Tema 2: Diseño en construcción

Docente : Héctor Xavier Limón Riaño
email: xavier120@hotmail.com

Introducción

Introducción

- Es posible argumentar que el diseño no es parte de la construcción
- En proyectos pequeños, muchas actividades son pensadas como construcción cuando en realidad son de diseño, análisis, entre otras
- En proyectos grandes, el diseño suele ser muy detallado, lo que permite que la codificación sea algo mecánico
- Sin embargo, rara vez el diseño es totalmente completo, el programador puede tomarse libertades para diseñar subrutinas

Introducción

- Incluso en un proyecto pequeño e informal puede existir diseño en forma de pseudo-código o diagramas informales
- Sin importar como está hecho, tanto proyectos grandes y pequeños se benefician del diseño
- Reconocer al diseño como una actividad explícita maximiza su beneficio

Introducción

- El diseño es una actividad muy extensa, podría hablarse de ello durante varios cursos
- En el curso sólo se verán algunos aspectos de él referentes a construcción

Retos de diseño

Introducción

- La frase "Diseño de Software" significa la concepción, invención, o estratagema para convertir la especificación de un programa de computadora en un programa operacional
- El diseño es una actividad que liga los requerimientos al código y depuración
- Un buen diseño de alto nivel provee una estructura que fácilmente puede contener múltiples sub-niveles de diseño
- El buen diseño es útil en proyectos pequeños e indispensable en grandes

El diseño es un problema perverso

- Horst Rittel and Melvin Webber definen como perverso aquel problema que sólo puede ser definido claramente resolviéndolo, o resolviéndolo en parte
- En un ámbito académico rara vez los estudiantes se topan con un problema de este tipo
- Imaginen que se les encarga un proyecto y tan pronto terminan el diseño se les cambia los requerimientos, luego, justo antes de que entreguen el proyecto completo se vuelven a cambiar los requerimientos. Este es el día a día en el desarrollo profesional de software

El diseño es un proceso torpe

- El diseño de software final debe verse bien organizado y limpio, pero el proceso para llegar desarrollar el diseño no es tan bonito y organizado como el producto final
- El diseño es torpe debido a que se toman muchos pasos en falso, rutas a ciegas y se cometen errores.
- Cometer errores es el punto de diseñar, de esta forma pueden corregirse en una etapa temprana
- Es torpe porque a menudo la diferencia entre una solución buena y una mala es pequeña
- También es torpe porque es difícil determinar en que momento el diseño es suficientemente bueno (normalmente la respuesta a esta pregunta es: Cuando ya se te acabó el tiempo)

El diseño se trata sobre balanceos y prioridades

- En un mundo ideal todos los aspectos del sistema son importantes y pueden optimizarse hasta la perfección
- En un sistema del mundo real, es importante que el diseñador balancee diversos aspectos de diseño
- Por ejemplo: es posible que el sistema tenga como prioridad un tiempo de respuesta rápido, más aun que el tiempo de desarrollo. En ese caso el diseñador adaptará el diseño a cumplir ese requerimiento. Si por el contrario lo más importante es minimizar el tiempo de desarrollo, un buen diseñador creara un diseño diferente

El diseño involucra restricciones

- EL punto de diseñar es crear posibilidades y al mismo tiempo restringir posibilidades
- Los recursos limitados fuerzan la realización de simplificaciones en la solución
- Esto ayuda a mejorar la solución al ser necesariamente menos rebuscada

El diseño es no deterministico

- Si tres personas diferentes hacen un diseño para el mismo sistema, es altamente probable que los tres tendrán diseños muy diferentes, pudiendo ser aceptables los tres
- Usualmente hay muchas maneras de diseñar programas computacionales

El diseño es un proceso heurístico

- Como el diseño es no deterministico, las técnicas de diseño tienden a ser heurísticas, reglas prácticas, o simplemente algo que puede aplicarse y que a veces funciona
- Las técnicas de diseño no son procesos que garanticen resultados predecibles y satisfactorios
- El diseño involucra prueba y error
- Una técnica que funciono una vez en un proyecto no necesariamente funcionara en otro
- Ninguna herramienta es adecuada para todo

El diseño es emergente

- El diseño no aparece espontáneamente de forma completa en la mente de alguien
- El diseño evoluciona y mejora a través de revisiones de diseño, discusiones informales y a través de escribir y revisar el propio código.
- Prácticamente cualquier sistema pasa por algún grado de cambios de diseño durante el diseño inicial
- Conforme diversas versiones del sistema son creadas, también su diseño cambia ampliamente

Conceptos clave de diseño

Introducción

- El buen diseño depende del entendimiento de ciertos conceptos clave
- En este curso se verán los siguientes:
  - Complejidad
  - Características de diseño deseables
  - Niveles de diseño

Manejar complejidad

El desarrollo de software es complicado debido a dos tipos de problemas:

- Esenciales: son aquellos que definen al propio problema
- Accidentales: son problemas que se derriban de la forma de trabajo (lenguaje de programación, plataforma de desarrollo, sistemas operativos, etc.)

