Continue

Metodologia orientada a objetos

La programación orientada a objetos, o POO, es una metodología de diseño y programación que se basa en la organización de datos y comportamientos en estructuras llamadas "objetos". Aunque puede sonar técnico al principio, este enfoque revoluciona la forma en que creamos y estructuramos software… …al alinear más estrechamente nuestro código con cómo vemos y categorizamos el mundo real, la POO nos permite escribir software más organizado, reutilizable y fácil de entender. Repasemos los conceptos La carrera de programador en 2017 y en el futuro (con Javier Santana) 0. Abstracción: simplificando la Realidad La abstracción es uno de los pilares fundamentales de la programación orientada a objetos: nos permite modelar el mundo real en términos de código, eligiendo qué propiedades y comportamientos son esenciales para nuestro software y cuáles pueden ser ignorados. Esto hace que el diseño del software sea más manejable, manteniendo el enfoque en la funcionalidad clave. En la programación, la abstracción a menudo se logra mediante la creación de clases y objetos. Estas clases sirven como plantillas o modelos que capturan la esencia de algo, sin entrar minuciosamente en todos los detalles (o atributos). 1. ¿Qué es un objeto? En el mundo real, estamos rodeados de objetos: tu taza de café, tu teléfono, tu bicicleta. Estos objetos tienen características (color, tamaño, modelo) y comportamientos (una taza contiene líquido, un teléfono puede hacer llamadas). En POO, modelamos estas entidades reales como "objetos" en código. Por ejemplo, considera una sencilla definición de un coche: class Coche: # Constructor de la clase def init(self, marca, modelo): self.marca = marca self.modelo = modelo # Método para describir el coche def describir(self): return f"Este coche es un {self.marca} {self.modelo}" Aquí, marca y modelo son características, mientras que describir es un comportamiento. 2. Clases: Los moldes de los objetos La definición anterior (Coche) es lo que llamamos una clase. Piensa en la clase Coche como un plano que nos dice cómo debería ser cada coche que salga de la fábrica en términos de programación. O podemos plantearlo en términos más domésticos: imaginemos que estás en la cocina, listo para hacer galletas. Tienes un molde de galletas que te ayudará a darles una forma específica. Este molde es lo que en POO llamamos una clase. Es una definición o plantilla que nos dice cómo se verán y comportarán nuestras galletas. Ahora bien, cuando usas el molde para hacer una galleta, esa galleta individual es un objeto. Aunque proviene de un molde común, cada galleta es única y puede tener sus propias particularidades, como chispas de chocolate o trocitos de nuez. 3. Creación de objetos: instanciando una clase Una vez que tengas una clase, puedes crear objetos específicos o instancias de esa clase. Siguiendo con nuestro ejemplo: mi_coche = Coche("Toyota", "Corolla") print(mi_coche.describir()) # Salida: Este coche es un Toyota Corolla 4. Atributos y métodos Un mismo objeto 'Chico' puede tener varios atributos, diferentes o en común con respecto a otros objetos de la misma clase Piensa en cómo describirías a tu mejor amigo: tiene nombre, edad, color de ojos, hobbies, etc. Estas características individuales son similares a los atributos en POO. Es decir, son las propiedades específicas que un objeto (como nuestra galleta) puede tener. Además de tener características, las personas pueden hacer cosas: cantar, bailar, correr, etc. En POO, estos comportamientos o acciones son los métodos. Son acciones que un objeto puede realizar. El mismo objeto 'monigote' haciendo uso de tres métodos distintos… En nuestros ejemplos anteriores, 'Coche', 'marca' y 'modelo' son atributos, mientras que 'describir()' es un método. 5. Herencia: Tomando lo mejor de otros Cuando hablamos de familias, a menudo escuchamos que alguien "ha heredado" el color de ojos de su madre o el talento musical de su padre. En POO, la herencia permite que una clase (recuerda, el molde de galletas) tome atributos y métodos de otra clase. Es como si una nueva receta de galletas tomara ingredientes de una receta anterior y agregara algo más. Supongamos que queremos tener una clase CocheElectrico que comparta algunas características con Coche pero que tenga algunas adicionales: class CocheElectrico(Coche): def init(self, marca, modelo, autonomia): super().init(marca, modelo) self.autonomia = autonomia def autonomia_restante(self, distancia_recorrida): return self.autonomia - distancia_recorrida El CocheElectrico es un tipo especial de Coche, pero tiene una característica adicional: autonomia. También tiene un método adicional para determinar la autonomía restante. 6. Polimorfismo: Una función, muchas formas El polimorfismo se refiere a la capacidad de diferentes clases de ser tratadas como instancias de la misma clase a través de la herencia. Es una forma elegante de decir que diferentes objetos pueden realizar una acción de la misma manera. Considerando las clases anteriores, aunque Coche y CocheElectrico son diferentes, ambos pueden usar el mismo método describir(). Este es un simple ejemplo de polimorfismo. 7. Encapsulamiento: Protegiendo nuestro código El encapsulamiento es la práctica de ocultar los detalles internos de cómo una clase realiza sus acciones. Imagina que tienes una mochila donde guardas tus cosas. Cuando la cierras, no necesitas saber exactamente cómo está todo organizado dentro. En POO, el encapsulamiento es como esa mochila: protege y oculta los detalles internos de un objeto, permitiendo que interactuemos con él de manera simple y segura. En Python, por ejemplo, usamos guiones bajos para denotar atributos o métodos privados: class Coche: def __init__(self, marca, modelo): self._codigo_secreto = "12345" self.marca = marca self.modelo = modelo Aquí, _codigo_secreto es un atributo protegido, que idealmente no debería ser accedido directamente fuera de la clase. Pero, ¿por qué deciemos que es más seguro recurrir al encapsulamiento? Veamos: Protección de datos internos: El encapsulamiento permite restringir el acceso directo a determinados atributos o métodos de una clase. Esto significa que podemos proteger y controlar cómo se accede y modifica la información interna de un objeto, evitando manipulaciones indebidas. Flexibilidad y mantenimiento: Si en el futuro necesitas cambiar la forma en que se implementa una característica particular, gracias al encapsulamiento, puedes hacerlo sin alterar el código que utiliza esa clase. Esto reduce el riesgo de errores cuando se hacen modificaciones. 8. Composición: construyendo objetos a partir de otros La composición implica construir clases utilizando otras clases como partes. Imagina construir un robot usando piezas individuales; cada pieza es esencial para la función completa del robot. Por ejemplo, un objeto Coche puede estar compuesto objetos como Motor, Ruedas o Chasis. 