Introducción

Hacer código **entendible** es tan importante como hacer código ejecutable.

En empresas se producen menos líneas de código por persona por mes. Porque hay mucho extra que toma tiempo y esfuerzo. Por ejemplo se agrega documentación, y testing.

La **ingeniería del software** se define como la aplicacion de un enfoque sistematico, disciplinado y cuantificable al desarrollo, operación y mantenimiento del software.

El Dominio del Problema

El dominio del problema es el software industrial (*industrial strength software*).

Tarde y No Confiable

El presupuesto y la planificación están fuera de control. Código *runaway* (desbocados).

Los errores de software son diferentes de otras ingenierías porque no envejecen (no se modifican con el tiempo). **En el software los errores son introducidos en el proceso de diseño y desarrollo**.

Mantenimiento y Rehacer trabajo

Una vez que se entrega el software se entra en la *fase de mantenimiento*. Los costos de mantenimiento en general exceden los costos de desarrollo del sistema.

Mantenimiento correctivo	Mantenimiento adaptativo
Correcciones de errores residuales que quedaron en el sistema y hay que corregirlos a medida que se detectan.	Adaptación del software a las necesidades del **entorno cambiante**.

** Uno de los mayores problemas en el desarrollo del software es entender lo que se desea del producto. Tanto los clientes como los desarrolladores deben *visualizar* cómo debería ser el comportamiento del software una vez que este listo. A veces eso no ocurre y hay que rehacer trabajo. También puede suceder que al cambiar el entrono cambien las necesidades y por eso se debe acomodar el software a los nuevos requerimientos.

El desafío de la Ingeniería del Software

Enfoque sistemático: Para predecir el trabajo que va a hacer falta y calcular un presupuesto. Las metodologías para el desarrollo deben ser *repetibles*.

Desafíos principales

1. **Escala**: Queremos software escalable de chico a grande y de grande a chico: no exija recursos de más.
2. **Calidad**
3. **Productividad**
4. **Consistencia**
5. **Cambios**: de programadores o hardware.

Calidad

Es uno de los desafíos principales de la ingeniería del software. Para medir la calidad, se utilizan los siguientes parámetros:

1. **Funcionalidad**: capacidad de proveer las funciones que uno quiere y necesita
2. **Confiabilidad**: que no haga más cosas de las que queremos. Que se hagan las cosas en tiempo y forma.
3. **Usabilidad**: Fácil de usar. Nunca lo usa la persona lo programó. Debe ser comprensible.
4. **Eficiencia**: Usar la menor cantidad de recursos y tiempo. No siempre se puede acceder al hardware potente, los usuarios no tienen los mejores procesadores.
5. **Mantenibilidad**: Es importante para adaptarse al usuario. Es clave la legibilidad.
6. **Portabilidad**: que funcionen en diferentes entornos (sin necesidad de toquetear el código o tocando poquito).

Cambios:

Dos tipos:

- **Cambios de programadores**: (aumento de sueldo a través de cambios) El código se debe entender para modificarse.
- **Cambio del hardware**.

El enfoque de la Ingeniería del Software

Para alta C&P (relación calidad y precio) necesitamos buena **tecnología**, buenos **procesos** o **métodos** y **gente** que haga bien su trabajo.

El proceso: es lo que menos varía en el tiempo. Queremos la sistematización de procesos.

Administración del proceso: Fases del proceso de desarrollo

Es importante separar en fases para la separación de incumbencias (echar culpas). Cada grupo tiene una tarea específica. Cuando la tarea general falla, es importante saber qué parte falló.

Permite verificar la calidad de cada clase. El proceso en fases es central en el enfoque de la Ingeniería del Software para solucionar la *crisis del software* (no saber dar un buen presupuesto y fecha de entrega).

En general los modelos de procesos consisten de:

1. **Análisis de requisitos y especificación:** qué quiere el cliente
2. **Arquitectura y Diseño:** el sistema se separa en módulos
    i. Diseño
    ii. Diseño detallado
    iii. Diseño de alto nivel
3. **Codificación**
4. **Testing**
5. **Entrega e instalación:** entregar, instalar y mantener

Queremos que cada una tenga una entrada y una salida definidas para testear.

Análisis y especificación de los Requisitos del Software

- **Entrada**: Las necesidades se encuentran en la cabeza de alguien (ideas abstractas)
- **Salida** : Un detalle preciso de lo que será el sistema futuro: **SRS** (*System Requirements Specification*).
- Identificar y especificar los requisitos (involucra **interacción con la gente** y no puede automatizarse).
- **Análisis**: *Entender* al cliente. Descubrir qué se pretende del producto. Ida y vuelta entre el analista y el cliente.
  Un buen analista debe asesorar y convencer y que el cliente acepte en lo que uno es bueno.

SRS

Debe dar una clara comprensión de lo que se espera de un determinado software.

La SRS es el medio para **reconciliar las diferencias** y **especificar las necesidades** del cliente/usuario de manera que todos entiendan.

Hay abogados y escribanos involucrados. Es un **contrato** o acuerdo entre cliente/usuario y quién suministrará el software. No debe ser ambigua.

Requerimientos del software

Hay que tenerlos en concreto para producir el software.

Requerimientos (IEEE)

1. Una condición o capacidad necesaria de un usuario para solucionar un problema o alcanzar sus objetivos.
2. Una condición o capacidad necesaria que debe poseer o cumplir un sistema [...].

¿Por qué la SRS es necesaria?

1. **Una SRS establece el contrato entre el ciente y el proveedor respecto a lo que el software va a hacer.**
   - El cliente no comprende el proceso de desarrollo de software. Y el desarrollador no conoce el problema ni su área de aplicación.
   - La SRS ayuda a comprender las necesidades del cliente (tanto para el cliente mismo como para el desarrollador).
2. **La SRS provee un referencia para la validación del producto final.**
   - (Determina el resultado correcto). Verificación: "el software satisface la SRS".
3. **Una SRS de alta calidad es esencial para obtener un software de calidad.**
   - Los errores de requerimientos solo se manifiestan en el software final. Los defectos de requerimientos no son pocos. Ejemplo: 45% de los errores en testing correspondían a errores de requerimientos (siendo el 25% del total de defectos encontrados en el proyecto).
4. **Una buena SRS reduce los costos de desarrollo.**
   - Contribuye a minimizar cambios y errores.
   - Los cambios de requerimientos pueden costar demasiado (hasta un 40%).
   - En cada etapa es más caro el precio de encontrar y resolver un error de los requerimientos.

