UNIVERSIDAD PRIVADA-DE-TACNA

INGENIERIA DE SISTEMAS

TITULO:

INFORME DE LABORATORIO No 01

CURSO:

BASE DE DATOS II

DOCENTE(ING):

Patrick Cuadros Quiroga

Integrantes:

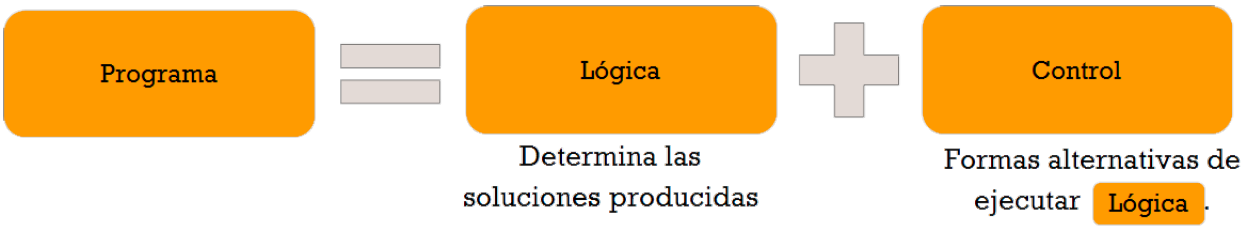
Orlando Antonio Mostacero Ortiz	(2015052775)
Orestes Ramirez Ticona	(2015053236)
Marlon Xavier Villegas Arando	(2015053890)
Nilson Laura Atencio	(2015053846)
Roberto Zegarra Reyes	(2010036175)
Richard Cruz Escalante	(2013047247)

Índice

1. Paradigma No 03 – Programación Logica

1. Programacion Logical:

   La programación lógica es un paradigma que se encuentra dentro del paradigma de la programación funcional. Aunque no es tan conocido como otros paradigmas de programación, es realmente interesante. Se basa en la declaración de hechos y reglas que permiten ir creando lo que para nosotros sería el conocimiento. Aunque inicialmente sea un poco complejo de entender, la programación lógica trabaja de forma muy similar a los humanos en cuanto al manejo de información y conocimientos se refiere. Veamos un ejemplo para entender mejor cómo es la declaración de las reglas y hechos Paradigma de programación basado en la lógica de primer orden. La Programación Lógica estudia el uso de la lógica para el planteamiento de problemas y el control sobre las reglas de inferencia para alcanzar la solución automática. La Programación Lógica, junto con la funcional, forma parte de lo que se conoce como Programación Declarativa, es decir la programación consiste en indicar como resolver un problema mediante sentencias, en la Programación Lógica, se trabaja en una forma descriptiva, estableciendo relaciones entre entidades, indicando no como, sino que hacer, entonces se dice que la idea esencial de la Programación Lógica es



   Asume que partimos de un conjunto de hechos y reglas conocidos. Solamente es la declaración del componente lógico de un algoritmo. El sistema desarrolla el componente de control de secuencia. En este paradigma, la evaluación asume que cuando se selecciona una regla, es porque ésta es la única posibilidad o la necesaria para resolver el problema. Es decir, se encuentra una solución si un conjunto de reglas adecuado y las sustituciones a dichas reglas producen un conjunto de reglas aterrizadas (sin variables libres), suficientes para deducir el resultado de los hechos conocidos. La programación lógica intenta resolver lo siguiente: Dado un problema S, saber si la afirmación A es solución o no del problema o en que casos lo es. Además queremos que los métodos sean implantados en maquinas de forma que la resolución del problema se haga de forma automática La programación lógica: construye base de conocimientos mediante reglas y hechos

   Características

2. Los programas para los lenguajes de programación lógicos son un conjunto de hechos y reglas.
3. La sintaxis de los lenguajes de programación lógicos es notablemente diferente de los lenguajes de programación imperativos.
4. Unificación de términos.

5. Mecanismos de inferencia automática.
6. Recursión como estructura de control básica.
7. Visión lógica de la computación.
8. La aplicación de las reglas de la lógica para inferir conclusiones a partir de datos.
9. El programa se transforma en un conjunto de declaraciones formales de especificaciones que deben ser correctas por definición.
10. No tiene un algoritmo que indique los pasos que detallen la manera de llegar a un resultado.
11. Las salidas son funcionalmente dependientes de las entradas.

