TEORIA-POO.pdf

Shainee

Programación Orientada a Objetos

2º Grado en Ingeniería Informática

Escuela Politécnica Superior de Córdoba
Universidad de Córdoba

THEORY WITHOUT COMPUTERS

Ft. PerroSanxeCabron en Wuolah

Tema 1: Abstracción y diseño de software

La abstracción es básicamente **simplificar**, en medida de lo posible, un sistema complejo como puede ser un sistema software. Para ello, vas poniendo capas de abstracción una encima de otra, como en un pastel, **ocultando lo que no quieres que se vea porque es difícil y mostrando lo que es relevante y/o sencillo de entender**.

P.ej.: cómo se implementan la tirada de dados no te importa, sólo el poder tirar los dados, así que ocultas la implementación. A veces no te importa tanto la tirada de dados porque estás jugando al monopoly, así que muestras el turno del monopoly sólo.

La abstracción se utiliza en el diseño: localizas los componentes del problema, y miras lo que hacen, las funciones, sin llegar a diseñarlas. Muestras lo verdaderamente importante.

Las TADs o clases en POO hacen exactamente eso; te muestran el componente y **lo que hace, no cómo lo hace**, mediante abstracción. Otra ventaja es que son fácilmente reutilizables, pues sólo necesitas saber lo que hacen; sólo es mostrada su parte pública o interfaz.

Esto último nos lleva a la siguiente definición: la **encapsulación**. Una clase no muestra cómo hacer las cosas, así que tendrá **datos inaccesibles desde fuera: su parte privada**. Las **operaciones de la clase serán su parte pública, y sólo se podrá comunicar con la clase mediante ellas**, encapsulando los datos privados. Como no nos importa el cómo hacer algo, protegemos eso para que si cambia, no haya que cambiar el resto.

Tema 2: Software de calidad

Para definir si un software es de calidad o no, se utilizan los factores de calidad. Éstos se pueden clasificar de dos formas: externos (calidad funcional, cara al usuario) e internos (calidad estructural, cara al programador), o product operation (funcionalidad), product revision (mantenimiento) y product transition (adaptación y relación con otros softwares)

Product operation

- Corrección (externo): el programa hace aquello para lo que fue diseñado.
- Robustez (externo): el programa se comporta con exactitud, precisión y satisfactoriamente siempre. Maneja bien situaciones imprevistas.
- Eficiencia (externo/interno): el programa realiza su trabajo de la mejor forma posible (ahorrando memoria, etc.)
- Integridad (externo): el programa no se corrompe ni es vulnerable.
- Facilidad de uso (externo): *self-explanatory*.

Product revision

- Mantenibilidad (interno): se puede encontrar y corregir un defecto en el programa.
- Extensibilidad / Flexibilidad (interno): facilidad de adaptar el programa a cambios en la especificación de requisitos.
- ¿Testabilidad? (interno): Habilidad de poder validar el programa.

Product transition

- Portabilidad (externo/interno): capacidad de ejecutar el programa en otro dispositivo o SO diferente.
- Reusabilidad (interno): el programa o parte de él puede ser utilizado por otros productos (diseño en clases)
- Compatibilidad (externo): facilidad del programa para combinarse con otros.

Otros factores

- Seguridad (externo): capacidad del programa de proteger sus componentes ante accesos no autorizados o pérdida de información.
- Accesibilidad (externo): que el programa pueda ser utilizado por cualquiera, sin importar discapacidades.
- Oportunidad: el programa es lanzado en el momento que se necesita.
- Economía: *que no cueste un riñón*.

Tema 3: Descomposición modular

Dado que hay muchas formas de descomponer un problema en clases, ¿cómo elegimos la mejor? Para lograrlo, se siguen unos criterios, reglas y principios.

Criterios

- Descomposición modular: el método debe ayudar en la tarea de descomponer el problema en un número de subproblemas interconectados.

- Composición modular: el método debe favorecer que los elementos se puedan combinar para producir nuevos sistemas ajenos al inicialmente pensado.
- Comprensibilidad modular: Que una persona pueda entender cada módulo sin tener que recurrir a los otros.
- Continuidad modular: un cambio en los requisitos sólo ocasiona cambios en un módulo o muy pocos.
- Protección modular: si se da una situación anómala o inesperada, que el error se propague lo menos posible, preferiblemente no fuera del módulo.

Reglas

- **Correspondencia directa**: La estructura modular obtenida debe ser compatible con la estructura modular del dominio del problema. (Que pases todas las clases del diseño a software, vamos)
- **Pocas interfaces**: Un módulo debe comunicarse con el menor número de módulos posible, para ganar independencia.
- **Pequeñas interfaces**: Los módulos deben intercambiar la menor información posible.
- **Interfaces explícitas**: Con leerlas debería ser suficiente para saber qué hacen.
- **Ocultación de la información**: Se debe seleccionar un subconjunto de información para mostrar públicamente al resto de módulos. El resto será privado e inaccesible desde fuera.

