Metodología de la Programación

Sesión 1:

Tipos de Lenguajes

- Naturales: (español, inglés, etc.) dependen del contexto, por ejemplo, necesito tener un nexo común con el otro hablante, porque con ese nexo, consigo poder hablar con esa otra persona. Porque la manera de interpretar a dos personas de diferente contexto puede ser erróneo. El contexto es algo muy volátil, dos personas cercanas con el aparente mismo contexto, puede variar. Para que esto no suceda, necesito establecer un contexto, el problema es que no puedo hacer esto con un ordenador.

- Formales: (lenguajes de programación) son la solución al problema del lenguaje natural, porque en este lenguaje no existe el contexto de antes.

Dentro de los Lenguajes de Programación:

1)Lenguaje Maquina

2)Lenguaje Ensamblador

3)L. Alto Nivel

4)L. Orientados a Problemas: describen la solución, no cómo conseguirla

¿Necesitamos más lenguajes?

La respuesta es NO, pero Si, como las tecnologías se implantan y se desechan a gran velocidad, cuando hago un lenguaje para un determinado tipo de tecnología, puede que después ya no sea tan útil o bueno como algún nuevo lenguaje que haya sido creado específicamente para ese tipo de lenguaje. Es decir, si los problemas evolucionan, es entendible que los programas también lo hagan.

Traductores, Intérpretes y Compiladores

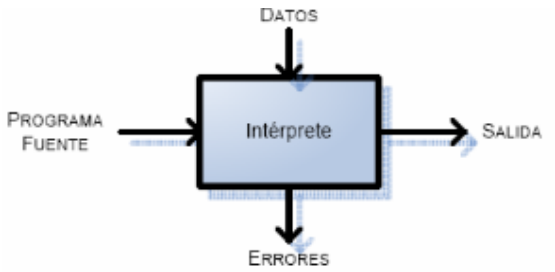
Traductor: Un traductor es un programa que lee un programa escrito en lenguaje fuente de alto nivel, y lo traduce a un lenguaje objeto.

El traductor tiene dos funcionalidades:

1)Intérprete: Ejecuta directamente lo que traduce, sin almacenar en disco la traducción realizada

1. Cada vez que se escribe una línea el programa comprueba si es correcta, si lo es, la ejecuta.

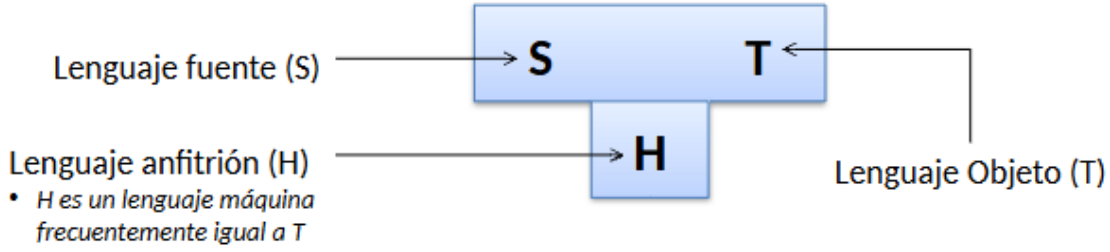
2. La ejecución es interactiva.
3. Los intérpretes más puros no guardan copia del programa que se está escribiendo



2)Compilador: Genera un fichero del lenguaje objeto estipulado, que puede posteriormente ser ejecutado tantas veces se necesite sin volver a traducir. Además, el compilador informa de los errores del programa fuente

- El programa ejecutable, una vez creado, no necesita el compilador para funcionar

Los compiladores pueden estar implementados en lenguajes de programación distintos del lenguaje fuente (Source) y del lenguaje objeto (Target). A este tercer lenguaje se le denomina lenguaje anfitrión (Host).



(Ver pdf imágenes)

Entornos de Ejecución

Procesamiento de un Lenguaje

Sesión 2:

(Parte Labo)

(explicación de la forma de trabajo en GitHub, su funcionamiento)

Comandos: commit, push, pull, fetch.

Funcionamiento: en mi fichero le doy a guardar, es más, guarda cada cambio, pero lo guarda en el fichero todo el rato.

1º comando, “commit”, en el Git local, marca un conjunto de cambios que funcionan o que son lo suficientemente relevantes para mí, como para guardarlos. Guarda una versión nueva en el Git local por si llegará a perder el trabajo, por poner un ejemplo. Pero si por algún casual, me explota el ordenador, yo necesito sacar la información del Git local a otro Git, y eso se logra con

2º comando, “push”, toma las versiones locales y las manda al servidor, pero solo manda trabajos terminados.

