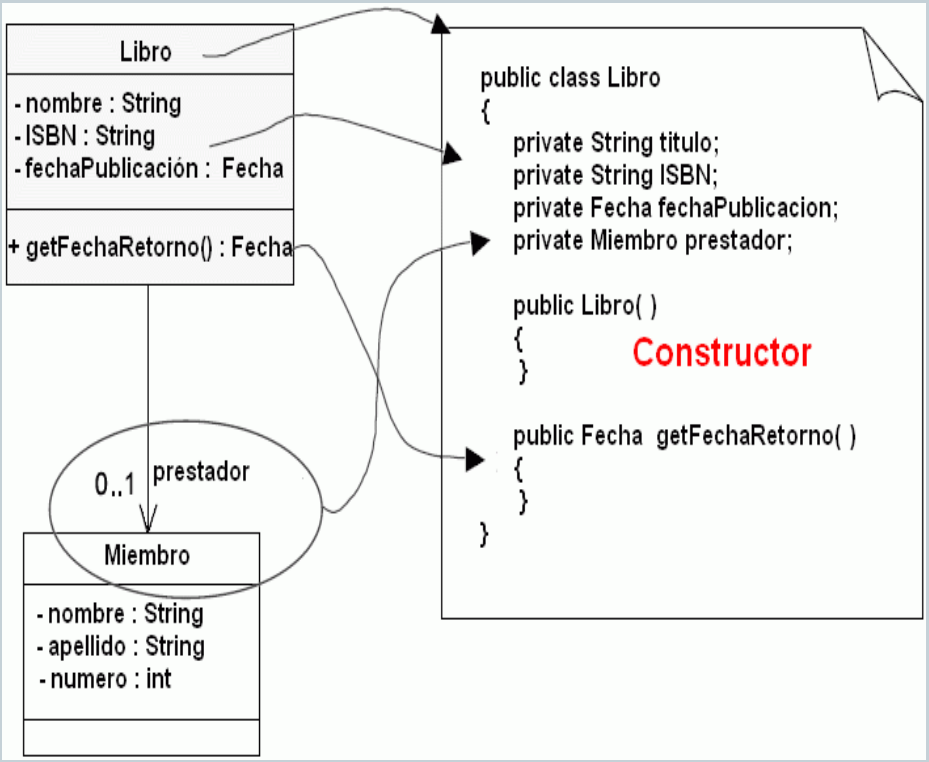
Facultad de Ciencias de la Computación

Codificación del sistema

ABRAHAM SÁNCHEZ LÓPEZ
GRUPO MOVIS
FCC-BUAP

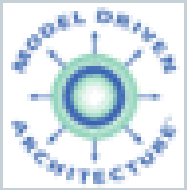
Introducción

- Cómo pasar del modelado a código?



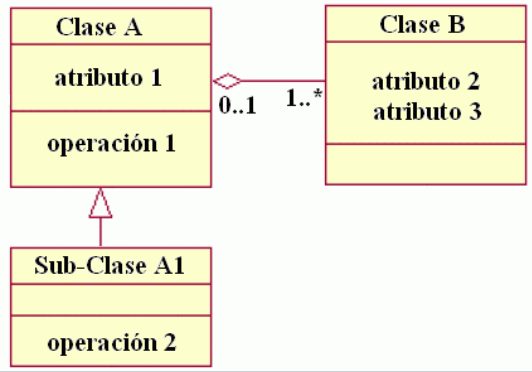
Conceptos básicos, I

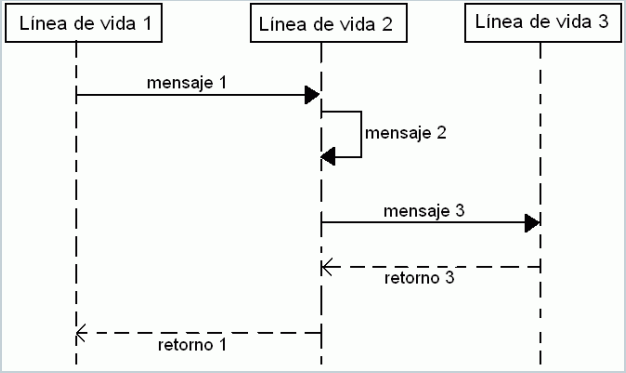
- UML es el lenguaje estándar en la industria para especificar, visualizar, construir y documentar el diseño y la estructura de sistemas de software.
- UML 2.0 define trece tipos de diagramas dentro de las categorías de estructura, de comportamiento y de interacción.
- El OMG (Object Management Group) añade, que modelar es un proceso esencial que asegura la generación de componentes completos y correctos, especialmente para software complejo.
- En 2001 el OMG presentó la Arquitectura Guiada por Modelos (MDA, siglas en inglés) como un enfoque para la MDE.



Conceptos básicos, II

- En el análisis de sistemas bajo el enfoque OO, se producen los diagramas de casos de uso, clase, estado e interacción (secuencia y de comunicación o de colaboración).
- Durante el diseño, estos artefactos son incrementados hasta obtener un formato utilizable por los programadores.
- Los diagramas pueden estar elaborados en exceso o en forma insuficiente.
- Al refinar los diagramas UML durante el modelaje de análisis y de diseño pueden omitirse atributos indispensables para generar código y los diagramas tendrían que completarse para ser procesados por las herramientas que permiten la generación automática de código basadas en UML (GCM).





Características generales de los GCM

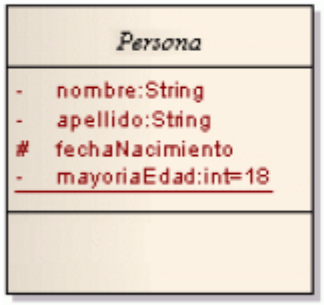
Funcionalidad del GCM
- Generación de código a nivel de producción y pruebas en distintos lenguajes: C, C++, Java, C#, Visual Basic, Perl, MOFM2T, Python, PHP, entre otros. - Procesamiento de diagramas UML para generar código. Unos GCM procesan hasta 14 diagramas UML 2.4; otros no procesan diagramas de objeto, paquetes, temporización, ni de entorno de interacción. - Versión de UML: 2.4, 2.1, 2, 1.3. - "Round-trip" para crear y ajustar diagramas a partir del código. - Generación de código para definir y manipular datos en DBMS específicos. - Enfoque de generación de código: estructural, de comportamiento o de traducción. - Integración con verificadores de modelos como SPIN. - Capacidades de lenguajes de dominio específico (DSL, siglas en inglés).
Compatibilidad con los ambientes de desarrollo y operaciones
- Plataforma Operativa: Linux, Mac OS X, MS Windows, MDA para modelos independientes de plataforma. - Formato de archivo para intercambio de información: OCL, XMI, .uml (de StarUML). - DBMS. - Integración con ambientes y herramientas como son: Visual SourceSafe, CVS, MS Word, Eclipse, Visual Studio.NET.
Otras características
- Proveedor: código abierto o comercial. - Uso especializado o de propósito general. - Continuidad de la herramienta y estabilidad de la versión.

