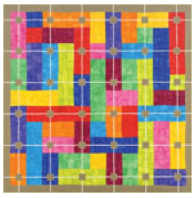
V.2.1



# Modelar Sistemas con Orientación a Objetos

**Contenido:**

Definiciones

Juego de Dados

Análisis y Diseño Orientado a Objetos
- Clases
- Responsabilidades de las clases
- Cohesión y acoplamiento de clases
- Reglas de oro para definir clases
- Encontrando clases
  - Análisis de sustantivos y verbos
  - Análisis CRC
  - Otras fuentes de clases

Ejercicio
- Modelar un curso de carrera universitaria
  - Solución:
  - Diagrama de Modelo de Dominio
  - Diagrama de Clases

Referencias

Martín Agüero
2019

# Definiciones

---

- ***Análisis*** *se enfoca en investigar el problema y sus requerimientos, más que en la solución.*
- ***Diseño*** *se orienta en la solución conceptual (software + hardware) que cumple con los requerimientos, más que en la implementación.*
- ***UML*** *(Lenguaje Unificado de Modelado) es un conjunto de herramientas que incluye diagramas para visualizar el modelo y construcción de un sistema orientado a objetos.*

---

Nota: Los diagramas se crearon con PlantUML.


A continuación se presentará un ejemplo introductorio sobre cómo modelar con UML un juego de dados.

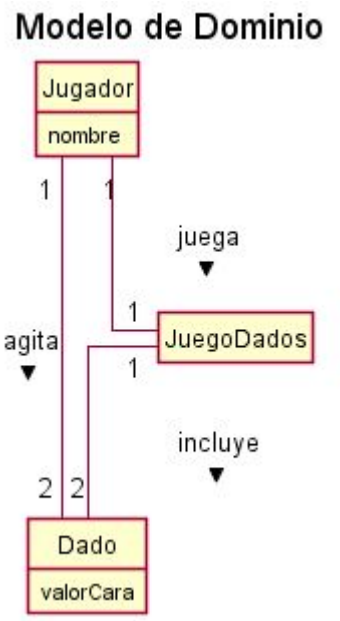
## Paso 1: Casos de Uso

El análisis de los requerimientos deben incluir historias o escenarios de cómo las personas utilizan la aplicación, ésto puede ser documentado como casos de uso. Los casos de uso no son un artefacto orientado a objetos (OO), son simples historias escritas. A continuación un caso de uso para el juego de dados:

**Caso de Uso:** Juego de dados
**ID:** 1
**Breve descripción:** Jugador tira los dados y evalúa el resultado.
**Actores principales:** Jugador
**Actores secundarios:** Sistema
**Precondiciones:**
-ninguna-
**Flujo principal:**
1. El jugador solicita tirar los dados.
2. El sistema presenta los resultados: si la cara superior de los dados suman 7, el jugador gana.
**Post condiciones:**
-ninguna-
**Flujos alternativos:**
2.1 Si la cara superior de los dados es distinta de 7, el jugador pierde.

## Paso 2: Modelo de dominio

El análisis OO tiene como objetivo fundamental crear la descripción del dominio desde la perspectiva de los objetos, es decir, identificar conceptos, atributos y las asociaciones. El resultado de esta fase puede ser expresada con un Modelo de Dominio:

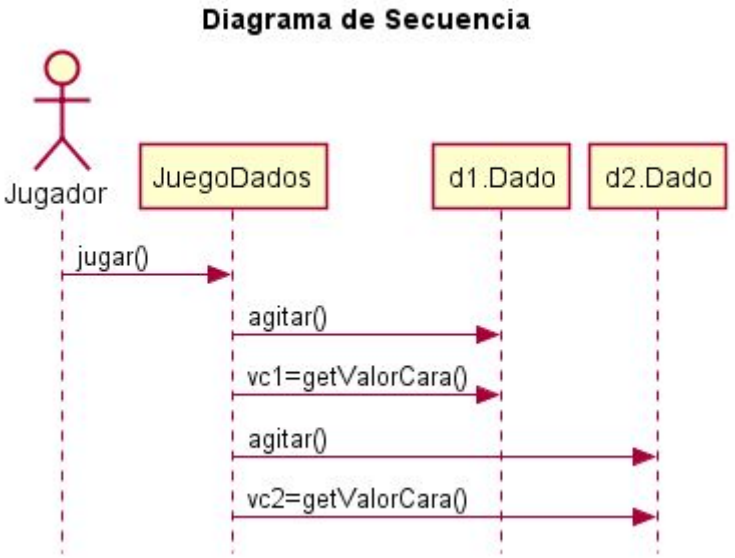



```
@startuml
title Modelo de Dominio
skinparam style strictuml
hide methods
hide empty members
skinparam linetype ortho
class Jugador {
  nombre
}
class Dado {
  valorCara
}
class JuegoDados {
}
Jugador "1" - "2" Dado : agita >
Jugador "1" - "1" JuegoDados : juega >
JuegoDados "1" - "2" Dado: incluye >
@enduml
```