Manejar complejidad

- La raíz de todas las dificultades (esenciales o accidentales) es la complejidad
- Cuando un proyecto falla rara vez es por errores técnicos
- Normalmente los fallos vienen por una mala extracción de requerimientos, mala planificación, o mala administración
- Sin embargo, si el proyecto llega a fallar por razones técnicas, esta razón es a menudo complejidad descontrolada

Manejar complejidad

- El software puede ser tan complejo como se quiera, y si esta complejidad es demasiado alta es posible que nadie entienda el sistema
- Si ya no se tiene idea del impacto que una modificación en el código de un modulo tendrá en los demás módulos, entonces el proyecto falla

Manejar complejidad

- El manejo de complejidad es el tópico técnico de mayor importancia en desarrollo de software
- La idea es descomponer los problemas en partes más pequeñas, de tal forma que podamos enfocarnos a cada parte de manera independiente
- La meta es minimizar la fracción de sistema en la que cada programador piensa en un momento dado
- Al nivel de arquitectura de software, la complejidad se reduce al dividir el sistema en subsistemas

Manejar complejidad

- Los humanos pueden entender más fácilmente un conjunto de piezas de información simples que una sola pieza muy complicada
- La meta de cualquier técnica de diseño de software es descomponer un problema complejo en pequeñas piezas
- Entre más independencia haya entre las piezas (subsistemas) será más fácil enfocarse solo a esa pieza, reduciendo la complejidad
-

Manejar complejidad

Se pueden seguir los siguientes consejos generales

- Dividir el código en módulos o paquetes de subsistemas
- Mantener funciones y rutinas cortas
- Escribir programas en términos del dominio del problema
- Utilizar el nivel de abstracción más alto disponible (usar objetos apropiadamente por ejemplo)

Manejar complejidad

Existen tres fuentes de diseño inefectivo y costoso:

- Una solución compleja para un problema simple (sobre diseño)
- Una solución simple e incorrecta a un problema complejo
- Una solución inapropiada y compleja para un problema complejo

Manejar complejidad

- El software moderno es inherentemente complejo
- En algún punto debe lidiarse con la complejidad
- Habrá casos donde las simplificaciones no sean posibles, la complejidad puede ser parte del problema
- Dos aproximaciones básicas para manejar complejidad:
  - Minimizar la cantidad de complejidad esencial que cada programador tiene que trabajar al mismo tiempo
  - Mantener la complejidad accidental a raya

Manejar complejidad

- Una vez se entiende que las demás metas de técnicas del desarrollo de software son secundarias al manejo de complejidad, muchas consideraciones de diseño se vuelven directas

Características deseadas del diseño

- Cualquier diseño de alta calidad tiene características en común
- Si se cumple con dichas características el diseño se considera bueno
- Algunas metas deseables contradicen otras metas deseables, eso es parte de los retos de diseño
- Algunas características del diseño de calidad también las tiene el código de alta calidad: legibilidad, eficiencia, entre otras
- En lo que sigue se presentarán diversas características deseadas de diseño

Características deseadas del diseño

Complejidad mínima

- Debe ser la principal meta del diseño
- Se deben evitar los diseños "ingeniosos" que a menudo son difíciles de entender
- El diseño debe ser simple y fácil de entender
- Si el diseño no te permite ignorar de forma segura la mayoría de partes del sistema cuando estás enfocado en una parte específica, entonces el diseño no sirve

Características deseadas del diseño

Facilidad de mantenimiento

- Significa diseñar para el programador que dé mantenimiento
- Continuamente imagínate las preguntas que el programador de mantenimiento te hará con respecto a tu código
- Piensa en el programador de mantenimiento como si fuese tu audiencia, y entonces diseña el sistema para que se explique así mismo

Características deseadas del diseño

Conectividad mínima

- Significa mantener el menor número de conexiones (acoplamiento) entre las diferentes partes del sistema
- Se debe utilizar el principio de cohesión fuerte, bajo acoplamiento y ocultamiento de información (encapsulación) para diseñar clases
- Esto reduce el trabajo cuando se realiza integración, pruebas y mantenimiento

Características deseadas del diseño

Extensibilidad

- Significa que es posible mejorar el sistema sin causar cambios radicales a la estructura básica del sistema.
- Se puede cambiar una parte del sistema sin afectar a otras partes

Características deseadas del diseño

Reusabilidad

- Significa diseñar el sistema de tal forma que partes de él se puedan reutilizar en otro sistema

Características deseadas del diseño

Alta ventilación interna

- Se refiere a tener un alto número de clases que utilicen una misma clase dada
- Esto implica que el sistema ha sido diseñado para hacer buen uso de clases de utilería en los niveles más bajos del sistema

Características deseadas del diseño

Baja ventilación externa

- Significa tener una clase dada que utilice un número bajo o medio de otras clases
- La ventilación externa alta (más de siete clases) indica que una clase puede ser demasiado compleja
- Este mismo principio puede ser aplicado también cuando se considera el número de rutinas que una rutina dada manda a llamar

Características deseadas del diseño

Portabilidad

- Significa diseñar el sistema de tal forma que pueda ser movido a otro ambiente (sistema operativo, tipo de dispositivo, etc.)

Características deseadas del diseño

Ligereza

- Significa diseñar sistemas de tal forma que no tenga partes extra
- El filosofo Voltaire dijo que un libro está terminado no cuando ya no se le pueda añadir algo más, sino cuando no se le puede quitar nada
- El código extra debe ser desarrollado, revisado, probado y considerado cuando el demás código es modificado
- Versiones futuras del software deben ser retro-compatibles con el código extra
- Lo que deben evitar preguntarse es lo siguiente: Es fácil, ¿Porqué no agregarlo? ¿Que daño puede hacer?