UML para GCM

- La GACM utiliza el diagrama de clase como punto inicial para generar la estructura básica del código y luego el cuerpo de los métodos se complementa con diagramas de secuencia y diagramas de máquinas de estado.
- Los diagramas que resultan de las iteraciones de los flujos de trabajo de requisitos, análisis y diseño, intentan comunicar modelos con diferente nivel de detalle, entre usuarios y desarrolladores.
- Estos diagramas pueden estar muy pobremente elaborados para generar código. Los diagramas UML más mencionados son los de clase, de estado, de secuencia y de actividad.
- Deben conocerse las entradas indispensables para que un GCM genere un tipo de componente: interfaz de usuario, algoritmo y almacenamiento en estructuras de datos, entre otros.
- Un modelo de diseño incompleto resultará en diagramas UML sin suficiente elaboración para permitir la programación manual o automática.

Correspondencia UML-LP, I

UML	Java

	abstract public class Persona {
	  private String nombre;
	  private String apellido;
	  protected Date fechaNacimiento;
	  private static int mayoriaEdad=18;
	}

	C#

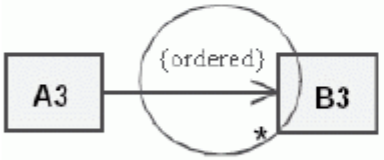
	abstract public class Persona {
	  private string nombre;
	  private string apellido;
	  protected DatTimee fechaNacimiento;
	  private static int mayoriaEdad=18;
	}

Correspondencia UML-LP, II

UML	Java	C#

	public class A1 {
	  private B1 laB1;
	  …
	}
		public class A1 {
		  private B1 laB1;
		  …
		}

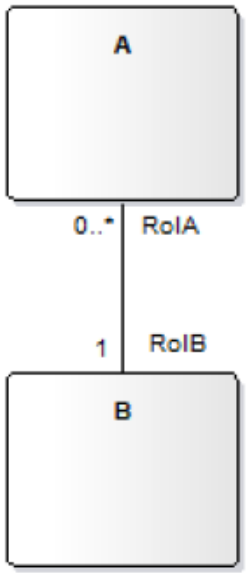
	public class A2 {
	  private B2 lasB2[];
	  …
	}
		public class A2 {
		  private B2[] lasB2;
		  …
		}

	public class A3 {
	  private List<B3> las B3
	  = new ArrayList<B3>();
	  …
	}
		public class A3 {
		  private IList lasB3 =
		  new List<B3>();
		  …
		}

	public class A4 {
	  private Map<Q,B4> las B4
	  = new HashMap<Q,B4>();
	  …
	}
		public class A4 {
		  private IDictionary <Q,B4>
		  las B4 = new Dictionary
		  <Q,B4>();
		  …
		}

Correspondencia UML-LP, III

Asociación N a 1

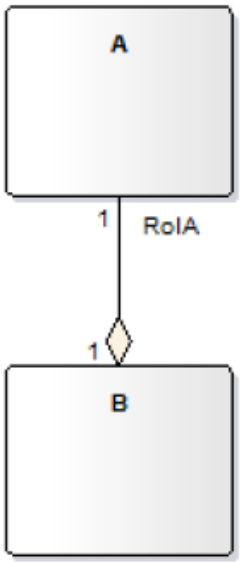


Clase A idéntica a la de una asociación 1 a 1

```
#include <set>
using namespace std;
// Clase B
class B {
public:
   B();
   B(const B& derecha);
   ~B();

   const B& operator = (const B& derecha);
   bool operator == (const B& izquierda) const;
   bool operator != (const B& izquierda) const;

// Para la relación de asociación de multiplicidad 0..*
   const set<A*> get_RolA() const;
   void Set_RolA(const set<A*> valor);

private:
   set<A*> Ra;
};

const set<A*> B::get_RolA() const { return Ra; }
void B::set_RolA(const set<A*> valor)
{ Ra = valor; }
```

Correspondencia UML-LP, IV

Agregación 1 a 1




```
// Declaración anticipada
class B;

class A
{
// constructores, destructores y
// operadores parecidos a las clases
// anteriores

// relación de agregación
   const B* get_B() const;
   void set_B(B* const valor);
private:
   B *_B;
};
```

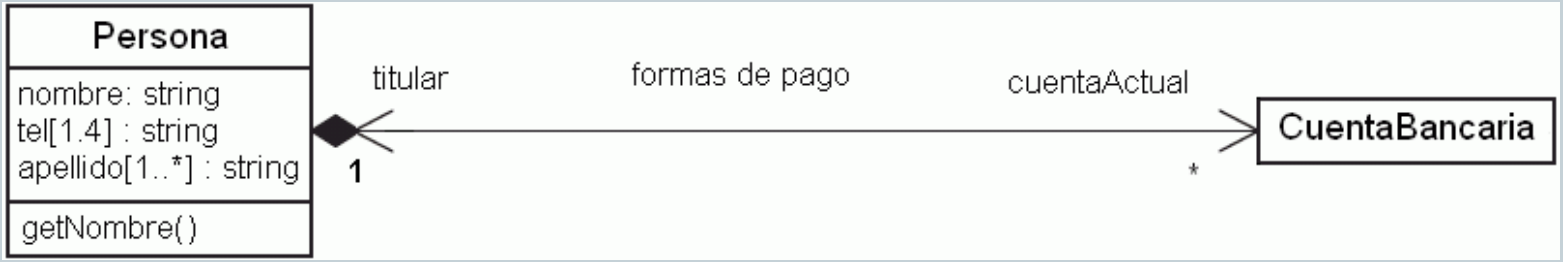
```
class B {
// constructores, destructores y
// operadores parecidos a las clases
// anteriores

// relación de agregación
   const A* get_RolA() const;
   void set_RolA(A* const valor);
private:
   A *RolA;
};
```

Dificultades y limitaciones

- La GACM se apoya en el diseño de lenguajes de modelado, el cual posee una base teórica reducida.
- Carece también de un mapeo completo entre diagramas UML y código.
- La traducción de diagramas UML a código OO, no asegura la consistencia entre el código y los diagramas ni genera ciertas asociaciones entre clases.
- Estas limitaciones pueden dificultar la determinación de relaciones precisas entre diagramas UML y tipos de componente como la interfaz de usuario, el manejo de base de datos y la lógica empresarial.
- Para complementar la GACM puede ser necesario utilizar lenguajes de dominio específico (DSL, siglas en inglés) o pseudocódigo.
- Los DSL permiten un énfasis en decisiones de diseño relevantes al dominio y utilizar los conceptos y abstracciones más adecuados.