3ºcomando, “pull”, Mi amigo pepe se quiere descargar mi trabajo del servidor, lo hace con “pull”.

4ºcomando, “fetch”, tu no sabes si tu amigo que tiene diferente forma horaria, no sabes que está haciendo, para coordinarlo se utiliza fetch, se salta todo lo anterior, y solo informa de lo que hay, pero no te descarga nada.

En el trabajo en grupo, lo que debemos hacer es un pull antes de hacer cualquier otra cosa, porque si yo genero cambios, y lo subo y me compañero también ha realizado cambios, pero no se descarga mis cambios, le generará un error. De esta forma, puedo ver cómo evoluciona en proyecto

Operación “merge” si me equivoco, puedo volver a unir dos versiones

Ej.

Yo hago algo: 1ab79c “commit”, lleva una descripción < 50 caracteres

Mi compañero hace otra versión: 7bc139

Hago un pull y me descargo los cambios, hago otra versión: 9ba7cc

Mi amigo hace otra versión, sin verificar que se han hecho cambios: 9ba7cc.1

También está el comando “Branch”, para hacer una segunda versión (otra rama) si se tiene la intención de trabajar en ello.

(Parte Teoría)

Escuelas de Pensamiento: (Las llamamos diferentes debido a la forma de abordar los problemas)

- Programación Funcional

  ¡¡¡Viva las Mates!!!, se aplican funciones y se anidan, solo se aplican métodos matemáticos. Cuando se usa, encontramos un modelo abstracto,

- Programación Orientada a Objetos

  Plantear una solución en base a una idea (de un problema). La idea general de uso me permite acceder de manera rápida a cada idea, y ya decido yo hasta donde llegar

(características)

Abstracción (de las ideas)

Reutilización (de código)

Encapsulamiento – Principio de Ocultación

Modularidad (separación del código en partes)

Herencia y Polimorfismo (son herramientas de diseño)

La herencia nos permite en el momento del diseño, como establecer las relaciones de las clases. EL polimorfismo nos permite: si existen dos clases con la misma funcionalidad (pila y cola), tiene la misma finalidad,

Recolección de Basura

Cunado creo objetos y luego los destruyo, debe quedar un registro, son funcionalidades que sirven para cuando creo un objeto nuevo, y lo apunta en una tabla virtual de la cual desconocemos. Se ejecuta cada cierto tiempo (x milisegundos) que hace una pasada al programa para buscar objetos que han dejado de utilizarse. Ahora bien, esta recolección no es estrictamente necesario, ya que puedes hacer esto tú mismo, pero actualmente todos los lenguajes lo utilizan.

Clase: definición de cómo se va a comportar un objeto

Objetos: son implementaciones de las clases

Métodos: funcionalidad de la clase

Mensaje: invocación de un método

Evento: mensaje extremo

Cuando un usuario toca un ordenador, sucede lo que se llama “evento”, se da un mensaje que se envía al método. Cualquier cosa que suceda fuera del ordenador, se llama evento, porque es externo.

Atributo/propiedad: Dato, variable de clase

Identificador/referencia

Estado Interno (conjunto de valores)

Las posiciones en memoria son una referencia, que es un identificador. Cuando veo dos objetos (o1 y o2) se diferencia por su estado interno, los valores de sus atributos. Si dos objetos tuvieran el mismo conjunto de valores, aunque sus referencias son distintas, podemos decir que uno es una copia del otro

Hay muchos tipos de diagramas distintos. Vamos a ver el diagrama de Casos de Usos, D. de Clases, D. Secuencias, D. Estado, D. Componentes

Fases canónicas: análisis (establecimiento de requisitos y se estudia con D. de Casos de Usos), diseño, programación y prueba.

El d. clase,

El d. secuencia, te da la idea feliz de como tendría que salir todo (ideal sin errores).

D. estados, se utiliza para aquellos objetos con un estado interno especial, el estado modifica el comportamiento del objeto

D. componentes, nos da a alto nivel los módulos agrupados y como interactuan entre ellos.

(libros)

Lenguaje unificado de modelado Grady Booch

Utilización de UML Booch, jacobson y rumbaugh

Sesión 4;

(LAB)

Usar “package” que son los “paquetes” que se usan para dividir un trabajo.