Características deseadas del diseño

Estratificación

- Significa tratar de mantener los niveles de descomposición del sistema de forma estratificada, de tal forma que el sistema pueda ser observado en capas, las cuales son por si mismas consistentes
- Es diseñar el sistema de tal forma que al observar un nivel no sea necesario preocuparse por los otros niveles
- Si estás haciendo un programa que se comunica con un sistema legacy pobremente implementado, es buena idea agregar una capa que interactué de forma limpia con ese sistema, de esta forma la capa oculta los detalles feos del sistema legacy, y las nuevas partes no se preocupan de ello

Características deseadas del diseño

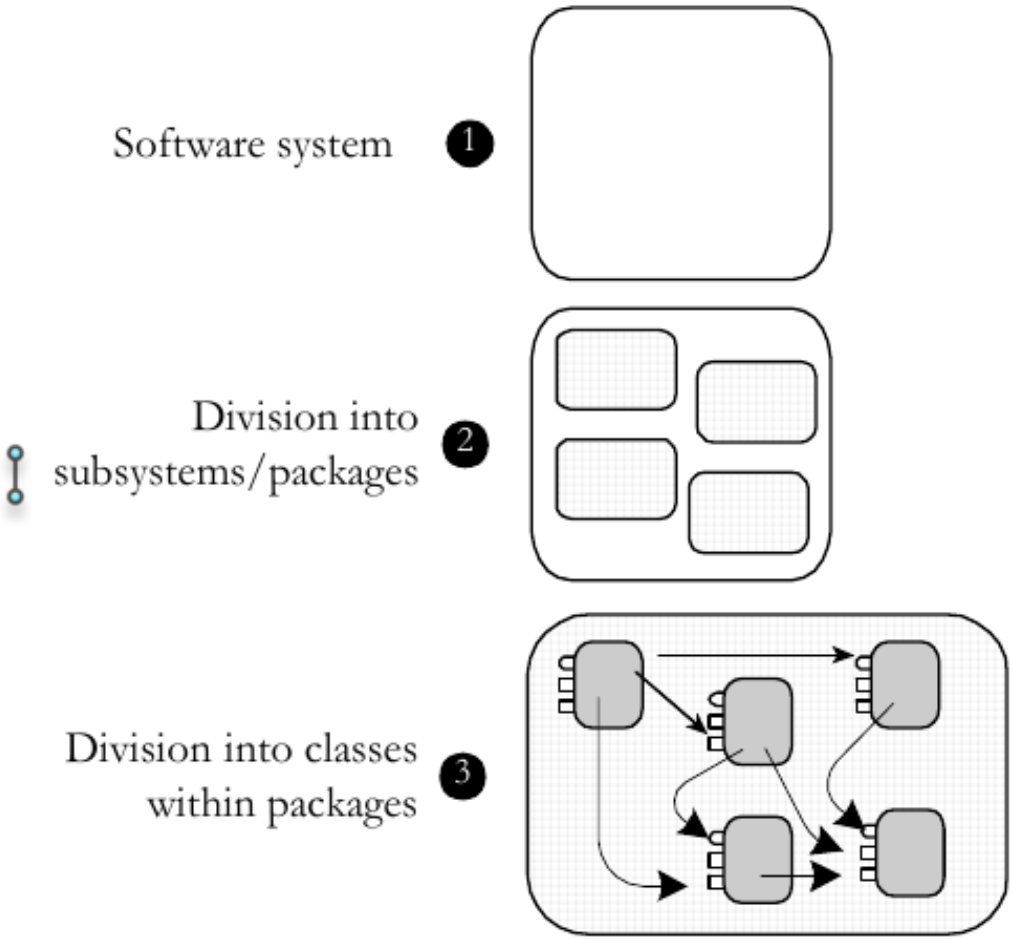
Técnicas estándar

- Entre mayor dependencia tenga un sistema a piezas exóticas, más intimidante será para alguien entender el sistema por primera vez
- Trata de darle al sistema un estilo familiar al utilizar aproximaciones comunes y estandarizadas

Niveles de diseño

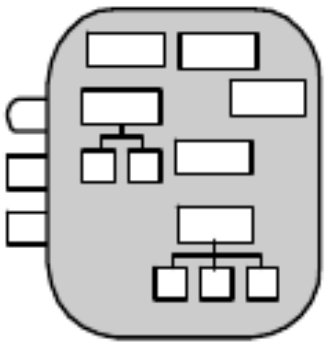
- El diseño se requiere en diversos niveles de detalle del sistema de software
- Algunas técnicas de diseño se aplican a todos los niveles, mientras otras a uno o dos niveles

Niveles de diseño

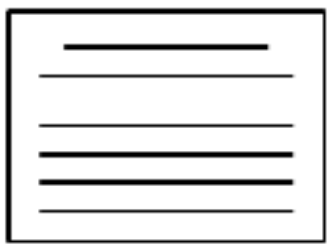


Niveles de diseño

Division into data and routines within classes 4



Internal routine design 5



Niveles de diseño

Nivel 1: sistema de software

- Es el sistema completo
- Algunos programadores saltan del nivel de sistema al diseño de clases, sin embargo, es usualmente beneficioso pensar a través de un nivel superior, como el de subsistemas o paquetes
- Si se toma la decisión de no hacer subsistemas porque el proyecto es pequeño, esa decisión debe ser conciente