Proceso de requerimientos

Secuencia de pasos que se necesita realizar para convertir las necesidades del usuario en la SRS

	Es iterativo y en paralelo: Algunas partes pueden estar siendo especificadas mientras otras están aún bajo análisis.

Análisis del problema

Objetivo: Lograr una buena comprensión de las necesidades, requerimientos y restricciones del software.

El análisis incluye:

1. Entrevistas con el cliente y usuarios.
2. Lectura de manuales.
3. Estudio del sistema actual.
4. Ayudar al cliente/usuario a comprender las nuevas posibilidades.

Es importante:

- Obtener la información necesaria
- Lluvia de ideas: discutir. **Interactuar** con el cliente para establecer propiedades
- Relaciones interpersonales
- Habilidades de comunicación
- Organizar la información
- Asegurar **completitud**
- Asegurar **consistencia**
- Evitar diseño interno: evitar tratar de resolver el problema.

  Particionar el problema Estrategia: Descomponer e problema en pequeñas partes, comprenderlas y relacionarlas entre ellas.

Comprender los sub-problemas y la relación respecto a:

- Funciones (análisis estructural)
- Objetos
- Eventos del sistema

Método de análisis estructurado

1. Dibujar el diagrama del contexto.
2. Dibujar el DFD del sistema existente: refinar el anterior. Es importante la comunicación con el cliente.
3. Modelado del sistema propuesto. Dibujar el **DFD del sistema propuesto** e identificar la frontera hombre-máquina.

Diagrama de Contexto

- Identifica el contexto.
- Es un DFD con un único transformador (el sistema), con entradas, salidas, fuentes y sumideros del sistema.

DFD del sistema existente

- El **sistema actual*** se modela tal como es con un DFD con el fin de comprender el funcionamiento.
- Se refina el diagrama de contexto.
- Pueden usarse DFD en niveles jerárquicos.

- Para obtenerlo se debe interactuar intensamente con el usuario.
- El DFD obtenido se valida junto a los usuarios.

* no es necesariamente de software. Es un sistema que queremos automatizar.

Modelado del sistema propuesto

- El DFD debe modelar el sistema propuesto completo: ya sean procesos automatizados o manuales.
- Validar con el usuario
- Establecer la frontera hombre máquina: qué procesos se automatizarán y cuáles permanecerán manuales.
- Mostrar claramente la interacción entre los procesos manuales y los automáticos

Ejemplo

Una dueña de restaurante, quiere automatizar algunas partes del negocio. Entrevistas y cuestionarios. Recolectamos información en general.

Paso	Diagrama	Comentarios
Diagrama de contexto		**Partes involucradas**: clientes (dueña) y usuarios (mozos, operador de la caja). No todos los aspectos serán computables.
DFD del sistema existente		Se interactúa con el cliente. Pueden agregar nuevos aspectos.
DFD del sistema propuesto		Se genera el sistema propuesto.
Diccionario de datos del DFD del sistema		Formato de los datos, a través de expresiones regulares

Proceso de modelado

1. Identificar los **objetos** y las **clases** en el dominio del problema: los objetos se corresponden con sustantivos.
2. Identificar **estructuras**: herencia
3. Definir las clases identificando cuál es la información del estado que ésta encapsula (es decir, los **atributos**). Características. No agregar atributos innecesarios
4. Identificas **asociaciones**: hacer jerarquía de las clases. Las relaciones entre los objetos de las distintas clases, ya sea en la jerarquía o a través de llamadas a métodos.

5. Definir **servicios**: proveen el elemento activo en el modelado OO

- Pasos más significativos para el modelado orientado a objetos. Identificar objetos y clases

En estos pasos, puede haber "huecos" por ejemplo clases que todavía no se sabe cuales son los atributos.

Clase de modelados: 2a-modelados

Prototipado

Un prototipo de software se puede definir como una **implementación parcial** de un sistema cuyo propósito es **aprender** algo del problema a resolver o del enfoque de la solución.

El prototipado enfatiza que la experiencia práctica es la mejor ayuda para entender las necesidades. Se necesita una constante interacción con el cliente/usuario durante el prototipado para entender sus respuestas.

Enfoques:

- *Throwaway:* (Descartable) Se construye con la idea de que va a descartarse luego de que se complete el análisis.
- *Evolutionary:* (Evolutivo) Se construye con la idea de que eventualmente se va a convertir en el sistema final.

Para el análisis y entendimiento el prototipado descartable es más adecuado.

Los requerimientos se pueden dividir en tres grupos:

- Los entendidos.
- Parcialmente comprendidos: son los que deben ser incluidos en el prototipado.
  - Críticos en el diseño: En estos debería enfocarse el prototipo.
  - No críticos: Después pueden ser fácilmente incorporados en el sistema.
- Desconocidos.

Si el conjunto de requerimientos es substancial (en particular si se trata de los requerimientos críticos), entonces debería construirse un prototipo descartable.

Tipos de prototipado:

- *Vertical*: Una parte elegida del sistema, que no ha sido bien comprendida, se construye completamente.

- *Horizontal*: El sistema es organizado como una serie de capas y alguna capa es el foco del prototipo.

Para que sea posible hacer un prototipo en la etapa de análisis de requerimientos, el **costo** debe mantenerse **bajo**. Por eso solo se incluyen las características valiosas para la experiencia de usuario, solo se produce mínima documentación del desarrollo y se reduce el testing. Además, la eficiencia no es una preocupación en el prototipado, y usualmente se usan lenguajes interpretados de alto nivel para hacer prototipos.

Gracias al prototipado se pueden reducir substancialmente los errores de requerimientos y la cantidad de cambios de requerimientos.

Especificación de los requerimientos

Características de la SRS

Correcta:

Cada requerimiento representa precisamente alguna característica deseada por el cliente en el sistema final.

```
SRS -> representa -> cliente
```

Completa:

Todas las características deseadas por el cliente están descritas.

```
cliente -> determina -> SRS
```

No ambigua:

Cada requerimiento tiene exactamente un significado. Que no haya cosas subjetivas: "rápido" "lindo" . Los lenguajes formales ayudan a "desambiguar".

Consistente:

Que no se contradiga. Que toda la información esté ordenada ayuda a que sea consistente (es más fácil detectar inconsistencias).

Verificable:

Casa requerimiento es verificable (existe algún proceso efectivo que puede verificar que el software final satisface el requerimiento). Es esencial la No-ambigüedad, y que la SRS sea comprensible (debe poder ser revisada por el desarrollador, el usuario y el cliente).

