

LABORATORIO GUIADO - UNIDAD 2

Análisis y UML: Modelado con Draw.io + OCL con USE + Implementación Python

Curso: Introducción a la Arquitectura y Diseño de Software

Instructor: Ing. Roberto Salguero

Unidad: 2 - Análisis y UML

Modalidad: Laboratorio Guiado

Tiempo Estimado: 8-10 horas

TABLA DE CONTENIDOS

1. [Introducción y Objetivos](#)
2. [Preparación del Entorno](#)
3. [Parte A - Modelado en Draw.io](#)
4. [Parte B - OCL con USE](#)
5. [Parte C - Implementación en Python](#)
6. [Trazabilidad UML → OCL → Código](#)
7. [Entregables y Rúbrica](#)
8. [Recursos Adicionales](#)
9. [Errores Comunes y Soluciones](#)
10. [Anexos](#)

1. INTRODUCCIÓN Y OBJETIVOS

1.1 Introducción

Este laboratorio guiado integra **UML**, **OCL** y **Python** para desarrollar un flujo completo de análisis y diseño orientado a objetos. Trabajaremos con el **Sistema de Reservas de Tutorías**, un dominio familiar que permite practicar conceptos fundamentales sin perderse en la complejidad del negocio.

El laboratorio está estructurado en tres partes progresivas:

- **Parte A:** Modelado visual con Draw.io
- **Parte B:** Especificación formal con OCL y USE
- **Parte C:** Implementación y pruebas en Python

Nota importante: Este laboratorio utiliza EXACTAMENTE el mismo código y ejemplos del material de clase para garantizar consistencia y evitar confusión.

1.2 Objetivos del Laboratorio

Al finalizar este laboratorio, el estudiante será capaz de:

1. **Modelar un problema con UML estructural** (casos de uso y clases) y UML de comportamiento (secuencia/actividad/estados)
2. **Ejecutar un flujo completo:** requisitos → análisis OO → UML → OCL → implementación en Python
3. **Escribir restricciones OCL** (invariantes y pre/post) para eliminar ambigüedad y "blindar" reglas de negocio
4. **Traducir correctamente el modelo a código** (POO en Python) manteniendo trazabilidad
5. **Validar modelos con USE** detectando y corrigiendo violaciones de invariantes
6. **Implementar pruebas automatizadas** que validen las reglas de negocio modeladas en OCL

1.3 Dominio del Problema: Sistema de Reservas de Tutorías

Descripción del sistema:

- **Estudiantes** pueden buscar y reservar tutorías con **tutores** especializados
- **Tutores** publican su **disponibilidad** en horarios específicos
- Una **reserva** conecta un estudiante con un tutor en un horario específico
- Las reservas tienen un **ciclo de vida**: CREADA → CONFIRMADA → CANCELADA/EXPIRADA

Entidades principales:

- Estudiante (nombre, carnet)
- Tutor (nombre, especialidad)
- Reserva (fechaHora, estado)
- Disponibilidad (fechaHora, estado)

Reglas de negocio clave:

1. Un tutor no puede tener dos reservas activas en el mismo horario
2. Solo se puede reservar en horarios donde el tutor está disponible
3. Las reservas solo se pueden cancelar si están CREADAS o CONFIRMADAS

1.4 Entregables Esperados

Al finalizar el laboratorio, deberás entregar:

- **Diagrama de Casos de Uso** (PDF exportado de Draw.io)
- **Diagrama de Clases del Dominio** (PDF exportado de Draw.io)
- **Diagrama de Secuencia "Crear Reserva"** (PDF)
- **Diagrama de Estados de Reserva** (PDF)
- **Archivo Cars.use** (ejercicio de primer contacto)
- **Archivo University.use** (ejercicio con association class)
- **Archivo Reservas.use** (modelo simplificado)
- **Archivo Tutorias.use** (modelo completo con pre/post)
- **Scripts .cmd** (Cars.cmd, University.cmd, Reservas.cmd)
- **Código Python completo** (domain.py, repositories.py, services.py)
- **Pruebas automatizadas** (test_reservas.py)
- **Capturas de USE** mostrando violaciones y correcciones
- **Tabla de trazabilidad** UML → OCL → Código

1.5 Herramientas Necesarias

- **Draw.io** (<https://app.diagrams.net>) - Modelado UML
- **USE** (UML-based Specification Environment) - Validación OCL
- **Java 8+** - Requerido para ejecutar USE
- **Python 3.9+** - Implementación
- **pytest** - Framework de pruebas