Este modelo ilustra los conceptos de Jugador, Dado y JuegoDeDados con sus asociaciones y atributos. Un modelo no es una descripción de objetos de software, es simplemente una visualización de conceptos o un modelo mental de dominio del “mundo real”. También se lo denomina Modelo Conceptual de Objetos.

**Paso 3: Interacciones**

El diseño OO tiene como objetivo definir a los objetos de software sus responsabilidades y colaboraciones. Una visualización muy empleada es el Diagrama de Secuencia (un tipo de diagrama de interacción de UML). En este diagrama se muestra el flujo de mensajes entre los objetos a través de la invocación de métodos. El siguiente diagrama muestra los pasos esenciales del juego de datos.




```
@startuml
title Diagrama de Secuencia
skinparam style strictuml
actor Jugador
Jugador -> JuegoDados : jugar()
JuegoDados -> d1.Dado : agitar()
JuegoDados -> d1.Dado :
vc1=getValorCara()
JuegoDados -> d2.Dado : agitar()
JuegoDados -> d2.Dado :
vc2=getValorCara()
@enduml
```

**Paso 3: Clases**

Una vista estática de la definición de las clases se logra a través del Diagrama de Clases, el cual explica visualmente los atributos y métodos de las clases.


```
@startuml
title Diagrama de Clases
skinparam style strictuml
hide empty members
class JuegoDados {
dado1:Dado
dado2:Dado
jugar()
}
class Dado {
valorCara:int
getValorCara():int
agitar()
}
JuegoDados "1" -> "2" Dado
@enduml
```

En contraste con el modelo de dominio que representa las clases del "mundo real", el diagrama de clases muestra una visión de software de las clases.

**Conclusión:**
Los diseños y lenguajes OO pueden dar soporte, con bastante facilidad, y mediante componentes de software, a esa representación mental de un modelo de dominio.

## Análisis y Diseño Orientado a Objetos

El dominio del problema es el dominio donde se precisa contar con un sistema de software. El más importante aspecto de una clase es que debe "mapear" de forma clara y no ambigua con algún concepto del mundo real, tal como cliente, producto o cuenta. Es una habilidad esencial del analista OO, poder clarificar conceptos del negocio. El primer paso para crear software orientado a objetos es comprender el dominio del problema. Gran parte de este trabajo se logra a partir del estudio del flujo de los requerimientos a fin de crear el modelo de casos de uso. Es importante entender que el verdadero objetivo del análisis OO es encontrar las clases de esos objetos. De hecho, encontrar las clases apropiadas es la clave del análisis y diseño OO.

## Clases

Una clase debe presentar un conjunto de atributos de alto nivel. Las operaciones de las clases especifican los servicios clave que debe ofrecer.

¿Qué se considera una "buena clase"? Es decir, apropiada al dominio del problema.

- ➢ Que su nombre refleje su propósito.
- ➢ Que sea una auténtica abstracción que modele un elemento específico del dominio del problema.
- ➢ Que mapee a una clara e identificable característica del dominio del problema.
- ➢ Que posea un pequeño pero bien definido conjunto de responsabilidades.
- ➢ Que posea alta cohesión
- ➢ y bajo acoplamiento.