Ejemplos, I

- UML a Java



- Ejecutando la operación de generación de código a partir del modelo mostrado en la figura anterior, obtenemos el siguiente código Java.

```
public class Persona{
   ArrayList cuentaActual ;
}
```

```
public class cuentaBancaria{
      persona titular ;
}
```

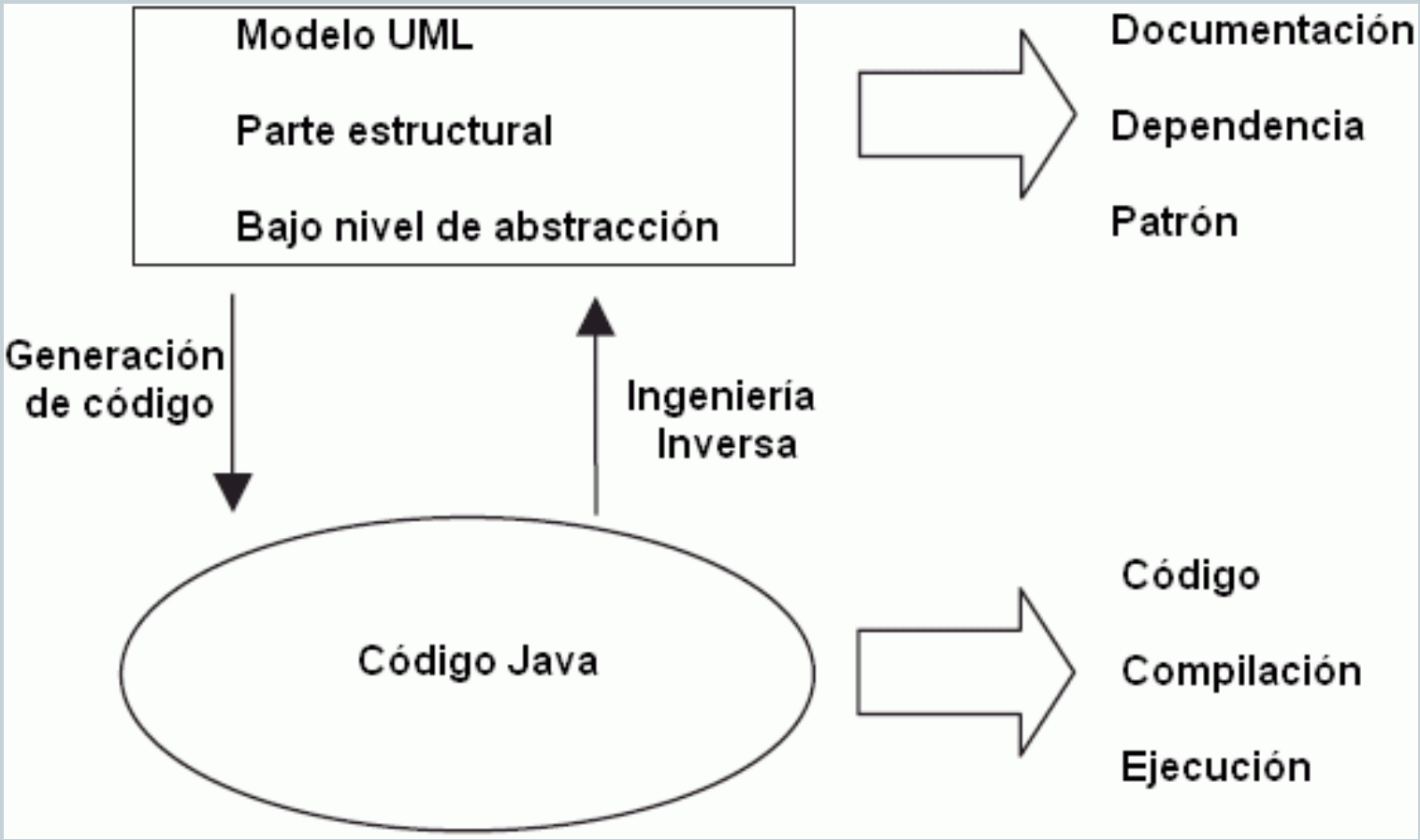
- Después de la ejecución de una Ingeniería Inversa, este código permite obtener el modelo ilustrado a continuación, que es completamente diferente del modelo de origen.

Ejemplos, II

- El modelo queda como sigue:

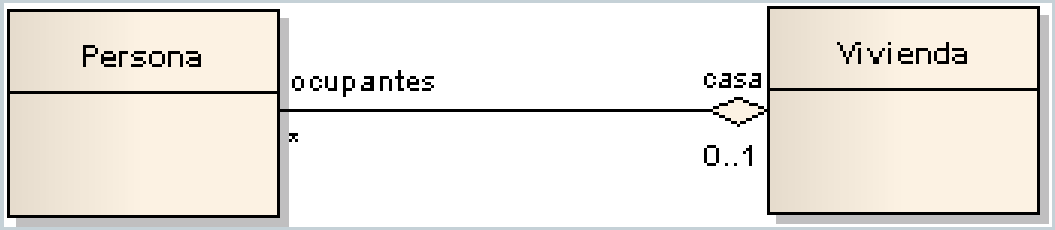
- Los efectos generados por las operaciones de generación de código e Ingeniería Inversa son dictados por las reglas de correspondencias que definen estas operaciones.
- Precisamos que las reglas de correspondencias no son en ningún caso estándares y que cada herramienta propone las suyas.
- Los modelos UML no deben contener herencias múltiples entre clases.
- Los modelos UML no deben contener asociaciones no navegables.
- Los modelos UML no deben contener asociaciones de agregación o composición.
- Los modelos UML no deben contener asociaciones navegables que especifican que el número máximo de objetos que pueden ser conectados es superior a 1. En cambio, procuraremos que los modelos UML utilicen la clase ArrayList.
- Los modelos UML deben contener una nota de código vinculada a cada operación.