Niveles de diseño

Nivel 2: división en subsistemas o paquetes

- El producto principal de diseño en este nivel es la identificación de todos los subsistemas importantes
- Ejemplos de subsistemas son: interfaz de usuario (GUI), lógica de negocios, manejo de BD, motor de reportes, etc.
- Las actividades de diseño principales de este nivel son decidir cómo partir el sistema en subsistemas y cómo cada subsistema puede comunicarse con los otros (si es que puede)
-

Niveles de diseño

Nivel 2: división en subsistemas o paquetes

- Este nivel se necesita típicamente en cualquier proyecto que requiere más de unas cuantas semanas de desarrollo
- En cada subsistema se pueden utilizar diferentes métodos de diseño
- Se deben establecer las reglas de comunicación entre subsistemas
- La comunicación entre subsistemas debe estar lo más restringida posible (esto reduce complejidad, baja el acoplamiento y ayuda a la reusabilidad)
- Es mejor restringir completamente la comunicación entre subsistemas al principio y luego agregar las requeridas, a dejar abierta la comunicación desde el principio

Niveles de diseño

Nivel 2: división en subsistemas o paquetes

- Para mantener las conexiones fáciles de entender y mantener, es mejor cometer errores referentes a simplificar demasiado las relaciones entre subsistemas
- La relación más simple es tener una llamada a una rutina de un subsistema dentro de otro
- Una relación más compleja es tener un subsistema que contiene clases de otro subsistema
- La relación más compleja es tener clases en un subsistema que hereden de clases de otro subsistema
- Lo ideal es nunca tener relaciones circulares entre subsistemas

Niveles de diseño

Nivel 2: división en subsistemas o paquetes, subsistemas comunes

- Lógica de negocios
- Interfaz de usuario (GUI cuando es gráfica)
- Acceso a BD
- Dependencias de sistema

Niveles de diseño

Nivel 2: división en subsistemas o paquetes. Lógica de negocios

- Son las leyes, regulaciones, políticas y procedimientos que codificas en el sistema de cómputo
- Es el código más fuertemente ligado con el dominio del sistema
- Es donde se define y resuelve el problema en si (problema esencial)

Niveles de diseño

Nivel 2: división en subsistemas o paquetes. Interfaz de usuario

- Es recomendable para que la interfaz de usuario pueda evolucionar sin dañar el resto del sistema
- En la mayoría de casos este subsistema utiliza muchos subsistemas y bibliotecas subordinadas o clases para GUIs, para el manejo de menús, ventanas, sistemas de ayuda, etc.

Niveles de diseño

Nivel 2: división en subsistemas o paquetes. Acceso a BD

- Se pueden ocultar los detalles de implementación del acceso a BD de tal forma que la mayoría del programa no se preocupa acerca de detalles de majo nivel referentes a BD (sintaxis de querys, tablas, etc.)
- De esta forma la interacción es de más alto nivel (mayor nivel de abstracción), pudiéndose preocupar más el programador por los datos en un sentido del dominio del problema (nivel negocio)
- Elevar el nivel de abstracción siempre reduce complejidad
- Reduce los errores al trabajar con datos, al centralizar todas las operaciones con datos
- Es más sencillo modificar el diseño de la BD sin tener que realizar demasiados cambios en el programa

Niveles de diseño

Nivel 2: división en subsistemas o paquetes. Dependencias de sistema

- Significa separar todos los aspectos específicos de una plataforma de ejecución (como un SO) en un subsistema a parte
- Aspectos específicos pueden ser bibliotecas, programas propios de la plataforma, etc.
- Si luego se quiere dar soporte a otra plataforma, simplemente se crea otro subsistema con la misma interfaz

Niveles de diseño

Nivel 3: división en clases dentro de paquetes

- El diseño a este incluye identificar todas las clases de el sistema.
- Por ejemplo, la lógica de negocios puede ser dividida en clases, una clase para cada elemento del dominio y sus interacciones
- Detalles de la forma en que cada clase interactúa con el resto del sistema son también especificados

Niveles de diseño

Nivel 3: división en clases dentro de paquetes

- En particular, la interfaz de la clase es definida
- La principal actividad de este nivel es asegurarse de que todos los subsistemas se han descompuesto al nivel de detalle adecuado, de tal forma que sus partes puedan ser implementadas como clases individuales
- Esta labor es típica para proyectos que toman más de unos cuantos días
- En proyectos pequeños es posible moverse directamente a esta etapa sin pasar por la segunda

Niveles de diseño

Nivel 4: división en datos y rutinas dentro de clases

- El diseño en este nivel incluye la división de clases en rutinas
- Las interfaces de clase definidas en el nivel 3 definirán algunas rutinas
- En este nivel se detallan las rutinas privadas
- Algunas rutinas son planas, mientras que otras pueden estar compuestas a partir de rutinas jerárquicas, las cuales requieren aun más diseño

Niveles de diseño

Nivel 4: división en datos y rutinas dentro de clases

- El acto de definir completamente las rutinas de una clase a menudo resulta en una mejor comprensión de la interfaz de clase, lo que puede provocar que se cambien aspectos de la interfaz (el nivel 3)
- Normalmente se le deja al programador individual
- Es necesario para cualquier proyecto que lleve más de unas cuantas horas
- No es necesario realizar este nivel de manera formal, pero si mental