1.6 Tiempo Estimado

- **Preparación del entorno:** 1 hora
- **Parte A (UML en Draw.io):** 2-3 horas
- **Parte B (OCL con USE):** 3-4 horas
- **Parte C (Python):** 2-3 horas
- **Total:** 8-10 horas

2. PREPARACIÓN DEL ENTORNO

2.1 Instalación de Draw.io

Opción 1: Versión Web (Recomendada)

1. Ve a <https://app.diagrams.net>
2. No requiere instalación, funciona directamente en el navegador
3. Permite guardar archivos localmente (.drawio) y exportar a PDF

Opción 2: Aplicación de Escritorio

1. Ve a <https://github.com/jgraph/drawio-desktop/releases>
2. Descarga la versión para tu sistema operativo
3. Instala siguiendo las instrucciones estándar

Configuración inicial:

1. Abre Draw.io
2. Ve a "More Shapes..." en el panel izquierdo
3. Marca "**UML**" para habilitar la librería de diagramas UML
4. Haz clic en "Apply"

Tip: Guarda tus diagramas con extensión .drawio para poder editarlos después, y exporta como PDF para entrega.

2.2 Instalación de USE (UML-based Specification Environment)

Paso 1: Verificar Java

```
java -version
```

Debes ver Java 8 o superior. Si no tienes Java instalado:

- **Windows:** Descarga desde <https://adoptopenjdk.net/>
- **Linux:** `sudo apt install openjdk-11-jdk`
- **Mac:** `brew install openjdk@11`

Paso 2: Descargar USE

1. Ve a <https://sourceforge.net/projects/useocl/>
2. Descarga la versión más reciente (archivo .zip)
3. Descomprime en una carpeta de tu preferencia (ej: C:\USE o /home/usuario/USE)

Paso 3: Verificar instalación

```
# Windows
cd C:\USE
use.bat

# Linux/Mac
cd /home/usuario/USE
./use
```

Si aparece el prompt `use>` o se abre la interfaz gráfica, la instalación fue exitosa.

Comandos básicos de USE:

```
use modelo.use          # Cargar modelo
open script.cmd         # Ejecutar script
check                  # Verificar invariantes
? expresion_ocl         # Evaluar consulta OCL
info                   # Información del modelo
help                   # Lista de comandos
quit                   # Salir
```

2.3 Configuración de Python

Verificar Python:

```
python --version
# o
python3 --version
```

Instalar pytest:

```
pip install pytest
# o
pip3 install pytest
```

Verificar pytest:

```
pytest --version
```

2.4 Estructura de Carpetas Recomendada

Crea la siguiente estructura para organizar tu trabajo:

```
laboratorio-uml-ocl/
└── diagramas/
    ├── casos_uso.drawio
    ├── clases.drawio
    ├── secuencia.drawio
    └── estados.drawio
└── ocl/
    ├── Cars.use
    ├── Cars.cmd
    ├── University.use
    ├── University.cmd
    ├── Reservas.use
    ├── Reservas.cmd
    └── Tutorias.use
└── python/
    └── tutorias/
        ├── __init__.py
        ├── domain.py
        ├── repositories.py
        └── services.py
    └── tests/
        └── test_reservas.py
```

Comandos para crear la estructura:

```
mkdir -p laboratorio-uml-ocl/{diagramas,ocl,python/{tutorias,tests}}
cd laboratorio-uml-ocl/python/tutorias
touch __init__.py
cd ../../
```

3. PARTE A - MODELADO EN DRAW.IO

3.1 Diagrama de Casos de Uso del Sistema de Tutorías

Actores Identificados:

- **Estudiante** - Usuario principal que reserva tutorías
- **Tutor** - Proporciona tutorías y gestiona disponibilidad
- **Administrador** - Gestiona usuarios y sistema
- **Sistema de Notificaciones** - Envía confirmaciones (opcional)

Casos de Uso Principales:

1. **Registrarse / Iniciar sesión**
2. **Buscar tutor**
3. **Ver disponibilidad del tutor**
4. **Crear reserva** ★ (caso principal)
5. **Confirmar reserva**
6. **Cancelar reserva**
7. **Ver historial de reservas**
8. **Recibir notificaciones**
9. **Pagar tutoría** (opcional)

Relaciones Include/Extend:

- "Crear reserva" <> "Validar disponibilidad"
- "Crear reserva" <> "Enviar notificación"
- "Cancelar reserva" <> "Notificar cancelación"

Pasos para crear el diagrama:

1. **Crear nuevo diagrama en Draw.io**
 - File → New → Blank Diagram
 - Nombre: `casos_uso.drawio`
2. **Dibujar la frontera del sistema**
 - Usar un rectángulo grande
 - Título: "Sistema de Reservas de Tutorías"
3. **Agregar actores**
 - Usar la forma "Actor" de la librería UML
 - Colocar fuera de la frontera del sistema
4. **Agregar casos de uso**
 - Usar elipses dentro de la frontera
 - Nombres claros y verbos en infinitivo
5. **Conectar con asociaciones**
 - Líneas simples entre actores y casos de uso
 - Flechas discontinuas con <> y <>

✓ Checklist de Verificación - Casos de Uso:

- 4 actores identificados y posicionados correctamente
- 9 casos de uso con nombres claros
- Frontera del sistema definida
- Al menos 2 relaciones include/extend
- Diagrama exportado como PDF

3.2 Diagrama de Clases del Dominio

Clases Identificadas (9 clases principales):

1. **Tutor**
 - nombre: String
 - especialidad: String
2. **Estudiante**

- o nombre: String
- o carnet: String

3. Reserva

- o fechaHora: String
- o estado: EstadoReserva

4. Disponibilidad

- o fechaHora: String
- o estado: EstadoDisponibilidad

5. Curso

- o codigo: String
- o nombre: String

6. Tema

- o nombre: String

7. Ubicacion

- o tipo: String
- o enlace: String

8. Pago

- o monto: Real
- o estado: EstadoPago

9. Notificacion

- o canal: String
- o mensaje: String

Enumeraciones:

- **EstadoReserva:** {CREADA, CONFIRMADA, CANCELADA, EXPIRADA}
- **EstadoDisponibilidad:** {LIBRE, OCUPADA}
- **EstadoPago:** {PENDIENTE, PAGADO, REEMBOLSADO}

Relaciones y Multiplicidades:

Relación	Multiplicidad	Tipo
Tutor ↔ Reserva	1 ↔ *	Asociación
Estudiante ↔ Reserva	1 ↔ *	Asociación
Tutor ↔ Disponibilidad	1 ↔ *	Asociación
Reserva ↔ Curso	0..1 ↔ 0..1	Asociación
Curso ↔ Tema	1 ↔ *	Asociación
Reserva ↔ Ubicacion	1 ↔ 1	Composición
Reserva ↔ Pago	0..1 ↔ 0..1	Asociación
Reserva ↔ Notificacion	1 ↔ *	Asociación

Pasos para crear el diagrama:

1. Crear clases principales

- o Usar rectángulos divididos en 3 secciones
- o Sección 1: Nombre de la clase
- o Sección 2: Atributos con tipos
- o Sección 3: Métodos (opcional para modelo de dominio)

2. Agregar enumeraciones

- o Usar <> como estereotipo
- o Listar valores posibles

3. Conectar con asociaciones

- o Líneas sólidas para asociaciones
- o Diamante relleno para composición
- o Agregar multiplicidades en ambos extremos
- o Roles opcionales para claridad

4. Organizar layout

- Clases centrales (Reserva, Tutor, Estudiante) en el centro
- Clases de soporte alrededor
- Minimizar cruce de líneas

Checklist de Verificación - Clases:

- 9 clases con atributos y tipos correctos
- 3 enumeraciones definidas
- 8 asociaciones con multiplicidades
- 1 relación de composición identificada
- Layout organizado y legible
- Diagrama exportado como PDF

3.3 Diagrama de Secuencia - "Crear Reserva"

Participantes:

1. **Estudiante** (actor)
2. **API/InterfazWeb**
3. **ReservaService**
4. **DisponibilidadRepo**
5. **ReservaRepo**
6. **NotificacionService** (opcional)

Flujo de Mensajes:

1. **Estudiante** → **API**: `solicitarReserva(tutor, fechaHora)`
2. **API** → **ReservaService**: `crearReserva(estudiante, tutor, fechaHora)`
3. **ReservaService** → **DisponibilidadRepo**: `findFree(tutorId, fechaHora)`
4. **DisponibilidadRepo** → **ReservaService**: `disponibilidad`
5. **ReservaService** → **ReservaService**: `validarNoDobleReserva()`
6. **ReservaService** → **ReservaRepo**: `add(reserva)`
7. **ReservaService** → **DisponibilidadRepo**: `setState(OCUPADA)`
8. **ReservaService** → **NotificacionService**: `enviarConfirmacion()`
9. **ReservaService** → **API**: `reserva`
10. **API** → **Estudiante**: `confirmación`

Elementos a incluir:

- Líneas de vida para cada participante
- Mensajes sincrónicos (flechas sólidas)
- Mensajes de retorno (flechas discontinuas)
- Activaciones (rectángulos en líneas de vida)
- Fragmentos alt/opt si hay condicionales

Checklist de Verificación - Secuencia:

- 6 participantes con líneas de vida
- 10 mensajes en orden correcto
- Activaciones mostradas
- Mensajes de retorno incluidos
- Diagrama exportado como PDF

3.4 Máquina de Estados - Reserva

Estados:

- [●] (Inicial) → CREADA
- CREADA (estado normal)
- CONFIRMADA (reserva confirmada)
- CANCELADA (estado final)
- EXPIRADA (estado final)
- [●] (Final)

Transiciones Válidas:

- [Inicial] → CREADA [crear reserva]
- CREADA → CONFIRMADA [confirmar()]
- CREADA → CANCELADA [cancelar()]
- CREADA → EXPIRADA [timeout]
- CONFIRMADA → CANCELADA [cancelar()]
- CANCELADA → [Final]
- EXPIRADA → [Final]

Transiciones Inválidas (que deben prevenirse):

- ~~X~~ CANCELADA → CONFIRMADA
- ~~X~~ EXPIRADA → CONFIRMADA
- ~~X~~ CANCELADA → CREADA

Elementos a incluir:

- **Estados** representados como rectángulos con esquinas redondeadas
- **Estado inicial** como círculo relleno
- **Estados finales** como círculo con borde doble
- **Transiciones** como flechas etiquetadas
- **Condiciones/Eventos** entre corchetes

✓ Checklist de Verificación - Estados:

- 4 estados principales definidos
- Estado inicial y finales marcados
- 7 transiciones válidas
- Eventos/condiciones etiquetados
- Diagrama exportado como PDF

3.5 Resumen de la Parte A

Al completar esta sección deberías tener:

- 4 **diagramas UML** creados en Draw.io
- 4 **archivos PDF** exportados para entrega
- **Comprensión del dominio** a través del modelado visual
- **Base sólida** para la implementación en OCL y Python

Próximo paso: Con los modelos UML listos, procederemos a especificar formalmente las reglas de negocio usando OCL.

4. PARTE B - OCL CON USE

4.1 ¿Qué es OCL?

OCL (Object Constraint Language) es un lenguaje declarativo para expresar restricciones sobre modelos UML. Su propósito es eliminar la ambigüedad del lenguaje natural y proporcionar especificaciones precisas y verificables.

OCL permite expresar:

- **Invariantes:** Condiciones que siempre deben cumplirse
- **Precondiciones:** Condiciones antes de ejecutar una operación
- **Postcondiciones:** Condiciones después de ejecutar una operación
- **Consultas derivadas:** Valores calculados a partir del modelo

Lo que OCL NO es:

- No es un lenguaje de programación imperativo
- No "hace" cosas, declara condiciones
- No reemplaza el código, lo complementa y valida

4.2 ¿Qué es USE?

USE (UML-based Specification Environment) es una herramienta para:

- Escribir modelos UML como texto estructurado
- Agregar restricciones OCL al modelo
- Crear y manipular estados del sistema (snapshots)
- **Verificar automáticamente** si los invariantes se cumplen

Flujo de trabajo típico:

1. Escribir modelo UML en archivo .use
2. Definir restricciones OCL en el mismo archivo
3. Crear script .cmd con objetos y datos de prueba
4. Ejecutar USE para validar automáticamente

4.3 Ejercicio 1: Primer Contacto (Cars)

Objetivo: Validar un invariante simple sobre una clase.

Paso 1: Crear archivo Cars.use

Crea el archivo ocl/Cars.use:

```
model Cars
  class Car
    attributes
      mileage : Integer
  end

  constraints
    context Car
    inv MileageNotNegative: self.mileage >= 0
```

Explicación del código:

- model Cars: Declara el modelo
- class Car: Define una clase con un atributo
- constraints: Inicia la sección de restricciones OCL
- inv MileageNotNegative: Invariante que debe cumplirse siempre

Paso 2: Crear script Cars.cmd

Crea el archivo ocl/Cars.cmd:

```
!create smallCar : Car
!create bigCar : Car
!set smallCar.mileage := 2000
!set bigCar.mileage := -1500
```

Explicación de comandos:

- !create: Crea una instancia de la clase
- !set: Asigna valor a un atributo

Paso 3: Ejecutar USE

```
cd ocl/
use Cars.use
```

En el prompt use>:

```
use> open Cars.cmd
use> check
```

Resultado esperado:

- smallCar cumple el invariante (mileage = 2000 >= 0)
- bigCar **viola** el invariante (mileage = -1500 < 0)

USE detectará y reportará la violación automáticamente.