**Por ejemplo, si se desea modelar a un cliente en un sistema bancario, se deseará capturar el nombre del cliente, la dirección, pero no será de interés conocer si prefiere sentarse en ventanilla o pasillo cuando vuela.**
**Es fundamental enfocarse en los aspectos de las cosas del "mundo real" que son verdaderamente importantes desde la perspectiva del sistema que se está construyendo.**

Usualmente se puede tener una primera impresión acerca de si una clase es apropiada simplemente a partir de su nombre. Por ejemplo en un sistema de e-commerce, Cliente aparenta refererirse a algo preciso del mundo real, es decir, puede ser un buen candidato para una clase. CarritoDeCompras también parece ser una buena abstracción, se sabe casi intuitivamente cuál es su semántica. No obstrante, algo como VisitanteDeWebsite aparenta tener una semántica no del todo clara y hasta parece superponerse al rol de Cliente. Siempre se deben buscar las abstracciones evidentes, que poseen una clara y obvia semántica.

## Responsabilidades de las clases

Una responsabilidad es un contrato u obligación que una clase posee con sus clientes. Esencialmente, una responsabilidad es un servicio que una clase ofrece a otras clases. Volviendo al ejemplo de CarritoDeCompras, sus responsabilidad serán:

- Agregar item
- Quitar item
- Mostrar items

Este es un cohesivo grupo de responsabilidades. Es cohesivo porque todas las responsabilidades poseen el mismo objetivo. Luego, si se desea agregar las siguientes nuevas responsabilidades a la misma clase CarritoDeCompras:

- Validar tarjeta de crédito

- Aceptar pago
- Imprimir factura

## Cohesión y acoplamiento de clases

Es evidente que no parecen ser apropiadas para esa clase. No son cohesivas y claramente deberían ser asignadas a otra cosa, posiblemente una clase EmpresaTarjetaDeCrédito o Pago. Es importante distribuir las responsabilidades de forma apropiada sobre las clases a fin de maximizar la cohesión en cada clase. Por último, las clases deberían poseer un mínimo de acoplamiento entre sí. Se mide el acoplamiento entre clases por el número de clases sobre las cuales una clase tiene relación.

## Reglas de oro para definir clases

A continuación una serie de buenas prácticas para la creación de clases:

- ***Establecer entre 3 y 5 responsabilidades por clase:*** Típicamente las clases deben ser lo más simple posible.
- ***Evitar clases solitarias:*** La esencia de OO es definir clases que colaboran entre sí.
- ***Atención con muchas clases pequeñas:*** Si el modelo posee muchas clases pequeñas con 1 o 2 responsabilidades cada una, debería considerarse fusionar algunas clases en otras algo mayores.
- ***Atención con pocas clases grandes:*** En estos casos la estrategia es verificar si esas clases pueden ser fraccionadas en 2 o más más pequeñas con el mismo número de responsabilidades.
- ***Atención con los "fuctoides":*** Un functoide es una función procedural disfrazada en una clase.
- ***Atención con las clases omnipotentes:*** Estas son clases que aparentan "hacer todo". Prestar atención con las clases "sistema" o "controlador". La estrategia es verificar si las responsabilidades. De la clase omnipotente puede derivarse en subconjuntos cohesivos. Entonces puede que cada uno de esos conjuntos pueda ser factorizada en clases separadas.
- ***Evitar árboles de herencia profundos:*** La esencia del buen diseño de la herencia es que cada nivel de abstracción pueda poseer un bien definido propósito. A veces es fácil agregar niveles que no tengan un propósito específico.

# Encontrando clases

No existe un algoritmo para encontrar las clases. Sin embargo existen una serie de técnicas probadas que pueden ayudar a descubrirlas.

## Análisis de sustantivos y verbos

Es una manera simple de encontrar clases, atributos y responsabilidades. Sin embargo con esta técnica hay que prestar atención a los sinónimos y homónimos para no crear clases espúreas

El primer paso consiste en recolectar toda la información relevante que sea posible, las fuentes pueden ser: Especificaciones de requerimientos, casos de uso, un glosario del proyecto, etc. Se debe analizar resaltando lo siguiente:

- Sustantivos, por ejemplo: vuelo;
- Frases nominales: número de vuelo;
- Verbos: ubicar;
- Frases verbales: verificar tarjeta de crédito;

Los sustantivos y las frases pueden estar indicando las clases o los atributos de las clases. Los verbos y las frases verbales pueden estar indicando las responsabilidades de las clases.