Principios

- **Unidades modulares lingüísticas**: Si vas a hacer módulos, usa lenguajes que tengan buen soporte de módulos
- **Auto-documentación**: Toda la información relativa al módulo debería formar parte del propio módulo.
- **Acceso uniforme**: Utilizar la misma implementación y notación para todos los módulos. Si en un sitio se llama ‘x’, que no se llame ‘y’ en otro.
- **Principio Abierto-Cerrado**: Un módulo debe estar abierto a futuras modificaciones, pero cerrado porque ya puede ser utilizado e implementado en el sistema.
- **Elección única**: Siempre que un software deba admitir un conjunto de alternativas, habrá un único módulo que conozca su lista completa. De forma que si se incluyen nuevos módulos, sólo haya que modificar dicha base.

Tema 4: TDD y POO. Implementación

Solucionar un problema mediante el uso del TDD (top-down design) presenta desventajas, como la poca reutilización, la compartición de estructuras de datos, la dependencia entre módulos, la dificultad de repartir el trabajo en un equipo…

Con las clases se gestiona todo de una manera más sencilla e intuitiva. La especificación de una clase tiene la siguiente forma:
TAD Clase
Descripción lo que hace
Operaciones
Procedimiento x **Devuelve** algo **Excepciones**
Requiere algo
Modifica nada
Efectos hace esto

Para la implementación se pueden utilizar bloques try-catch para las excepciones, que el código sea el mínimo que cumpla la especificación, stubs para prototipar funciones rápidamente, constructores, modificadores, observadores, destructores…

Tema 5: Reutilización del Software

No se trata de simplemente copiar y pegar código, sino de poder reutilizar componentes como módulos o librerías, otorga beneficios como la rapidez de desarrollo, menor mantenimiento, fiabilidad, menor coste…

Cuántas más personas tengan acceso al código, más fáciles de detectar serán los bugs. Sirve para facilitar la creación de componentes y aprender de los ya existentes.

¿Qué reutilizar? Pues todo lo que se pueda. Personal de desarrollo, metodologías como POO, patrones de diseño… Si algo se repite mucho durante el diseño, hazlo un módulo. Los módulos son idóneos para la reutilización: desempeña una única tarea, es independiente, oculta detalles irrelevantes, pequeño tamaño…

Las comunidades de desarrolladores favorecen la reutilización por la puesta en común de corrección de errores, mantenimiento, portabilidad..

Hay diferentes estructuras de reutilización:

- Librerías: aunque son de tamaño reducido, no presentan apenas adaptación.
- Clases: perfectas para la reutilización.

- Paquetes: tiene utilidades de todo tipo sobre un tema: funciones, namespaces, módulos… Son soluciones amplias a problemas.
- Frameworks: facilitan el desarrollo de aplicaciones e incorporan elementos estructurales de diseño de éstas.

Tema 6: Los 4 pilares de POO

En la computación, hay tres 'fuerzas' fundamentales: acción, objeto y procesador. Tradicionalmente se diseñaba en torno a la acción, basada en funciones, con demasiada atención a la interfaz, etc… pero al basarse en objetos, se simplifica pensando a qué se lo hace el sistema, no qué hace.

En resumen, la tecnología orientada a objetos tiene cuatro pilares fundamentales:

Abstracción
Utilizar la clase de manera abstracta, mediante interfaces. Da igual lo que la clase haga, se trata de evitar que el usuario acceda dentro.

Encapsulamiento
La implementación está dentro de una cápsula, inaccesible. Sólo se puede acceder mediante los métodos de la clase. Impide el acceso al estado para evitar operaciones no permitidas, simplificar comprensión, si se modifican no cambia nada fuera…

Herencia
Estructura de reutilización jerárquica. Útil para la descomposición modular, reutilización, extensibilidad… Las clases pueden ser diferidas (algún comportamiento no se especifica en la propia clase, sino que sirve como interfaz genérica para sus subclases) y efectivas (no diferidas).

Polimorfismo
Cuando se dispone de la misma interfaz para entidades de distinto tipo. Varios tipos:
- Estático o ad hoc: sobrecarga de funciones/operadores. Se define una diferente para cada tipo.
- Paramétrico: se usa un tipo genérico que se especifica como parámetro cuando se instancia. Plantillas de clase y de función.
- De subtipo o subtipado: en tiempo de ejecución, permite construir código genérico independientemente del tipo mediante punteros a clase base y funciones virtuales. Los subtipos comparten interfaz y se puede escribir el mismo código para todas ellas.