Rastreable:

Se debe poder determinar el origen de cada requerimiento y cómo éste se relaciona a los elementos del software.

```
requerimiento está en SRS <=> elemento está en el producto final
```

=> Dado un requerimiento se debe poder detectar en qué elementos de diseño o código tiene impacto.
<= Dado un elemento de diseño o código se debe poder rastrear que requerimientos está atendiendo.

Modificable:

La estructura y estilo de la SRS es tal que permite incorporar cambios fácilmente preservando completitud y consistencia.
Debe suceder que no haya requerimientos en varios lugares (no haya redundancia).

Ordenada en aspectos de importancia y estabilidad:

Los requerimientos pueden ser *críticos*, *importantes pero no críticos*, *deseables pero no importantes*. Algunos requerimientos son *esenciales* y difícilmente cambien con el tiempo. Otros son propensos a cambiar.

Se necesita definir un orden de prioridades en la construcción para reducir riesgos debido a cambios de requerimientos.

Componentes de la SRS

Requerimientos sobre funcionalidad

- Conforma la mayor parte de la especificación
- Especifica toda la **funcionalidad** que el sistema debe proveer.*
- Especifica qué **salidas** deben producir para cada entrada dada y las relaciones entre ellas.
- Describe todas las **operaciones** que el sistema debe realizar.
- Describe las **entradas válidas** y las **verificaciones de validez** de la entrada y salida.*

* Sirve para *testing*

Especificación funcional con Casos de Uso

- Adecuado para sistemas interactivos.
- Consiste en especificar cada función provista por el sistema. (enfoque tradicional).
- Cada sistema tiene muchos casos de uso.
- Siempre usar reglas estrictas.
- Son útiles porque son muy entendibles para los usuarios y para los clientes.

- Permiten brainstorming con los clientes.

Formato de los casos de uso

- Caso de uso # `nombre del caso de uso`
- Actor primario: `nombre del AP`
- (Pre-condición: `descripción de la pre-condición`)
- (Ámbito: `subsistema al cual se aplica el caso de uso`)
- Escenario exitoso principal: `paso 1, ..., paso n`
- Escenarios excepcionales: `excepción 1, ..., excepción n`
- (Post-condiciones)

Caso de uso

- Los casos de uso capturan el comportamiento del sistema como **interacción** de los actores con el sistema.
- Un caso de uso es una colección de muchos escenarios.
- El nombre del caso de uso especifica el objetivo del actor primario.
- Pueden ordenarse jerárquicamente.
- Se enfocan en el comportamiento externo.
- La forma básica es textual. Existen notaciones gráficas (diagramas) como soporte.
- Los casos de uso **no** forman la SRS completa, sólo la parte funcional.
- Los casos de uso se enumeran para referencias posteriores.
- Se pueden organizan en jerarquías (subobjetivos)

Actor

Persona o **sistema** que interactúa con el sistema propuesto para alcanzar un objetivo.

Un actor es una **entidad lógica** y hay actores *receptores* y actores *transmisores* (y pueden ser mismo individuo).

- El **Actor Primario** es el actor principal que inicia el caso de uso. El caso de uso debe satisfacer su objetivo.

Escenario

- Es un conjunto de acciones realizadas con el fin de alcanzar un objetivo bajo determinadas condiciones.
- Se representan en secuencia pero no necesariamente esa es su implementación.

- La lista de acciones puede contener acciones que no son necesarios para el objetivo del actor primario. Sin embargo el sistema deba asegurar que todos los objetivos puedan cumplirse.
- Para cada punto actúa uno de los actores, en el siguiente debe ser otro actor el que realiza las acciones.
- No hay nada del manejo interno del sistema.
- Se pueden mencionar casos de uso de otra jerarquía, que estén especificados aparte.
- `paso_n = actor1 + acción1, ..., actor1 + acción_m`
- `paso_n+1 = actor2 + respuesta1, ...`
- El **escenario exitoso principal** sucede cuando todo funciona normalmente y **se alcanza el objetivo**. Mientras más exhaustivo, mejor. Porque estos casos serán usados como oráculo.
- Los **escenarios alternativos** (de extensión o de excepción) suceden cuando algo sale mal y **el objetivo no puede ser alcanzado**.
    - Las listas de excepciones no son exhaustivas: Solo se ponen los más básicos
    - Se ponen los que son errores específicos y necesitan un salida específica.
    - `escenario de error = error + respuesta`

Elaboración de los Casos de Uso

Pueden elaborarse haciendo refinamientos paso a paso. Se presentan cuatro niveles de abstracción:

1. **Actores y objetivos**
    i. Preparar una lista de actores y objetivos.
    ii. Proveer un breve resumen del caso de uso.
    iii. Evaluar completitud.
    - Esto define el ámbito del caso de uso
2. **Escenarios exitosos principales**
    i. Para cada caso de uso, expandir el escenario principal.
    ii. Revisar para asegurar que se satisface el interés de los participantes y actores.
3. **Condiciones de falla**
    - Siempre listar las que parezcan más importantes, la exhaustividad es casi nula.
    i. Por cada paso, identificar cómo y por qué puede fallar.

ii. Listar las posibles condiciones de falla para cada caso de uso.

4. **Manipulación de fallas**

    i. Especificar el comportamiento del sistema para cada condición de falla.
    - El realizar esta etapa emergerán nuevas situaciones y actores.

Se deben usar reglas de buena escritura técnica:

- Usar gramática/oraciones *simples* y *claras*.
- Especificar claramente todas las partes del caso de uso.
- Cuando sea necesario, combinar o dividir pasos.

Validación de los requerimientos

Métricas

Para poder **estimar costos y tiempos**, y planear el proyecto se necesita "medir" el esfuerzo que demandará. Una métrica es importante solo si es útil para el seguimiento o control de costos, calendario o calidad.

Punto función

Se determina solo con la SRS.
Define el tamaño en términos de la "funcionalidad".
Métrica en términos de LOC.

Tipo de funciones:

1. **Entradas externas:** Tipo de entrada (dato/control) externa a la aplicación.
2. **Salidas externas:** Tipo de salida que deja el sistema.
3. **Archivos lógicos internos:** Grupo lógico de dato/control de información generado/usado/manipulado.
4. **Archivos de interfaz externa:** Archivos pasados/compartidos entre aplicaciones.
5. **Transacciones externas:** Input/output inmediatos (queries)

Contar cada tipo de función diferenciado según sea *compleja*, *promedio* o *simple*. Y se pondera con un número `w_ij` .

