Programación Orientada a Objetos: Lenguajes, Metodología y Herramientas

Master de Computación

PROGRAMACION ORIENTADA A OBJETOS

J.M. Drake

Notas:

LA CRISIS DEL SOFTWARE.

- Conjunto de tópicos relacionados con la problemática asociada al desarrollo de software:

  "Construir una aplicación software es una tarea mucho más compleja de lo que parece al iniciarla"

- Aspectos de esta problemática son:
  - **Resposiveness:** No satisfacen las espectativas del usuario.
  - **Reliability:** Presentan fallos y su depuración es muy difícil.
  - **Cost:** El costo es difícil de evaluar y mas alto de lo esperado.
  - **Modificability:** Son productos muy rígido y difíciles de mantener.
  - **Timeless:** Requieren para su ejecución mas tiempo del previsto.
  - **Transportability:** Hay problemas para migrar entre plataforma.
  - **Efficiency:** Sólo utilizan una parte de la capacidad de hardware.

Notas:

Desde 1968 se identifica con la denominación **"crisis del software"** el conjunto de tópicos relacionados con la problemática asociada con el desarrollo de software. Aspectos identificados con esta crisis son:

* **Responsiveness:** Frecuentemente los sistemas basados en computador no satisfacen las expectativas que tiene el usuario.

* **Reliability:** Los programas suelen presentar fallos, y su depuración es muy difícil de garantizar.

* **Cost:** El costo del software es de evaluación difícil, y es habitual que resulte mas caro de lo que se preveía.

* **Modifiability:** Los programas son productos muy rígidos y de difícil modificación. El costo de su mantenimiento es muy alto.

***Timeless:** El desarrollo del software requiere siempre más tiempo que el previsto.

***Transportability:** Cuando se traslada un software de un equipo a otro, siempre se presentan problemas de adaptación.

***Efficiency:** Los programas utilizan solo una fracción pequeña de la capacidad del hardware en que se ejecuta.

El espíritu de la crisis del software se puede resumir en la frase: **"Construir una aplicación software es una tarea mucho más compleja de lo que de antemano parece"**

Un pequeño paquete con algún millar de líneas de código es es asequible mentalmente a un programador,sin embargo, un paquete mediano o grande con cientos de miles de líneas, desborda la capacidad de cualquier programador. En estos casos se hace crítico la problemática de la crisis del software.

CAUSAS DE LA CRISIS DELSOFTWARE.

- Causas profundas de la crisis del software son:
  - La metodología en cascada que linealiza el proceso de desarrollo.
  - La metodología de modularización estructurada hace que el software sea inflexible y difícil de mantener.
  - Programadores sin formación en ingeniería software.
  - Las empresas e instituciones tienen inercia a introducir las innovaciones.
  - La estructura secuencial de Von Newman no se adapta a los problemas que se abordan.

Notas:

Causas profundas que dan lugar a la crisis del software son:

* El paradigma de **diseño estructurado** de software es de naturaleza lineal, y es difícil evaluar a nivel de proyecto el efecto posterior de las decisiones que se toman en cada fase del ciclo de vida de un software.

* El conjunto de programadores que actualmente desarrollan software **no tienen una formación o no aplican la ingeniería software**.

* Las empresas y organizaciones que desarrollan software tienen una gran **inercia a introducir novedades** de eficacia demostrada.

* La **estructura secuencial de Von Newmann** y el estilo de programación que induce, no es el adecuado a los problemas que se abordan.

Complejidad del software

- El desarrollo de software es una tarea muy compleja.
- La complejidad del software es una característica esencial.
- La ingeniería software debe abordar la complejidad.
- Los componentes de la complejidad del software son:
  - Complejidad de los problemas que resuelve.
  - Naturaleza del proceso de desarrollo del software.
  - Libertad de la representación abstracta de los problemas.
  - La naturaleza discreta del software.

Notas:

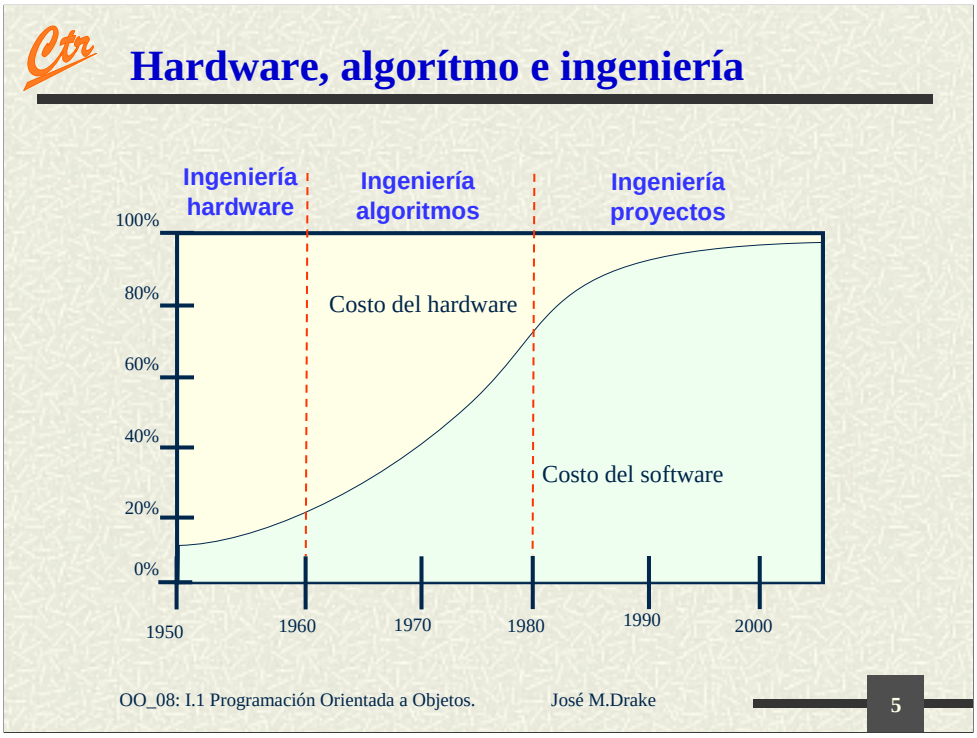
El análisis, especificación, diseño, codificación, verificación y validación de un programa software que resuelve un problema de la vida real es una tarea que requiere una considerable capacidad intelectual.

