{desafío} latam_

UML (Parte II)

Diagrama de Clases

Competencias

- Relacionar el diagrama con la POO.
- Reconocer las notaciones como cajas y generalizaciones.
- Construir diagrama de clases.
- Llevar los diagramas de clase a código en Java.

Introducción

Para modelar las clases e interfaces, incluidos sus atributos, operaciones, relaciones y asociaciones con otras clases, el UML proporciona el diagrama de clases, que aporta una visión estática o de estructura de un sistema, sin mostrar la naturaleza dinámica de las comunicaciones entre los objetos de las clases.

El diagrama de clase, además de ser de uso extendido, también está sujeto a la más amplia gama de conceptos de modelado. Aunque los elementos básicos son necesarios para todos, los conceptos avanzados se usan con mucha menor frecuencia. Es por eso que se toman los temas más importantes y suficiente para lograr los objetivos propuestos.

En la herramienta, nos vamos a model -> add diagram -> Class Diagram.

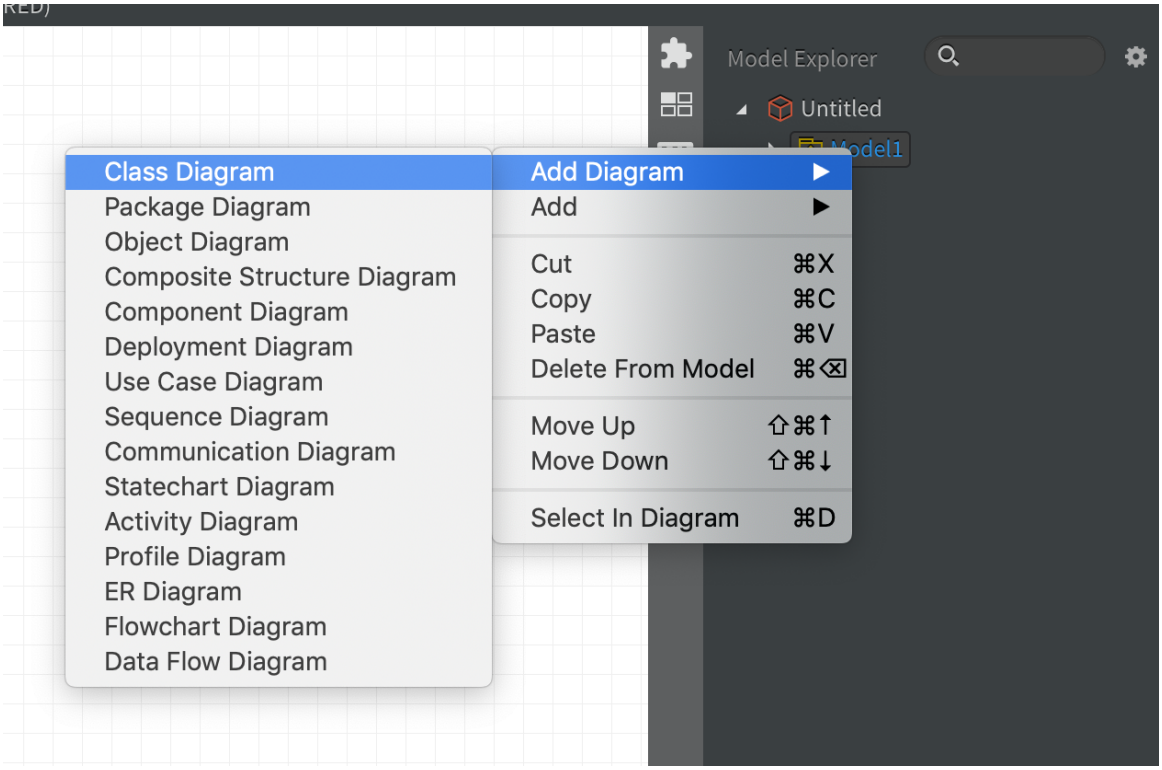

Imagen 1: Agregar Diagrama de Clase.