9. Interfaces: un contrato entre clases Las interfaces juegan un papel fundamental en muchos lenguajes de programación orientada a objetos. Puedes pensar en una interfaz como un contrato que una clase decide firmar. Este contrato estipula que la clase implementará ciertos métodos, pero no dicta cómo deben implementarse esos métodos. En otras palabras, la interfaz define qué se debe hacer, pero deja el cómo hacerlo a la clase que la implementa. Imagina un control remoto universal: tiene botones para encender, apagar, subir el volumen, bajar el volumen, etc. No importa la marca o el modelo de televisión, si deseas que ese televisor sea compatible con el control remoto universal, debe cumplir con ciertas funciones básicas. De manera similar, una interfaz define un conjunto de métodos que cualquier clase que implemente esa interfaz debe tener. Vamos a ver un ejemplo simplificado usando Python. Aunque Python no tiene un soporte directo para interfaces como otros lenguajes (como Java), podemos simularlo usando clases abstractas: from abc import ABC, abstractmethod class Encendible(ABC): @abstractmethod def encender(self): pass @abstractmethod def apagar(self): pass class Bombilla(Encendible): def encender(self): print("Bombilla encendida!") def apagar(self): print("Bombilla apagada!") Aquí, Encendible es nuestra interfaz que dicta que cualquier objeto que pueda ser encendido y apagado debe tener los métodos encender() y apagar(). La clase Bombilla implementa esta interfaz y define cómo se enciende y se apaga una bombilla. Las interfaces son una herramienta poderosa en la POO, ya que promueven la reutilización de código y aseguran que ciertas clases tengan comportamientos esperados, actuando como un estándar que debe seguirse. Además, ayudan a desacoplar el código, permitiendo que los sistemas sean más flexibles y adaptables a cambios futuros. Epílogo: Historia de la Programación Orientada a Objetos (POO) La programación orientada a objetos (POO) se desarrolló como una respuesta a las crecientes necesidades y desafíos de la ingeniería de software, buscando proporcionar estructuras más modulares, reutilizables y fácilmente mantenibles. 1960s - Simula: El origen de la POO se encuentra en el lenguaje de programación Simula, que introdujo los primeros conceptos de programación orientados a objetos, aunque el propio término "orientado a objetos" aún no había sido acuñado. 1970s - Smalltalk: En la década de 1970, el lenguaje Smalltalk se convirtió en el primer lenguaje de programación totalmente orientado a objetos. Introdujo conceptos como clases, instancias, herencia y encapsulamiento, y popularizó la POO. 1980s - C++: Durante la década de 1980, Bjarne Stroustrup desarrolló C++, que añadió características orientadas a objetos al lenguaje C. C++ jugó un papel crucial en hacer de la POO una técnica dominante en la ingeniería de software, dada su compatibilidad con C y su poderosa combinación de programación procedural y orientada a objetos. 1990s - Java: En la década de 1990, Sun Microsystems introdujo Java, diseñado con una orientación a objetos desde su núcleo. Java llevó la POO a la vanguardia de la ingeniería de software a gran escala y al desarrollo web, gracias a su portabilidad y su promesa de "Escribirlo una vez, ejecutarlo en cualquier sitio". Otros Lenguajes: A lo largo de los años, varios otros lenguajes han incorporado soporte para POO, incluidos Python, Ruby, C#, PHP y muchos más. Estos lenguajes han adaptado y evolucionado los principios básicos de POO para satisfacer diferentes necesidades y paradigmas. Imágenes | Marcos Merino mediante IA En Genbeta | Bill Gates cuenta cómo consiguió convertirse en un gran desarrollador: este es su principal consejo 1 - Metodologias de Análise OO - Introdução Para fazer um estudo das metodologias OO, é necessário conhecer as pessoas que estão por trás desta comunidade, pois lá, existe uma grande diversidade de idéias. Muitas destas pessoas discutem as suas opiniões apenas em termos da sintaxe e semântica das linguagens de programação por eles selecionadas. São utilizados os termos análise e desenho de uma forma bastante informal. Análise pode significar ouvir os clientes, tomar algumas notas e apontamentos, pensar acerca do problema e das potenciais soluções, e construir alguns protótipos de software. Desenho pode significar a codificação dos objetos individuais, o desenvolvimento de uma hierarquia de objetos ou a definição informal e implementação de um determinada classe. Existe outro grupo de pessoas que está interessada na formalidade e no rigor. Para estas pessoas, a engenharia de software é altamente sistemática e repetível. Eles vêem a engenharia de software OO como um processo de engenharia muito bem definido. A qualidade dos produtos resultantes pode ser avaliada de uma forma quantitativa, bem como qualitativa. Assim, a avaliação e comparação deste tipo de metodologias é uma atividade difícil e complexa. Por exemplo, o método de Booch usa o termo objeto e instância como sinônimos. Berard já utiliza o termo objeto de uma forma mais genérica, incluindo classes e meta-classes. Enquanto que esta diferença em termos de terminologia pode parecer puramente acadêmica, a aplicação destes métodos é significativamente influenciada por esta distinção. A completa especificação das várias metodologias varia dramaticamente. Por exemplo, alguns descrevem apenas o processo, outros apresentam uma notação gráfica, enquanto que outros combinam as notações com o processo. Sem compreender a cultura que está por trás de uma determinada metodologia, é impossível efetuar uma correta avaliação e comparação. Para tentar passar, de uma forma mais ampla e genérica, o conceito de O.O. na análise de sistemas, foram escolhidos seis métodos que me parecem os mais significativos em termos de aplicação, bem como divulgação. Não se trata de uma exposição exaustiva dos referidos métodos, mas sim, de uma descrição dos conceitos, processos, técnicas e ferramentas de suporte utilizadas em cada um dos métodos, de forma a ser possível encontrar semelhanças e diferenças, bem como problemas subjacentes. Assim, são apresentados os seguintes métodos: Booch Jacobson Coad/Yourdon Shlaer-Mellor Rumbaugh Wirfs-Brock 2. Método Booch Visão Global Geral: Conceitos Diagramas de Classe Diagramas de Transição de Estados Diagramas de Interação de Objetos Enquadramento Conceitual Análise de Requisitos Análise do Domínio Desenho Processos e Técnicas Análise de Requisitos Análise do Domínio Desenho Ferramentas de Suporte O método de Booch é constituído, basicamente, por três fases: · Análise de requisitos, que permite o estabelecimento das operações fundamentais do sistema; · Análise do domínio, que permite a construção da estrutura lógica referente ao domínio em questão; · Desenho, que permite a construção da estrutura física do sistema, fazendo o mapeamento da estrutura lógica anteriormente construída, conduzindo, assim, à construção de protótipos; O objetivo do método de Booch é providenciar uma notação e um processo de desenvolvimento, dando especial atenção ao aspecto da comunicação entre a fase de análise e desenho de um sistema de software OO. O método de Booch tem um caráter amplo que, endereçando os aspectos das ferramentas de análise e desenho OO, inclui modelação dos objetos, modelação da análise, desenho da aplicação, desenho da implementação e ciclo de vida dos processos. Booch propõe diferentes visões para descrever os sistemas OO. O modelo lógico, isto é, o domínio do problema, é representado na estrutura de classes e objetos. O diagrama de objetos mostra como os objetos interagem uns com os outros, enquanto que os diagramas de classe são de índole mais estática. Assim, os diagramas de objetos descrevem o comportamento dinâmico do sistema. O conceito de subsistema, neste método, é entendido como um diagrama de módulos, tendo um paralelismo, em termos do papel que representa, com os diagramas de categoria e de classes. Os subsistemas representam conjuntos de módulos relacionados de uma forma lógica. Um sistema pode consistir em múltiplos programas, executando num conjunto de computadores distribuídos. O diagrama de processos é utilizado para visualizar a forma como os processos são alocados aos processadores, em termos de desenho físico do sistema. Tipicamente, pode ser incluído apenas um diagrama de processo. A arquitetura de módulos e de processos lida com a alocação física das classes e objetos aos respectivos módulos, processadores, dispositivos, e ligações de comunicação entre eles, isto é, descreve os requisitos de hardware concretos em relação aos componentes de software do sistema. Na seguinte figura são mostradas as diferentes perspectivas explicitadas pelo método de Booch. *Clique Aqui para ver a Figura! 2.1. Conceitos Este método utiliza uma série de diagramas para descrever as decisões estratégicas tomadas nas fases de análise e desenho, que devem ser consideradas quando é criado uma sistema segundo o paradigma dos objetos. 