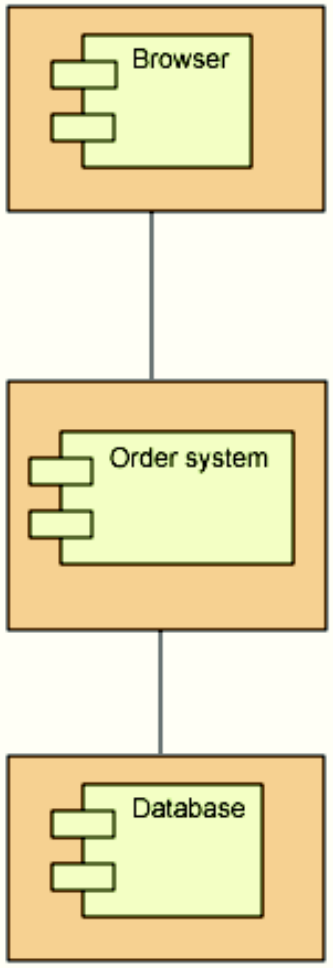


Vista Física (1)

- Define cómo se instala la aplicación y cómo se ejecuta en una red de computadoras.
- Toma en cuenta requerimientos no funcionales como disponibilidad, confiabilidad, desempeño y escalabilidad.

Vista Física (2)

- Diagrama de Desarrollo

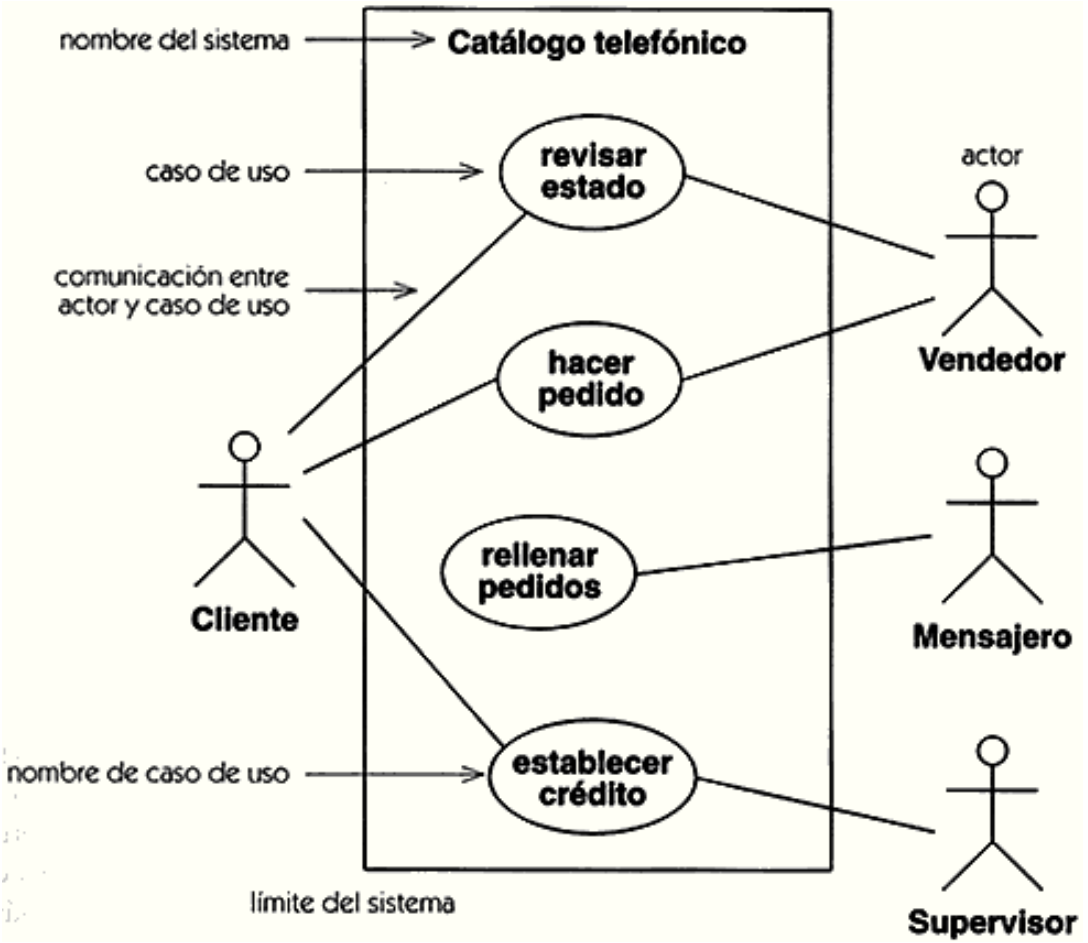


Vista +1 (Casos de Uso) (1)

- Consiste en casos de usos y escenarios que detallan o complementan las otras vistas.
- Los casos de uso representan la parte funcional del sistema, explican la funcionalidad y estructuras descritas por otras vistas.