Componentes de un diagrama de clases

Atributos de una clase

Un atributo es algo que un objeto de dicha clase conoce o puede proporcionar todo el tiempo. Por lo general, los atributos se implementan como campos (variables de instancia) de una clase, pero no necesitan serlo. Podrían ser valores que la clase puede calcular a partir de sus variables o valores de instancia y que puede obtener de otros objetos de los cuales está compuesto. Por ejemplo, un objeto puede conocer siempre la hora actual y regresarla siempre que se solicite. Por tanto sería adecuado mencionar la hora actual como un atributo de dicha clase de objetos. Sin embargo, el objeto muy probablemente no tendría dicha hora almacenada en una de sus variables de instancia, por que necesitaría actualizar de manera continua ese campo. En vez de ello, el objeto probablemente calcularía la hora actual (por ejemplo, a través de consulta con objetos de otras clases) en el momento en el que se le solicite la hora.

Operaciones

Una operación, es lo que pueden hacer los objetos de la clase. por lo general, se implementa como un método de la clase.

Cajas

Elementos principales de un diagrama de clase, son los íconos utilizados para representar clases e interfaces. Cada caja se divide en partes horizontales. La parte superior contiene el nombre de la clase. La sección media menciona sus atributos. La tercera sección del diagrama de clase, contiene las operaciones o comportamientos de la clase.

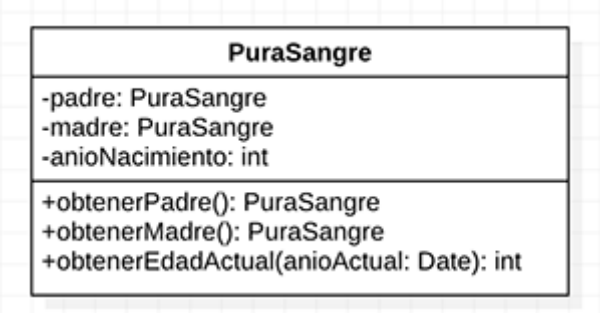

Imagen 2: Representación gráfica de una clase.

La Imagen 2, presenta un ejemplo simple de una clase `PuraSangre`, esta modela caballos de pura sangre y osee tres atributos:

- `padre`.
- `madre`.
- `anioNacimiento`.

Además de tres operaciones:

- `obtenerPadre()`.
- `obtenerMadre()`.
- `obtenerEdadActual()`.

Puede haber otros atributos que no se muestren en el diagrama. Cada atributo, puede tener un nombre, un tipo y un nivel de visibilidad. El tipo y la visibilidad son opcionales. El tipo sigue al nombre y se separa de él mediante dos puntos. La visibilidad se indica, anteponiendo cualquiera de los siguientes símbolos:

- `-` Visibilidad privada.
- `#` Visibilidad protegida.
- `~` Paquete.
- `+` Visibilidad privada.

También es posible, especificar si un atributo es del tipo static subrayándolo.

Para los atributos especificamos el nombre y el tipo de retorno de la siguiente forma:

- `nombre` : `tipoDeRetorno`

En el caso de las operaciones, tenemos estas más opciones:

- `nombreOperacion(nombreParametro:tipo, ...)`
- `nombreOperacion():tipoRetorno`
- `nombreOperacion(nombreParametro:tipo, ...):tipoRetorno`

La implementación de la caja de ejemplo en Java sería la siguiente:

```
package cl.desafiolatam.uml.diagramaclase;

import java.util.Date;

public class PuraSangre {
  private PuraSangre padre;
  private PuraSangre madre;
  private int anioNacimiento;

  public PuraSangre obtenerPadre() {
    // TODO implementar acá
    return null;
  }

  public PuraSangre obtenerMadre() {
    // TODO implementar acá
    return null;
  }

  public int obtenerEdadActual(Date anioActual) {
    // TODO implementar acá
    return 0;
  }

}
```

Vemos que ya podemos comenzar a preparar el código base de nuestra aplicación, incluso la herramienta `starUml` posee una opción de llevar nuestro modelo a código.

Paso 1: Debemos tener nuestro diagrama de clases, en un modelo como indica la siguiente imágen:

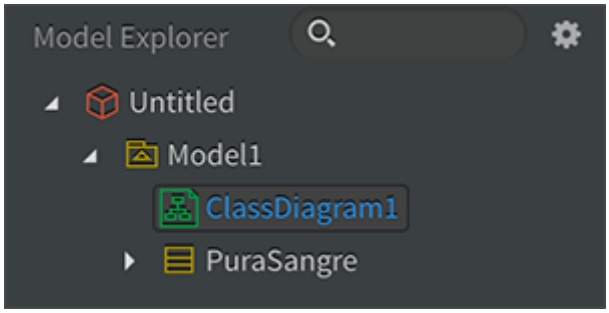

Imagen 3: Paso 1.