Paso 4: Consultas OCL en consola

Prueba estas consultas en el prompt use>:

```
use> ? Car.allInstances()->size()
use> ? Car.allInstances()->collect(c | c.mileage)
use> ? Car.allInstances()->select(c | c.mileage < 0)
use> ? Car.allInstances()->forAll(c | c.mileage >= 0)
```

Explicación de operaciones OCL:

- allInstances(): Todas las instancias de la clase
- collect(): Mapea una expresión sobre la colección
- select(): Filtra elementos que cumplen la condición
- forAll(): Verdadero si todos cumplen la condición

Checklist - Ejercicio Cars:

- Archivo Cars.use creado correctamente
- Script Cars.cmd ejecutado sin errores de sintaxis
- Violación de MileageNotNegative detectada por USE
- Consultas OCL ejecutadas y comprendidas

4.4 Ejercicio 2: Asociaciones (University)

Objetivo: Trabajar con association class y múltiples invariantes.

Paso 1: Crear archivo University.use

Crea el archivo ocl/University.use:

```
model University

class Student
    attributes
        id : Integer
        name : String
        email : String
end

class Course
    attributes
        code : String
        title : String
        credits : Integer
end

associationclass Enrollment
    between
        Student[*] role student
        Course[*] role course
    attributes
        grade : Integer
end

constraints
    context Student
        inv IdPositive: self.id > 0

    context Student
        inv UniqueEmail: Student.allInstances()->isUnique(s | s.email)

    context Course
        inv CreditsPositive: self.credits > 0

    context Enrollment
        inv GradeRange: self.grade >= 0 and self.grade <= 100
```

Elementos nuevos:

- **associationclass:** Asociación que es también una clase
- **between ... role:** Define los extremos de la asociación
- **isUnique():** Verifica que todos los valores sean únicos

Paso 2: Crear script University.cmd

Crea el archivo ocl/University.cmd:

```
-- 1) crear estudiantes
!create s1 : Student
!set s1.id := 1
!set s1.name := 'Ana'
!set s1.email := 'ana@uni.edu'

!create s2 : Student
!set s2.id := 2
!set s2.name := 'Luis'
!set s2.email := 'ana@uni.edu'    -- viola UniqueEmail

-- 2) crear curso
!create c1 : Course
!set c1.code := 'ARQ101'
!set c1.title := 'Arquitectura'
!set c1.credits := 4

-- 3) crear matrícula (Enrollment es association class)
!create e1 : Enrollment
!set e1.grade := 95
!insert (s1, c1) into Enrollment -- crea el vínculo de la association class
```

Comando nuevo:

- **!insert (obj1, obj2) into Association:** Crea enlace en association class

Paso 3: Ejecutar y validar

```
use University.use
```

En el prompt:

```
use> open University.cmd
use> check
```

Resultado esperado:

- ✗ Violación de UniqueEmail porque s1 y s2 tienen el mismo email

Paso 4: Corregir la violación

En el prompt use>:

```
use> !set s2.email := 'luis@uni.edu'
use> check
```

Ahora todos los invariantes deberían cumplirse ✅

✓ Checklist - Ejercicio University:

- Archivo University.use con association class correcto
- 4 invariantes definidos (IdPositive, UniqueEmail, CreditsPositive, GradeRange)
- Violación de UniqueEmail detectada inicialmente
- Corrección aplicada exitosamente
- Comprensión de isUnique() y association classes

4.5 Ejercicio 3: Sistema Completo de Tutorías

Objetivo: Modelo completo con las 3 restricciones OCL principales del sistema.

