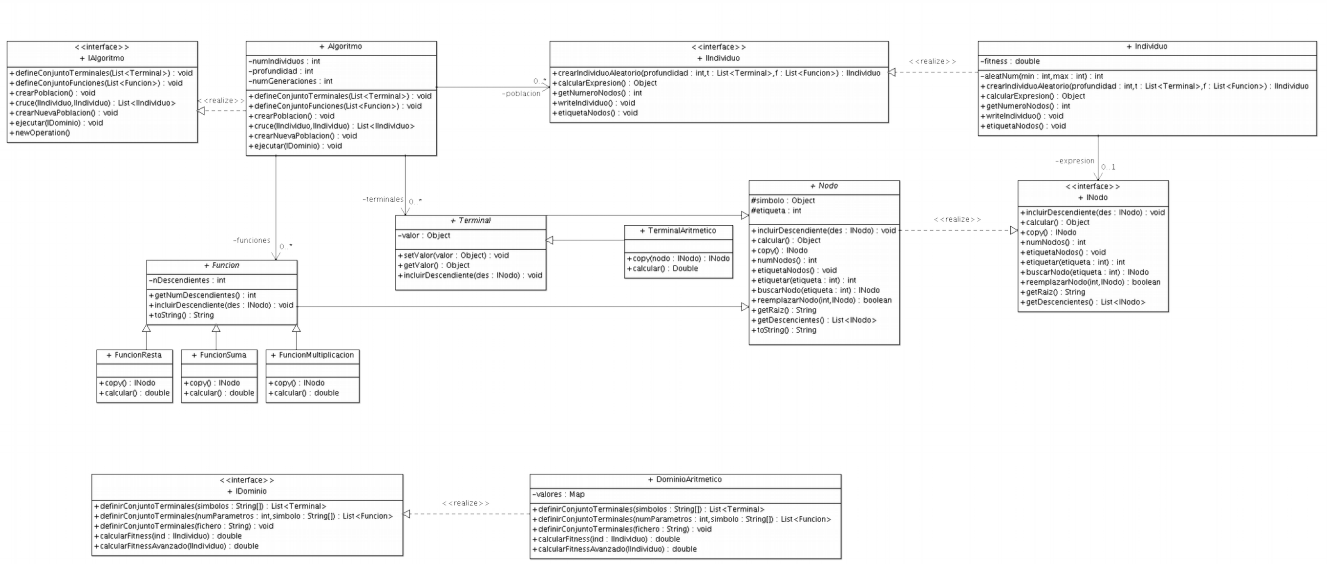
Javier Delgado del Cerro
Javier López Cano

PRACTICA 4

Diagrama de clases

En esta práctica hemos desarrollado un algoritmo genético que nos permite calcular un polinomio que pasa por una serie de puntos guardados en un fichero.
El diagrama de clases del proyecto seria el siguiente:



Como se puede ver, tanto la interfaz *IDominio* como la clase *DominioAritmetico* no estan unidas con nada, pues se inicializan en los testers al ser usadas.
No hemos incluido en dicho diagrama, los testers ni las excepciones, pues constan simplemente de un main los testers, y un *toString* en el caso de las excepciones, y el añadirlo “ensuciaba” el diagrama.

Respecto a la extensibilidad del diseño, hemos intentado, como se aprecia en el diagrama, implementar todo de la forma más genérica posible, mediante el uso de interfaces y clases abstractas siempre que fuese posible, y usando el polimorfismo carácterístico de Java.
Así, el unico sitio donde no tenemos una clase abstracta de por medio, es en *DominioAritmetico*, pues cada *IDominio* va a trabajar con un tipo de valores, de terminales, de funciones, etc, y por tanto no tenía sentido programar una implementacion por defecto que no sería de ninguna utilidad.
Hemos usado tambien *Object* como valores para los terminales, de forma que nuestra aplicación puede trabajar con Integer, *Double*, *Byte*, *Boolean*, etc.

Javier Delgado del Cerro
Javier López Cano

Decisiones de diseño

En primer lugar, dado que las funciones usadas en esta practica eran unicamente de una variable, y para simplificar el codigo, hemos implementado el metodo calcular de forma que todos los *TerminalAritmetico* tienen el mismo valor, una variable estática que se inicializa al llamar a *Terminal.setValor(valor);*
Sin embargo, y como hemos dicho anteriormente, al ser el valor un objeto de tipo *Object*, podriamos definir un tipo de *Terminal* que usase una lista o un mapa para dicho valor, de forma que podemos admitir tambien distintas variables.

Tuvimos problemas a la hora de medir el fitness de un individuo para calcular cuales eran los mejores, pues con el algoritmo proporcionado, simplemente se tiene en cuenta que la expresion se acerque en algunos puntos a los datos dados, y no si en otros puntos se aleja excesivamente, por ejemplo.

Para solucionar esto, hemos implementado otro metodo en la interfaz *IDominio* llamado *calcularFitnessAvanzado*, de forma que en *DominioAritmetico*, este metodo calcula y devuelve un promedio de la diferencia entre la expresion y los datos dados, con lo que es mucho mas exacto, y permite clasificar mejor a los *IIndividuos*. Dicha funcion devuelve un double entre 0.0 e infinito, siendo mejor la aproximacion cuanto menor es el valor devuelto, tal y como se explica en la documentacion, y ha sido este metodo el que hemos usado en el algoritmo para cruzar los individuos entre si.

Por esto mismo, en nuestro tester del algoritmo, el fitness va decreciendo, hasta llegar a 0,0 al encontrar el IIndividuo buscado, pero gracias a este algoritmo, hemos conseguido que funcione correctamente la gran mayoria de las veces, en un numero d iteraciones muy pequeño.

Hemos eliminado tambien la posibilidad, en el metodo *cruceAlgoritmo*, de que surga una execepcion *CruceNuloException* porque los dos nodos seleccionados para cruzar sean cero, pues es algo inevitable que sucede de vez en cuando dada la aleatoriedad del programa, y la impresion de que algo ha fallado. Sin embargo, dentro del metodo *cruce* de *PruebaCruce* sigue implementado, con lo que se sigue podiendo probar en la clase *TesterCruce*.

Por último, respecto al funcionamiento de nuestro algoritmo genético en sí, hemos usado una implementacion basada en la selección natural: dada una generacion, solo la mitad de los individuos, aquellos con mejor fitness, sobreviven. De estos, la primera mitad se reproduce con el resto. De esta forma, siempre se van a reproducir los mejores entre si, y mantenemos el numero de la poblacion constante, evitando saturar la RAM.