Niveles de diseño

Nivel 5: diseño interno de la rutina

- Consiste en definir la funcionalidad detallada de las rutinas individuales
- Se le deja típicamente al programador individual que trabaja en una rutina individual
- Actividades asociadas pueden ser:
  - Escribir pseudo-código
  - Ver algoritmo en referencias
  - Decidir como organizar los párrafos y líneas de código
- Este nivel siempre debe realizarse

Bloques de construcción

Introducción

- A los desarrolladores de software les gusta el comportamiento determinista
- Cuando se programa se definen flujos de acción de acuerdo a las entradas, obteniéndose salidas esperadas
- Se pude definir un proceso en pasos y el comportamiento es predecible
- Pero el diseño de software es una historia diferente

Introducción

- Ya que el diseño es no determinista, lo más cercano que se tiene son las heurísticas
- Una buena aplicación de heurísticas es el núcleo de un buen diseño
- Las heurísticas pueden ser entendidas como guías
- A continuación se presentan diversas heurísticas para el manejo de complejidad (lo más importante del diseño)

Encuentra objetos del mundo real

- Aproximación orientada a objetos
- Consiste en identificar objetos sintéticos o del mundo real
- Los pasos son los siguientes:
  - Identificar los objetos y sus atributos
  - Determinar qué se puede hacer con cada objeto
  - Determinar que le puede hacer un objeto a otros (¿Qué objetos pueden contener a otros o heredar?)
  - Determinar las partes de un objeto que le serán visibles a otros (parte pública y privada)
  - Definir la interfaz pública del objeto
- Los pasos no se ejecutan necesariamente en orden y pueden repetirse, iterar es importante

Ejercicio

- Toma como referencia el problema de la administración de una tiendita
- Identifica al menos 3 objetos en el dominio del problema
- Para esos 3 objeto completa los pasos de la heurística vistos anteriormente

Forma abstracciones consistentes

- Abstraer es la habilidad de involucrarse con un concepto mientras al mismo tiempo se ignoran de forma segura algunos de sus detalles
- Significa manejar diferentes detalles en niveles diferentes
- Por ejemplo: siempre que uno dice "casa" está realizando una abstracción, no es necesario referirse al conjunto de paredes, material de construcción, etc.
- De la misma forma, pueblo es una abstracción

Forma abstracciones consistentes

- Las clases base te permiten concentrarte en atributos que varias clases poseerán sin necesidad de preocuparte por dichas clases
- Una buena interfaz de clase es una abstracción que te permite enfocarte en la interfaz sin necesidad de preocuparte de cómo funciona internamente la clase
- El principio anterior aplica también a la interfaz entre paquetes

Forma abstracciones consistentes

- Las abstracciones es una gran ayuda en el manejo de la complejidad
- En el lenguaje natural y las conversaciones cotidianas las personas usan continuamente abstracciones
- Algunos desarrolladores a menudo construyen software a nivel molecular, esto hace que el sistema sea muy difícil de entender ya que es demasiado complejo
- Los buenos programadores crean abstracciones al nivel de interfaz de rutina, nivel de interfaz de clase y nivel de interfaz de paquete

Encapsula detalles de implementación

- La encapsulación continua el trabajo de la abstracción
- La abstracción dice: tienes permitido mirar a un objeto a un nivel ele evado de detalle
- La encapsulación dice: consecuentemente, no tienes permitido mirar un objeto a ningún otro nivel de detalle, sólo el elevado
- La encapsulación te ayuda a manejar complejidad al prohibirte mirar la complejidad

Hereda cuando la herencia simplifica el diseño

- Es común encontrarse objetos que se parecen a otros exceptuando algunos campos
- Para este tipo de casos conviene utilizar herencia donde las características en común se describen en una clase base, mientras que las diferencias en clases específicas que heredan de la base
- Parte del comportamiento también se pude manejar desde la clase base de forma genérica lo que ayuda al diseño

Hereda cuando la herencia simplifica el diseño

- La herencia es una de las características más poderosas de la POO
- Permite polimorfismo
- Puede ser de gran ayuda cuando se usa bien y perniciosa cuando se usa mal (recuerden que crea acoplamiento)

Esconde secretos

- El ocultar información es parte fundamental del diseño estructurado y del diseño orientado a objetos
- En diseño estructurado se cuenta con el concepto de "cajas negras"
- En POO da origen a los conceptos de encapsulación y modularidad y está asociado con el concepto de abstracción
- Enfatiza el ocultar complejidad
- Lo más importante como diseñadores y programadores es preguntarnos constantemente ¿Qué debo ocultar?