2.1.1. Diagramas de classe Um diagrama de classe consiste num conjunto de classes e relacionamentos entre elas. Segundo esta notação, existem vários tipos de classes, cada uma representando um objetivo específico. As decisões tomadas são capturadas nos diagramas de classe e nas suas especificações. Os tipos de classes existentes são os seguintes: · Class - Conjunto de objetos que partilham uma estrutura e um comportamento comum. Uma classe é uma abstração de um item do mundo real. Quando estes itens existem, são instâncias da classe respectiva e são denominados objetos; · Parameterized class - Neste tipo de classes são declarados, formalmente, parâmetros genéricos. · Class utility - São classes não instanciáveis, contendo um ou mais métodos de classe; · Metaclass - São classes cujas instâncias são classes. Providenciam operações para inicialização de variáveis de classe, servindo como repositórios de suporte às variáveis de classe, necessários para todos os objetos da classe definida; Os relacionamentos são utilizados para indicar ligações semânticas entre as classes. Cada relacionamento tem associado um label, indicando que tipo de relação é que existe. O tipo de relacionamentos existentes são os seguintes: · Association - Utilizado para indicar que existe um determinado tipo de relacionamento, mas a decisão sobre que tipo exato de relacionamento existe, pode ser deferida; · Contains - Indica uma relação de estrutura entre duas classes. Pode ser utilizada cardinalidade. Os atributos e a agregação são casos particulares deste tipo de relacionamento; · Inheritance - Indica que uma classe partilha a estrutura ou comportamento definido numa ou mais classes. · Uses - Indica que uma classe é cliente de outra classe, isto é, utiliza os seus recursos; · Instantiation - São relacionamentos entre uma parameterized class e uma classe instanciável; · Metaclass - Mostra o relacionamento entre uma metaclass e as suas instâncias, que são classes; Os tipos de visibilidade dos relacionamentos podem ser os seguintes: público, protegido e privado. A especificação de uma determinada classe é utilizada para apreender toda a informação importante sobre essa classe, sobre uma forma textual. Esta especificação contém uma série de campos que a definem. Para grandes sistemas, as classes podem ser agrupadas de uma forma lógica. Cada categoria pode conter um ou mais diagramas de classe. Os diagramas de categoria são utilizados para denotar a visibilidade entre as categorias das classes. 2.1.2. Diagramas de transição de estados. O comportamento dinâmico associado com determinadas classes é caracterizado utilizando este tipo de diagramas. São utilizadas, neste método, as máquinas de estado hierárquicas de Harel's. Cada transição de estados liga dois estados. Um estado pode ter uma transição para ele próprio, e muitas transições podem partir do mesmo estado. Na seguinte figura é mostrado um exemplo de aplicação desta técnica de modelação. Clique aqui para ver a Figura! 2.1.3. Diagramas de Interação de objetos. Estes diagramas mostram a existência dos objetos e dos relacionamentos, no modelo lógico do sistema. Existem dois tipos de diagramas de objetos. Um diagrama de cenários, que mostra como é que um conjunto de objetos colaboram, de forma a implementar um mecanismo, mostrando o conjunto de operações que devem ocorrer, como resposta a um determinado cenário. Os diagramas de instância mostram a existência dos objetos, bem como os relacionamentos de estrutura existentes entre eles. Os diagramas de Interação mostram a execução de cenários da mesma forma que os diagramas de objetos. Assim, é outra representação dos diagramas de objetos. Mas, desta forma, segundo Booch, é mais fácil visualizar a invocação das mensagens na sua ordem. Os objetos são especificados no cimo das colunas horizontais do diagrama. As mensagens são mostradas com setas horizontais do cliente para o objeto servidor. A caixa vertical representa o tempo relativo que o fluxo de controle demora em cada objeto. Estes diagramas focam mais em eventos que em operações, resultando, assim, um melhor resultado na captação da semântica dos respectivos cenários. A seguinte figura explicita um exmplo dos digramas de Interação de objetos. Clique aqui para ver a Figura! 2.2. Enquadramento Conceitual . Um dos grandes pressupostos do método de Booch é criar um modelo do sistema, passível de ser implementado. Os vários conceitos deste método estão distribuídos, com diferentes níveis de abstração, nas três fases do processo, indicados a seguir. 2.2.1. Análise de requisitos. Neste método, a análise de requisitos produz dois tipos de especificações formais. A primeira é uma declaração das funções principais do sistema. Devem ser definidos inputs e outputs do sistema, ou uma lista de referências para planos de ação, procedimentos, etc. A segunda é a especificação de um conjunto de mecanismos chave que o sistema deve providenciar, incluindo o estado de entrada, saída e as mudanças de estado esperadas. Estes mecanismos chave servem para validar o domínio do modelo, descobrir operações, bem como testar determinados casos. 2.2.2. Análise do domínio. Esta fase inclui: · Diagramas de classe abstratos, que identificam as classes chave do domínio em questão, bem como os relacionamentos entre elas; · Especificações das classes, que contém todas as definições de índole semântico das respectivas classes, os seus relacionamentos, atributos e operações principais; · Herança, que são diagramas de classe que revelam os níveis de especialização entre as classes; · Diagramas de cenários dos objetos, que ilustram como é que os objetos interagem de forma a conduzir os mecanismos delineados na análise de requisitos; · Especificação dos objetos, faz o mapeamento dos objetos nas respectivas classes; Trata-se, portanto, de visões diferentes do modelo subjacente que contém a definição das classes, objetos, relacionamentos, etc. 2.2.3. Desenho. Nesta fase, é feito o refinamento do modelo criado na fase de análise, adicionando as estruturas e comportamentos necessários para implementação do domínio em estudo. O modelo iniciado durante a análise serve como base para a construção do modelo de desenho. Nesta fase, são adicionados as seguintes contribuições ao modelo: · Descrições arquiteturais - São capturadas as maiores decisões desta fase, tal como, escolha de processadores, gestores de bases de dados, sistemas operativos, linguagens, etc.; · Descrições do protótipo - Define os objetivos e conteúdo das implementações sucessivas do protótipo, dirigindo o processo de desenvolvimento, testando o desenho e os requisitos; · Diagramas de categoria - Modela as partições do sistema em grupos de classes e objetos, a um nível elevado; · Diagramas de classe - Descreve as abstrações da implementação a um nível físico, tipos de dados detalhados, ou estruturas, fazendo o mapeamento das abstrações do nível lógico; · Diagramas de objeto - Mostra a lógica operacional detalhada, de forma a cumprir as funções, incluindo a utilização dos objetos físicos; · Novas especificações - São feitas, suportando estes diagramas; · Aperfeiçoamentos às especificações das classes - Descreve as especificações operacionais, de uma forma integral, para as operações com algoritmos complexos, implementação de relacionamentos e tipos de atributos; 2.3. Processos e técnicas. Os utilizadores deste método desenvolvem modelos de um determinado sistema, implementando-os diretamente em unidades de código, denominados protótipos. O modelo é construído em estágios que permitem a concentração em determinados aspectos do sistema, um determinado momento. Podem ser escolhidos diferentes diagramas para focar áreas críticas do sistema. O resultado é uma série de visões claras e expressivas, como resposta a questões específicas sobre o desenho do sistema ou análise de requisitos. Este método tem como base o desenvolvimento de um sistema como um processo iterativo, isto é, conceitualizações anteriores devem ser complementadas ou refinadas, servindo o resultado como input do próximo estágio, de uma forma incremental. Assim, as primitivas codificações da fase de desenho podem servir como forma de descoberta dos requisitos do sistema. 