Vista +1 (Casos de Uso) (2)

- Diagrama de Casos de Uso



Conclusión Modelo vistas 4+1

- Es un método útil y estandarizado para estudiar y documentar un sistema de software desde una perspectiva arquitectural.
- Cada vista que compone el modelo proporciona una ventana a diferentes niveles del sistema.

ESTILOS ARQUITECTONICOS

Estilos Arquitectonicos (modelos de referencia)

- An architectural pattern expresses a fundamental structural organization schema for software systems.
- It provides:
  - a set of predefined subsystems
  - specifies their responsibilities
  - includes rules and guidelines for organizing the relationships between them

Estilos Arquitectonicos

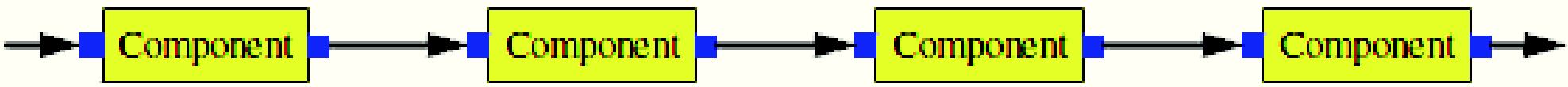
- Data flow systems
  - Batch sequential
  - Pipes and filters
- Call-and-return systems
  - Main program and subroutines
  - Client-server systems
  - Object-oriented systems
  - Hierarchical layers
- Independent components
  - Communicating processes
  - Event-based systems
- Virtual machine
  - Interpreters
  - Rule-based systems
- Data-centered systems
  - Database systems
  - Blackboards

Data Flow Systems

- The availability of data controls the computation process.
- Data centered: what to compute (not how).
- Conceptual elements:
  - Process
  - File
  - data flow
- Data flows from one process to the next, through a series of processes that do operations on the data.
- Common forms:
  - Batch Sequential
  - Pipes and Filters

The Data Flow Architecture

- Each component is isolated from performance of upstream component,
  - Different speeds is taken care of by buffers .Might lead to overflow error
- Components may be memoryless (no internal state preserved).
- Only memoryless components can be replaced at runtime.



Batch Sequential Systems

- In Batch Sequential Systems
  - a number of programs are executed sequentially
  - each program runs to completion
  - data is transmitted between programs via files stored on external storage.
    - a program reads data from one or more files,
    - processes it,
    - and writes one or more files
  - after exiting the next program is started, if not the last one.
- Issues
  - Low memory requirement
  - High latency for I/O
  - Limited concurrency
  - No interactivity

Pipes and Filters Systems

- In Pipes and Filters Systems
  - data sets are processed in discrete, separable stages.
  - each stage is represented by a component.
  - each component operates on the entire data set and passes the results to next stage. the component act as a filter.
- connectors serves as links
  - act as pipes