Esconde secretos

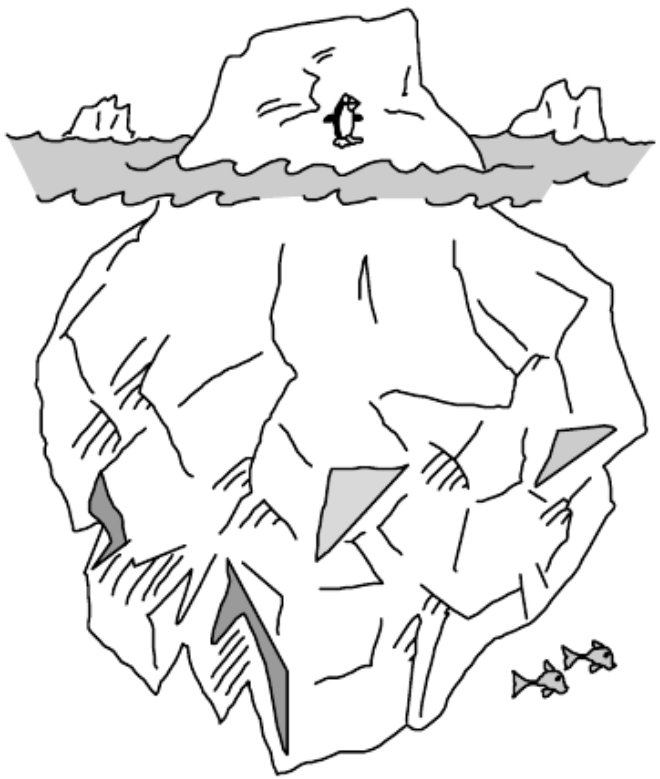
- Cada clase, paquete o rutina está caracterizada por los secretos que guarda
- Los secretos pueden ser áreas propensas a cambiar, el formato de un archivo, la forma en que un tipo de dato está implementado, o una área que se encuentra aislada del resto del programa para aislar también los errores que pueda generar
- El trabajo de la clase/paquete es mantener los secretos como privados
- Cambios menores en un sistema pueden afectar a varias rutinas dentro de una clase, pero esos efectos nunca deben ir más allá de la interfaz

Esconde secretos

- Una tarea clave es decidir qué características mostrar y qué características mantener en secreto
- Una clase por ejemplo puede utilizar una gran variedad de tipos de datos y no exponer ninguna información sobre ellos
- A esto también se le conoce como visibilidad

Esconde secretos

- La interfaz de una clase debe revelar tan poco como sea posible de la forma en que la clase trabaja internamente
- Una clase debe ser como un iceberg (la mayoría no se ve)



Esconde secretos

- Diseñar la interfaz de clase es un trabajo iterativo, como cualquier otro aspecto del diseño
- Si no logras tener la interfaz bien la primera vez, prueba hasta que se estabilice
- Si no se estabiliza necesitas probar una nueva aproximación
- Mencionar ejemplo de IDs

Esconde secretos

Dos categorías de secretos:

- Esconder complejidad de tal forma que tu cerebro no tenga que lidiar con ella a menos que estés específicamente trabajando con ella
- Esconder fuentes de cambio de tal forma que cuando un cambio ocurra sus efectos estén localizados

Esconde secretos

- En algunos casos esconder información es verdaderamente imposible
- Pero en la mayoría de los casos si se cree que es imposible es posible que se trate de un bloqueo mental inducido por el uso habitual de alguna técnica

Esconde secretos

Barreras para esconder información:

- Excesiva distribución de información: por ejemplo, utilizar un valor literal en vez de una constante y repetir ese valor literal múltiples veces
- Dependencias circulares: por ejemplo, la clase A depende de B y B de A. No es posible probar ninguna de las dos hasta que las dos estén completas
- Datos de clase interpretados como datos globales: las propiedades de clase no son tan riesgosas como las variables globales
- Penalidades de rendimiento percibidas: si tomas en cuenta el esconder información y el rendimiento, puedes conseguir ambas

Identifica áreas que pueden cambiar

- Una característica que comparten todos los buenos diseñadores es su habilidad para anticiparse a los cambios
- Acomodar cambios es uno de los retos más grandes del diseño de un programa
- El objetivo es aislar áreas inestables, de tal forma que su efecto al cambiar sólo perturbe a una clase

Identifica áreas que pueden cambiar

Pasos:

1. Identificar elementos que sean propensos a cambiar. Si los requerimientos están bien hechos, éstos deben incluir una lista de cambioas potenciales y su probabilidad
2. Separar elementos que sea probable que cambien. Todos los elementos volátiles deberían estar en una clase separada
3. Aísla los elementos que sea probable que cambien. Diseña la interfaz de clase para que no sea sensible a los cambios potenciales. El resto del sistema no debería enterarse si hubo un cambio. La interfaz de clase debe ocultar los secretos de la clase (que puede haber cambios)

Identifica áreas que pueden cambiar

Algunas áreas que son propensas al cambio:

- Lógica de negocios: las reglas de negocio son muy propensas a cambiar. Por ejemplo, el gobierno puede cambiar la forma en que se calculan y cobran impuestos. Las reglas de negocio deben estar bien ocultas
- Dependencias de hardware: como por ejemplo impresoras específicas, dispositivos de almacenamiento específicos, etc. Las dependencias de hardware deberían estar en su propio subsistema o clase, de esta forma es más fácil adaptar el sistema a otro hardware
- Entrada y salida: si tu sistema crea sus propios archivos, es posible que el formato de dichos archivos cambie a la par que el sistema se vuelve más sofisticado

Identifica áreas que pueden cambiar

Algunas áreas que son propensas al cambio: (continuación)

- Características no estándar del lenguaje: esto se refiere a bibliotecas externas, éstas podrían no estar disponibles en cualquier sistema, o ser compatibles con cualquier hardware. Esconde todas estas características en clases a parte, de esta forma pueden reemplazarse de ser necesario
- Partes complicadas de diseño y construcción: es importante ocultar las partes del diseño y de la construcción que se considere fueron realizados pobremente