(TEORIA)

Paquetes, anotaciones, herencia (clases abstractas), interfaces, Test Unitario

(paquetes)

Cuando tengo algo en público es accesible desde fuera, pero uno protegido, yo puedo determinar quién puede acceder al archivo. Existen tres tipos, públicos, privados y protegidos.

Tiene la siguiente estructura;

package es.uah.matcomp.mp.el1.eja.e1 (donde cada punto determina donde se guarda)

Public class M.Class {

Protected int datos; (solo clases de mi paquete pueden acceder)

}

(anotaciones)

@override “declaro que voy a sobrecargarlo”

Public String to String

Nos sirven para decirle a java, que tenga cuidado al utilizar algo con mis anotaciones, sirve para redefinirlo. Todas las clases de objetos, todas heredan de object.

(Herencias, Clases Abstractas)

Nombre Clase

Privado – Dato 1

Protegido – Dato 2

Público - Dato 3

Método 1

Método 2

Método 3

Ahora definimos;

Clase Vehículo

- Matrícula

Puedo acceder a la matrícula con:

- getMatrícula()

- setMatrícula(M)

Como todos los vehículos no usan ruedas, por poner un ejemplo. Lo que vamos a hacer ahora es reutilizar el código, es decir la “Herencia”, para luego ajustarlo a cada caso. La herencia se marca con un Triángulo. Con estos nuevos casos, genero hijos que se ajustan cada vez más a ciertas especificaciones, como:

1 V. Ruedas 2 V. Volador 3 V. Marítimo

- Número Ruedas

Get ruedas()

SetRuedas(M)

Ahora puedo hacer

V_Ruedas V1: new V.Ruedas

V1.setMatricula(“M7145FJ”)

En código puedo hacer;

```
Public class Vehículo {
    Private String matricula;
    Public String getMatricula() {
        Return matricula;
    }
    Public void getMatricula(String m) {
        This.matricula = m;
    }
}
```

(para añadir el nº ruedas)

```
Public class V_Ruedas extends Vehículo {
    Private mt n.ruedas;
    Public int getNRuedas() {
       Return n.ruedas;
   }
    Public void setRuedas(int n) {
        This.n.ruedas = n;
    }
}
```

En la clase Vehículo, no puedo hacer new, porque nunca puedes instanciarlo.

Hay otra cosa llamada Herencia múltiple, es decir, una clase como anfibios, puede tomar herencia de la clase terrestres y acuáticos. Pero hay programas donde esto no funciona. Por ejemplo, en Java, no se puede hacer esto, al hacer (extends “”) entre comillas solo puedes poner un extends. Entonces, ¿cómo se soluciona? Con los interfaces.

(interfaces)

```
Interface terrestre { //Firma de métodos
    Public getEdad();
}
Interface acuatico {
    Public get.....();
}
Class anfibio implements terrestre, acuatico {
    Public getEdad() {
    }
}
Terrestre t1 = new Perro ()
```

(Test Unitario)

(mucho texto, ha explicado todo por ejemplo en el programa de IntelliJ)

Sesión 5;

Multiplicidad o Cardinalidad

Asociación: Agregación: Composición

En la relación de Persona y Coche, puedo marcar el mínimo en 0, -1, 1 por ejemplo. Pero hay que tener en cuenta que este minimo no es igual para otro tipo de relacion, como puede ser en nº de brazos que tiene la persona. Entonces vamos a usar 0 como no relacion, 1 como si relacion, y n para decir que existen multiples relaciones. Si el max es 1, no hay multiplicidad, es una relacion 1-1, o puedo tener n, donde n son más de un coche, y n me dice que existe una multiplicidad.

Me surgue el siguiente conjunto de posibilidades: (0,1), (0,n), (1,1), (1,n)

Class Persona

(1 ,1)PoseeCoche

(1,n)PoseeCoche[ ] -> los corchetes son un Array y sirve para introducir más valores

(Si tenemos PoseeCoche[] y EsposeidoPor[], entonces tenemos una relacion (n,m))

Class Coche

La relación de Agregación(por ejemplo el Author del apartadoB) nos dice que existe algún tipo de propiedad “común creo que ha dicho”

Mi clase coche, tiene la clase asientos, ruedas, motor, ...

Sesión 6 (14/02)

Tomo apuntes en libro, es más fácil cuando dibuja.