La complejidad del software es una característica inherente a la propia naturaleza del software y no es una característica accidental consecuencia de la forma en que se desarrolla.

Es mas rentable que la ingeniería software centre su esfuerzo en desarrollar metodologías con capacidad para abordar su complejidad inherente, que se oriente en buscar procedimientos que traten de reducirla.

La complejidad inherente del software tiene cuatro componentes básicas:

- La propia complejidad de los problemas que se abordan con la aplicación software.
- La naturaleza del proceso de desarrollo del software que se utiliza.
- La libertad que proporciona la formulación abstracta de los problemas concretos que se resuelven.
- La naturaleza discreta del software que proporciona respuestas cualitativamente diferentes por cambios cuantitativamente pequeños en el código o en los datos.


Hardware, algorítmo e ingeniería
Ingeniería hardware
Ingeniería algoritmos
Ingeniería proyectos
100%
80%
60%
40%
20%
0%
Costo del hardware
Costo del software
1950
1960
1970
1980
1990
2000
OO_08: I.1 Programación Orientada a Objetos.
José M.Drake
5

Notas:

Complejidad de los problemas

Las aplicaciones software actuales resuelven problemas muy complejos:

- Requieren enormes cantidades de componentes.
- Hay que dar coherencia a cientos de requerimientos.
- La especificación del problema es informal.
- Pueden requerir especificaciones no funcionales:
  - Facilidad de uso
  - Prestaciones temporales de respuesta.
  - Costo
  - Tiempo de vida útil.
  - Fiabilidad.
  - ....

Notas:

Los problemas concretos que actualmente se resuelven con aplicaciones software tienen una complejidad muy alta. Esta complejidad es consecuencia de:

•Requieren ensamblar miles o cientos de miles de componentes. La ingeniería software es la ingeniería que actualmente aborda los problemas con mayor número de componentes.

•Debe dar coherencia a cientos de requerimientos que frecuentemente compiten entre si y que incluso pueden presentar aspectos contradictorios.

•La formulación de sus requerimientos suele ser de naturaleza informal. Se utilizan muchas hojas de texto que describen con detalle el problema, pero habitualmente se desconoce la solución que se busca.

•A los programas se le requieren muchos requerimientos que son de naturaleza no funcional y que son difíciles de cuantificar, tales como:

- •Facilidad de uso.
- •Prestaciones de respuesta de naturaleza temporal.
- •Costo.
- •Tiempo de vida útil.
- •Seguridad de operación o fiabilidad.
- •Etc.