Identifica áreas que pueden cambiar

- Variables de estatus: indican el estado del programa (por ejemplo códigos de errores) y tienden a cambiar con mucha frecuencia. El consejo general es no utilizar booleanos (mejor enum o clase) como variables de estatus y tener una rutina de acceso para cambiar la variable en vez de hacerlo directamente
- Restricciones de tamaño en datos: por ejemplo, cuando estableces que un arreglo es de tamaño 15, estás exponiendo información al mundo que el mundo no necesita saber, mejor crea una constante MAX para ocultar el 15

Identifica áreas que pueden cambiar

Anticipa diferentes grados de cambio:

- Diseña el sistema de tal forma que el efecto o ámbito de un cambio sea proporcional a la probabilidad de que ese cambio ocurra
- Si un cambio es muy probable, asegúrate de que el sistema puede acomodarlo fácilmente
- Si un cambio es extremadamente improbable se permite que el cambio afecte a varias clases
- Anticiparse al cambio tiene costos, enfócate en los más probables

Identifica áreas que pueden cambiar

Anticipa diferentes grados de cambio:

- Una buena técnica es identificar la parte central del sistema (la parte que el usuario necesita forzosamente)
- Ese núcleo probablemente no cambié
- Luego define incrementos mínimos del sistema, esos incrementos constituyen mejoras que tienen más probabilidades de cambiar
- Diseña esas mejoras incrementales utilizando el principio de esconder información

Mantén el acoplamiento bajo

- El acoplamiento describe que tanto una clase o rutina está relacionada con otra
- La meta es crear clases y rutinas con relaciones pequeñas, directas, visibles y flexibles (bajo acoplamiento)
- El concepto aplica de la misma forma para clases y para rutinas (las dos se pueden agrupar en el término módulo)

Mantén el acoplamiento bajo

- Un acoplamiento sano entre módulos es suficientemente bajo para que un módulo pueda fácilmente utilizar otros módulos
- El acoplamiento en software debería ser similar al acoplamiento en vagones de tren (no hay que soldar nada, o amarrar, los vagones pueden ser diferentes, etc.)
- Las conexiones entre módulos deberían ser lo más simple posible

Mantén el acoplamiento bajo

- Trata de crear módulos que dependan poco de otros módulos (o nada)
- Has que esos módulos se puedan remover más que quedar pegados (como siameses)
- Dos clases que dependen de los datos globales de la una y la otra están fuertemente acopladas

Mantén el acoplamiento bajo

Criterios para evaluar acoplamiento:

- Tamaño: se refiere al número de conexiones entre módulos. Una rutina que sólo recibe un parámetro tiene menor acoplamiento con rutinas que la usan que una rutina que recibe 6 parámetros. Una clase con 4 métodos públicos tiene menor acoplamiento con módulos que la usan que una clase con 50 métodos públicos
- Visibilidad: se refiere a que tan obvias son tus conexiones. Las conexiones deben ser lo más obvias posibles. Por ejemplo, no usen variables globales para crear la conexión
- Flexibilidad: que tan fácil puedes cambiar una conexión. Esto ayuda a que la misma conexión pueda ser utilizada por otros módulos

Mantén el acoplamiento bajo

Tipos de acoplamiento:

- Acoplamiento simple por parámetros: se da cuando los módulos sólo intercambian información a través de parámetros primitivos. Este acoplamiento es normal y aceptable
- Acoplamiento simple de objetos: sucede cuando un objeto instancia a otro. Es normal y está bien
- Acoplamiento por parámetros de objetos: es como el acoplamiento simple de parámetros pero pasando objetos como parámetros

Mantén el acoplamiento bajo

Tipos de acoplamiento: (continuación)

- Acoplamiento semántico: el peor de todos. Es cuando un módulo ocupa no sólo un elemento sintáctico de otro módulo, sino que también conocimiento semántico de cómo trabaja internamente otro módulo. Por ejemplo, el módulo M1 le pasa a M2 una bandera que le indica qué hacer a M2. Esto requiere que M1 sepa o asuma cómo funciona internamente M2 (que va a hacer con la bandera de control).
  Este tipo de acoplamiento es muy peligroso porque los cambios pueden tener un gran impacto y ser difíciles de detectar y depurar

Mantén el acoplamiento bajo

Recuerden:

- Las clases y rutinas son ante todo herramientas intelectuales para reducir la complejidad
- Si éstas no hacen tu trabajo más simple, entonces no están cumpliendo con su trabajo

Utiliza patrones de diseño comunes

- Los patrones de diseño proveen el núcleo de soluciones ya realizadas que pueden ser utilizadas para solucionar muchos de los problemas de software comunes
- La mayoría de problemas tienen aspectos similares a problemas pasados, dichos problemas pueden ser solucionados utilizando soluciones similares
- A esas soluciones similares le llamamos patrones de diseño

Utiliza patrones de diseño comunes

Beneficios:

- Reducen complejidad al proveer abstracciones ya hechas
- Reducen errores al institucionalizar detalles de soluciones comunes: un símil puede ser el uso de bibliotecas ya hechas y provadas
- Proveen valor heurístico al sugerir alternativas de diseño: es más fácil analizar patrones conocidos y evaluar el que mejor se ajusta a proponer algo desde cero
- Simplifican la comunicación referente al diseño al moverla a un nivel más elevado