2.3.1. Análise de requisitos. A análise de requisitos é o processo de determinação daquilo que o cliente quer que o sistema faça. É uma fase de alto nível que identifica as funções chave que o sistema deve desempenhar, definindo o alcance do domínio que o sistema deve suportar, documentando os processos e planos de ação da organização suportados pelo sistema. Os passos desta fase não são definidos formalmente, pois este processo varia de uma forma dramática, já que depende de uma série de fatores. 2.3.2. Análise do domínio. Nesta fase, é definido, de uma forma precisa e concisa, o modelo da organização, relevante para o sistema. É neste processo que, é ganho um detalhe de conhecimento do domínio necessário para que o sistema cumpra as funções exigidas. Durante esta fase, são executados os seguintes passos: · Definição das classes, que inclui a identificação dos objetos, bem como a sua definição; · Definição de relações, que inclui a descrição das associações entre os objetos; · Procura de atributos, que inclui a determinação dos dados que descrevem as classes e determinam quais as classes que necessitam da informação para as descrever; · Definição da herança, que inclui a procura de generalizações e especializações, dentro de cada tipo de domínio; · Definição de operações, que inclui a identificação das principais operações necessárias para suportar a estrutura das classes e as funções do sistema; · Validação e iteração, que inclui a revisão, teste e reparo do modelo; Este processo é altamente iterativo, isto é, podem ser encontrados relacionamentos de herança, quando são descobertos os atributos. 2.3.3. Desenho. Nesta fase, são determinadas, de uma forma eficaz, eficiente e de baixo custo, implementações físicas que cumpram a funcionalidade pretendida, bem como o armazenamento dos dados definidos na análise do domínio. É feito o mapeamento da análise lógica, feita durante a fase de análise de domínio, para uma estrutura que permita que os objetos e as classes sejam passíveis de codificação e execução. É feito um refinamento contínuo e estendido ao modelo conseguido durante a fase de análise de domínio através de: · Determinação da arquitetura inicial, que inclui a tomada de decisões a um nível elevado de implementação de recursos e serviços, as categorias de classes a utilizar e o protótipo a ser desenvolvido; · Determinação do desenho lógico, que inclui a especificação de todos os tipos de dados e estruturas, operações detalhadas, e decisões de acesso aos objetos; · Mapeamento para a implementação física, que inclui o desenvolvimento do interface e o mapeamento de input/output para esses serviços através de classes intermediárias; · Refinamento do desenho, que inclui o refinamento do protótipo e modificação do desenho para conseguir alguns requisitos de desempenho; Tal como a análise, o desenho também é um processo iterativo e incremental. Pode ser necessário voltar à análise, quando são encontradas ambigüidades e omissões. A iteração também é feita pelos passos desta fase, à medida que os protótipos vão sendo construídos, novas partes do sistema são integradas e protótipos existentes vão sendo estendidos, de forma a construir o sistema completo. 2.4. Ferramentas de suporte. Este método é suportado pelas seguintes ferramentas, entre outras: System Architect, de New York; Paradigm Plus, de Houston; Palladio Software, de Wisconsin; Rational Rose, da California; ObjectMaker, da California; Semaphore Tools, de Massachusetts; 3. Jacobson Visão Global Geral do Método: Conceitos. Processos. Análise de Requisitos Análise de Robustez Desenho Implementação Teste Técnicas. Ferramentas de Suporte. Este método é adequado para o desenvolvimento de software numa escala industrial. É uma aproximação à análise e desenho orientado aos objetos, centrando-se na compreensão da forma como o sistema está e será utilizado antes de ser feita a alteração pretendida. Organizando a análise e o desenho através de seqüências de interações com o utilizador e utilizando cenários, o método produz sistemas bastante robustos e amigáveis, adaptando as alterações de uma forma suave. O método divide o desenvolvimento em processos, que, diferentemente das fases de desenvolvimento tradicionais, podem ser iterados e sobrepostos. Estes processos produzem uma série de modelos interligados, facilitando posteriormente a junção dos mesmo através de uma modelação consistente. Todos os sistemas se alteram durante o seu ciclo de vida, tendo a manutenção dos mesmos um peso muito grande em termos de custos de desenvolvimento. Muitos métodos de desenvolvimento adequam-se a novos desenvolvimentos, mas tratando as revisões do modelo de uma forma pouco adequada. Os desenvolvimentos iniciais devem ser vistos como uma atividade importante, estabelecendo uma arquitetura e uma filosofia que constitui a base do sistema. O método é descrito como um conjunto de processos que interagem entre si durante o desenvolvimento do projeto, através de diferentes modelos. Os processos principais são a análise, construção e teste. No processo de análise é criado um modelo Conceitual do sistema a construir. Nesta fase, os modelos são desenvolvidos de forma a facilitar a compreensão do sistema, privilegiando a comunicação com o cliente. No processo de construção, é desenvolvido o sistema a partir dos modelos criados anteriormente. A construção inclui o desenho e a implementação, resultando um sistema completo. O processo de teste integra os componentes do sistema, testando-o e decidindo se está pronto a ser distribuído ao cliente. O conhecimento do sistema aumenta sucessivamente à maneira que o trabalho vai progredindo. Quando se está trabalhando na primeira versão de desenvolvimento do sistema, surgem novos requisitos e outros são alterados. Deste modo o sistema não pode ser completamente desenvolvido, até que as especificações dos requisitos se mantenham constantes. Uma forma de resolver este problema é através de desenvolvimento incremental. Na prática, o sistema é dividido em partes, correspondentes aos serviços requeridos pelo cliente. Cada novo estágio de desenvolvimento estende o sistema com nova funcionalidade até que o produto esteja terminado. De tal modo que, esta estratégia incremental providencia um grande feedback durante o processo de desenvolvimento, resultando na execução dos processos várias vezes durante a mesma versão do sistema. Nesta aproximação, o modelo de use cases é o modelo principal no qual todos os outros modelos são derivados. Um modelo de use cases descreve a funcionalidade completa do sistema, identificando como é que o sistema externo interage com o sistema. O modelo de use cases é a base das fases de análise, construção e teste. O objetivo da análise é compreender o sistema de acordo com os seus requisitos funcionais. Os objetos são encontrados, organizados e as suas interações são descritas. As operações dos objetos e a visão interna é descrita durante a fase de análise. É importante que os objetos sejam encontrados na fase de análise. Na fase de teste, o sistema é verificado, significando que a correção do sistema é verificada de acordo com a sua especificação. 