`C_ij` determina la cantidad de funciones tipo "i" con complejidad "j"

Ajustar el UFP de acuerdo a la **complejidad del entorno**. Se evalúa según las siguientes características:

1. Comunicación de datos
2. Procesamiento distribuido
3. Objetivos de desempeño
4. Carga en la configuración de operación
5. Tasa de transacción
6. Ingreso de datos online
7. Eficiencia del usuario final
8. Actualización online
9. Complejidad del procesamiento lógico
10. Reusabilidad
11. Facilidad para la instalación
12. Facilidad para la operación
13. Múltiples sitios
14. Intención de facilitar cambios

Casa uno de estos items debe evaluarse como:

1. No presente: `p_i = 0`
2. Influencia insignificante: `p_i = 1`
3. Influencia moderada: `p_i = 2`
4. Influencia promedio `p_i = 3`
5. Influencia significativa `p_i = 4`
6. influencia fuerte: `p_i = 5

```
puntos_función = CAF * UFP;
// 1 punto función = 125 LOC en C
//                 =  50 LOC en C++ o Java
```

Arquitectura del software

Ya se empieza a detallar el *cómo* se va a resolver el problema Es la primera de las tres etapas de **diseño** y la de más alto nivel:

1. **Arquitectura**
2. Diseño Alto Nivel
3. Diseño Bajo Nivel

Da una vision de las partes del sistema y de las relaciones entre ellos. La arquitectura permite que los cambios sean simples de hacer en sistemas grandes y complejos. Permite *information hiding* y *encapsulación*.

El objetivo de la arquitectura es identificar subsistemas y la forma que interactúan entre ellos.

Definición: La *arquitectura de software* de un sistema es la estructura del sistema que comprende los elementos del software, las propiedades externamente visibles de tales elementos, y la relación entre ellas.

No son importantes los detalles de como se aseguran las propiedades externas. En general se tienen varias estructuras (aspectos ortogonales).

A este nivel se hacen las elecciones de tecnología, productos a utilizar, servidores, ... Es la etapa más temprana donde algunas propiedades como confiabilidad y desempeño pueden evaluarse. Aunque a nivel arquitectura se omiten muchos detalles.

¿Por qué es importante?

- **Comprensión y comunicación**: Muestra la *estructura de alto nivel* del sistema ocultando la complejidad de sus partes, lo cual define un marco de *comprensión común* entre los distintos interesados (usuarios, cliente, arquitecto, diseñador, ...).
- **Reuso**: Una de las principales técnicas para incrementar la *productividad*. Se elige una arquitectura tal que las *componentes existentes encajen* adecuadamente con otras componentes a desarrollar.
- **Construcción y evolución**: la división provista servirá para *guiar* el desarrollo y *asignar equipos de trabajo*. Durante la evolución del software, al arquitectura ayuda a dimensionar cuán caro puede costar un cambio.

- **Análisis**: La arquitectura permite predecir parcialmente propiedades de *confiabilidad y desempeño*. Se requerirá descripción precisa de la arquitectura así como de las propiedades de las componentes.

No hay una única arquitectura de un sistema.

Vistas de la arquitectura

Una vista consiste de **elementos** y **relaciones** entre ellos y **describe una estructura**. Destacan un aspecto. Por lo tanto, no hay una única vista de un sistema. Una vista que se enfoca en algún aspecto reduce la complejidad con la que debe enfrentarse el lector.

La mayoría de las vistas propuestas pertenece a alguno de estos tipos:

1. **Vista de módulos**: estructuras de código, planeamiento.
2. **Vista de Componentes y conectores**: estructuras de ejecución, análisis de desempeño.
3. **Vista de Asignación de recursos**: co-estructuras de software y entorno.

Existen relaciones entre los elementos de una vista y los de otra; y tal relación puede ser muy compleja.

Vista de módulos

Un sistema es una colección de **unidades de código** (no representan entidades en ejecución). Por ejemplo: clases, paquetes, funciones, procedimientos, métodos, etc.

La **relación** entre ellos está **basada en el código**. Por ejemplo: "parte de", "usa a", "depende de", llamadas, generalización o especialización, etc.

Vista de Asignación de Recursos

Se enfoca en *cómo* las unidades de software se asignan a recursos como hardware, sistemas de archivos, personas, etc.

Exponen propiedades estructurales como qué proceso ejecuta en qué procesador, qué archivo reside dónde, etc.

Vista de Componentes y conectores

Los elementos son **entidades de ejecución** denominados componentes. La vista C&C define las componentes y cómo se conectan entre ellas a través de conectores. Describe un estructura en ejecución del sistema: qué componentes existen y cómo interactúan entre ellos en **tiempo de ejecución**.

Componentes

Son **elementos computacionales** o **de almacenamiento** de datos. Por ejemplo: objetos, procesos, `.exe`, `.dll`, etc. Las componentes utilizan interfaces o puertos para comunicarse con otras componentes.

Conectores

Son mecanismo de **interacción** entre las componentes. Los conectores describen el **medio** en el cual la interacción entre componentes toma lugar.

Por ejemplo: pipes, sockets, memoria compartida, protocolos,llamada a procedimiento/función, puertos TCP/IP, RPC, etc.

Los conectores no necesariamente son binarios. Tienen nombre y tipo.

Estilos arquitectónicos para la vista de C&C

Define una familia de arquitecturas que satisfacen las restricciones de ese estilo. Distintos estilos pueden combinarse.

En general se usan diferentes combinaciones de estilos o se usan estilos nuevos.

Pipe and Filter

Adecuado para sistemas que fundamentalmente realizan *transformaciones de datos*. Un filtro realiza transformaciones y le pasa los datos a otro filtro a través de un tubo.

- Tipo de componente: filtro: es **independiente** y **asíncrona**, se limita a consumir y producir datos.
- Tipo de conector: pipe (tubo). Unidireccional y no tiene bifurcaciones. Cada tubo solo conecta 2 componentes.

Restricciones

- Cada filtro debe trabajar sin (necesariamente) conocer la identidad de los filtros productores o consumidores.
- Un tubo debe conectar un puerto de salida de un filtro a un puerto de entrada de otro filtro.
- Un sistema *puro* de *Pipe&Filter* usualmente requiere que cada filtro tenga su propio hilo de control.