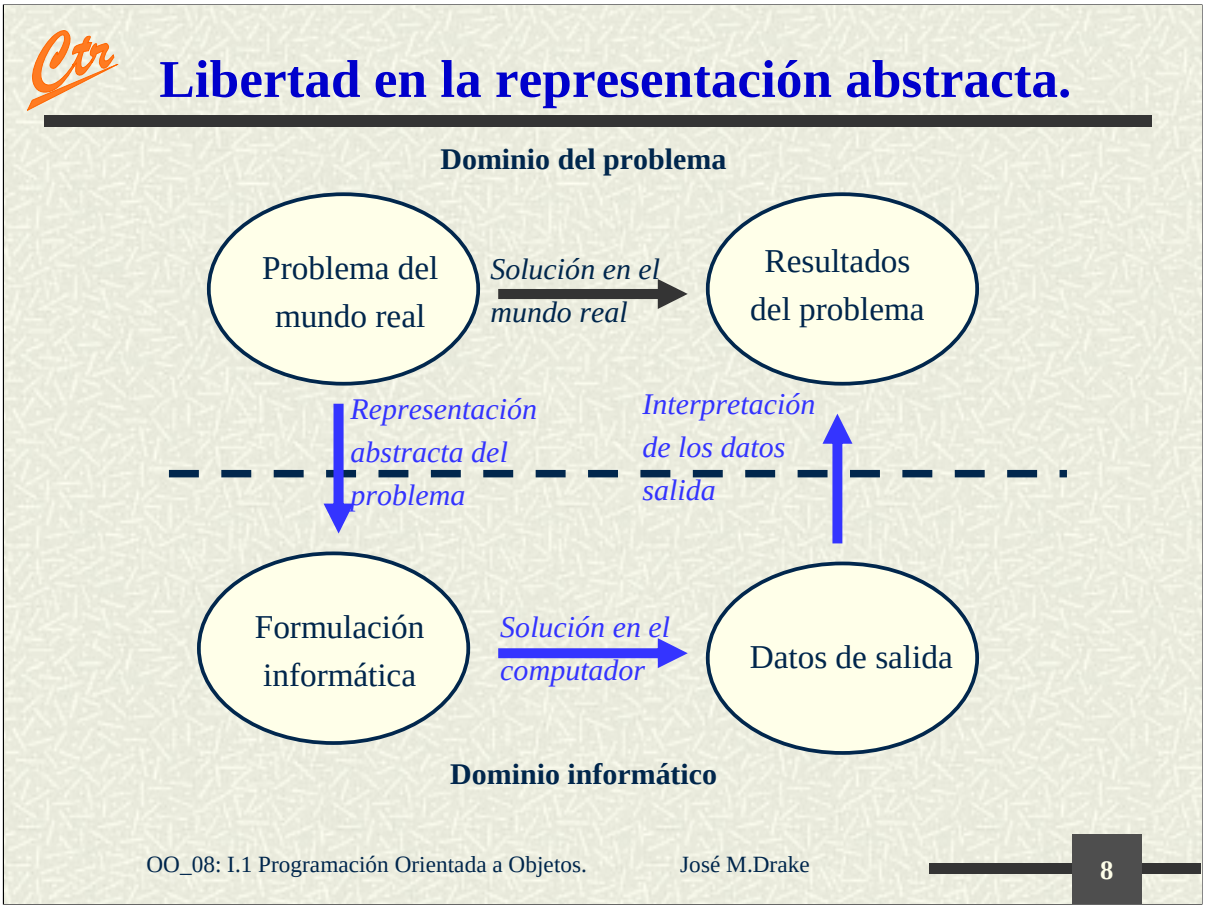
Complejidad por el proceso de desarrollo.

- Una aplicación software requiere manejar cientos de módulos y un programador puede manejar de forma ágil solo algunos componentes.
- La herramientas de nueva generación generan una falsa ilusión de simplicidad.
- Un proyecto software requiere ser desarrollado por un equipo, requiere especialmente métodos de ingeniería.
- No se ha generalizado estrategias productivas basadas en ensamblado de componentes.

Notas:

La metodología que se utiliza para desarrollar una aplicación software frecuentemente no es adecuada a la complejidad inherente que posee:

- Un proyecto software de mediana complejidad tiene decenas de miles de líneas de código y cientos de módulos diferentes que se deben concebir, diseñar y acoplar correctamente para que la aplicación funcione. Esto contrasta con la capacidad mental de un programador humano que solo es capaz de manejar de forma ágil y segura un número muy pequeño de componentes.
- La introducción de las metodologías software de últimas generación (lenguajes, sistemas operativos, generadores de código, etc.) generan la ilusión de simplicidad, pero la complejidad real continua dentro y bajo cualquier problema el programador queda impotente.
- Un proyecto software de mediana complejidad requiere la participación de un equipo de programadores y para que no se convierta en un caos se requiere que se apliquen sistemáticamente métodos de ingeniería de proyectos.
- En la ingeniería software no se ha generalizado el uso de estrategias de diseño de sistemas complejos utilizando componentes básicos estandarizados. Cada proyectos se aborda desde la nada, como si no tuviese nada en común con otros proyectos.



Notas:

La complejidad de los problemas que se resuelven es consecuencia del desacoplo entre el lenguaje y los conceptos del dominio del problema que se resuelve y los del dominio informático.

El cliente que plantea el problema y el ingeniero que lo resuelve mediante una aplicación software, manejan diferentes interpretaciones del problema:

- Tiene diferente visión de su naturaleza.
- Conciben soluciones diferentes.
- Intercambian entre ellos especificaciones muy complejas.

Pequeños cambios en las especificaciones del problema pueden requerir cambios muy importantes (cualitativamente y cuantitativamente) en la correspondiente formulación informática.

Naturaleza discreta del software.

- Una aplicación tiene un espacio de estados discreto y de una complejidad muy grande.
- Un evento inadecuado en su naturaleza o en el instante en que se produce puede cambiar drásticamente el comportamiento.
- Dada la complejidad del espacio de estados es imposible garantizar el correcto funcionamiento.
- La estrategia actual es incrementar la robustez de la aplicación introduciendo código que prevea la existencia de fallos.

Notas:

Una aplicación software contiene miles de variables y de líneas de control, lo que da lugar a un espacio de estados que es de naturaleza discreta y de una complejidad explosiva. Cada evento externo o cada ejecución de una sentencia de código hace cambiar el estado interno de la aplicación.

Un evento inadecuado en su naturaleza o en el instante en que se genera, y aunque sea en un aspecto nimio es potencialmente capaz de llevar a la aplicación a un estado en el que su comportamiento es drásticamente diferente del esperado.