Ciclo de desarrollo con UML



Ejemplos, III

- UML a C++
- Consideremos el siguiente ejemplo:



agregación!!

- En C++, no hay diferencias entre la asociación y la agregación, las dos se implementan de la misma manera.
- La composición, por ejemplo, induce un vínculo fuerte: el ciclo de vida de los objetos es el mismo.
- Una clase C++ que compone a otra clase debe por lo tanto a toda costa **administrar** la **memoria** de ella. Esto se puede implementar de dos maneras:
  - ya sea que el atributo es un **valor** y no un apuntador
  - ya sea que el atributo es un apuntador y la clase debe definir un **destructor**, un **constructor por copia**, un operador de afectación

Ejemplos, IV


```
class Vivienda;
class  Persona
{
        Vivienda* casa;
    public:
        Persona();
        const Vivienda* getCasa() const  (return casa;)
        void setCasa(Vivienda* house);
};
```

```
class Persona;
class Vivienda
{
      std::set<Persona*> ocupantes;
   public:
      Vivienda();
       void  mudarse(Persona*  pepito);
       void trasladarse(Persona* pepito);
};
```

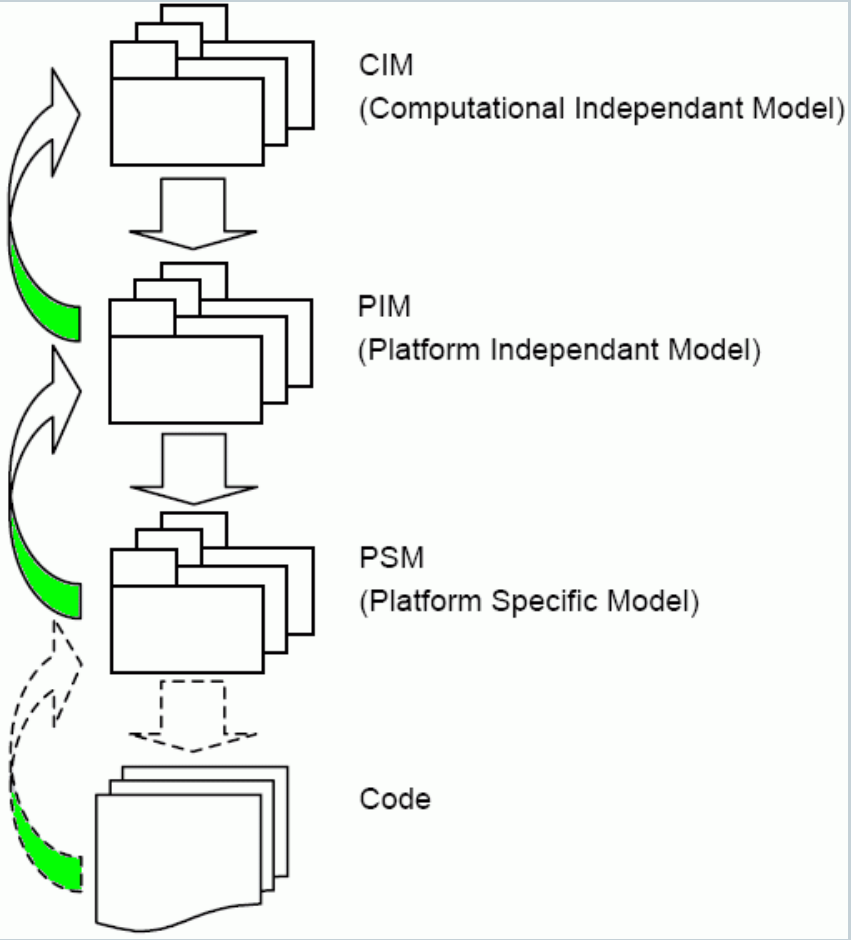
```
int main()
{
      Vivienda casa1;
      Persona pepito;
      pepito.setCasa(&casa1);
      Persona paul;
      paul.setCasa(&casa1);

      return 0;
}
```

Arquitectura general de MDA, I

- La siguiente figura proporciona una vista general del enfoque MDA.
- Constatamos que la construcción de una nueva aplicación comienza por la elaboración de uno o de varios modelos de requisitos (CIM).
- Se continúa con la elaboración de los modelos de análisis y diseño abstracto de la aplicación (PIM).
- Éstos deben en teoría parcialmente generarse a partir de los CIM para que en algunos se establezcan los vínculos de trazabilidad.
- Los modelos PIM son modelos persistentes, que no contienen ninguna información sobre las plataformas de ejecución.
- Para realizar concretamente la aplicación, es necesario a continuación construir los modelos específicos de las plataformas de ejecución.
- Estos modelos se obtienen por una transformación de PIM añadiendo la información técnica relativa a las plataformas.
- Los PSM no tienen por vocación ser persistentes. Su principal función es facilitar la generación de código.