Estilo de Datos Compartidos

- Componentes: **Repositorio** de datos y **usuario** de datos.
  - Repositorio de datos: provee **almacenamiento** permanente y confiable.

    - Usuarios de datos: acceden a los datos en el repositorio, **realizan cálculos** y ponen los resultados otra vez en el repositorio.
- Conector: un solo tipo, de **lectura/escritura**.

La comunicación entre los usuarios sólo se hace a través del repositorio, los componentes no actúan directamente entre ellas.

Estilo pizarra
Cuando se modifica el repositorio, se **informa** a todos los usuarios.

Estilo repositorio
Es pasivo.

Estilo cliente-servidor

- Componentes: **clientes** y **servidores**
    - Los clientes solo se comunican con el servidor, no con otros clientes.
    - La comunicación es siempre iniciada por el cliente y usualmente sincrónica.
- Conector: solicitud/respuesta (*request/reply*), es asimétrico.

Usualmente el cliente y el servidor residen en distintas máquinas.

Estructura multinivel

Nivel	Descripción
del cliente	Contiene a los clientes.
intermedio	Contiene las reglas del servicio.
de base de datos	reside la información.

Otros estilos

Publicar-suscribir

- Componentes: los que publican eventos y los que se suscriben a eventos.
- Se publica un evento → se invoca a las componentes suscritas a dicho evento.

Estilo peer-to-peer Componente: Peer (único tipo), pueden pedir servicios a otros peers. Parecido al modelo de computación orientada a objetos.

Estilo de procesos que se comunican Procesos que se comunican a través de mensajes.

Relación entre Arquitectura y Diseño

- La arquitectura **es** un diseño: se encuentra en el dominio de la solución y no en el del problema.
- Es un diseño de **muy alto nivel** que se enfoca en las componentes principales. La arquitectura no considera la estructura interna.
- La arquitectura impone restricciones sobre elecciones que pueden realizarse en otras fases del diseño y en la implementación.
- Para que la arquitectura tenga sentido, ésta debe *acompañar el diseño y el desarrollo del sistema*.

Evaluación de las arquitecturas

La arquitectura tiene **impacto** sobre los **atributos no funcionales** (como modificabilidad, desempeño, confiabilidad, portabilidad, etc). Por lo tanto se deben evaluar estas propiedades en la arquitectura propuesta.

Métodos para evaluar propiedades: Técnicas formales: redes de colas, model checkers, lenguajes de especificación, ... Otra posibilidad: metodologías rigurosas. Como el método de análisis ATAM.

Método de análisis ATAM

Architecture Trade off Analysis Method
Analiza las propiedades y las concesiones entre ellas.

Pasos principales

1. **Recolectar escenarios**:
   - Los escenarios *describen las interacciones del sistema*.
   - Elegir los escenarios de interés para el análisis (escenarios *críticos*).
   - Incluir escenarios excepcionales solo si son importantes
2. **Recolectar requerimientos y/o restricciones**:
   - Definir *lo que se espera del sistema* en tales escenarios.
   - Deben especificar los *niveles deseados* para los **atributos de interés** (preferiblemente cuantificados).
3. **Describir las vistas arquitectónicas**
   - Las vistas del sistema que serán evaluadas son recolectadas.
   - Distintas vistas pueden ser necesarias para distintos análisis.
4. **Análisis específicos a cada atributo**.
   - Se analizan las vistas bajo distintos escenarios separadamente para cada atributo de interés distinto; esto determina los niveles que la arquitectura puede proveer en cada atributo.

- Se comparan esos niveles con los requeridos: Esto forma la base para la elección entre una arquitectura u otra o la modificación de la arquitectura propuesta.
- Puede utilizarse cualquier técnica o modelado.

5. **Identificar puntos sensitivos y de compromisos**
   - Análisis de sensibilidad: cuál es el impacto que tiene un elemento sobre un atributo de calidad. Los elementos de mayor impacto son los puntos de sensibilidad.
   - Análisis de compromiso: Los puntos de compromiso son los elementos que son puntos de sensibilidad para varios atributos.

Diseño de alto nivel

- Se realiza luego de que los requerimientos estén definidos y antes de la implementación.
- Es el lenguaje intermedio entre los requerimientos y el código.
- Se comienzan a hacer representaciones más concretas.
- El resultado es un plano del sistema que **satisfaga los requerimientos** y que se utilizará para la implementación.
- Determina las mayores características de un sistema.
- Tiene un gran **impacto en testing** y **mantenimiento**.

Lo ideal es que sea **simple** y **entendible**.

Niveles en el proceso

1. Diseño arquitectónico:
    - Identifica las componentes necesarias del sistema, su comportamiento y relaciones.
    - Es más general que el diseño de alto nivel.
2. **Diseño de alto nivel**:
    - Es la vista de los módulos del sistema.
    - Cuáles son los módulos del sistema, qué deben hacer, y cómo se organizan/interconectan
    - Opciones:
        - Orientado a funciones
        - Orientado a objetos
3. Diseño detallado o diseño lógico:
    - Establece *cómo* se implementan las componentes de manera que satisfagan sus especificaciones.
    - Muy cercano al código: incluye detalles del procesamiento lógico (por ejemplo algoritmo) y de estructuras de datos.

Criterios para Evaluar el Diseño

Los criterios son usualmente subjetivos y no cuantificables. :(
Principales criterios para evaluar:

1. **Corrección**
    - ¿El diseño implementa todos los *requerimientos*?

    - ¿Es *factible* el diseño dada las restricciones?

2. **Eficiencia**

    - Apropiado *uso de los recursos* del sistema (principalmente CPU y memoria).

3. **Simplicidad**

    - Facilita la _comprensión_ del sistema. * Facilita el testing, modificación de código, mantenimiento, descubrimiento y corrección de bugs.

Eficiencia y simplicidad no son independientes => el diseñador debe encontrar un balance.

Principios de diseño

No hay una serie de pasos que permitan derivar el diseño a partir de los requerimientos. Pero existen "ayudas" que son **principios fundamentales** que guían el proceso de diseño:

Partición y jerarquía.

Consiste en dividir el problema en pequeñas partes, simplificando el diseño y facilitando el mantenimiento. Cada parte debe poder *solucionarse* y *modificarse* independientemente. Aunque no son totalmente independientes: deben **comunicarse** para solucionar el problema mayor.

La comunicación agrega complejidad: A medida que la cantidad de componentes aumenta, el *costo del particionado* + la *complejidad de la comunicación* también aumenta. Hay que detener el particionado cuando el costo supera al beneficio.