En una aplicación de complejidad media es imposible garantizar que está libre de errores, y de existir alguno, no es posible prever sus consecuencias.

La estrategia con la que se aborda actualmente este problema es la introducción de código extra que sea capaz de cazar los fallos si se producen, y reconducir su efecto. Actualmente no es extraño que el volumen de código que se introduce para tratar los errores sea mayor que el que se introduce para conseguir la funcionalidad requerida.

Conclusiones sobre la complejidad del software.

- La complejidad es una característica esencial del software y es mas rentable dedicar los esfuerzos a gestionar la complejidad que a reducirla.

- La complejidad afecta de forma especialmente crítica en las fases de verificación y mantenimiento.

Notas:

Calidad del software

Las metodologías orientadas a objeto se introducen para mejorar la calidad del software.

- Los factores de calidad del software se clasifican en:
  - Factores de calidad externos: Son los que pueden ser estimados por el usuario.
  - Factores de calidad internos: Son los que solo pueden ser detectados por los programadores de desarrollo o de mantenimiento.
- Los factores de calidad externos son el objetivo final.
- Los factores de calidad internos son el medio de conseguir y garantizar los factores de calidad externos.

Notas:

Las metodologías orientadas a objetos tienen su origen en la investigación sobre los métodos conceptuales que deben introducirse para mejorar la calidad del software que se desarrolla.

Las características que debe tener un desarrollo software para que sea de calidad se pueden clasificar en dos grupos:

•Factores de calidad externos: son aquellas características cuya presencia o ausencia pueden ser detectadas por el usuario o por el experto en el dominio del problema.

•Factores de calidad internos: son aquellos que solo pueden ser detectados por los profesionales informáticos que desarrollan la aplicación o la mantienen.

Los factores de calidad externos constituyen el objetivo final que hay que satisfacer. Los factores de calidad internos son necesarios para poder garantizar que se van a satisfacer los factores externos.

Factores de calidad externa: Mejoradas por OO

Correctitud: Es la capacidad de una aplicación software para operar de acuerdo a como está definida en su especificación.

- **Robustez:** Es la capacidad de un producto software para dar una res-puesta satisfactoria bajo situaciones no establecidas en la especificación.
- **Extensibilidad:** Es la facilidad que presenta un producto software para ser adaptado tras un cambio de las especificaciones.
- **Reusabilidad:** Es la adecuación de un producto software para poder ser reusado en otras aplicaciones para las que no fue diseñado.
- **Compatibilidad:** Es la facilidad que presenta un producto software para poder ser combinado con otros.

Notas:

CORRECTITUD (Correctness): Es el factor de calidad externo fundamental. Es un factor muy fácil de definir, pero solo se puede verificar si las especificaciones son exhaustivas.

ROBUSTEZ (Robustness): La respuesta de la aplicación fuera de la especificación no es de interés, pero la aplicación debe dar respuestas correctas cuando se retorna a condiciones especificadas. El concepto de robuztez es difuso, básicamente se suele requerir que nunca se alcancen estados catastróficos.

EXTENSIBILIDAD (Extendebility): Aparentemente es consustancial con el concepto de software, pero en los proyectos grandes no es posible estimar los efectos colaterales de los cambios. Las dos bases de la extensibilidad son: La simplicidad del diseño y el diseño descentralizado.

REUSABILIDAD (Reusability): La reusabilidad es el objetivo esencial de la industria del software. Aparentemente debería alcanzarse ya que las operaciones básicas de software son muy pocas.

COMPATIBILIDAD (Compatibility): Las claves para conseguir la compatibilidad son: la homogeneidad y la estandarización.

Factores de calidad externa: No mejoradas por OO

- **Eficiencia:** Es la capacidad de un producto software de hacer uso completo de los recursos que le proporciona la plataforma.
- **Portabilidad:** Es la capacidad de un producto software para ser transferido a diferentes plataformas hardware y entornos software.
- **Verificabilidad:** Es la facilidad que proporciona un software para detectar y trazar fallos durante las fases de validación y operativa.
- **Integridad:** Es la capacidad que ofrece un producto software para proteger su recursos contra accesos y modificaciones no autorizadas.
- **Facilidad de uso:** Es la facilidad que presenta el producto software para ser manejada e interpretar los resultados por el operador humano.

Notas:

EFICIENCIA (Efficency): Es la capacidad de un producto software de hacer un uso completo de los recursos que le proporciona la plataforma sobre la que se ejecuta.

PORTABILIDAD (Portability) : Es la capacidad de un producto software para ser transferido a diferentes plataformas hardware y entornos software.

VERIFICABILIDAD (Verfiability): Es la facilidad que proporciona un software para detectar y trazar fallos durante las fases de validación y operativa.

INTEGRIDAD (Integrity): Es la capacidad que ofrece un producto software para proteger su recursos (código, datos, ficheros, etc.) contra accesos y modificaciones no autorizadas.

FACILIDAD DE USO (Ease of use): Es la facilidad que presenta el producto software para ser utilizada, ser gestionada, preparar la información de entrada, interpretar los resultados, etc.

Balance entre factores de calidad externos (tradeoffs)

Los factores de calidad no siempre son compatibles entre sí.