Utiliza patrones de diseño comunes

Algunos de los más comunes:

- Adapter
- Bridge
- Decorator
- Fecade
- Factory Method
- Observer
- Singleton
- Strategy
- Template Method

Tarea: investigar cada uno de ellos y hacer un reporte al respecto

Heurísticas extra

- Apunta a la cohesión fuerte: todos los elementos que definas deben concentrarse a hacer sólo una cosa
- Construye jerarquías: no sólo se refiere a herencia de clases sino también a jerarquías en llamadas entre rutinas
- Formaliza contratos entre clases: osea crea interfaces rigurosas (creando un Interface en Java por ejemplo)
- Asigna responsabilidades: debes preguntarte todo el tiempo cuál es la responsabilidad de un objeto, su objetivo
- Diseña para probar: si diseñas pensando en hacer las pruebas más sencillas, el resultado es mejores interfaces, cohesión y modularización

Heurísticas extra

- Evita el fallo: no sólo pienses en el producto final o copies soluciones que funcionaron en el pasado, también piensa en lo que puede salir mal
- Crea puntos centrales de control: es conveniente agregar una capa de control entre subsistemas
- Considera utilizar la fuerza bruta: puede tomar mucho tiempo llegar a una solución elegante y eficiente, si tienes una solución más sencilla pero costosa puede ser una buena idea llevarla a cabo
- Dibuja un diagrama: te permitirá ver el problema de forma más general
- Mantén tu diseño modular: la meta es que cada rutina sea como una caja negra (sabes lo que entra, lo que sale, pero no lo que sucede adentro)

Consejos para utilizar heurísticas

- Entiende el problema
- Visualiza un plan
- Lleva a cabo tu plan
- Revisa lo que ya has hecho

Prácticas de diseño

Introducción

Se revisarán las siguientes:

- Itera
- Divide y vencerás
- Aproximaciones Top-down y Bottom-up
- Prototypos experimentales
- Diseño colaborativo

Itera

- Los ciclos de diseño son cortos y sin embargo tienen un gran impacto en el desarrollo posterior
- Dado lo anterior es conveniente repetir (iterar) varias veces los ciclos de diseño
- Iterar te permite ver las cosas en alto nivel y en bajo nivel dandote un mejor panorama
- Aunque hayas llegado a un diseño que consideres bueno, no te detengas, es posible que se te haya escapado algo. Cada intento mejora al anterior

Divide y vencerás

- Divide el sistema en diferentes áreas cohesivas
- Ataca cada área de manera individual
- Si te topas con un problema en alguna de las áreas, itera el diseño

Aproximaciones Top-down y Bottom-up

- Top-down se refiere comenzar en un nivel elevado de abstracción
- En top-down se definen primero clases base, u otros elementos no específicos de diseño
- Conforme se desarrolla el diseño se incrementa el nivel de detalle (clases derivadas, interfaces de clase, etc.)
- Si sigues un diseño basado en los niveles de detalle vistos antes, entonces estás haciendo diseño Top-down

Aproximaciones Top-down y Bottom-up

- En un diseño Bottom-up empiezas con consas muy específicas y luego diseñas cosas generales
- Por ejemplo, primero identificas objetos y luego creas cases base con las características parecidas
- Hay quienes opinan que la única forma correcta de trabajar es Top-down
- También hay quienes opinan que no es posible identificar aspectos generales hasta que se han entendido los detalles
- La recomendación es seguir una aproximación Top-down general y en ciertas ocaciones utilizar Bottom-up (como en la identificación de clases base)
- Al final del día, si realizas iteraciones estarás trabajando con ambas aproximaciones

Prototypos experimentales

- A veces es posible que no sepas si un diseño va a funcionar hasta que entiendas algunos detalles de implementación
- Es posible que no sepas si un subsitema va a funcionar apropiadamente hasta que lo ligues al GUI
- Todo esto tiene que ver con que el diseño es una labor perversa (no puedes tener el diseño hasta que hayas solucionado el problema parcialmente)
- Una forma barata de resolver este dilema es creando prototipos

Prototypos experimentales

- En este contexto prototipo significa escribir la cantidad absolutamente mínima de código desechable para responder a una pregunta específica de diseño
- Si no se realiza realmente el mínimo necesario la aplicación de prototipos no resulta conveniente
- De la misma forma no funciona de manera conveniente si la pregunta de diseño no es específica
- Un riesgo final es no tratar el código como desechable
- Usados correctamente y con disciplina los prototipos son la mejor herramienta para tratar debilidades de diseño
- Si no se utilizan con disciplina agregan complejidad

Diseño colaborativo

- En diseño, dos cabezas suelen ser mejor que una
- La colaboración tiene diversas formas:
  - Informalmente le pides consejos a un colega
  - Te reúnes con un colega a discutir alternativas de diseño y hacer esquemas
  - Te reunes con un colega y entre los dos desarrollan diseño detallado para un lenguaje
  - Realizas reuniones con varios colegas
  - No trabajas con nadie, guardas tu trabajo en un cajón y una semana después regresas. Para entonces habrás olvidado suficiente de lo que hiciste como darte a ti mismo una buena revisión