3.1. Conceitos. O primeiro modelo desenvolvido é a especificação dos requisitos. Consiste num modelo orientado ao caso em estudo, com descrições do interface com o utilizador, e um modelo dos objetos do domínio. Este modelo é baseado nos conceitos de actors e use cases. Deste modo, define-se o que existe fora do sistema e o que deve ser executado pelo sistema, respectivamente. Os actors podem ser pessoas ou máquinas. Cada um deles pode executar um número diferente de tarefas, bem como participar na operação do sistema. Executa uma seqüência de transações em permanente diálogo com o sistema. A uma seqüência específica é denominado use case. Cada use case é uma forma específica de utilizar o sistema. O conjunto de use cases corresponde aos requisitos funcionais do sistema a construir. O modelo de requisitos facilita a discussão sobre os mesmos e as preferências do sistema, de forma a assegurar a definição correta do mesmo. Para suportar o modelo de use cases é apropriado o desenvolvimento de modelos de interface, tendo os protótipos um papel importante na sua simulação. Adicionalmente, para comunicar com os utilizadores potenciais e obter uma concordância estável das descrições dos use cases, um modelo lógico dos objetos do domínio é apropriado. O modelo de requisitos formula as especificações dos requisitos funcionais baseadas nas necessidades dos utilizadores do sistema. Uma vez definido e aprovado o modelo de requisitos, o desenvolvimento do sistema passa para a construção do modelo. Este modelo define a estrutura lógica do sistema, independentemente do ambiente de implementação selecionado. Embora possa ser utilizado o modelo de objetos anteriormente definido na fase de análise, como base para a implementação, não resulta numa estrutura robusta. O modelo de análise representa uma estrutura bastante mais estável e fácil de posterior manutenção. Este método reconhece três tipos diferentes de objetos, separando o controle e coordenação, da funcionalidade relacionada com o interface e das entidades que personificam os conceitos da área de aplicação. Desta forma, os tipos de objetos utilizados são entity, interface e control. Cada tipo de objeto tem um objetivo diferente. Os objetos entity modelam a informação do sistema que deve ser utilizada por um determinado período de tempo. Todo o comportamento ligado a esse tipo de informação deve ser colocado neste tipo de objetos. Os objetos interface modelam o comportamento e informação que depende do atual interface do sistema. Os objetos control modelam a funcionalidade que não é genérica, mas específica a um ou mais use cases, tipicamente comportamento que consiste em operações sobre diferentes objetos entity, executando algumas computações e retornando o resultado para um objeto interface. A distribuição da funcionalidade desta forma ajuda ao isolamento das alterações, facilitando a posterior manutenção. Este método não obriga à criação de um modelo rígido baseado nestes tipos de objetos. O resultado é um modelo robusto de uma aplicação, isto é, um sistema que é, fundamentalmente, mais fácil de ser adaptado a extensões e alterações, junto com conjuntos de componentes que são mais facilmente reutilizáveis. Assim, futuras alterações ao sistema, não afetam a estrutura lógica do mesmo. O modelo de desenho é entendido como um refinamento e formalização do modelo de análise. O principal objetivo é adaptar o modelo à implementação. Podem-se encontrar neste modelo, os seguintes conceitos: Actor - Define um papel que um determinado utilizador pode representar na sua troca de informação com o sistema; Role - O papel é definido pelas operações dos objetos; User - Pessoa que atualmente utiliza o sistema. É uma instância da classe actor; Use Case - Curso completo de eventos especificando as ações entre o utilizador e o sistema; Object - É caracterizado pelas operações e estado associado; Entity Object - Informação que é guardada por um largo período de tempo, mesmo quando uma use case é completada; Interface Object - Objeto que contém funcionalidade da use case que interage diretamente com o ambiente; Control Object - Objeto que modela a funcionalidade que não está em qualquer outro objeto; Central Interface Object - Objeto de interface que contém outros objetos de interface; Attribute - Contém informação e o seu tipo; Class - Grupo de objetos que têm similar comportamento e estruturas de informação; Stimulus - Um evento que é enviado de um objeto para outro, iniciando uma determinada operação; Message - Estímulo intra-processo, isto é, uma chamada normal dentro de um processo; Signal - Estímulo intra-processo, mas uma chamada dentro de dois processos; Operation - Atividade dentro de um bloco que pode levar a uma mudança de estado do objeto correspondente; Subsystem - Grupo definido de objetos, de forma a estruturar o sistema; Na seguinte figura mostram-se as várias notações para os conceitos anteriormente descritos. Clique aqui para ver a Figura! 3.2. Processos. 3.2.1. Análise de requisitos. Os modelos desenvolvidos descrevem o que é que o sistema faz. A base deste modelo são os requisitos do cliente ou utilizador final, expressos de uma determinada forma. É a compreensão por parte do analista, daquilo que o cliente quer, que deve ser expressada, podendo servir como um contrato entre o analista e o cliente. Este modelo consiste, normalmente, no modelo de use case com as descrições de interface e o modelo dos objetos do domínio em questão. O sistema é descrito como um número de use cases que são executados pelos actors. Os actors constituem aquilo que é externo ao sistema a ser desenvolvido, podendo os limites do sistema serem estabelecidos. Cada um deles executa uma ou mais tarefas. Uma vez definido aquilo que está fora do sistema, a funcionalidade interna pode ser definida através da especificação de use cases. Os actors são a principal ferramenta para encontrar use cases. Um use case é uma forma específica de utilizar alguma da funcionalidade total do sistema. De forma a encontrar use cases podem ser colocadas questões como: Quais são as principais tarefas de cada actor ? O actor tem de informar o sistema acerca de alterações externas? Um dos princípios fundamentais do desenho de interface é a manutenção de consistência entre a visão Conceitual do utilizador do sistema e o comportamento atual do mesmo. Utilizando um modelo dos objetos do domínio como uma base Conceitual para definir as construções semânticas do sistema e interfaces, poder-se-á chegar a uma consistência entre o interface com o utilizador e a sua perspectiva lógica. Como os use cases focam certas áreas funcionais da aplicação, é possível fazer o seu desenvolvimento para diferentes áreas, juntando-as mais tarde para completar o modelo de requisitos do sistema. 3.2.2. Análise de robustez. Uma vez desenvolvido o modelo de requisitos e aprovado pelo cliente, pode-se centrar os esforços no desenvolvimento propriamente dito e na estruturação do sistema. O modelo desenvolvido é totalmente orientado para a aplicação e não para a tecnologia de implementação, como seja o sistema operativo, sistema de gestão da base de dados, distribuição dos processadores, ou configuração do hardware. É assim gerado um modelo Conceitual de configuração do sistema, consistindo nas várias categorias de objetos. O objetivo deste modelo é encontrar a robustez necessária e a estrutura do sistema como uma base para a construção. Cada tipo de objeto tem o seu objetivo especial para a robustez, e juntos, oferecem a funcionalidade total especificada no modelo de requisitos. Toda a funcionalidade que está diretamente dependente do ambiente do sistema é colocado nos objetos de interface, sendo através destes objetos que os actors comunicam com o sistema. A tarefa dos objetos de interface é traduzir as ações dos actors em eventos do sistema. Para modelar a informação que o sistema deve manipular ao longo do tempo são utilizados os objetos entity. Quando o comportamento não pode ser colocado de uma forma natural nestes dois tipos de objetos, são colocados nos objetos do tipos control. Geralmente, são objetos relacionados com seqüências específicas a um ou mais use cases, de forma a isolar os outros dois tipos de objetos, sendo, assim, facilitada a manutenção do sistema. Uma outra forma de facilitar esta tarefa é alocar um objeto do tipo control para cada actor. Os objetos identificados são agrupados em packages, servindo como input para a fase de desenho. Estes packages reduzem a complexidade e estrutura do sistema, de forma a facilitar posteriores desenvolvimentos e manutenção do próprio sistema. Um package, no nível mais baixo, deve estar relacionado apenas com um actor. Adicionalmente, todos os objetos com relacionados funcionalmente devem ser colocados no mesmo package. Outro critério para a divisão é a redução ou minimização da comunicação entre os diferentes packages. 