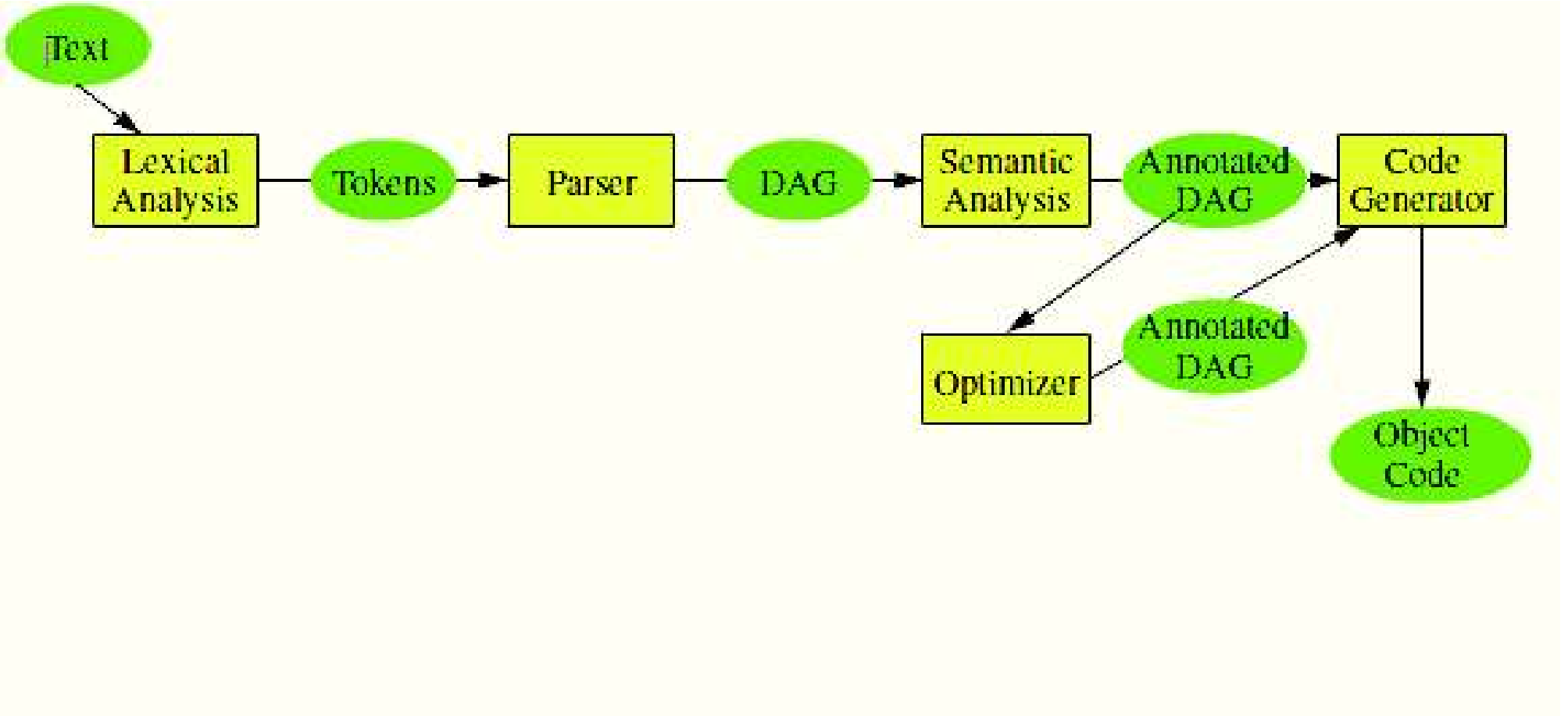
Filters

- Basic activities:
  - enrich input data, add data, e.g. from a data store or computed values
  - refine input data, delete or sort data
  - transform input data, e.g. from one type to another
- Two types of Filter:
  - Active Filter
    - drives the data flow
  - Passive Filter
    - driven by the dataflow
- In a Pipes and Filters Architecture at least one of the filters must be active. This active filter can be the environment, e.g. user input.

Pipes

- A Pipe
  - transfers data from one data source to one data sink
  - may implement a (bounded/unbounded) buffer
  - may connect
    - two threads of a single process
      - may contain references to shared language objects
    - two processes on a single host computer
      - may contain references to shared operating systems objects (e.g. files)
    - two processes on any host computer
      - distributed system

Pipes and Filters — Example



Pipes and Filters Systems

- Easy to understand system input/output.
- Support for reuse
  - Any two filters can be hooked together provided they agree on data transmission.
- Easy maintenance:
  - new filters can be added
  - old filters can be replaced
- Throughput and deadlock analysis possible.
- Natural support for concurrency.
  - Different programs.
  - No shared memory.