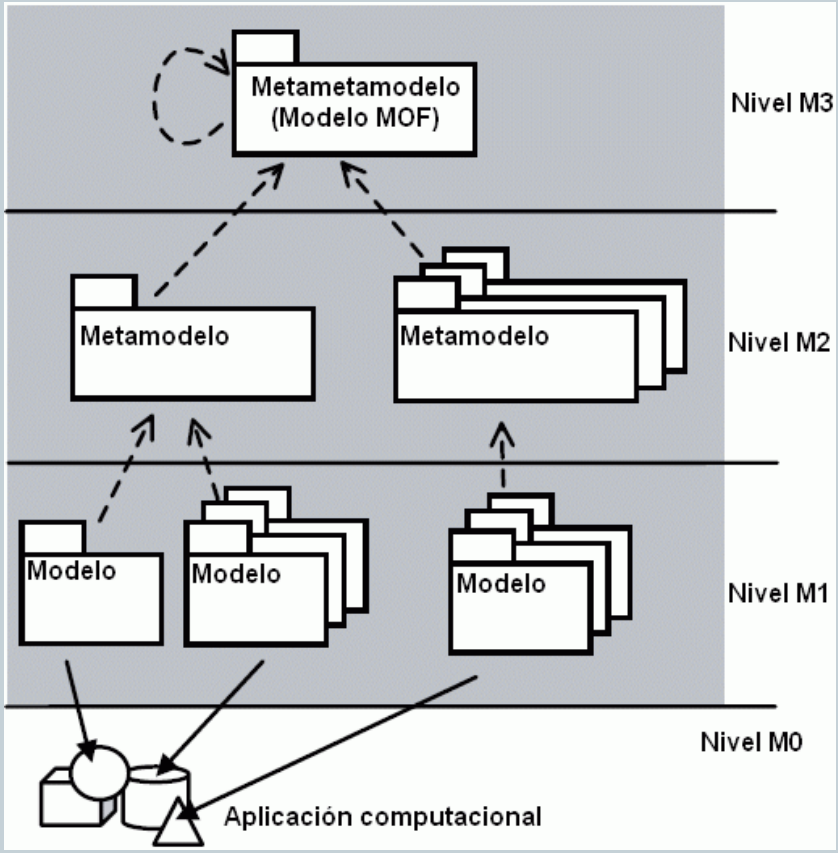
Arquitectura general de MDA, II

- MDA no considera la generación de código a partir de los modelos PSM por otra parte realmente. Ésta se vinculó más bien con una traducción de los PSM en un formalismo textual.
- Si la primera vocación del enfoque MDA es facilitar la creación de nuevas aplicaciones, este procura por otro lado numerosas ventajas para el retro-diseño de aplicaciones existentes.
- Esta es la razón por la que las transformaciones opuestas también se definen, PSM hacia PIM y PIM hacia CIM.
- Estas transformaciones no están presentes aún, se encuentran en fase de investigación