El particionado del problema determina una *jerarquía* de componentes en el diseño. Usualmente la jerarquía se forma a partir de la relación "es parte de".

La abstracción es esencial en el particionado del problema, pues `Jerarquía => Abstracción` aunque no necesariamente se cumple `Abstracción => Jerarquía`.

Abstracción

La abstracción de una componente describe el *comportamiento externo* sin dar detalles internos de cómo se produce dicho comportamiento. Es decir, trata de *ocultar* detalles de lo que pasa en niveles más bajos.

Abstracción durante el proceso de diseño:

- Para decidir como interactúan las componentes sólo el comportamiento externo es relevante.
- Permite concentrarse en una componente a la vez.

- Permite considerar una componente sin preocuparse por las otras.
- Permite que el diseñador controle la complejidad.
- Permite una transición gradual de lo más abstracto a lo más concreto.
- Necesaria para solucionar las partes separadamente.

Mecanismos comunes de abstracción

1. *Abstracción funcional*
    - Especifica el comportamiento funcional de un módulo.
    - Los módulos se tratan como funciones de entrada/salida.
    - Puede especificarse usando pre y post condiciones.
2. *Abstracción de datos*
    - Una entidad del mundo que provee servicios al entrono.
    - Los datos se tratan como objetos junto a sus operaciones (orientación a objetos). Y las operaciones definidas para un objetos solo pueden realizarse sobre este objeto.
    - Lenguajes que soportan abstracción de datos: Ada, C++, Modula, Java

Modularidad

Un sistema se dice modular si consiste de *componentes discretas* que se pueden *implementar separadamente*; donde un cambio a una de ellas tenga *mínimo impacto* sobre las otras.

- Prove la abstracción en el software.
- Es el soporte de la estructura jerárquica de los programas.
- Mejora claridad de diseño y facilita la implementación.
- Reduce costos de testing, debugging y mantenimiento.

Necesita *criterios de descomposición*: resulta de la conjunción de la *abstracción* y el *particionado*.

Diseño orientado a funciones

Módulos

Un módulo es una **parte lógicamente** separable de un programa. Es una unidad **discreta** e **identificable**.

Criterios para seleccionar módulos que soporten abstracciones bien definidas: acoplamiento y cohesión.

Acoplamiento

Definición

El acoplamiento es un concepto *inter-modular*, que determina la forma y nivel de *dependencia* entre módulos.

Dos módulos son independientes si cada uno puede funcionar completamente sin la presencia del otro.

El nivel de acoplamiento se define a nivel de diseño arquitectónico y de alto nivel. Y no puede reducirse durante la implementación.

Ventajas de la independencia

Los módulos se pueden implementar, modificar y testear separadamente.

No es necesario comprender todos los módulos para comprender uno en particular: Cuanto más conexiones hay entre dos módulos, más dependientes son uno del otro, es decir, se requiere más conocimiento de un módulo para comprender el otro.

Objetivo

Los módulos deben estar tan *débilmente acoplados* como sea posible.

En un sistema *no existe la independencia entre todos los módulos* pues deben cooperar entre sí. Pero queremos que la dependencia sea mínima.

Factores que influyen en el acoplamiento

1. **Tipo de conexiones entre módulos**: La *complejidad* y *oscuridad* de lasinterfaces. Ejemplo: ¿Utilizo solo las interfaces especificadas o también uso datos compartidos?
2. **Complejidad de las interfaces**: ¿Estoy pasando solo los parámetros necesarios? De todas maneras, cierto nivel de complejidad en las interfaces es necesario para soportar la comunicación requerida con el módulo.
3. **Tipo de flujo de información entre módulos**: ¿Estoy pasando parámetros de control (flags)? si se pasan flags que determinan el un caso de la función, es probable que se pueda dividir en dos funciones. Transferencia de información de control permite que las acciones de los módulos dependan de la información; y hace que los módulos sean más difíciles de comprender.

El acoplamiento disminuye si:

- Solo las entradas definidas en un módulo son utilizadas por los otros.
- La información se pasa exclusivamente a través de parámetros.
- Sólo se pasa la información estrictamente necesaria.
- Las interfaces solo contienen comunicación de datos.

El acoplamiento se incrementa si:

- Se utilizan interfaces indirectas y oscuras.
- Se usan directamente operaciones y atributos internos al módulo
- Se utilizan variables compartidas.
- Se pasan parámetros que contienen más información de la necesaria y hay que parsear (depende del formato).
- Las interfaces contienen comunicación de información híbrida (datos+control).

Cohesión

Definición

Es *intra-modular*. Tiene que ver con la relación de las componentes del mismo módulo. Y determina cuán fuertemente *vinculados* están los elementos de un módulo.

Consiste en minimizar las relaciones entre los elementos de los distintos módulos y maximizando las relaciones entre los elementos del mismo módulo.

Objetivo

Alta cohesión. Usualmente, a mayor cohesión de los módulos, menor acoplamiento.

Tipos de cohesión

1. **Casual**: La relación entre los elementos del módulo no tiene significado.
2. **Lógica**: Existe alguna relación lógica entre los elementos del módulo; los elementos realizan funciones dentro de la misma clase lógica.
3. **Temporal**: Los elementos están relacionados en el tiempo y se ejecutan juntos. Ejemplo: inicialización, clean-up, finalización.
4. **Procedural**: Contiene elementos que pertenecen a una misma unidad procedural. Ejemplo: un ciclo o secuencia de decisiones.
5. **Comunicacional**: Tiene elementos que están relacionados por una referencia al mismo datos. Ejemplo: pedir los datos de una cuenta personal y devolver todos los datos del registro.
6. **Secuencial**: Los elementos están juntos porque la salida de uno corresponde a la entrada del otro. Es relativamente buena cohesión y relativamente fácil de mantener, pero difícil de reusar.
7. **Funcional**: Todos los elementos del módulo están relacionados para llevar a cabo una sola función. Ejemplo: Calcular el seno de un ángulo.

Metodología del diseño estructurado

La estructura se decide durante el diseño y la implementación NO debe cambiar la estructura. La metodología de diseño estructurado (SDM: *Structured Design Method*) apunta a controlar la estructura y proveer pautas para auxiliar al diseñador en el proceso de diseño. SDM es una metodología orientada a funciones.