Ejemplos:

- Incrementos de la integridad reducen la facilidad de uso.
- Incrementos de la eficiencia degrada la portabilidad.

- En la especificación de la aplicación se debe establecer un balance ponderado de los factores de calidad que se requieren.

Notas:

Los factores externos de calidad que se han descrito no son necesariamente compatibles entre sí:

•Si se incrementa la integridad del sistema, deben introducirse barreras de control que arbitren el acceso a la aplicación, y esto obviamente, degradará su facilidad de uso.

•Si se incrementa la eficiencia del programa, se debe adaptar el programa a los recursos hardware disponibles, y esto normalmente supone una degradación de la portabilidad, extensibilidad y reusabilidad.

En estos casos, en la especificación del problema se debe establecer la importancia que se asigna a los diferentes factores de calidad. En el diseño se debe plantear un balance ponderado entre los factores de calidad que se requieren.

Diseño modular del software

- Las **herramientas** básicas para **abordar la complejidad** son:
  - **Modularización:** Capacidad para descomponer los componentes complejos en otros mas simples.
  - **Abstracción:** Capacidad de reducir la información de un componente a la necesaria para manejarlo en un nivel de desarrollo.
  - **Herencia:** Capacidad de jerarquizar los componentes del dominio de acuerdo con las características comunes que presentan.
- Son las bases de la estrategia que da lugar a la **metodología orientada a objetos** de desarrollo de software .

Notas:

Las dos herramientas básicas que utiliza el hombre para abordar sistemas muy complejos es la modularización y la abstracción.

La modularización consiste en descomponer un componente complejo en un conjunto reducido de subcomponentes mas sencillos. La estrategia de modularización debe iterarse hasta que las complejidad de los módulos que resultan sea abordables por el programador.

La abstracción es la capacidad reducir la información que describe un componente a la justa necesaria para manejarlo dentro de la fase de desarrollo en que se está.

La herencia es la capacidad de agrupar jerárquicamente los componentes de forma que aquellas características que tengan en común muchos de ellos solo se necesite describir una vez a través de la descripción del correspondiente anctecesor común.

Criterios de modularización.

- **Descomponibilidad modular**: Permita descomponer sucesivamente cada módulo en otros mas simples que puedan ser abordados por sí.
- **Componibilidad modular**: Genere módulos que puedan ser libremente combinados para generar módulos mas complejos.
- **Compresibilidad modular**: Cada módulo que se genera puede ser descrito y comprendido por sí y con independencia de otros.
- **Continuidad modular**: La descomposición debe ser tal que pequeñas modificaciones de las especificación del problema induzca cambios en poco módulos y en proporción de las modificaciones introducidas.
- **Protección modular**: Los errores que se produzcan en un módulo queden confinados y puedan tratarse en él.

Notas:

Descomponibilidad modular: El criterio requiere, que los módulos resultantes sean consistentes por sí y puedan ser tratados y manejados de forma independiente por diferentes diseñadores. El método de diseño top-down es el mas utilizado. El diseñador comienza con la descripción mas general del sistema, y luego la refina paso a paso, descomponiéndolo en nuevos subsistemas de complejidad mas simple.

Componibilidad modular: Este criterio tiene como finalidad la reusabilidad. Se busca que el trabajo desarrollado en un proyecto pueda ser aprovechado en los siguientes proyectos. La componibilidad y descomponibilidad son aspectos independientes ya veces, estan enfrentados. Por ejemplo: Los módulos generados utilizando la estrategia top-down y obtenidos para una aplicación específica suelen ser no componibles, ni reusables.

Compresibilidad modular: Este criterio es esencial para las fases de mantenimiento. En un sistema complejo, se necesita poder comprender, analizar y modificar un módulo individual, sin necesitar conocer al resto de módulos.

Continuidad modular: Es importante para la extensibilidad y el mantenimiento. Las estructura debe ser tal que una tareas simple de mantenimiento no pueda conducir a situaciones de colapso catastrófico.

Protección modular: Este criterio acepta la existencia de los fallos de código, de plataforma de falta de recursos o de datos no previstos. Busca una estructura modular que haga viable y sencilla la recuperación de los errores imprevistos.

Recursos para la modularización

- **Unidad lingüística:** Se necesita disponer de una unidad sintáctica que permita formular el ente individualizado.
- **Minimización del número de interfaces:** Cada módulo debe comunicarse con el mínimo numero de otros módulos.
- **Acoplamiento débil**: Debe minimizarse la cantidad de información que se intercambia entre módulos.
- **Interfaces explícitas**: Debe declararse explícitamente los mecanismos de comunicación y los tipos de información que se intercambia.
- **Ocultación de información**: Toda la información que gestiona un módulo debe ser privada, salvo la que se intercambia que se declara explícitamente pública.

Notas:

Unidad lingüística: Para conseguir Descomponibilidad, Componibilidad y Protección modular en la modularización de una aplicación se necesita formular el módulo como una unidad sintáctica.

Minimización del número de interfaces: Para conseguir la Continuidad y Protección modular se deben minimizar las interdependencia entre los módulos.