Paso 2: Seleccionamos `tools -> Java -> generate code`.

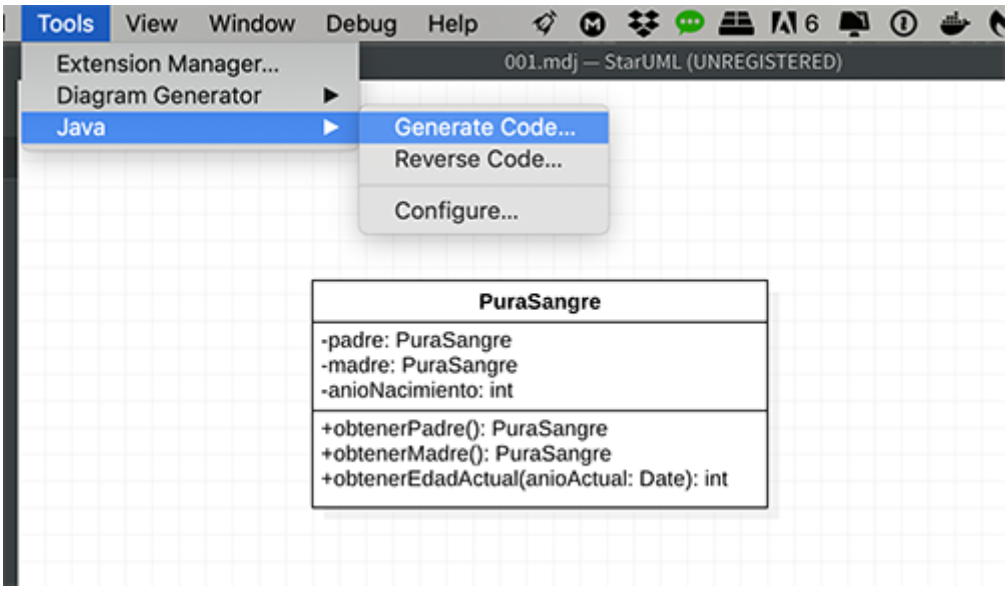

Imagen 4: Paso 2.

Paso 3: Seleccionamos la el modelo y la ubicación en donde lo queremos dejar. Esto genera una carpeta con el nombre del modelo y si revisamos su interior, encontraremos los archivos generados, en este caso es solo uno.

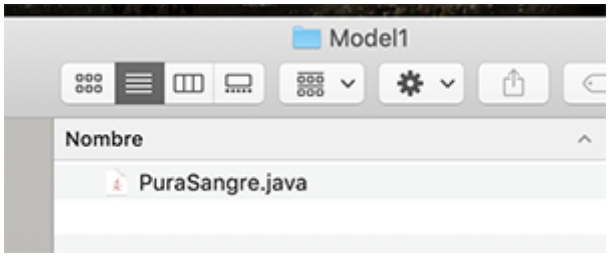

Imagen 5: Paso 3.

Podemos entonces revisar su contenido:

```
import java.util.*;
/**
 *
 */
public class PuraSangre {

    /**
     * Default constructor
     */
    public PuraSangre() {
    }

    /**
     *
     */
    private PuraSangre padre;

    /**
     *
     */
    private PuraSangre madre;

    /**
     *
     */
    private int anioNacimiento;

    /**
     * @return
     */
    public PuraSangre obtenerPadre() {
        // TODO implement here
        return null;
    }

    /**
     * @return
     */
    public PuraSangre obtenerMadre() {
        // TODO implement here
        return null;
    }

    /**
     * @param anioActual
     * @return
     */
    public int obtenerEdadActual(Date anioActual) {
        // TODO implement here
        return 0;
    }
}
```

Generalización (Herencia)

Los diagramas de clase, también pueden mostrar relaciones entre las clases, una clase que sea una subclase de otra clase se conecta con ella mediante una flecha con una línea sólida y con una punta triangular hueca. La flecha apunta de la subclase a la superclase. Podemos relacionar esto con la relación de herencia en la POO.