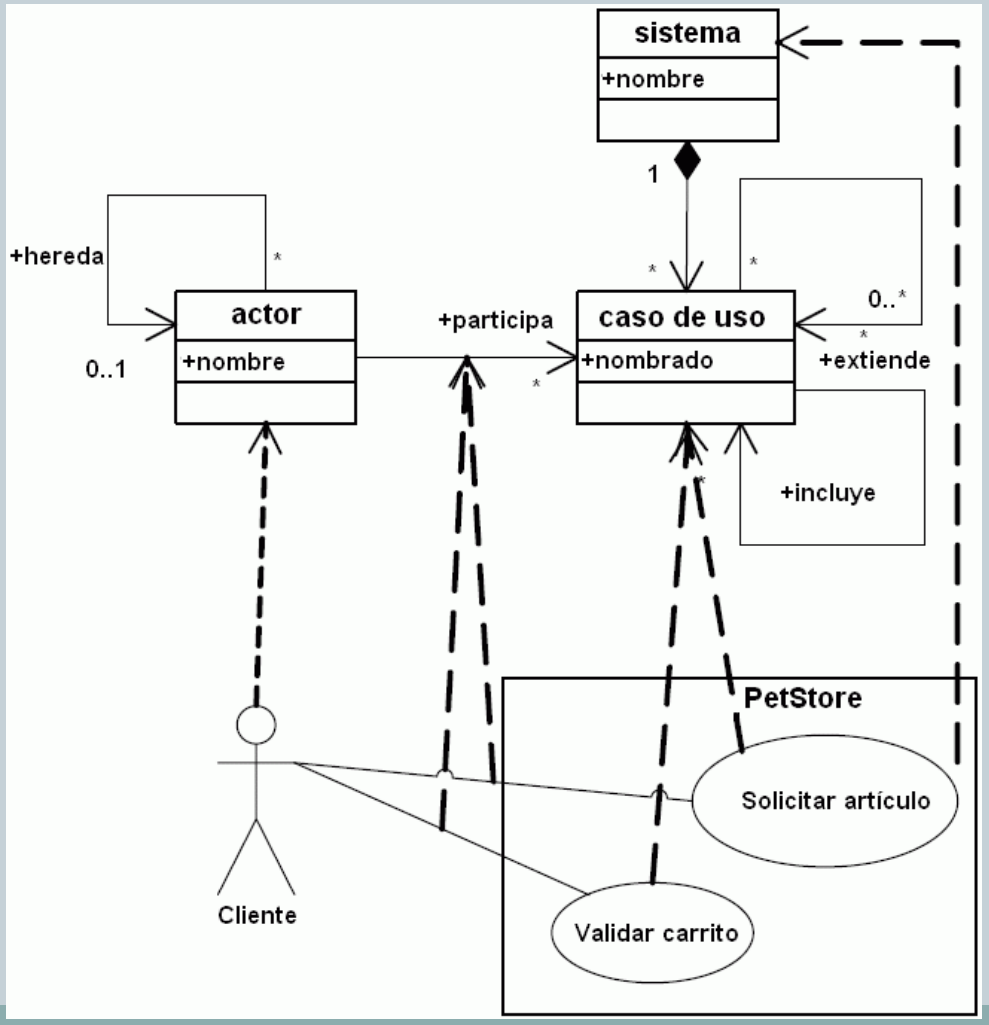


Metamodelos

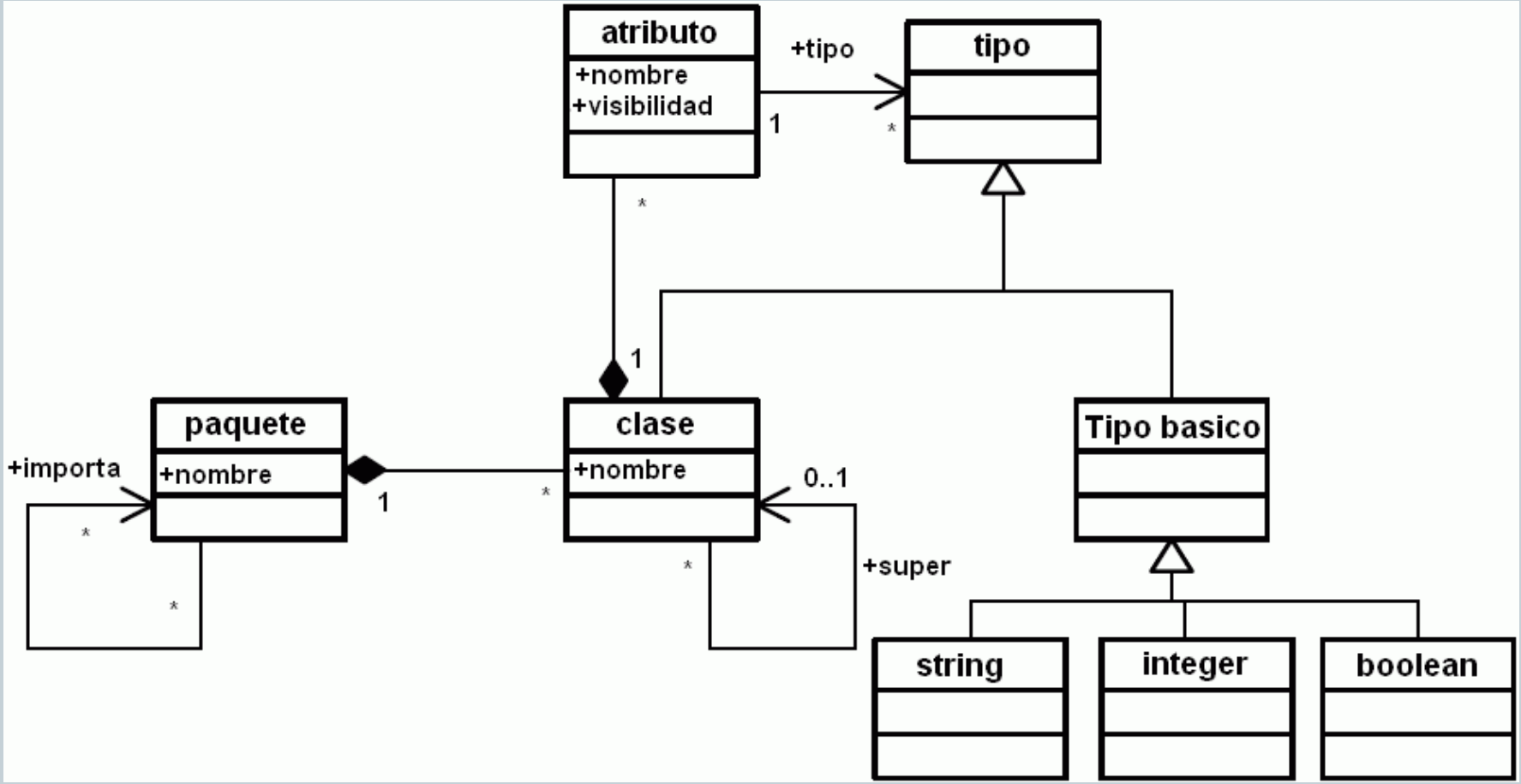
- MDA considera que, cualesquiera que sean los niveles M1, M2 y M3, todos los elementos son modelos. Es decir, el metametamodelo y los metamodelos son también modelos.