Paso 1: Crear archivo Reservas.use

Crea el archivo ocl/Reservas.use:

```
model Reservas

-- =====
-- ENUMS
-- =====
enum EstadoReserva { CREADA, CANCELADA, CONFIRMADA, EXPIRADA }
enum EstadoDisponibilidad { LIBRE, OCUPADA }

-- =====
-- CLASES
-- =====
class Tutor
  attributes
    nombre : String
end

class Estudiante
  attributes
    nombre : String
end

class Reserva
  attributes
    fechaHora : String
    estado : EstadoReserva
end

class Disponibilidad
  attributes
    fechaHora : String
    estado : EstadoDisponibilidad
end

-- =====
-- ASOCIACIONES (con roles)
-- =====
association TutorReserva between
  Tutor[1] role tutor
  Reserva[*] role reservas
end

association EstudianteReserva between
  Estudiante[1] role estudiante
  Reserva[*] role reservas
end

association TutorDisponibilidad between
  Tutor[1] role tutor
  Disponibilidad[*] role disponibilidades
end

-- =====
-- OCL CONSTRAINTS
-- =====
constraints
```

```

-- (1) Estado válido de Reserva
context Reserva
  inv EstadoValido:
    Set{ EstadoReserva::CREADA,
      EstadoReserva::CANCELADA,
      EstadoReserva::CONFIRMADA,
      EstadoReserva::EXPIRADA }->includes(self.estado)

-- (2) Reserva siempre tiene tutor y estudiante
context Reserva
  inv TieneTutorYEstudiante:
    not self.tutor.oclIsUndefined() and not self.estudiante.oclIsUndefined()

-- (3) No doble reserva activa para el mismo tutor en la misma fecha/hora
context Tutor
  inv NoDobleReservaMismoHorario:
    self.reservas
      ->select(r | r.estado <> EstadoReserva::CANCELADA and r.estado <> EstadoReserva::EXPIRADA)
      ->isUnique(r | r.fechaHora)

```

Explicación de restricciones:

1. **EstadoValido:** El estado debe pertenecer al conjunto de valores válidos
2. **TieneTutorYEstudiante:** Verifica que las asociaciones existan usando `oclIsUndefined()`
3. **NoDobleReservaMismoHorario:** Filtra reservas activas y verifica unicidad por fechaHora

Paso 2: Crear script Reservas.cmd

Crea el archivo ocl/Reservas.cmd:

```

-- =====
-- 1) Crear objetos base
-- =====
!create t1 : Tutor
!set t1.nombre := 'Carlos Tutor'

!create e1 : Estudiante
!set e1.nombre := 'Ana Estudiante'

-- =====
-- 2) Crear disponibilidades (para la regla 4)
-- =====
!create d1 : Disponibilidad
!set d1.fechaHora := '2026-01-20T10:00'
!set d1.estado := #LIBRE
!insert (t1, d1) into TutorDisponibilidad

!create d2 : Disponibilidad
!set d2.fechaHora := '2026-01-20T11:00'
!set d2.estado := #OCUPADA
!insert (t1, d2) into TutorDisponibilidad

-- =====
-- 3) Crear una reserva válida
-- =====
!create r1 : Reserva
!set r1.fechaHora := '2026-01-20T10:00'
!set r1.estado := #CONFIRMADA
!insert (t1, r1) into TutorReserva
!insert (e1, r1) into EstudianteReserva

-- =====
-- 4) Crear una reserva duplicada (misma fechaHora) para violar la regla 3
-- =====
!create r2 : Reserva
!set r2.fechaHora := '2026-01-20T10:00'
!set r2.estado := #CREADA
!insert (t1, r2) into TutorReserva
!insert (e1, r2) into EstudianteReserva

-- =====
-- 5) Crear una reserva con estado "sin asignar" para provocar problemas
-- (En algunos entornos queda undefined y puede romper EstadoValido)
-- =====
!create r3 : Reserva
!set r3.fechaHora := '2026-01-20T12:00'
!insert (t1, r3) into TutorReserva
!insert (e1, r3) into EstudianteReserva

```

Comandos nuevos:

- `#LIBRE`: Referencia a valor de enum en comandos USE
- Múltiples `!insert` para crear enlaces en diferentes asociaciones

Paso 3: Ejecutar USE

`use Reservas.use`

En el prompt:

```
use> open Reservas.cmd
use> check
```

Resultado esperado:

- ✗ **Violación de NoDobleReservaMismoHorario** porque r1 y r2 tienen la misma fechaHora y están activas
- ✗ **Possible violación de EstadoValido** en r3 si el estado queda undefined

Paso 4: Corregir violaciones

En el prompt use>:

```
use> !set r2.estado := #CANCELADA
use> !set r3.estado := #CREADA
use> check
```

Ahora todos los invariantes deberían cumplirse ✓

Paso 5: Consultas OCL en consola

Experimenta con estas consultas:

```
use> ? t1.reservas->size()
use> ? t1.disponibilidades->exists(d | d.fechaHora = '2026-01-20T10:00' and d.estado = EstadoDisponibilidad::LIBRE)
use> ? t1.reservas->select(r | r.estado = EstadoReserva::CONFIRMADA)
use> ? t1.disponibilidades->select(d | d.estado = EstadoDisponibilidad::LIBRE)
use> ? Reserva.allInstances()->collect(r | r.fechaHora)
```

✓ Checklist - Ejercicio Reservas:

- Archivo Reservas.use compilado sin errores
- 3 invariantes OCL implementados correctamente
- Script Reservas.cmd ejecutado
- Violaciones detectadas automáticamente por USE
- Correcciones aplicadas exitosamente
- Consultas OCL de navegación funcionando

4.6 Modelo Completo con Precondiciones y Postcondiciones (Tutorias.use)

Objetivo: Modelo MÁS COMPLETO con todas las clases, asociaciones y restricciones OCL avanzadas (pre/post).