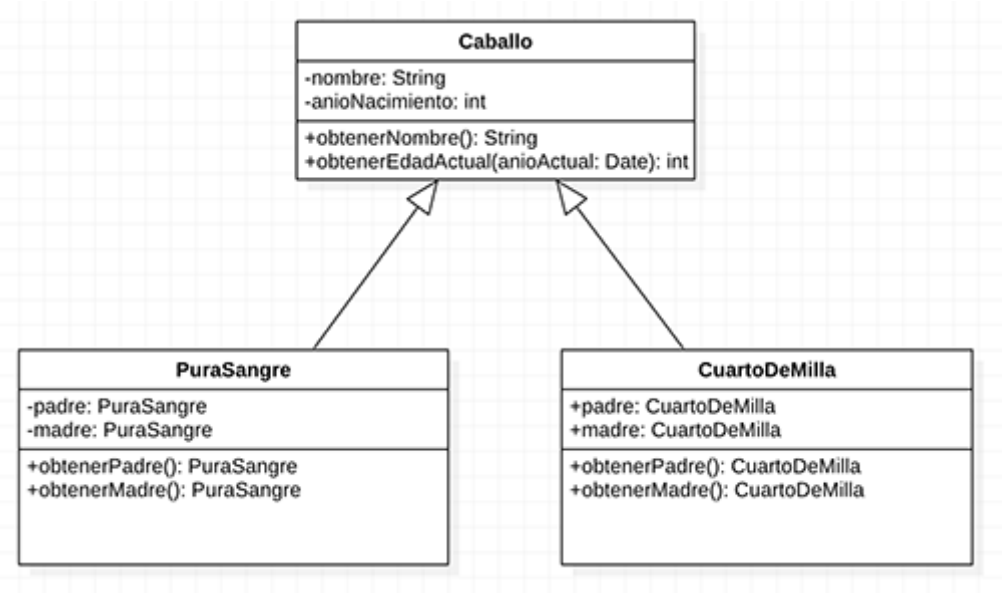

Imagen 6: Ejemplo de generalización (Herencia).

Clase `Caballo`:

```
package cl.desafiolatam.uml.diagramaclase;
import java.util.Date;
public class Caballo {
    private String nombre;
    private int anioNacimiento;

    public Caballo(String nombre, int anioNacimiento) {
      this.nombre = nombre;
      this.anioNacimiento = anioNacimiento;
    }

    public String obtenerNombre() {
        // TODO implementar aquí.
        return "";
    }

    public int obtenerEdadActual(Date anioActual) {
        // TODO implementar aquí.
        return 0;
    }
}
```

Clase `PuraSangre`:

```
package cl.desafiolatam.uml.diagramaclase;
import java.util.Date;
public class PuraSangre extends Caballo {
    private PuraSangre padre;
    private PuraSangre madre;

    public PuraSangre(PuraSangre padre, PuraSangre madre, String nombre, int anioNacimiento) {
      super(String nombre, int anioNacimiento);
      this.padre = padre;
      this.madre = madre;
    }

   public PuraSangre obtenerPadre() {
      // TODO implementar acá
      return null;
   }

    public PuraSangre obtenerMadre() {
      // TODO implementar acá
      return null;
    }
}
```

Clase `CuartoDeMilla`:

```
package cl.desafiolatam.uml.diagramaclase;
import java.util.Date;
public class CuartoDeMilla extends Caballo {
    public CuartoDeMilla padre;
    public CuartoDeMilla madre;

    public CuartoDeMilla(PuraSangre padre, PuraSangre madre, String nombre, int anioNacimiento) {
      super(String nombre, int anioNacimiento);
      this.padre = padre;
      this.madre = madre;
    }

    public CuartoDeMilla obtenerPadre() {
        // TODO implementar aquí.
        return null;
    }

    public CuartoDeMilla obtenerMadre() {
        // TODO implementar aquí.
        return null;
    }
}
```

Implementación de Interfaces en UML.

Podemos expresar además, la relación de implementación de interfaces. Esto nos ayuda a poder diseñar un bosquejo de lo que se pretende construir, de esta forma, podemos hacer un mapa completo de la estructura que tendrán nuestras clases e interfaces además de poder generar el código desde la misma herramienta