3.2.3. Desenho. A processo de construção é composto pelo desenho, implementação e teste. Isto não significa que tenha de se esperar que todas as partes estejam construídas antes de iniciar a certificação do sistema, pelo contrário, tenta-se fazer o mais possível em paralelo. O processo de construção deve ser ativado muito antes do processo de análise ter sido completado. O modelo de desenho é um refinamento e formalização do modelo de análise, onde as conseqüências do ambiente da implementação têm de ser tidas em conta. O modelo de análise não é suficientemente formal, e, em vez de estarmos permanentemente a alterar o código fonte, procedesse ao refinamento dos objetos, às operações que eles devem oferecer, às comunicações exatas que devem ter entre eles, que estímulos é que são enviados, etc. No entanto, se são descobertos pontos pouco claros em relação ao modelo de análise ou dos requisitos, devem ser feitas as clarificações necessárias, voltando ao processo de análise. O modelo de desenho define explicitamente os interfaces dos objetos, bem como as semânticas das operações. Este modelo reflete decisões relacionadas com as bases de dados, linguagens de programação, distribuição, etc.. É composto por packages e design objects. Os design objects vão ser mais tarde implementados em código fonte e, neste caso, estão abstraídos da futura implementação. Depois de identificar a arquitetura do sistema, deve ser descrito como é que os packages e os design objects comunicam. Para cada use case concreto deve ser desenhado um diagrama de Interação, descrevendo como é que o use case é realizado através da comunicação entre os packages, que recebem e enviam estímulos. Estes estímulos, incluindo os parâmetros enviados, devem ser definidos. Os diagramas de transição de estado podem ser utilizados num nível intermédio, antes de iniciar a implementação. Estes diagramas formam uma descrição simplificada, aumentando a compreensão dos design objects ou de uma package, sem estarem dependentes da linguagem de programação selecionada. Estes diagramas descrevem que estímulos podem ser recebidos e o que acontece quando o estímulo é recebido. 3.2.4. Implementação. A implementação dos packages em código fonte pode iniciar-se quando o interface do package estabilizar. Em muitos casos, cada design object corresponde exatamente a uma classe, tornando fácil o mapeamento. Em alguns casos, algumas classes podem ser implementadas para um dado design object, para implementação dos atributos, utilização de componentes ou separação de funcionalidade comum em classes abstratas. Por conseguinte, cada design object continuará a ter um mapeamento no código, mas suportando várias classes que foram adicionadas. Sempre que possível, duas classes não devem oferecer funcionalidade semelhante, a não ser que estejam relacionadas por um mecanismo de herança. Por exemplo, se metade das operações de uma classe acederem a metade das variáveis de instância e as outras operações acederem às outras variáveis, devem ser consideradas duas classes. Existem outras heurísticas para um bom desenho das classes, incluindo o julgamento da reutilização potencial das mesmas. É evidente que demora muito tempo a desenhar boas classes, e nem sempre é eficiente desenvolver todas as classes do sistema o mais genéricas possível. Note-se que é mantida uma linha de ação desde a fase de análise até ao código fonte. Quando se lê o código fonte, pode-se descobrir diretamente o que é que o originou no modelo de requisitos. 3.2.5. Teste. Nesta fase, é feita a verificação do sistema construído em termos de correção. Uma aproximação disciplinada e bem organizada ao desenvolvimento de sistemas resulta da necessidade de aumentar a qualidade do sistema, diminuindo os custos da fase de teste. Na metodologia, as atividades de teste são executadas inteiramente durante o processo de desenvolvimento. As instâncias de diferentes classes devem ser integradas continuamente ao longo do processo. O conceito de use case é crucial para a integração, com um use case em cada momento a ser integrado. Os planos de teste podem ser feitos muito cedo, juntos com a identificação e especificação dos use cases. 3.3. Técnicas. Uma das técnicas utilizadas são os diagramas de Interação, denominados message sequence charts pelo CCITT. Descrevem de que maneira é que cada use case funciona, através da Interação dos diversos objetos. Também é representado o interface com tudo aquilo que está fora do diagrama, como sejam os actors externos, que podem corresponder a interfaces externos do sistema. A descrição das seqüências é feita de uma forma textual, tendo estas construções a facilidade de migração para a linguagem de programação a implementar. Além desta técnica, são utilizadas outras bastante divulgadas noutros métodos de desenvolvimento orientados aos objetos. 3.4. Ferramentas de suporte. Este método disponibiliza várias configurações, cada uma delas para casos específicos de desenvolvimento. Assim, o processo e as ferramentas são utilizados de acordo com os casos específicos de desenvolvimento. Clique aqui para ver a Figura! DIAGRAMA DE ANÁLISE gerado numa ferramenta CASE Clique aqui para ver a Figura! DIAGRAMA DE TRANSIÇÃO DE ESTADOS gerado numa ferramenta CASE Clique aqui para ver a Figura! 4. Coad/Yourdon. Visão Global Geral: Conceitos Notação Arquitetura Níveis Múltiplos Processos Técnicas Pesquisa das Classes e Objetos Relacionamento entre Classes e Objetos Identificação dos Tópicos ou Temas Definição dos Atributos Definição dos Serviços Ferramentas de Suporte Vários relatórios indicam este método como adequado para o desenvolvimento de sistemas segundo o paradigma dos objetos. Esse mérito é devido à sua capacidade de integração entre as fases de análise, desenho e implementação, bem como à facilidade de compreensão subjacente ao método em questão. São mencionados 4 grandes benefícios na utilização deste método em relação aos métodos de análise tradicionais: Melhoramento da Interação entre o analista e o cliente do sistema; Aumento da consistência interna entre análise, desenho e codificação; Reutilização; Providencia uma representação consistente entre as diversas fases; De acordo com Coad/Yourdon, o maior benefício da análise OO, resulta da redução da complexidade de um determinado problema. Segundo os autores deste método, uma aproximação OO consiste de classes, objetos, herança e comunicação via mensagens. O principal objetivo deste método é permitir a construção de sistemas eficazes, providenciando notações, práticas e estratégias para a prossecução do mesmo. Trata-se de um método de desenvolvimento orientado ao objeto com um domínio de aplicação bastante alargado, incidindo na manipulação de classes e objetos respeitantes ao domínio do problema, no interface homem-máquina e na gestão das tarefas inerentes ao processo de desenvolvimento. Disponibiliza uma extensão que abarca o conceito de reengenharia, de forma a ser possível desenvolver um modelo com base no redesenho dos processos organizacionais. 4.1. Conceitos. Os conceitos associados ao método são evidentes na notação disponibilizada, na arquitetura, níveis múltiplos, padrões e requisitos de qualidade, que serão apresentados de seguida. 