Crea el archivo ocl/Tutorias.use:

```
model Tutorias

enum EstadoReserva { CREADA, CONFIRMADA, CANCELADA, EXPIRADA }
enum EstadoDisponibilidad { LIBRE, OCUPADA }
enum EstadoPago { PENDIENTE, PAGADO, REEMBOLSADO }

class Tutor
  attributes
    nombre : String
    especialidad : String
end

class Estudiante
  attributes
    nombre : String
    carnet : String
end

class Curso
  attributes
    codigo : String
    nombre : String
end

class Tema
  attributes
    nombre : String
end

class Ubicacion
  attributes
    tipo : String
    enlace : String
end

class Disponibilidad
  attributes
    fechaHora : String
    estado : EstadoDisponibilidad
end
```

```

class Reserva
  attributes
    fechaHora : String
    estado : EstadoReserva
end

class Pago
  attributes
    monto : Real
    estado : EstadoPago
end

class Notificacion
  attributes
    canal : String
    mensaje : String
end

class ReservaService
  operations
    crearReserva(estudiante : Estudiante, tutor : Tutor, fechaHora : String) : Reserva
    cancelarReserva(reserva : Reserva)
end

association TutorReserva between
  Tutor[1] role tutor
  Reserva[*] role reservas
end

association EstudianteReserva between
  Estudiante[1] role estudiante
  Reserva[*] role reservas
end

association TutorDisponibilidad between
  Tutor[1] role tutor
  Disponibilidad[*] role disponibilidades
end

association ReservaCurso between
  Reserva[0..1] role reserva
  Curso[0..1] role curso
end

association CursoTema between
  Curso[1] role curso
  Tema[*] role temas
end

association ReservaUbicacion between
  Reserva[1] role reserva
  Ubicacion[1] role ubicacion
end

association ReservaPago between
  Reserva[0..1] role reserva
  Pago[0..1] role pago
end

association ReservaNotificacion between
  Reserva[1] role reserva
  Notificacion[*] role notificaciones
end

constraints

context Reserva
  inv EstadoValido:
    Set{ EstadoReserva::CREADA,
        EstadoReserva::CONFIRMADA,
        EstadoReserva::CANCELADA,
        EstadoReserva::EXPIRADA }->includes(self.estado)

context Reserva
  inv TieneTutorYEstudiante:
    not self.tutor.oclIsUndefined() and not self.estudiante.oclIsUndefined()

context Tutor
  inv NoDobleReservaActivaMismoHorario:
    self.reservas
      ->select(r | r.estado <> EstadoReserva::CANCELADA and r.estado <> EstadoReserva::EXPIRADA)
      ->isUnique(r | r.fechaHora)

context ReservaService::crearReserva(estudiante : Estudiante, tutor : Tutor, fechaHora : String) : Reserva
  pre CrearReservaValida:
    tutor.disponibilidades->exists(d |
      d.fechaHora = fechaHora and d.estado = EstadoDisponibilidad::LIBRE
    )
    and
    tutor.reservas->select(r |
      r.estado <> EstadoReserva::CANCELADA and r.estado <> EstadoReserva::EXPIRADA
    )->forAll(r | r.fechaHora <> fechaHora)

context ReservaService::cancelarReserva(reserva : Reserva)

```

```

pre CancelarSoloSiActiva:
    reserva.estado = EstadoReserva::CREADA or reserva.estado = EstadoReserva::CONFIRMADA

context ReservaService::crearReserva(estudiante : Estudiante, tutor : Tutor, fechaHora : String) : Reserva
post ReservaQuedaCreada:
    result.estado = EstadoReserva::CREADA and
    result.fechaHora = fechaHora and
    result.tutor = tutor and
    result.estudiante = estudiante

```

Elementos avanzados:

- **9 clases completas** con todos los atributos
- **8 asociaciones** con multiplicidades específicas
- **3 invariantes** (reutilizados del modelo simplificado)
- **2 precondiciones** para operaciones de servicio
- **1 postcondición** que define el resultado esperado
- **result**: Palabra clave OCL que se refiere al valor de retorno

Este es el modelo **MÁS COMPLETO** que usaremos como base para la implementación Python.