Pautas

- Los módulos *subordinados* son los que realizan la mayoría de la computación. El procesamiento real se realiza en los módulos *atómicos* del nivel más bajo.
- El módulo *principal* se encarga de la coordinación.
- La *factorización* es el proceso de *descomponer* un módulo de manera que el grueso de la computación se realice en módulos subordinados.

Pasos principales

1. Reformular el problema como un DFD
2. Identificar las entradas y salidas más abstractas
3. Realizar el primer nivel de factorización
4. Factorizar los módulos de entrada, de salida, y transformadores
5. Mejorar la estructura (heurísticas, análisis de transacciones

Reformular el problema como un DFD

Se toma el *flujo de datos* de todo el sistema propuesto. Y se plantea un DFD, que proveerá una visión de alto nivel del sistema.

Si bien *se ignoran aspectos procedurales* y la *notación es la misma*, el *propósito* del DFD es *diferente* al del de análisis de requerimientos. (No nos interesa entender el problema, sino empezar a desarrollar la solución).

Identificar las entradas y salidas abstractas

MAI (*Most Abstract Input*)

- La *MAI* consiste en la *entrada* y todas las *transformaciones* que se le aplican a la misma para que esté en un *formato adecuado*.
- Elementos de datos en el DFD que están más distantes de la entrada real, pero que aún puede considerarse como entrada.
- Podrían tener poca semejanza con la entrada real.

MAO (*Most Abstract Output*)

- Es dual a la *MAI*.
- Elementos de datos en el DFD que están más distantes de la salida real, pero que aún puede considerarse como salida.

Determinar las *MAI* o las *MAO* es subjetivo, es decir, representan un juicio de valor.

Las **transformaciones centrales** entre la *MAI* y la *MAO* es donde el sistema realmente "hace algo". Y estos se concentran en la transformación sin importar el formato/validación/etc de las entradas y salidas.

Entonces, el primer diseño básico consiste en:

MAI	→	**Sistema intermedio**	→	**MAO**
entrada	→	transformaciones	→	salida

Realizar el primer nivel de factorización

Hay que subdividir en:

1. *Modulo principal*: Módulo coordinador
2. *Módulos subordinados*
    i. De entrada: responsables de entregar las entradas lógicas
    ii. Transformadores: consumen las entradas lógicas y obtienen las salidas lógicas
    iii. De salida: Consumen las salidas lógicas
    - Cada uno de los tres tipos de módulos pueden diseñarse separadamente y son independientes.

Factorizar los módulos de entrada

- El transformador que produce el dato de MAI se trata ahora como un transformador central.
- Se repite el proceso del primer nivel de factorización considerando al módulo de entrada como si fuera el módulo principal.
- Usualmente no debería haber módulos de salida.

Factorizar los módulos de salida

- Módulos subordinados: transformador y módulos de salida.
- Usualmente no debería haber módulos de entrada.

Factorizar los transformadores centrales

- Utilizar un proceso de refinamiento top-down.

- El objetivo es determinar los sub-transformadores que compuestos conforman el transformador.
- Graficar DFD.
- Repetir hasta alcanzar los módulos atómicos.

Mejorar la estructura

Mejorar la estructura (heurísticas, análisis de transacciones)

Esta etapa consiste en hacer las *modificaciones finales*. Dada una serie de heurísticas se determina si la estructura está bien hecha. Y si no lo es, modificar.

El objetivo es tener el menor grado de acoplamiento y tener mucha cohesión en los módulos.

Verificación

Objetivo: Asegurar que el diseño implemente los requerimientos (corrección).
Hacer análisis de desempeño, eficiencia, etc.
Si se usan lenguajes formales para representar el diseño => existen *herramientas* que asisten para el análisis... Se usan también *listas de control*.

La calidad del diseño se completa con una buena modularidad (*bajo acoplamiento y alta cohesión*).

Métricas

Objetivo: proveer una evaluación cuantitativa del diseño (así el producto final puede mejorarse).

Se aplican dentro de UN MISMO proyecto y siempre se debe ser consistente.

1. **Tamaño estimado:** `cantidad de módulos + tamaño estimado de c/u`
2. **Complejidad de módulos:** cuáles son los que van a tomar más tiempo testear.
3. **Métricas de red:** enfocado en la estructura del diagrama.
   i. Se considera un buen diagrama aquel en el cual cada módulo tiene sólo un módulo invocador (reduce acoplamiento).
   ii. Impureza del grafo = `nodos_del_grafo - aristas_del_grafo - 1` si impureza = 0 tenemos un árbol.
4. **Métricas de estabilidad:** Trata de capturar el *impacto de los cambios* en el diseño. La estabilidad es la cantidad de suposiciones por otros módulos sobre uno específico.
5. **Métricas de flujo de información:** Computar la complejidad inter-módulo que se estima con inflow y outflow (el flujo de info que entra o sale del módulo).
   - $DC = tamaño * (inflow * outflow)^2$

- `DC = fan_in * fan_out + inflow * outflow` Donde fan_in representa la cantidad de módulos que llaman a C y fan_out los llamados por C.
- Propenso a error si `DC > complejidad_media + desviación_estándar`
- Complejo si `complejidad_media < DC < complejidad_media + desviación_estándar.`

Diseño orientado a objetos

El propósito del diseño OO es el de definir las clases del sistema a construir y las relaciones entre éstas.

Análisis OO y Diseño OO

Análisis OO: Primer paso (previo a la SRS). Existe métodos que combinan análisis y diseño (ADOO)

Análisis	Diseño
Dominio del problema	Dominio de la solución
El objetivo es entender el sistema	El objetivo es modelar una solución
Los objetos representan un concepto del problemas. Son objetos semámticos	* Produce objetos semánticos, de interfaces, de aplicaciones y de utilidad
	Hace incapié en el comportamiento **dinámico** del sistema

*

- Objetos de interfaces: se encargan de la interfaz con el usuario
- Objetos de aplicaciones: Especifican los mecanismos de control para la solución propuesta.
- Objetos de utilidad: Son los necesarios para soportar los servicios se los objetos semánticos (ejemplo: pilas, árboles, diccionarios, etc).

Herencia múltiple: puede generar conflictos. El polimorfismo es incluso más complejo. Mejor no usarlas a menos que sea MUY necesario.

- **Herencia estricta:** No se redefinen métodos. Solo se agregan características para especializarla. (Es suficiente con mantener el invariante de clase y las pre/post condiciones en las interfaces).
- **Herencia no estricta:** Se reescriben métodos. Es mala costumbre.