## Análisis CRC

Una muy buena técnica es involucrar a los usuarios para encontrar las clases. CRC (Class Responsibility Collaboration) es una técnica de brainstorming (tormenta de ideas) donde se capturan en papeles autoadhesivos los asuntos importantes del dominio del problema.


Este análisis debe emplearse siempre en conjunto con el análisis de sustantivos y verbos sobre los casos de uso, requerimientos, el glosario y toda la documentación del proyecto. El análisis CRC se ejecuta en 2 fases:

**Fase 1: Tormenta de ideas - recopilar la información**
Los participantes son los analistas OO, los stakeholders (los interesados) y los expertos en el dominio.

El procedimiento.

1. Explicar qué es una tormenta de ideas.
   1.1. Todas las ideas son aceptadas como buenas ideas.
   1.2. Las ideas se registran pero no debaten - nunca se discuten, simplemente se toma nota y luego se sigue adelante. Todo será analizado más adelante.
2. Pedir a los miembros del equipo que nombren las "cosas" que operan en su ámbito de trabajo - por ejemplo, clientes, productos.
   2.1. Escribir cada “cosa” en una nota adhesiva - será una clase candidata, o el atributo de una clase.
   2.2. Se adhiere la nota en una pared o pizarra.
3. Solicitar a los presentes que indiquen las responsabilidades que pudieran tener esas cosas (Registrarlas en la columna de responsabilidades).
4. Trabajar con el equipo, tratar de identificar las clases que podrían funcionar juntas. Reorganizar las notas en la pizarra para reflejar esta organización y trazar líneas entre ellas. Alternativamente, registrar los colaboradores en la columna correspondiente.

**Fase 2: Analizar la información**

¿Cómo se decide qué notas adhesivas pasarán a ser clases y qué debería convertirse en atributos? Las clases deben representar una nítida abstracción del dominio del problema. Ciertas notas representarán conceptos clave del negocio y claramente serán clases. Si una nota aparenta ser parte de otra nota, puede que represente a un atributo.

### Otras fuentes de clases

Además del análisis sustantivo/verbo y análisis CRC, también existen otras fuentes de potenciales clases a considerar:

- Objetos físicos tales como aviones, personas u hoteles pueden indicar clases.
- Los formularios son otra fuente de clases. Facturas, órdenes de compra o cajas de ahorro pueden estar indicando posibles clases. De todas formas se debe ser muy cuidadoso con los formularios. En muchas empresas el papeleo ha evolucionado en un nuevo sistema que da soporte a un proceso de negocio anticuado. No es el deseo de ningún analista o diseñador OO reflejar con un sistema automático procesos obsoletos.
- Las interfaces con el mundo exterior tales como pantallas, teclados, periféricos u otros sistemas pueden ser clases candidatas especialmente para los sistemas embebidos.
- Entidades conceptuales que son cruciales para la operación de un negocio pero que no se manifiestan como cosas concretas también pueden ser modeladas como clases. Por ejemplo, un ProgramaDeFidelidad tal como una tarjeta de recompensa. Claramente el programa no es algo concreto, pero sí es una abstracción cohesiva y deberá ser modelada como una clase.

# Ejercicio

## Modelar un curso de carrera universitaria

Se precisa contar con un sistema que gestione los cursos que ofertan los diferentes departamentos de una facultad. Cada curso se formaliza mediante un programa que, además de los contenidos, también define la bibliografía obligatoria y complementaria de la asignatura. El programa establece una serie de unidades que deberá dictar el docente a lo largo del cuatrimestre y las características de los exámenes. A su vez, cada curso otorga un cierto número de créditos y presenta como requisito una serie de asignaturas correlativas directas aprobadas.

**Se pide:**

1. Modelar el dominio en OO y representarlo a través de un diagrama.
2. Crear el diagrama de clases.

**Supuestos:**

Curso y asignatura hacen referencia a la misma entidad.

# Referencias

Larman, C., Applying UML and Patterns.
Addison-Wesley. 2004.

Kendall & Kendall, Analysis and System Design.
Prentice Hall, 2011.

Arlow, J., UML 2 and the Unified Process Practical Object Oriented Analysis and Design.
Addison-Wesley. 2005.