Metamodelo de los casos de uso

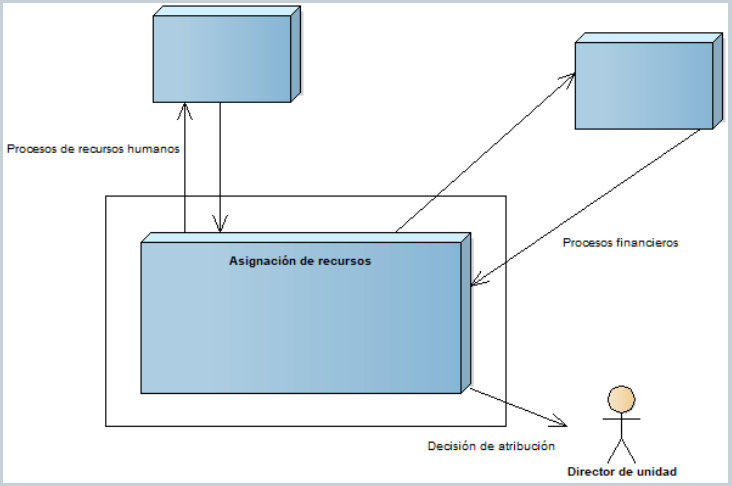


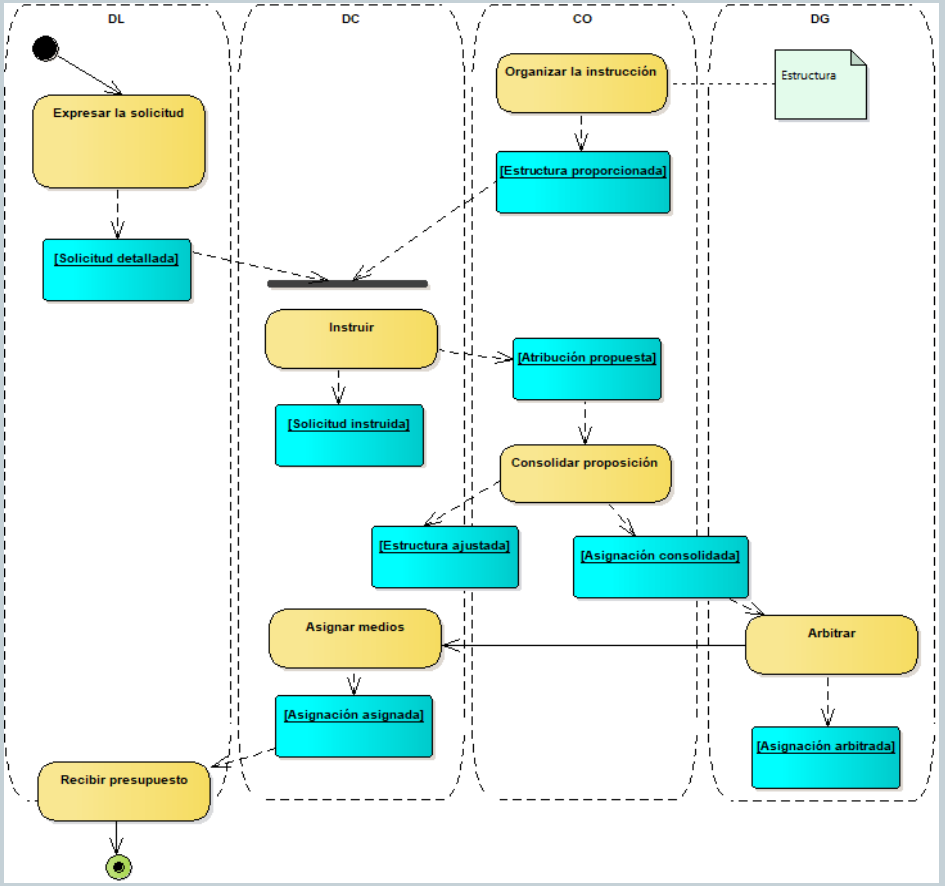
Metamodelo del diagrama de clases



Ejemplo de un sistema, I

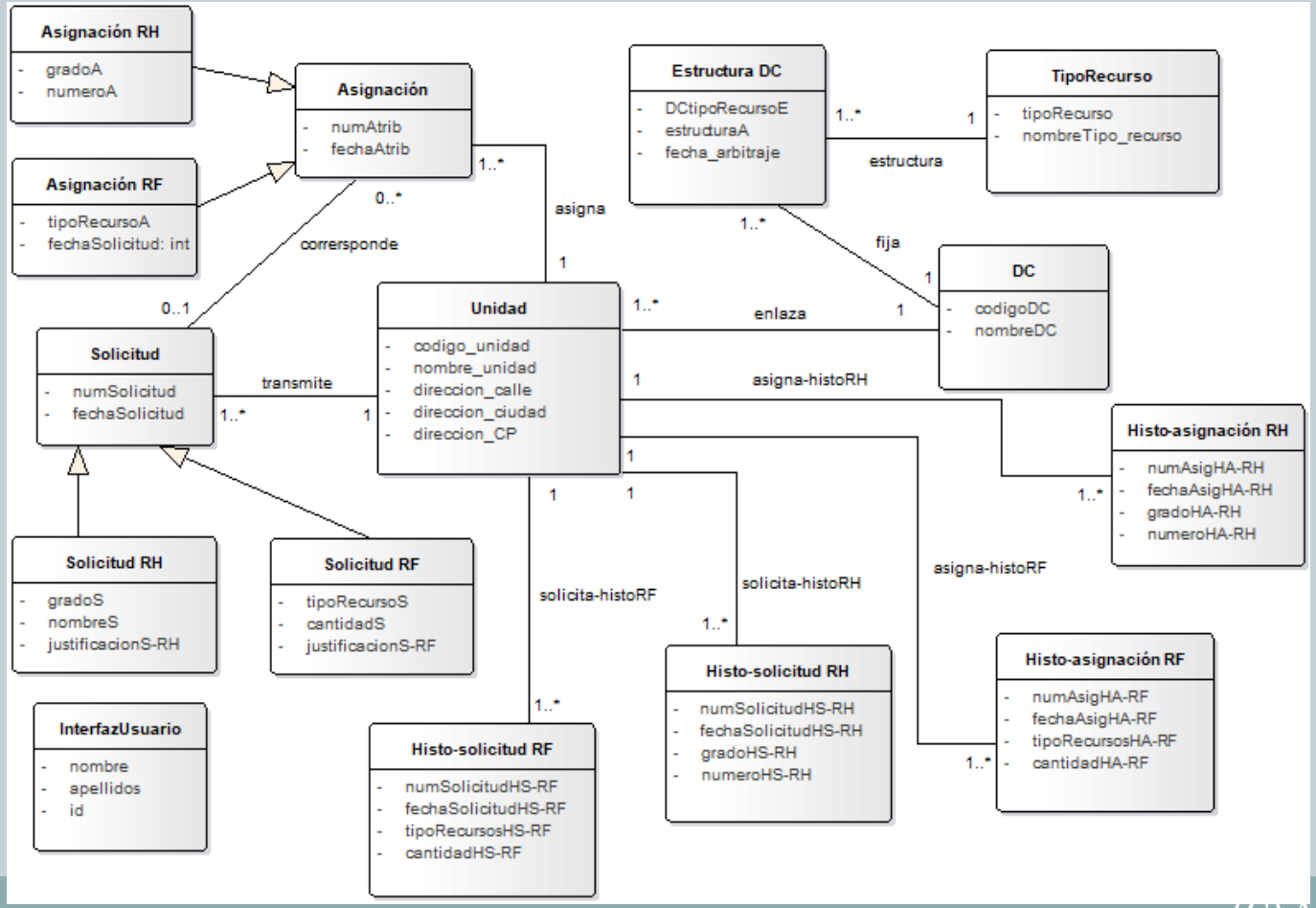
- Sistema para la gestión de recursos humanos