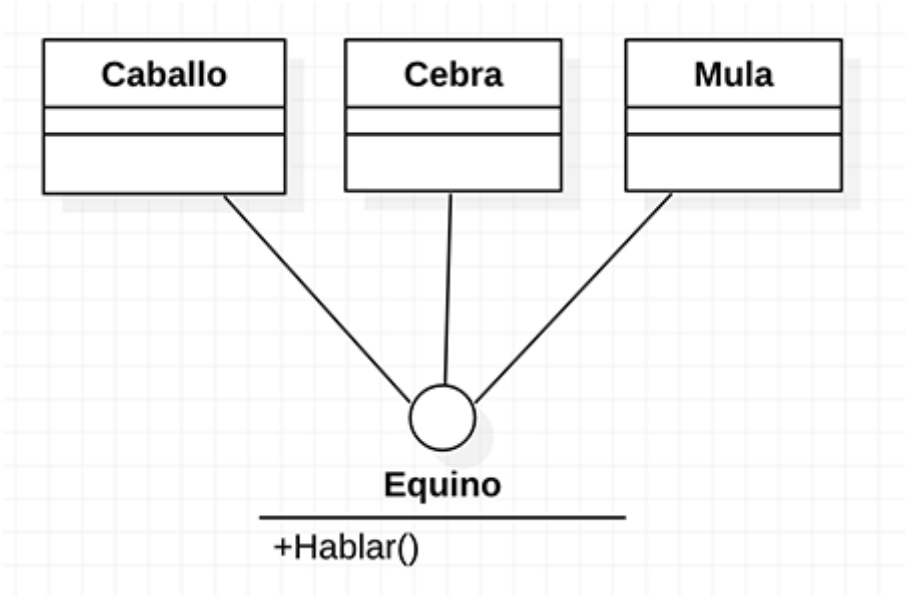

Imagen 7: Ejemplo de interfaces.

Clase: Equino

```
package cl.desafiolatam.uml.interfaces<p style="text-align: justify;">

public interface Equino {

    public abstract void Hablar();

}
```

Clase: Caballo

```
package cl.desafiolatam.uml.interfaces

public class Caballo implements Equino {

    public void Hablar() {
        // TODO implementar aquí.
    }

}
```

Clase: Cebra

```
package cl.desafiolatam.uml.interfaces


public class Cebra implements Equino {

    public Cebra() {}
    @Override
    public void Hablar() {
        // TODO implementar aquí.
    }

}
```

Clase: Mula

```
package cl.desafiolatam.uml.interfaces


public class Mula implements Equino {

    @Override
    public void Hablar() {
        // TODO implementar aquí.
    }

}
```

Podemos generar un código muy similar desde la herramienta, y ver que se auto-generan los métodos que se implementan desde la interfaz. De esta forma, podemos ver que si utilizamos el paradigma de programación orientado a objetos, este diagrama nos proporciona no solamente una idea de como debemos implementar el código, sino que además nos proporciona parte de dicho código.

Los sistemas de software son una de las tecnologías más importantes en todo el mundo. En los últimos 60 años, el software ha pasado a ser la solución de un problema especializado y herramienta de análisis de la información a una industria en sí misma. No obstante, aún hay problemas para desarrollar software de alta calidad a tiempo y dentro del presupuesto asignado.
"Una imagen vale más que mil palabras", pero antes de esto, debemos saber de qué imagen se trata y cuáles son esas mil palabras. Si no logramos generar diagramas que logren comunicar una correcta solución, surge una imagen errónea, que conduce al software erróneo.

Como consejo, se listan los principios del modelado, a lo que debemos apegarnos con la mayor fuerza posible, para que lo aprendido de UML, sea un real aporte al desarrollo de software.

- El equipo de software, tiene como objetivo principal, elaborar software y no modelos.
- Viajar ligero, no crear más modelos de los necesarios.
- Tratar de producir el modelo más sencillo que describa al problema o al software.
- Construir modelos susceptibles al cambio.
- Ser capaz de enunciar un propósito explícito para cada modelo que se cree.
- Adaptar los modelos que se desarrollan al sistema en cuestión.
- Tratar de construir modelos útiles, pero olvidarse de construir modelos perfectos.
- No ser dogmáticos respecto a la sintaxis del modelo. Si se tiene éxito para comunicar contenido, la representación es secundaria.
- Si su instinto dice que un modelo no es correcto a pesar de que se vea bien en el papel, hay razones para estar preocupados.
- Obtener retroalimentación tan pronto como sea posible.

Si se siguen estos principios, nuestros modelos tendrán un mejor aporte y no significaran una pérdida de tiempo, si no que todo lo contrario, por que hay que recordar que *meses de programación, nos ahorran unas horas de planificación*.