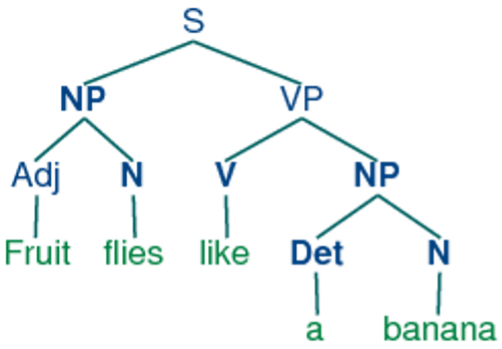
    Ventajas y Desventajas del uso de este paradigma
    Ventajas

12. Puede mejorarse la eficiencia modificando el componente de control sin tener que modificar la lógica del algoritmo.
13. Relaciones multipropósito.
14. Simplicidad.
15. Generación rápida de prototipos e ideas complejas.
16. Sencillez en la implementación de estructuras complejas.
17. Potencia.
    Desventajas

18. Altamente ineficiente.
19. Pocas áreas de aplicación
20. No existen herramientas de depuración efectivas.
21. En problemas reales, es poco utilizado.
22. Si el programa no contiene suficiente información para contestar una consulta responde false.

    Ejemplo

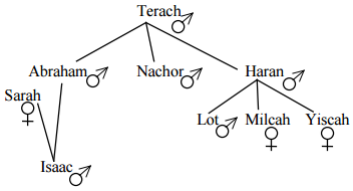
    De forma tradicional imperativa tendríamos que especificar como lo haremos




```
oracion(O) :- sintagma_nominal(SN),
              sintagma_verbal(SV),
              append(SN,SV,O).
sintagma_nominal(SN) :- nombre(SN).
sintagma_nominal(SN) :- articulo(A),
                        nombre(N),
                        append(A,N,SN).
sintagma_verbal(SV) :- verbo(V),
                       sintagma_nominal(SN),
                       append(V,SN,SV).

articulo([el]).
nombre([gato]).
nombre([perro]).
nombre([pescado]).
nombre([carne]).
verbo([come]).
```

```
7 ?- oracion([perro,come,pescado]).
true .
```

Ejemplo Un conjunto de hechos constituye un programa (la forma más simple de programa lógico) que puede ser visto como una base de datos que describe una situación. Por ejemplo, el Programa 1 refleja la base de datos de las relaciones familiares que se muestran en el siguiente gráfico.

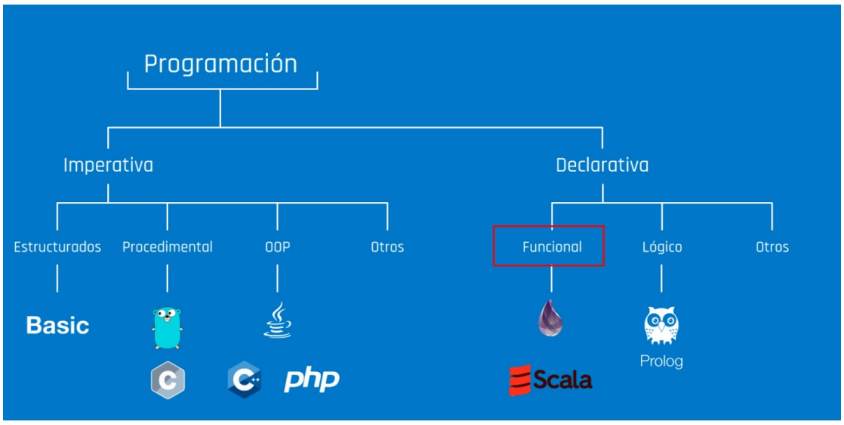



```
es_padre(terach, abraham).
es_padre(terach, nachor).
es_padre(terach, haran).
es_padre(abraham, isaac).
es_padre(haran, lot).
es_padre(haran, milcah).
es_padre(haran, yiscah).

es_madre(sarah, isaac).

es_hombre(terach).
es_hombre(abraham).
es_hombre(nachor).
es_hombre(haran).
es_hombre(isaac).
es_hombre(lot).

es_mujer(sarah).
es_mujer(milcah).
es_mujer(yiscah).

es_hijo(X,Y):- es_padre(Y,X), es_hombre(X).

es_hija(X,Y):- es_padre(Y,X), es_mujer(X).

es_abuelo(X,Z):- es_padre(X,Y), es_padre(Y,Z).
```

```
2 ?- es_padre(haran,lot), es_hombre(lot).
true .

3 ?- es_padre(abraham,lot), es_hombre(lot).
false.

4 ?- es_padre(abraham,X), es_hombre(X).
X = isaac.

5 ?- es_padre(haran,X), es_mujer(X).
X = milcah ;
X = yiscah.

6 ?- es_hijo(lot,haran).
true .

7 ?- es_hija(X,haran).
X = milcah ;
X = yiscah.
```

2. Paradigma No 04 – Programación Funcional

1. Programacion funcional:
   La Programación Funcional es una forma en la cual podemos resolver diferentes problemáticas ya que estaremos trabajando principalmente con funciones, evitaremos los datos mutables, así como el hecho de compartir estados entre funciones.