4.1.1. Notação. A notação utilizada personifica os princípios inerentes ao paradigma dos objetos, tais como: Classe & Objeto - Denomina uma classe e os objetos existentes para essa classe. Inclui uma seção de atributos, sendo, nos sistemas de tempo real, divididos em atributos de estado, atributos dependentes do estado e atributos independentes do estado. Inclui também uma seção de serviços, sendo, nos sistemas de tempo real, divididos em serviços dependentes do estado e serviços independentes do estado; Classe - Representa uma classe sem objetos associados, isto é, uma classe abstrata; Atributos - Informação de estado para o qual cada objeto de uma determinada classe tem os seus próprios valores; Estruturas de especialização/generalização - As classes de especialização herdam os atributos e serviços definidos nas classes de generalização, estando patente o conceito de herança; Estruturas Todo/Partes - Um objeto pode ser constituído, estruturalmente, por outros tipos de objetos; Mensagens - Mecanismo de pedido de execução de determinados serviços; Temas - Mecanismo de partição de sistemas complexos; Serviços - Comportamento específico que um determinado objeto é responsável; Estado - Valor dos atributos de um objeto; Transição - Alteração de estado; A notação gráfica para todos estes conceitos encontra-se explicitada na figura seguinte: Clique aqui para ver a Figura! 4.1.2. Arquitetura. Este método utiliza uma arquitetura de desenvolvimento constituída por quatro componentes principais: · Interação humana; · Domínio do problema; · Gestão de tarefas de tempo real; · Gestão dos dados; São aplicadas estratégias específicas para desenvolver cada um dos componentes, sendo possível a comunicação entre eles através de padrões de broadcast. Por exemplo, se o domínio do problema for alterado, os outros componentes recebem o broadcast, respondendo de uma determinada forma a essa alteração. 4.1.3. Níveis múltiplos. Com níveis múltiplos, pode-se selecionar o nível de complexidade que se pretende visualizar, ou trabalhar, num determinado momento. Assim, consegue-se reduzir a complexidade através de um modelo multifacetado. Os cinco níveis existentes são os seguintes: · Tópicos ou temas - Mostra os grupos de classes principais, para atribuição a equipas de trabalho ou para facilidade na revisão do modelo; · Classes & Objetos - Mostra as classes e os respectivos objetos; · Estruturas - Mostra as estruturas de generalização/especialização, bem como as estruturas de todo/partes; · Atributos - Mostra aquilo que cada objeto, numa determinada classe, conhece, isto é os seus próprios atributos e as ligações de atributos com outros objetos; · Serviços - Mostra aquilo que cada objeto faz na respectiva classe, isto é os seus próprios serviços e as mensagens que envia para obter serviços de outros objetos; O método utiliza o conceito de padrão como um grupo de classes e objetos que podem ser aplicados em mais que um domínio de problemas, permitindo uma modelação da fase de análise e desenho mais eficazes. Os constrangimentos de qualidade incluem, entre outros, consistência e disponibilidade dos objetos em questão. Podem ter um determinado alcance para um objeto, classe, cenário ou sistema. Os cenários representam um papel importante nesta área, especialmente nos sistemas de tempo real. 4.2. Processos. O desenvolvimento de software requer criatividade e inovação. Os métodos encorajam e promovem a criatividade e inovação, bem como as vantagens competitivas no mercado global. Este método consiste em atividades, e não passos. A fase de análise é modelada através da manipulação das classes respeitantes ao domínio do problema. A fase de desenho é modelada através da Interação humana, gestão de tarefas, e gestão de dados. As fases de análise e desenho podem ser executadas concorrentemente. Assim, este método é constituído por 5 grandes atividades: Procura de classes e objetos; Identificação de estruturas; Identificação de temas; Definição de atributos; Definição de serviços; Este método adequa-se ao desenvolvimento efetuado por equipas inter-disciplinares, como por exemplo: · Duas equipas para fazer a análise do domínio; · Duas equipas dedicadas à análise; · Uma equipa dedicada à Interação humana, na fase de desenho; · Uma equipa dedicada à gestão eficaz dos dados, na fase de desenho; · Uma equipa dedicada à gestão eficaz das tarefas, especialmente para os sistemas de tempo real; 4.3. Técnicas. O método de Coad/Yourdon suporta um conjunto amplo de atividades e estratégias associadas. Estas atividades e estratégias são aplicadas à fase de análise, isto é, ao domínio do problema, e à fase de desenho, isto é, à Interação humana, gestão de tarefas e gestão de dados. O comportamento dinâmico das classes pode ser capturado nos diagramas de estado dos objetos. Os algoritmos que são aplicados nos serviços são descritos nos diagramas de serviços. A ligação entre os serviços e o estado dos objetos é estabelecida por uma tabela Serviços/Estado. Todos estes tipos de diagramas não são descritos de uma forma extensiva. Os passos descritos a seguir não têm de ser feitos de uma forma seqüencial. Para sistemas complexos, é aconselhável fazer-se o refinamento do problema em temas preliminares, seguindo-se depois a identificação dos objetos e respectivas classes. 4.3.1. Pesquisa das classes e objetos. Um objeto é uma pessoa, um lugar ou um fato. Uma classe é uma descrição daquilo que o objeto conhece e faz. As estratégias de pesquisa são encaradas segundo três perspectivas: 1. Para onde olhar; ##O que olhar; ##O que considerar, estabelecendo um critério de filtragem, que decide que classes e objetos incluir no modelo; Para encontrar potenciais classes e objetos, podem-se ter em atenção os seguintes componentes: Estruturas; Dispositivos; Coisas ou eventos; Papeis desempenhados; Procedimentos operacionais; Localizações físicas; Unidades organizacionais; 4.3.2. Relacionamentos entre classes e objetos. Os relacionamentos denotam uma determinada forma de organização das classes e objetos entre si: Estruturas de generalização/especialização, que denotam a existência da partilha de características comuns entre as classes, quer dos atributos, quer dos serviços que disponibilizam; Relacionamentos entre classes definem uma hierarquia de herança para classes que são especialização de outras classes. Uma classe pode herdar de uma só classe (herança simples) ou de mais de uma classe (herança múltipla); Estruturas de todo/partes - que personificam o conceito de composição em termos estruturais. Nesta estrutura, um objeto é composto por vários objetos. Este relacionamento pode ter associado valores de cardinalidade, tais como nos diagramas de entidades-relacionamentos; Ligações de Mensagens - Modelação de dependência de processamento de um objeto,