Call-and-Return Systems

- Synchronous execution:
  - A component
    - calls another component
    - ceases execution and waits until the other component is ready
    - may get a return value
    - continues execution
  - is used when the order of computation is fixed.
- Common forms:
  - Main program and subroutines
  - Client-server systems
  - Object-oriented systems
  - Hierarchical layers

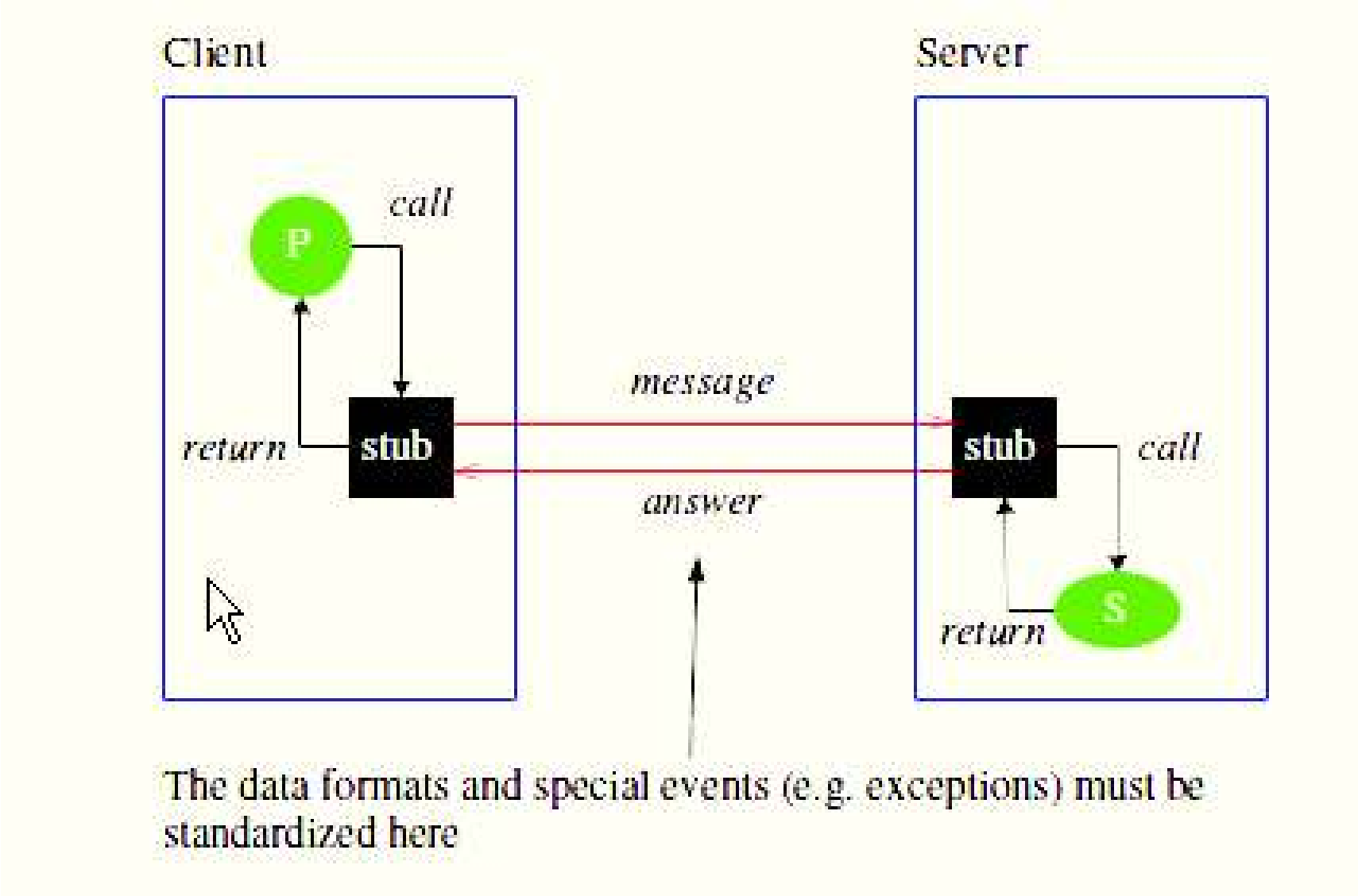
Main Program and Subroutines

- A main program acts as the controller.
- One or more Subroutines perform specific functions when called by the main program.
- A single thread of control.
- The correctness of execution is dependant on the correctness of he main program and all subroutines that are called.
- Hierarchical decomposition
- Single thread of control
- Subroutines typically aggregated into subsystems
- Hierarchical reasoning
- Old-fashioned, popular in the 1970s to 90s: Jackson Structured Programming