Acoplamiento débil: El intercambio entre módulos de solo los datos que son estrictamente necesarios, reduce la necesidad de revisión ante cambios (Continuidad modular) y la transmisión de condiciones anómalas y de excepción entre ellos (Protección modular).

Interfaces explicitas: Las posibilidades de interconexión entre módulos resulta normalizada (Componibili- dad) y completamente documentada (Comprensibidades).

Ocultación de información: Independiza la funcionalidad de la implementación del módulo lo que facilita la Componibilidad, Comprensibilidad y Continuidad modular.

Conclusiones sobre modularización

- La modularización, la abstracción y la herencia son herramientas para abordar la complejidad.
- Una modularización eficiente requiere que se generen módulos que satisfagan:
  - Representen la abstracción de algo útil, relevante, fácil de conceptualizar y con una funcionalidad bien definida.
  - Tengan una entidad propia e independiente del resto del sistema.
  - De lugar a una estructura del sistema que sea simple, natural y poco sensible a los cambios evolutivos.
  - Permita reducir la complejidad jerarquizando los módulos de forma que cada característica solo se describa una sola vez.

Notas:

La modularización, la abstracción y la herencia son las principales herramienta que se disponen para abordar la complejidad.

El análisis de un problema o el diseño de la aplicación software que lo resuelve debe llevarse a cabo mediante una eficiente modularización, lo cual requiere que el criterio que se utilice conduzca a la generación de módulos auténticos, esto es,

•Representen la abstracción de algo significativo y útil: y como consecuencia fácil de conceptualizar y de describir mediante una interfaz sencilla.

•Tengan una entidad propia e independiente del resto del sistema: y que por ello, se puedan diseñar, implementar, probar y utilizar de forma independiente y descentralizada.

•De lugar a una estructura del sistema que sea simple, natural y poco sensible a los cambios evolutivos.

•Permita organizar los componentes de forma que no se repita la información común.

Métodos orientados a objetos

Los métodos orientados a objetos son técnicas de abordar la complejidad del problema que utilizan como procedimiento de modularización la identificación de los objetos del dominio del problema.

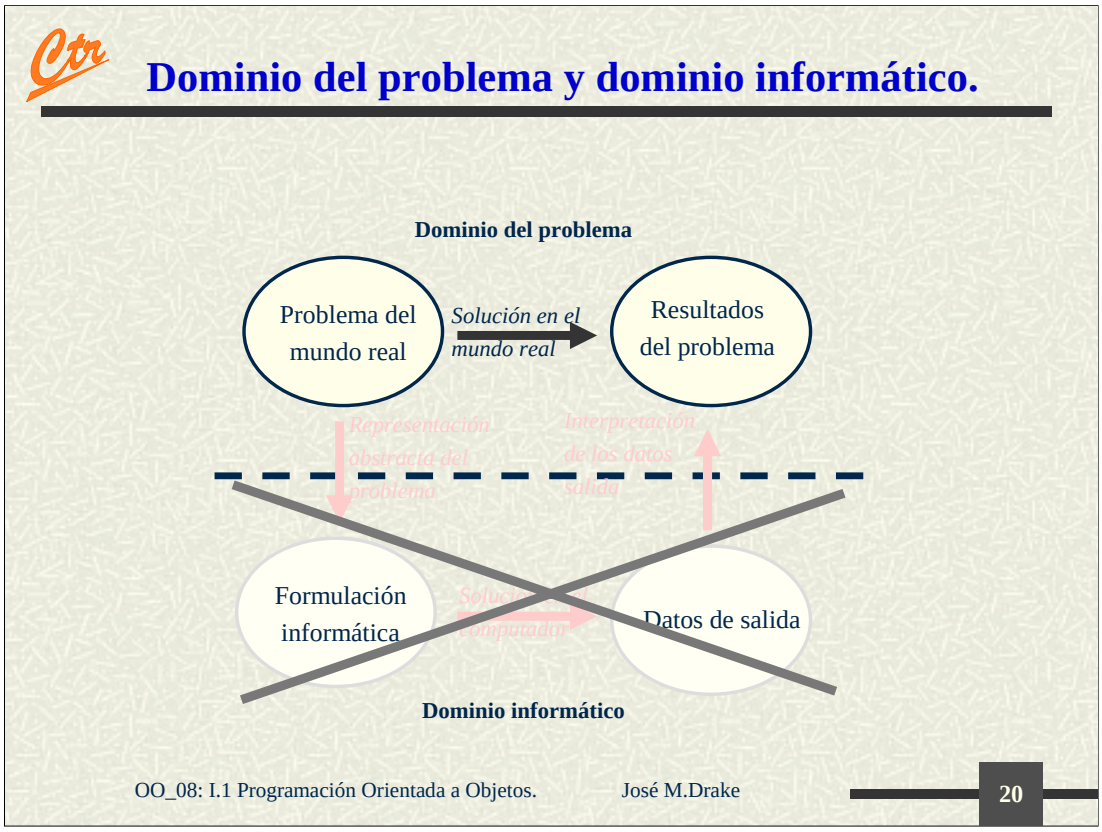
Cada módulo que se define corresponde a la abstracción de un objeto existente en el dominio del problema y en él se incluyen todos los aspectos (funcionalidad, estados, datos, etc.) que son propios del objeto.