Conceptos de diseño

El diseño se puede evaluar usando:

- **Acoplamiento**
- **Cohesión**
- **Principio abierto-cerrado**

Acoplamiento

Tipos de acoplamiento

Interacción:

- Métodos de una clase *invocan* a métodos de otra clase.
- No se puede eliminar del todo.
- *Menor acoplamiento*: si se comunican a través de la menor cantidad de parámetros pasando menos información y nada de control.
- *Mayor acoplamiento*: Los métodos acceden a partes internas de otros métodos o variables. O la información se pasa a través de variables temporales.

Componentes:

- Una clase *A* tiene variables de otra clase *C*. Si *A* tiene variables de instancia, parámetros o métodos con variables locales de tipo *C*.
- Cuando *A* está acoplada con *C*, también está acoplada con todas sus subclase.
- Mejor Si las variables de la clase C en A son, o bien atributos o parámetros en un método. Es decir, son *visibles*.

Herencia:

- Si una es subclase de otra.
- Lo malo es si se modifican la *signatura* de un método o *eliminan* un método (herencia no estricta). O si cambia la pre y post condición (*especificación*).
- Menor acoplamiento si la subclase solo agrega variables de instancia y métodos pero no modifica los existentes en la superclase.

Cohesión

Tipos de cohesión

Cohesión de método

- Es mayor si cada *método* implementa una *única función* claramente definida con todos sus elementos contribuyendo a implementar esta función.
- Se debería poder escribir una oración simple de lo que hace un método. **Cohesión de clase**
- Es mayor si una *clase* representa un *único concepto* con todos sus elementos contribuyendo a este concepto.
- Se pueden detectar múltiples conceptos si los métodos se pueden separar en diversos grupos, cada grupo accediendo a distintos subconjuntos de atributos.

Cohesion de Clase:

FALTA COMPLETAR

Cohesión de Herencia:

- Es mayor si la jerarquía se produce como consecuencia de la *generalización-especificación*.
- Cohesión por reuso no esta tan bueno.

Principio abierto-cerrado

"Las entidades de software deben ser abiertas para extenderlas y cerradas para modificarlas."

El comportamiento puede extenderse para adaptar el sistema a nuevos requerimientos, pero el código existente no debería modificarse. Es decir, permitir *agregar* código pero no modificar el existente.

Este principio se satisface si se usa apropiadamente la herencia y el polimorfismo. La herencia permite crear una nueva (sub)clase para extender el comportamiento sin modificar la clase original.

Evitar la herencia no estricta.

Si se cumple el principio de Sustitución de Liskov se cumple el principio abierto-cerrado:

Principio de sustitución de Liskov

Un programa que utiliza un objeto O con clase C debería permanecer inalterado si O se reemplaza por cualquier objeto de una subclase de C.

En general `Liskov => abierto-cerrado .`

Metodología de diseño

Pasos de la metodología OMT (*Object modeling technique*). El punto de partida es el modelo obtenido durante el análisis OO. El producto final debe ser un plano de la implementación (incluyendo algoritmos y optimización).

1. **Producir el diagrama de clases**
    - Básicamente el diagrama obtenido en el análisis. Explica qué pasa en el sistema. *Estructura estática*.
2. **Producir el modelo dinámico y usarlo para definir operaciones de las clases.**
    - Describe la *interacción* entre objetos. *Estructura de control*.
    - Apunta a especificar cómo cambia el estado de los distintos objetos cuando ocurren un evento (solicitud de operación).
    - Los escenarios son la secuencia de eventos que ocurren en una ejecución particular del sistema; permiten identificar los eventos. Todos los escenarios juntos permiten caracterizar el comportamiento completo del sistema.
    - Para el diagrama de secuencia: empezar por escenarios iniciados por eventos externos → escenarios exitosos → escenarios excepcionales.
    - Una vez reconocidos los eventos de los objetos, se expande el diagrama de clases. En general, para cada evento en el diagrama de secuencia habrá una operación en el objeto sobre el cual el evento es invocado.
3. **Producir el modelo funcional y usarlo para definir operaciones de las clases**
    - Define la *transformación* de los datos. *Estructura de cómputo*.
    - Describe las operaciones que toman lugar en el sistema; y especifica cómo computar los valores de salida a partir de los valores de la entrada.
    - No considera los aspectos de control (usar DFD).
4. **Identificar las clases internas y sus operaciones**
    - Considera cuestiones de implementación.
    - Evaluar críticamente cada *clase* para ver si es necesaria en su forma actual.
    - Considerar luego las *implementaciones* de las operaciones de cada clase.
5. **Optimizar y empaquetar**
    - Agregar asociaciones redundantes (optimizar el acceso a datos).
    - Guardar atributos derivados (evitar cálculos complejos repetidos y asegurar consistencia).
    - Usar tipos genéricos (permite reusabilidad de código).
    - Ajustar la herencia (considerar subir en la jerarquía operaciones comunes, considerar la generación de clases abstractas para mejorar la reusabilidad).

Métricas

Sirven para repensar el diseño o identificar que componentes necesitarán más atención en testing.

Métodos pesados por clases (WMC)

La complejidad de la clase depende de la **cantidad de métodos en la misma y su complejidad**. `M1, ..., Mn` son métodos de `c` y `C(Mi)` su complejidad (ej.: la longitud estimada, complejidad de la

$$WMC = \sum_{i=1}^{n} C(M_i)$$

interfaz, complejidad del flujo de datos, etc) Si WMC es alto, la clase es más propensa a errores.

Profundidad del árbol de herencia (DIT)

Una clase muy por debajo en la jerarquía de clases puede heredar muchos métodos y dificulta la predicción de su comportamiento.

`mayor DIT => mayor probabilidad de errores`

Cantidad de hijos (NOC)

Cantidad de subclases inmediatas de *C*. `mayor NOC => mayor reuso  mayor NOC => mayor influencia => mayor importancia en la corrección del diseño de esta clase`

Acoplamiento entre clases (CBC)

Cantidad de clases a las cuales esta clase está acoplada. Dos clases están acopladas si los métodos de una usan métodos o atributos de la otra.

`menor CBC => mayor independencia => más modificable  mayor CBC => mayor probabilidad de error`

Respuesta para una clase (RFC)

RFC captura el grado de conexión de los métodos de una clase con otras clases. RFC de una clase C es la cantidad de métodos que pueden ser invocados como respuesta de un mensaje recibido por un objeto de la clase C.

Es probable que sea más difícil testear clases con RFC más alto. Muy significativo en la predicción de clases propensas a errores.