Client-Server Systems

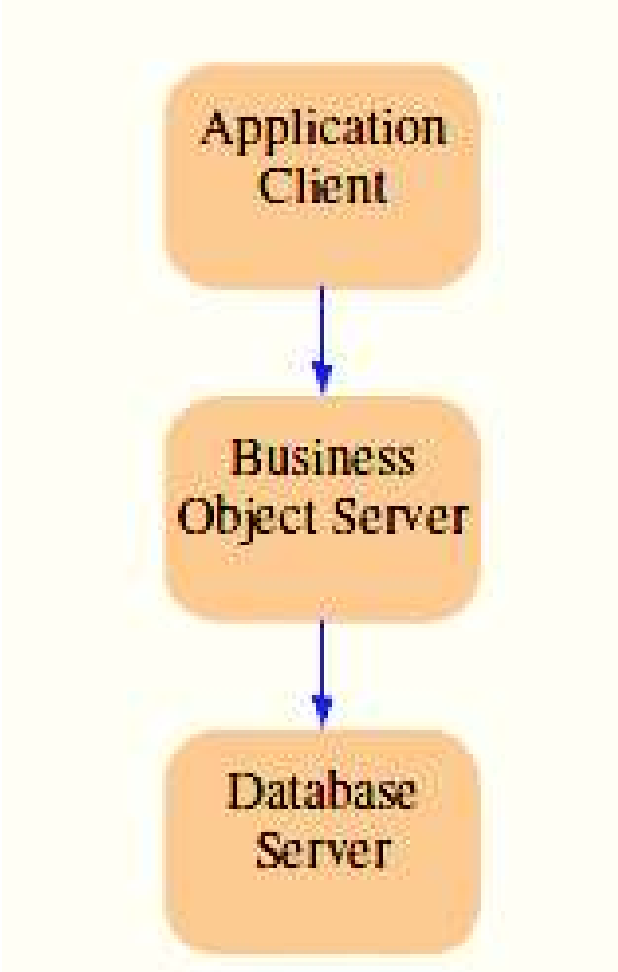
- A Server provides different services.
- A Client uses the services as (remote) subroutines from one ore more Servers.
- The Server might be a Client to another Server.
- A variant of the Main Program and Subroutines but the clients and servers
  - are implemented as separate entities (processes),
  - might be situated at different processors, and
  - can be implemented in another style (most often some variant of object oriented style).
- The mechanism for calling a server is called Remote Procedure Call, RPC.
  - This will require a protocol.

Remote Procedure Call



Client-Server Systems

- 3-tier: Functionality divided into three logical partitions:
  - Application services
  - Business services
  - Data services