Notas:

La descomposición del problema se realiza utilizando los criterios propios del dominio del problema, y no en función de criterios de representación informática.

En la práctica diaria, los sistemas se conciben como compuestos de objetos interrelacionados. La forma en que describimos y comprendemos un sistema se realiza describiendo de forma independiente los objetos y luego describiendo el sistema en función de ellos y de las relaciones que entre ellos se establecen.

Como el número de objetos puede ser muy grande y sus características muy específicas, se trata de describir los objetos jerarquizados según ciertos tipos abstractos. Esto reduce la necesidad de repetir descripciones muy semejantes, y permite reutilizar las descripciones de los objetos en muchos sistemas diferentes.



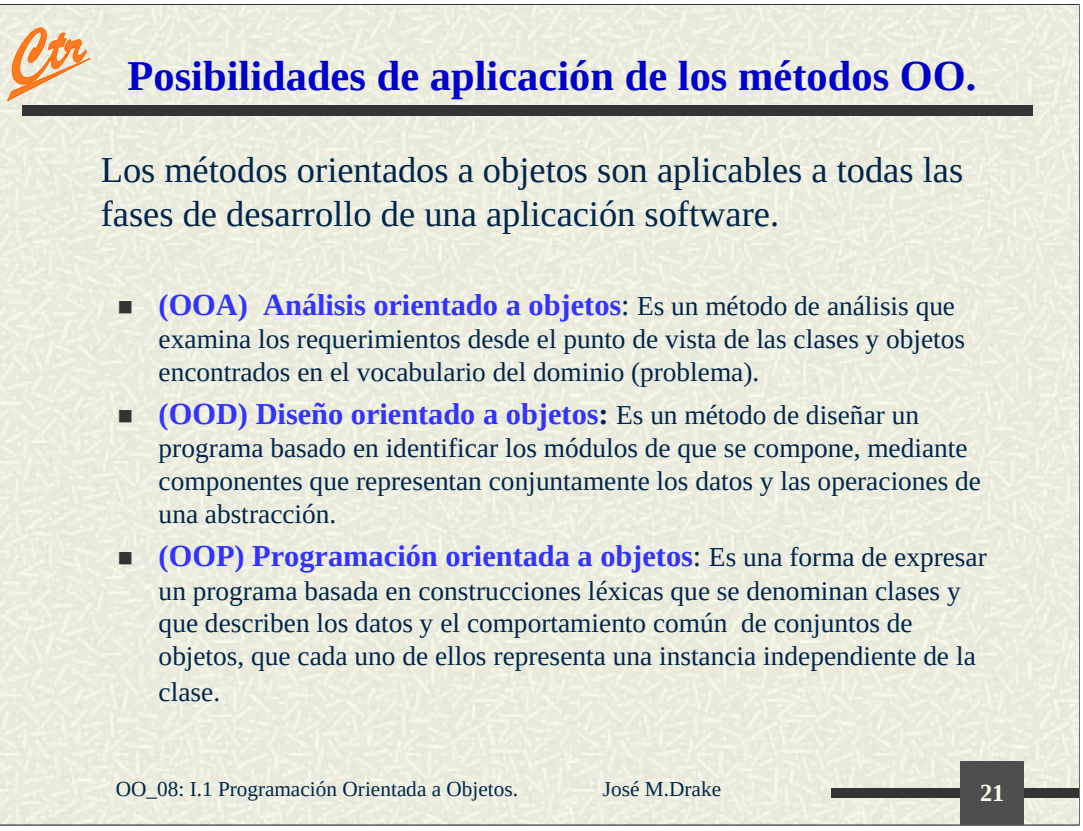
Notas:

La complejidad de los problemas que se resuelven es consecuencia del desacoplo entre el lenguaje y los conceptos del dominio del problema que se resuelve y los del dominio informático.

El cliente que plantea el problema y el ingeniero que lo resuelve mediante una aplicación software, manejan diferentes interpretaciones del problema:

- Tiene diferente visión de su naturaleza.
- Conciben soluciones diferentes.
- Intercambian entre ellos especificaciones muy complejas.

Pequeños cambios en las especificaciones del problema pueden requerir cambios muy importantes (cualitativamente y cuantitativamente) en la correspondiente formulación informática.



Notas:

Los métodos orientados a objetos son útiles en todas las fases del desarrollo del software:

Análisis orientado a Objetos (OOA): Es un método de análisis identifica y describe los requerimientos de un problema en función de los objetos que intervienen y en las relaciones de asociación e interacción que existen entre ellos.

Diseño Orientado a Objetos (OOD) : Es un método de diseño del software de una aplicación basada en construirla con una estructura modular en la que los módulos software se corresponden con abstracciones de los objetos del problema.

Programación Orientada a Objetos (OOP): Es un método de generación del código de una aplicación en el que las unidades lingüísticas que se utilizan corresponden a código que representan a los objetos del problema.

El método OO puede utilizarse indiferentemente en una fase independientemente o en las tres fases de forma coordinada.

Ventajas de la modularización OO.