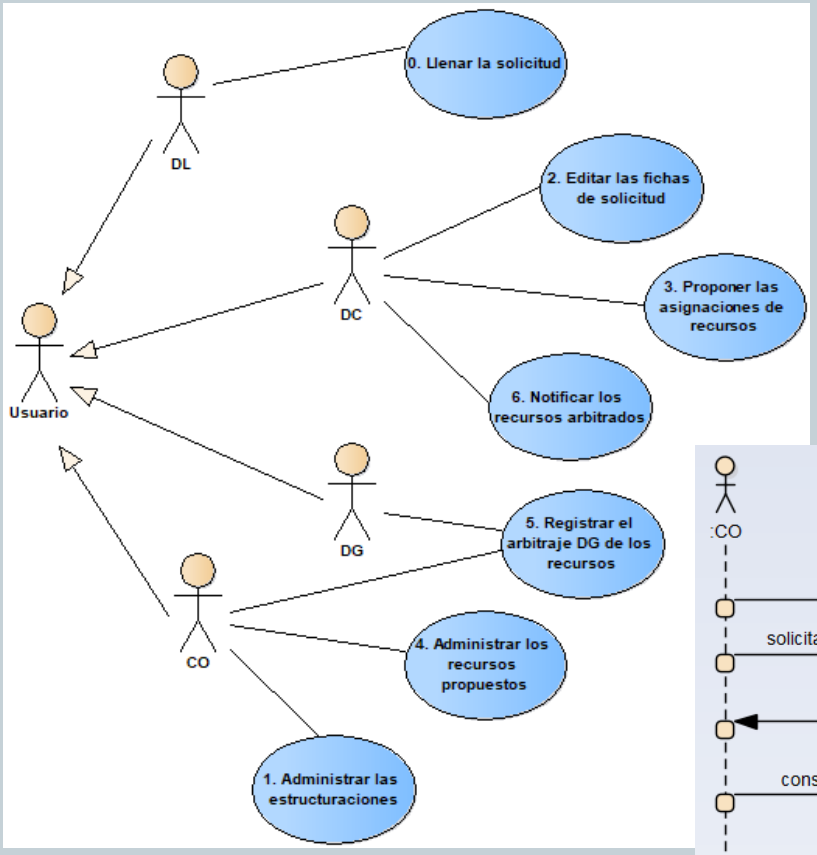
Ejemplo de un sistema, II

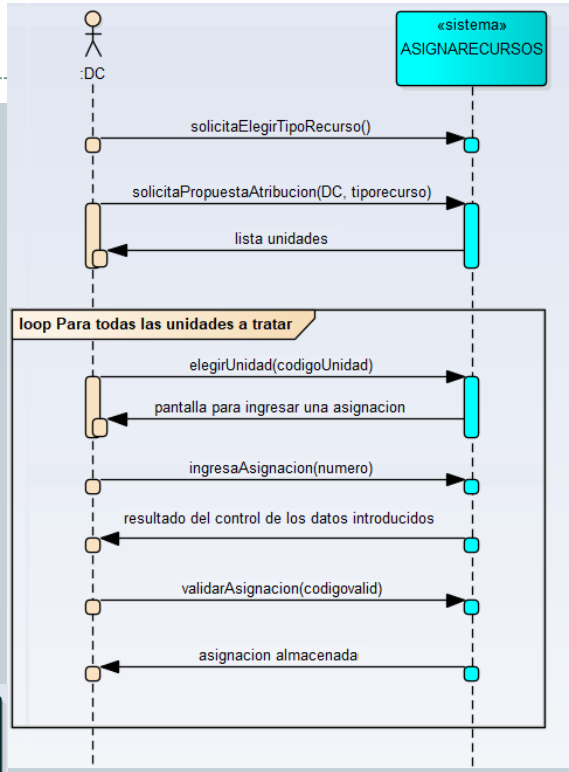
- Diagrama de clases de negocio

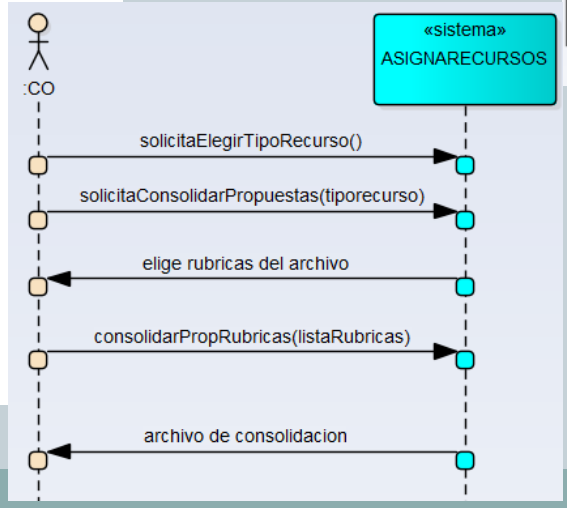


Ejemplo de un sistema, III

- Diagramas de caso de uso y secuencia

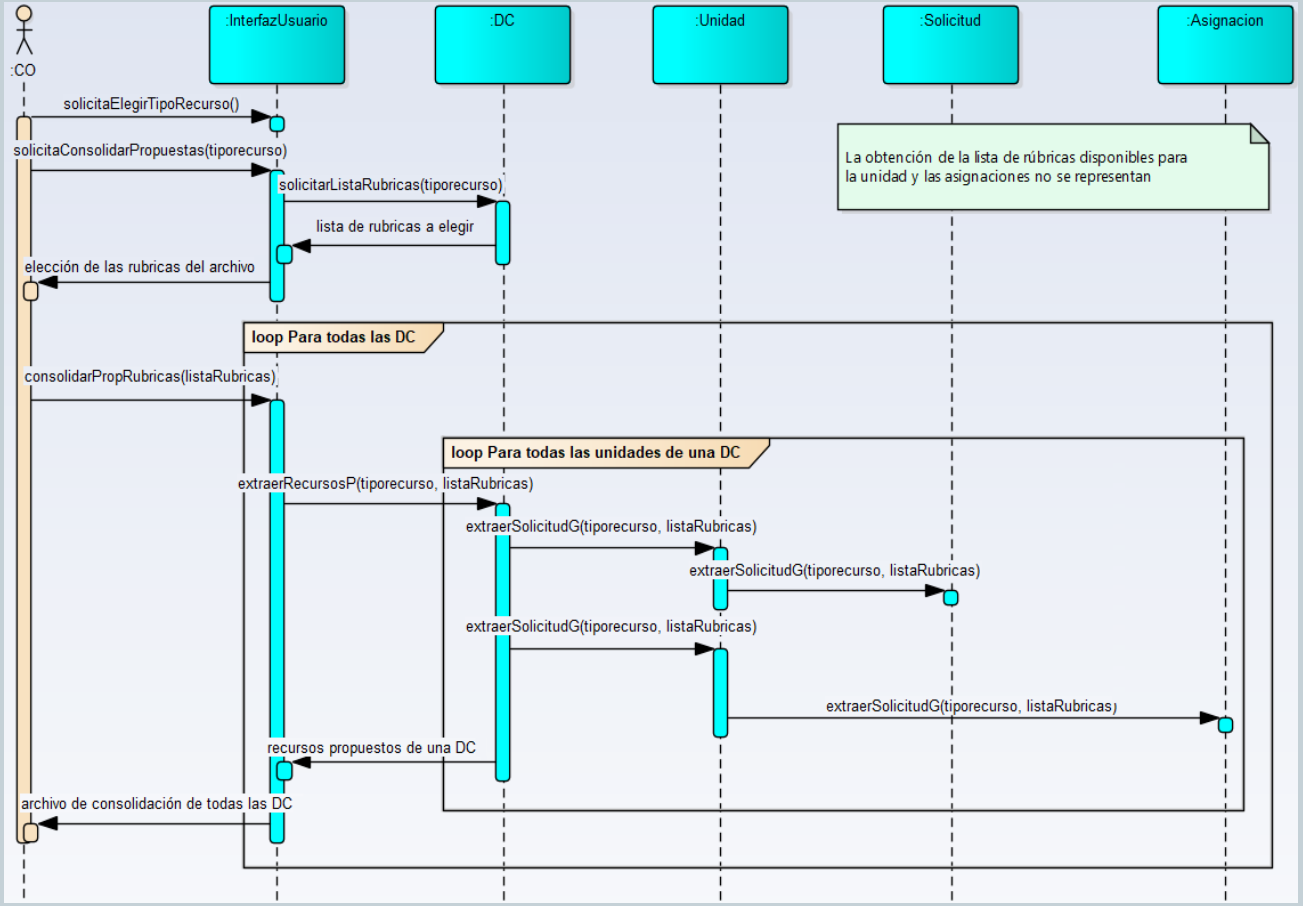






Ejemplo de un sistema, IV

- Diagramas de secuencia de diseño



Ejemplo de un sistema, V

- Diagrama de clases de diseño del diagrama de secuencia anterior