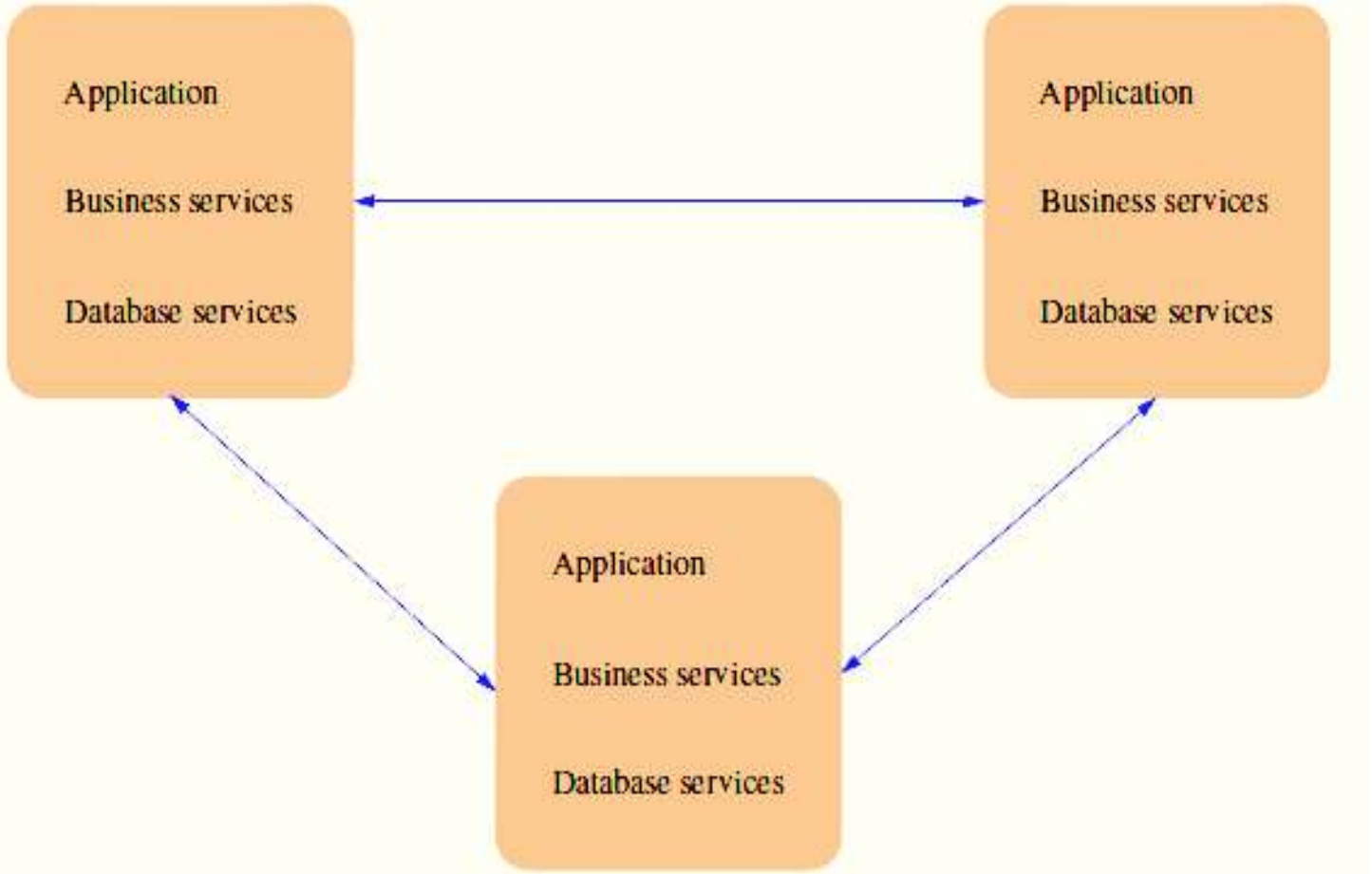


Client-Server Systems

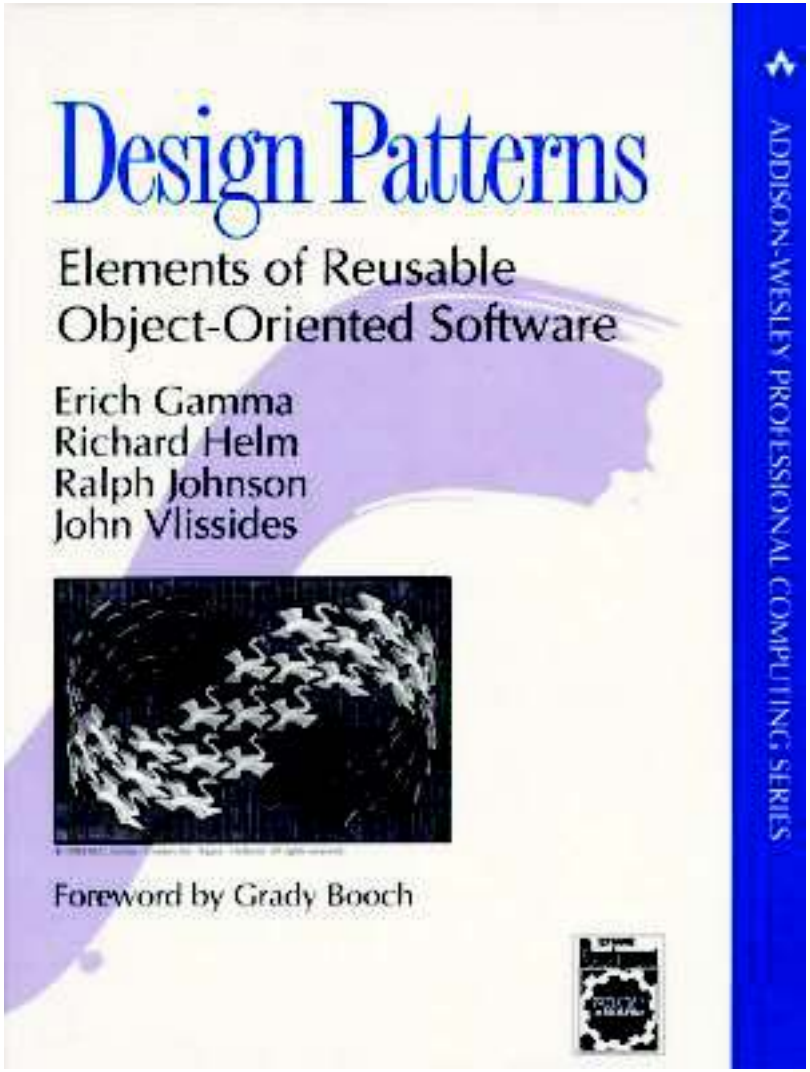
- When designing client-server systems the following must be considered:
  - Authentication:
    - Verifying the identity of the client and the server.
  - Authorization:
    - verifying the rights/privileges of the client.
  - Data security:
    - Protect the data stored on the server.
  - Protection:
    - Protect the server from malfunctioning clients.
  - Middleware:
    - How to connect the clients to the server.