FORMARIUM

¿Quieres aprobar las asignaturas técnicas de tu carrera?

Te ayudamos a hacerlo.

ONLINE Y PRESENCIAL

Ver más

Se denomina a un sistema cerrado si contiene todas las clases que necesita.

Tema 7: Patrones de diseño :DDDDDDD

Si ya hay una solución a un problema que tienes, para qué vas a crear la tuya propia, atontao? Usa la ley del mínimo esfuerzo.

Los principales patrones de sueño fueron especificados y estandarizados por el GoF (Gang of Four) //we gangsta on this motherfucka

Los patrones son **soluciones estándar a determinados problemas** a los que nos enfrentamos en la etapa de diseño. Es una solución reutilizable, elegante y de calidad a un problema complejo. No son simples instrucciones, son como modelos, donde el armazón ya viene.

Un patrón está compuesto por objetos, clases y sus relaciones, y se especializa en resolver un problema determinado en un contexto determinado. (como cheat sheets)
Se dividen en **tres grupos**:

- Behavioural Patterns: algoritmos, interacciones entre objetos…
- Creational Patterns: instanciar objetos de clases.
- Structural Patterns: crear clases o módulos.

Los patrones tienen como fichas a rellenar: nombre, tipo, descripción, aplicaciones y consecuencias (para la economía)

Template Method (Structural)
Para separar las cosas que pueden cambiar en un futuro de las que no. Imaginemos que tenemos una clase que puede hacer ‘x’ e ‘y’. Si en un futuro queremos que haga ‘z’, no tendremos que cambiar la clase con este patrón (cumple el principio abierto-cerrado)

¿En qué consiste?
Se crea una clase general para el módulo, abstracta, en la que se declaran las funciones de forma virtual, para servir como placeholders. Cada subclase define el comportamiento de esas funciones a su manera; de esa forma, no se modifica la clase principal. Si hay un nuevo comportamiento, se crea una nueva subclase. ggez.
(Los datos de la clase principal es recomendable que estén en protected para que puedan ser usados de forma sencilla por las subclases)

Parameterized Types (Structural)

En este patrón, se define un tipo, algo, sin especificar todos los tipos de sus atributos. template <class T> class MiClase{ private: T x_, y_;...

Iterator (Behavioural)
El viejo amigo. ¿Cómo podemos iterar algo sin modificar la estructura de ese algo? Haces una clase externa para ello. Dicha clase ya viene por defecto en la std library.

Observer (Behavioural)
En el caso de que haya unos datos y unas clases que dependen de los datos. Serán el objeto y los observadores. El patrón permite la subscripción siempre actualizada a los datos.

Se trata del par objeto-observador: el objeto tiene una función para registrar observadores y para notificar; los observadores al ser notificados tienen una función para actualizarse. Así ambos elementos son independientes, separando datos y vistas. Muy usado en el patrón MVC (modelo-vista-controlador)

Composite (Structural)
Para cuando hay objetos de diferentes niveles que tienen características y/o funciones muy parecidas. Tratamos uniformemente los objetos simples y sus composiciones.

Se crea una clase abstracta que impone una interfaz, y las subclases se comportan de manera uniforme sean sencillos o grupos. Los atributos privados cambiarán entre clases, claramente - los grupos tendrán un vector de objetos de las clases sencillas.

Strategy (Behavioural)
Tienes unos cuantos algoritmos, y puedes usar uno u otro para llevar a cabo una función. En ese caso, se prepara una descripción genérica del algoritmo para su posterior llamada del adecuado. Creas una clase strategy con subclases para algoritmos determinados, y al llamar a la función pasas el algoritmo que sea como objeto de clase. Con strategy se puede cambiar el algoritmo en tiempo de ejecución. Sirve para la prevención de distintos mecanismos en un futuro, como la adición de un nuevo algoritmo.

Model-View-Controller, MVC (Structural)
Se separa lo que cambia de lo que no: el modelo (objeto de la app), vista (la presentación) y el controlador (define el modo en el que se reacciona a las entradas). Se utiliza mucho en las aplicaciones web y está presente en muchos frameworks. Reparte las funciones fácilmente.

Builder (Creational)
Para crear instancias de objetos complejos o con muchos parámetros. Se crea una clase que controla los builders, y selecciona el determinado, y una clase builder para los diferentes objetos a crear.

Factory (Creational)
Se crea una clase para facilitar la creación y manipulación de instancias sin que el main dependa de las clases. La factoría (latinada) selecciona la clase correspondiente para que si se añade una nueva no haya que cambiar el main.

Singleton (Creational)
Para cuando sólo se necesita una única instancia de una clase. Se pone static. Se puede utilizar para gestionar datos globales. Static string = new string;