indicando a necessidade de determinados serviços de forma a completar as suas responsabilidades; 4.3.2. Identificação dos tópicos ou temas. Esta atividade pretende agrupar classes com ligações de estrutura, indicando um relacionamento forte entre essas classes. A identificação dos tópicos permite uma melhor compreensão do domínio do problema e uma maior simplicidade na construção do interface. A seleção de temas possíveis é feita pela promoção da classe de mais alto nível. O refinamento dos temas é feito pela procura de interdependências w interações mínimas entre classes e objetos de diferentes temas. De seguida os temas podem ser construídos e adicionados ao diagrama de análise. 4.3.4. Definição dos atributos. Um atributo é uma qualidade ou característica de um determinado objeto, sendo dele a sua responsabilidade. Podem ser atribuídos constrangimentos aos atributos, tais como unidades de medida, limites, enumeração de valores, precisão, valor por defeito, constrangimentos de acesso, constrangimentos baseados no valor de outros atributos, etc. Depois de serem identificados os atributos, devem ser identificadas as ligações de instância entre os objetos. Isto é feito adicionando linhas de ligações entre os objetos, refletindo mapeamentos com o domínio do problema. 4.3.5. Definição dos serviços. Um serviço denota aquilo que um determinado objeto faz. O primeiro passo na definição dos serviços é identificar os estados dos objetos através da descrição dos estados e dos objetos no diagrama de estados dos objetos. De seguida, os serviços podem ser identificados para cada Classe e objeto. As tabelas Serviços/Estados podem ser boas ferramentas para mostrar as ligações entre serviços e estados. Os serviços simples não são denotados no diagrama de análise. Os serviços podem ser subdivididos da seguinte forma: · Funções básicas, tais como criar, alterar, remover e consultar; · Comportamento dependente do estado, isto é, que muda ao longo do tempo; · Comportamento de um objeto como resposta a um evento; 4.4. Ferramentas de suporte. Este método é suportado por ferramentas oriundas de uma variedade de vendedores de todo o mundo. Existem centenas de produtos conhecidos. O método foi utilizado num grande número de problemas dos domínios mais diversos tais como bancos, CASE, manufatura de produtos, administração governamental, administração militar, seguros, integração de sistemas, telecomunicações, transportes, etc. 5. Shlaer-Mellor. Visão Global Geral: Processos Benefícios Ferramentas de Suporte Trata-se de um método bem definido e disciplinado, orientado para o desenvolvimento de software a uma escala industrial. É baseado no paradigma dos objetos, tendo sido desenvolvido à 11 anos atrás no ambiente de projetos do mundo real. Estes projetos incluíam manufatura e aplicações de controle de processos, operações bancárias, telecomunicações, e aplicações de defesa governamental. Este método inclui um conjunto de modelos de análise orientados ao objeto, que podem ser simulados para verificação, e uma aproximação inovadora à fase de desenho, denominado Recursive Design(RD), que produz o desenho do sistema através da translação dos modelos de análise. O modelo de informação de objetos é baseado no modelo relacional dos dados. Os modelos de estado são autômatos deterministicos finitos. Os diagramas de fluxos de dados mostram a seqüência, podendo cada processo ser definido numa linguagem formal. As regras e a formalidade permitem que os modelos sejam matematicamente transformados noutros tipos de modelos. Por exemplo, a estrutura relacional da definição dos dados pode ser transformada numa lista ligada, árvore, ou um conjunto de listas baseado em classes equivalentes. O proce La programación orientada a objetos o POO (OOP según sus siglas en inglés) es un paradigma de programación que usa los objetos en sus interacciones, para diseñar aplicaciones y programas informáticos. Está basado en varias técnicas, incluyendo herencia, cohesión, abstracción, polimorfismo, acoplamiento y encapsulamiento. Su uso se popularizó a principios de la década de los años 1990. En la actualidad, existe una gran variedad de lenguajes de programación que soportan la orientación a objetos. Características de la POO Existe un acuerdo acerca de qué características contempla la "orientación a objetos". Las características siguientes son las más importantes: -Abstracción Denota las características esenciales de un objeto, donde se capturan sus comportamientos. Cada objeto en el sistema sirve como modelo de un "agente" abstracto que puede realizar trabajo, informar y cambiar su estado, y "comunicarse" con otros objetos en el sistema sin revelar cómo se implementan estas características. Los procesos, las funciones o los métodos pueden también ser abstraídos, y, cuando lo están, una variedad de técnicas son requeridas para ampliar una abstracción. El proceso de abstracción permite seleccionar las características relevantes dentro de un conjunto e identificar comportamientos comunes para definir nuevos tipos de entidades en el mundo real. La abstracción es clave en el proceso de análisis y diseño orientado a objetos, ya que mediante ella podemos llegar a armar un conjunto de clases que permitan modelar la realidad o el problema que se quiere atacar. -Encapsulamiento Significa reunir todos los elementos que pueden considerarse pertenecientes a una misma entidad, al mismo nivel de abstracción. Esto permite aumentar la cohesión de los componentes del sistema. Algunos autores confunden este concepto con el principio de ocultación, principalmente porque se suelen emplear conjuntamente. -Modularidad Se denomina modularidad a la propiedad que permite subdividir una aplicación en partes más pequeñas (llamadas módulos), cada una de las cuales debe ser tan independiente como sea posible de la aplicación en sí y de las restantes partes. Estos módulos se pueden compilar por separado, pero tienen conexiones con otros módulos. Al igual que la encapsulación, los lenguajes soportan la modularidad de diversas formas. -Principio de ocultación Cada objeto está aislado del exterior, es un módulo natural, y cada tipo de objeto expone una interfaz a otros objetos que especifica cómo pueden interactuar con los objetos de la clase. El aislamiento protege a las propiedades de un objeto contra su modificación por quien no tenga derecho a acceder a ellas; solamente los propios métodos internos del objeto pueden acceder a su estado. Esto asegura que otros objetos no puedan cambiar el estado interno de un objeto de manera inesperada, eliminando efectos secundarios e interacciones inesperadas. Algunos lenguajes relajan esto, permitiendo un acceso directo a los datos internos del objeto de una manera controlada y limitando el grado de abstracción. La aplicación entera se reduce a un agregado o rompecabezas de objetos. -Polimorfismo Comportamientos diferentes, asociados a objetos distintos, pueden compartir el mismo nombre; al llamarlos por ese nombre se utilizará el comportamiento correspondiente al objeto que se esté usando. O, dicho de otro modo, las referencias y las colecciones de objetos pueden contener objetos de diferentes tipos, y la invocación de un comportamiento en una referencia producirá el comportamiento correcto para el tipo real del objeto referenciado. Cuando esto ocurre en "tiempo de ejecución", esta última característica se llama asignación tardía o asignación dinámica. Algunos lenguajes proporcionan medios más estáticos (en "tiempo de compilación") de polimorfismo, tales como las plantillas y la sobrecarga de operadores de C++. -Herencia Las clases no están aisladas, sino que se relacionan entre sí, formando una jerarquía de clasificación. Los objetos heredan las propiedades y el comportamiento de todas las clases a las que pertenecen. La herencia organiza y facilita el polimorfismo y el encapsulamiento, permitiendo a los objetos ser definidos y creados como tipos especializados de objetos preexistentes. Estos pueden compartir (y extender) su comportamiento sin tener que volver a implementarlo. Esto suele hacerse habitualmente agrupando los objetos en clases y estas en árboles o enrejados que reflejan un comportamiento común. Cuando un objeto hereda de más de una clase se dice que hay herencia múltiple. -Recolección de basura La recolección de basura o garbage collection es la técnica por la cual el entorno de objetos se encarga de destruir automáticamente, y por tanto desvincular la memoria asociada, los objetos que hayan quedado sin ninguna referencia a ellos. Esto significa que el programador no debe preocuparse por la asignación o liberación de memoria, ya que el entorno la asignará al crear un nuevo objeto y la liberará cuando nadie lo esté usando. En la mayoría de los lenguajes híbridos que se extendieron para soportar el Paradigma de Programación Orientada a Objetos como C++ u Object Pascal, esta característica no existe y la memoria debe desasignarse expresamente. Hosting: Drive (Direct Download for Desktop and Mobile) Archivo: Pdf Idioma: Español