Peer-to-Peer Systems

- Peer-to-Peer Systems, P2P are
  - like Client-Server but all processes can be both clients and servers.
  - more flexible b
    - deadlock
    - starvation



PATRONES DE DISEÑO



Los patrones han sido utilizados en muchas areas, desde organizaciones, procesos de enseñanza y arquitectura. A pesar de que los orígenes del diseño de patrones son mas enfocados al urbanismo y a la arquitectura, estos son aplicables a otras disiplinas, incluyendo la ingeniería de software.

Orígenes de los patrones de diseño.
Los patrones de diseño se volvieron populares gracias a la gran aceptación del libro "*Design Patterns: Elements of Reusable Object-Oriented Software*" de Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, conocidos como "the Gang of four".

Plantilla GoF

- Nombre: un nombre descriptivo del patrón.
- Clasificación : la clasificación del patrón (creación, comportamiento, estructural)
- Intención: que hace el patrón?
- También conocido como: nombre alternativo.
- Motivación: escenario ilustrativo mostrando como soluciona un problema el patrón.
- Aplicabilidad: cuándo, cómo y porqué
- Estructura: representación gráfica de clases
- Participantes: clases y/o objetos y responsabilidades en patrón
- Colaboración: como colaboran participantes en solución

- Consecuencias: Como el patrón soporta sus objetivos. Trade-offs
- Implementación: explicación de cómo implementarlo.
- Ejemplo: código en algún lenguaje de programación OO.
- Usos Conocidos: Ejemplos del patrón en sistemas reales
- Patrones relacionados: que otros patrones están relacionados con este?

- *Nivel: Clase única, componente, arquitectónico.*
- *Variantes. Posibles implementaciones alternativas y variaciones.*

Catálogo de Patrones GoF

- GoF propone una clasificación de patrones de diseño siguiendo dos criterios:
  - 1) Propósito. Refleja que hace el patrón. Los patrones pueden tener un propósito de:
    - Creación. Tienen que ver con el proceso de creación de objetos.
    - Estructurales. Tratan con la composición de clases u objetos.
    - Comportamiento. Caracterizan el modo en que las clases y objetos interactúan y se reparten la responsabilidad.
  - 2)Ámbito. Clasifica el patrón como:
    - Patrón de clases. Se ocupan de las relaciones entre clases y sus subclases a través de la herencia. Las relaciones son estáticas (creadas en tiempo de compilación).
    - Patrón de objetos. Tratan con las relaciones entre objetos, que pueden ser dinámicas (pueden cambiar en tiempo de ejecución).

Catalogo GoF

- Creación
  - Facilitan una de las tareas más comunes de programación OO, la creación de objetos en un sistema. Algunos de estos patrones son: *Abstract Factory, Builder, Factory Method, Prototype, Singleton.*
- Comportamiento
  - Están relacionados con el flujo de control de un sistema. Ciertas formas de organizar el control en un sistema pueden derivar en grandes beneficios para la eficiencia y mantenimiento del mismo. Algunos de estos patrones son: *Command, Interpreter, Iterator, Mediator, Memento, Observer,* entre otros.
- Estructurales
  - Describen formas efectivas de particionar y combinar los elementos de una aplicación. Algunos de estos patrones son: *Adapter, Bridge, Composite, Facade, Proxy,* entre otros.