- **Descomponibilidad** : Se consigue de forma natural al problema.
- **Componibilidad y Reusabilidad**: Los objetos son reales y se comparten con cualquier otra aplicación del mismo dominio.
- **Comprensibilidad y consistencia con el dominio del problema**: La función y abstracción de un objeto es obvia ya que coincide con la del objeto del dominio real que representa.
- **Continuidad y estabilidad frente a cambios**: La evolución de una aplicación es consecuencia de cambios en los objetos que la componen, y solo afecta a un módulo.
- **Robustez y protección frente a fallos**: Cada objeto se implementa como un ente independiente y los estados de excepción que se puedan generar, pueden ser tratados dentro del propio objeto.
- **Soporte inherente de la concurrencia**: La concurrencia propia de los dominios reales se transfiere de forma natural a la aplicación.
- **Escalabilidad**: La complejidad de la aplicación crece linealmente con la complejidad del problema que se aborda.

Notas:

Con este criterio se satisfacen de forma natural todos los criterios de modularización antes expuestos:

•Descomponibilidad modular: Se consigue descomponer de forma natural al problema. La descomposición de la aplicación coincide con la descomposición que resulta del análisis del dominio

•Componibilidad modular y reusabilidad: Los objetos de un problema son reales y y aparecerán en otros problemas del mismo dominio.

•Comprensibilidad modular y consistencia con el dominio del problema: La funcionalidad y abstracción de un módulo es obvia ya que coincide con la del objeto del dominio real que representa.

•Continuidad modular y estabilidad frente a cambios: Las evolución de un problema es consecuencia de cambios pequeños en los objetos que lo componen, y en consecuencia afecta solo al módulo que se representa.

•Robustez y protección frente a fallos: La mayoría de las situaciones anormales que pueden conducir a fallos, son consecuencias de agente externos, y por tanto se pueden definir y tratar a nivel del módulo que corresponde al objeto responsable.

•Soporte inherente de la concurrencia: Los objetos que resultan del análisis de un problema son concurrentes como lo es habitualmente el dominio al que corresponde y pueden transferirse a los objetos que constituyen la aplicación.

•Escalabilidad: Cuando crece la complejidad del problema, no crece patológicamente la complejidad de la aplicación.

Comparación con los métodos estructurados

- En los métodos estructurados el criterio de modularización es la funcionalidad.

¿Qué hace el sistema?

- En los métodos orientados a objetos el criterio de modularización son los componentes que componen el sistema.

¿Qué componentes tiene el sistema?

Notas:

En los métodos estructurados el criterio de modularización es la funcionalidad. La pregunta básica que se realiza en este caso, es:

¿Que hace el sistema?

como respuesta a esta pregunta aparecen las acciones que se identifican con los módulos que son de naturaleza s funcional. La estrategia es típicamente Top-down. Se parte del programa principal y luego se modulariza.

En los métodos orientados a objetos, el criterio de modularización son los componentes del problema. La pregunta básica que se realiza es:

¿Que componentes tiene el problema?

Como respuesta se identifican los objetos que intervienen en el problema y la forma que interactuan entre sí. Como consecuencia de su naturaleza y de las interacciones que se producen, resulta la funcionalidad. El programa principal, si existe, es lo último que se aborda.

Claves de las metodología orientadas a objetos.

- **Abstracción**: Busca una definición conceptual común a muchos objetos, tratando de identificar sus características esenciales y agrupándolos por clases.
- **Encapsulación**: define de forma independiente la abstracción o interfaz y su implementación y estructura interna.
- **Modularidad**: Describe el sistema como conjunto de módulos(objetos) descentralizados y débilmente acoplados.
- **Herencia**: Jerarquiza las clases de acuerdo con afinidades de sus abstracciones.
- **Tipado**: Clasifica los objetos de forma estricta, restringiendo las interacciones a solo aquellas que son coherentes.
- **Concurrencia**: Enfatiza la naturaleza independiente de cada objeto, adjudicándole si procede líneas de control de flujo independientes.
- **Persistencia**: Enfatiza la naturaleza independiente de los objetos, adjudicándole una existencia que sobrepasa a quien lo creó.

Notas:

•**Abstracción:** Trata de identificar las características esenciales de los objetos para clasificarlos por clases, y buscarle una definición conceptual común independiente de los detalles.

•**Encapsulación**: Trata de describir de forma separada la abstracción o interfaz que describe su funcionalidad y su capacidad de interacción con otros objetos que es pública, y su implementación y estructura interna que es privada.

•**Modularidad**: Trata de describir al sistema como conjuntos de módulos (objetos) descentralizados y débilmente acoplados.

•**Herencia**: Trata de jerarquizar las clases de objetos de acuerdo con la afinidades de sus abstracciones, a través de relaciones de relaciones de generalización y particularización.

•**Tipado**: Clasifica las características de los objetos de forma estricta, de forma que restringe las interacciones entre los objetos, a solo aquellas que son coherentes.

•**Concurrencia**: Enfatiza la naturaleza independiente de cada objeto, adjudicándole si procede líneas de control de flujo propio, lo que conduce a una actividad concurrente del sistema imagen de la del problema real.

•**Persistencia**: Enfatiza la naturaleza independiente de los objetos, adjudicándole una existencia que puede sobrepasar en el tiempo o en el espacio la del objeto que lo creó.