Las funciones serán tratadas como ciudadanos de primera clase. Las funciones podrán ser asignadas a variables además podrán ser utilizadas como entrada y salida de otras funciones

Ejemplo
De forma tradicional imperativa tendríamos que especificar como lo haremos

```
List<Integer> numeros = Arrays.asList(18, 6, 4, 15, 55, 78, 12, 9, 8);

//Imperativo
int contador = 0;
for(int i=0; i<numeros.size(); i++) {
int  numero = numeros.get(i);
if(numero > 10) {
    contador ++;
}
}
System.out.println(contador);
```

Pero de forma declarativa o programación funcional tendríamos

```
List<Integer> numeros = Arrays.asList(18, 6, 4, 15, 55, 78, 12, 9, 8);

//Declarativo
Long result = numeros.stream().filter( n -> n>10).count();
System.out.println(result);
```

3. Paradigma No 06 – PRINCIPIO DE DISEÑO SOLID

1. INTRODUCCION:

   Solid es un acrónimo inventado por Robert C. Martin para establecer los cinco principios básicos de la programación orientada a objetos y diseño. Este acrónimo tiene bastante relación con los patrones de diseño, en especial, con la alta cohesión y el bajo acoplamiento. El objetivo de tener un buen diseño de programación es abarcar la fase de mantenimiento de una manera más legible y sencilla, así como conseguir crear nuevas funcionalidades sin tener que modificar en gran medida código antiguo.

   El objetivo de tener un buen diseño de programación es abarcar la fase de mantenimiento de una manera más legible y sencilla, así como conseguir crear nuevas funcionalidades sin tener que modificar en gran medida código antiguo.

2. Responsabilidad simple:

   Este principio trata de destinar cada clase a una finalidad sencilla y concreta. En muchas ocasiones estamos tentados a poner un método reutilizable que no tienen nada que ver con la clase simplemente porque lo utiliza y nos pilla más a mano. En ese momento pensamos "Ya que estamos aquí, para que voy a crear una clase para realizar esto.

   El problema surge cuando tenemos la necesidad de utilizar ese mismo método desde otra clase. Si no se refactoriza en ese momento y se crea una clase destinada para la finalidad del método, nos toparemos a largo plazo con que las clases realizan tareas que no deberían ser de su responsabilidad.

3. Abierto/Cerrado:

   Es decir, el diseño debe ser abierto para poderse extender, pero cerrado para poderse modificar. Aunque dicho parece fácil, lo complicado es predecir por donde se debe extender y que no tengamos que modificarlo. Para conseguir este principio hay que tener muy claro cómo va a funcionar la aplicación, por donde se puede extender y cómo van a interactuar las clases.

   El uso más común de extensión es mediante la herencia y la reimplementación de métodos. Existe otra alternativa que consiste en utilizar métodos que acepten una interface de manera que podemos ejecutar cualquier clase que implemente ese interface. En todos los casos, el comportamiento de la clase cambia sin que hayamos tenido que tocar código interno.

4. Sustitucion Liskov:

   Este principio fue creado por Barbara Liskov y habla de la importancia de crear todas las clases derivadas para que también puedan ser tratadas como la propia clase base. Cuando creamos clases derivadas debemos asegurarnos de no Re implementar métodos que hagan que los métodos de la clase base no funcionases si se tratasen como un objeto de esa clase base.

5. SEGREGACION DEL INTERFACE:

   Este principio fue formulado por Robert C. Martin y trata de algo parecido al primer principio. Cuando se definen interfaces estos deben ser específicos a una finalidad concreta. Por ello, si tenemos que definir una serie de métodos abstractos que debe utilizar una clase a través de interfaces, es preferible tener muchos interfaces que definan pocos métodos que tener un interface con muchos métodos. El objetivo de este principio es principalmente poder reaprovechar los

interfaces en otras clases. Si tenemos una interface que compara y clona en el misma interface, de manera más complicada se podrá utilizar en una clase que solo debe comparar o en otra que solo debe clonar.

6. INVERSIÓN DE DEPENDENCIAS:

   También fue definido por Robert C. Martin. El objetivo de este principio conseguir desacoplar las clases. En todo diseño siempre debe existir un acoplamiento, pero hay que evitarlo en la medida de lo posible. Un sistema no acoplado no hace nada, pero un sistema altamente acoplado es muy difícil de mantener. El objetivo de este principio es el uso de abstracciones para conseguir que una clase interactúe con otras clases sin que las conozca directamente. Es decir, las clases de nivel superior no deben conocer las clases de nivel inferior. Dicho de otro modo, no debe conocer los detalles. Existen diferentes patrones como la inyección de dependencias o servicie locator que nos permiten invertir el control.