Patrones Creacionales

- Abstract Factory
  - Provee una interfaz para crear familias de objetos dependientes o relacionados
- Builder
  - Separa la construcción de un objeto complejo de sus representaciones, de forma tal que el mismo proceso de construcción puede crear diferentes representaciones
- Factory Method
  - Define una interfaz para la creación de un objeto. Pero deja a las subclases decidir que clase instanciar.
- Prototype
  - Especifica el tipo de objetos a crear usando una instancia prototipo y creando un nuevo objeto al copiar este prototipo.
- Singleton
  - Asegura que una clase tiene únicamente una instancia y provee una punto de acceso universal a la instancia.

Patrones Estructurales

- Adapter.
  - Convertir la interfaz de una clase en otra interfaz que ele cliente espera. Adapter, permite a las clases trabajar juntas que de otra forma serian incompatibles
- Bridge
  - Desacopla una abstracioón de su implentacion de tal forma que las dsos pueden variar de forma independiente
- Composite
  - Compone objetos en estructuras de árboles para representar jerarquias del tipo todo-partes.
- Decorator
  - Agrega responsabilidades adicionales a un objeto de forma dinámica. Una forma flexible de crear subclases.
- Façade
  - Provee de una interfaz uniforme a un conjunto de interfaces en un subsistema.
- Flyweight
  - Utiliza comparticion para soportar un gran numero de objetos pequeños en forma eficiente.
- Proxy
  - Provee un objeto surrogado o en lugar de otro objeto para controlar el acceso al objeto « protegido»

Patrones de Comportamiento

- Chain of Responsibility
  - Evita acoplar el invocador de una petición al receptor al permitir que más de un objeto responda. Encadena el objeto receptor y pasas la petición a lo largo de una cadena hasta que un objeto la atiende.
- Command
  - Encapsula una petición a un objeto, permitiendo que los clientes se parametricen con diferentes peticiones, cola o bitácora de peticiones y soportar operaciones que se pueden deshacer.
- Interpreter
  - Dado un lenguaje, define una representación de su gramática en conjunto con un intérprete que usa la representación para entender las instrucciones en el lenguaje.
- Iterator
  - Provee de un medio para acceder a los elementos de un objeto agregado secuencialmente sin exponer su representación interna.
- Mediator
  - Define un objeto que encapsula como un conjunto de objetos deben interactuar. Mediator promueve encapsulación mínima al dejar que los objetos se referencien explícitamente y variando su interacción independientemente

Patrones de Comportamiento

- Memento
  - Sin violar encapsulación, captura y externaliza el estado interno de un objeto de tal forma que el objeto pueda ser restaurado a un estado anterior.
- Observer
  - Define una dependencia de tipo uno-a-muchos entre objetos de forma que cuando un objeto cambia sus estado, todos sus dependientes son notificados y actualizados automáticamente.
- State
  - Permite a un objeto alterar su comportamiento cuando su estado interno cambia. El objeto aparecerá que cambia su clase.
- Strategy
  - Define una familia de algoritmos, encapsula cada uno y los hace intercambiables. Strategy permite a un algoritmo variar independientemente de los clientes que lo usen.
- Template Method
  - Define el esqueleto de un algoritmo en una operación, difiriendo algunos pasos a las subclases. Este patrón permite a las subclases redefinir ciertos pasaos de un algoritmo sin cambiar la estructura del algoritmo.
- Visitor
  - Representa una operación a ser realizada en los elementos de un objeto estructura. Visitor permite definir una operación nueva sin cambiar las clases o los elementos en los cuales opera,

Tecnológico de Monterrey

Bibliografía

- Libros:
  - Software Architecture in Practice, 3rd edition
    - Autores: Len Bass; Paul Clements; Rick Kazman
  - Design Patterns
    - Autores: Erich Gamma; Richard Helm; Ralph Johnson; John M. Vlissides; Grady Booch
- Estándares:
  - IEEE 1471
  - IEEE 610.12
  - ISO 12207
- Páginas de internet:
  - http://sourcemaking.com/design_patterns