5. PARTE C - IMPLEMENTACIÓN EN PYTHON

5.1 Estructura del Proyecto

Verifica que tienes la estructura correcta:

```

python/
└── tutorias/
    ├── __init__.py          # Hace que sea un paquete Python
    ├── domain.py            # Entidades del dominio
    ├── repositories.py      # Acceso a datos (in-memory)
    └── services.py          # Lógica de negocio
    └── tests/
        └── test_reservas.py  # Pruebas automatizadas

```

Principio importante: El dominio NO debe depender de frameworks externos. Solo Python puro con dataclasses y enums.

5.2 Archivo: domain.py

Crea el archivo python/tutorias/domain.py:

```

from __future__ import annotations
from dataclasses import dataclass
from enum import Enum

class EstadoReserva(str, Enum):
    CREADA = "CREADA"
    CONFIRMADA = "CONFIRMADA"
    CANCELADA = "CANCELADA"
    EXPIRADA = "EXPIRADA"

class EstadoDisponibilidad(str, Enum):
    LIBRE = "LIBRE"
    OCUPADA = "OCUPADA"

@dataclass
class Tutor:
    id: str
    nombre: str
    especialidad: str

@dataclass
class Estudiante:
    id: str
    nombre: str
    carnet: str

@dataclass
class Disponibilidad:
    id: str
    tutor_id: str
    fecha_hora: str
    estado: EstadoDisponibilidad

@dataclass
class Reserva:
    id: str
    tutor_id: str
    estudiante_id: str
    fecha_hora: str
    estado: EstadoReserva

```

Explicación del código:

- `from __future__ import annotations`: Permite usar type hints con strings
- `str, Enum`: Los enums heredan de str para serialización fácil
- `@dataclass`: Genera automáticamente `__init__`, `__repr__`, `__eq__`
- `Type hints`: Documentan los tipos esperados
- `IDs como string`: Más flexible para UUID o identificadores complejos

5.3 Archivo: repositories.py

Crea el archivo python/tutorias/repositories.py:

```
from __future__ import annotations
from dataclasses import replace
from typing import Dict, List, Optional

from .domain import Disponibilidad, EstadoDisponibilidad, Reserva

class InMemoryDisponibilidadRepo:
    def __init__(self) -> None:
        self._items: Dict[str, Disponibilidad] = {}

    def add(self, d: Disponibilidad) -> None:
        self._items[d.id] = d

    def find_free(self, tutor_id: str, fecha_hora: str) -> Optional[Disponibilidad]:
        for d in self._items.values():
            if d.tutor_id == tutor_id and d.fecha_hora == fecha_hora and d.estado == EstadoDisponibilidad.LIBRE:
                return d
        return None

    def set_state(self, disponibilidad_id: str, estado: EstadoDisponibilidad) -> None:
        d = self._items[disponibilidad_id]
        self._items[disponibilidad_id] = replace(d, estado=estado)

class InMemoryReservaRepo:
    def __init__(self) -> None:
        self._items: Dict[str, Reserva] = {}

    def add(self, r: Reserva) -> None:
        self._items[r.id] = r

    def get(self, reserva_id: str) -> Reserva:
        return self._items[reserva_id]

    def update(self, r: Reserva) -> None:
        self._items[r.id] = r

    def list_by_tutor(self, tutor_id: str) -> List[Reserva]:
        return [r for r in self._items.values() if r.tutor_id == tutor_id]
```

Explicación del código:

- **In-Memory**: Implementación simple usando diccionarios Python
- `replace()`: Función de dataclasses para crear copias con cambios
- **Type hints completos**: Documentan entradas y salidas
- **Métodos específicos**: `find_free()` implementa la consulta OCL
- **Separación de responsabilidades**: Un repo por aggregate root

5.4 Archivo: services.py

Crea el archivo python/tutorias/services.py:

```
from __future__ import annotations
from dataclasses import replace
from uuid import uuid4

from .domain import EstadoDisponibilidad, EstadoReserva, Estudiante, Reserva, Tutor
from .repositories import InMemoryDisponibilidadRepo, InMemoryReservaRepo

class DomainError(RuntimeError):
    pass

class ReservaService:
    def __init__(self, disp_repo: InMemoryDisponibilidadRepo, reserva_repo: InMemoryReservaRepo) -> None:
        self.disp_repo = disp_repo
        self.reserva_repo = reserva_repo

    def crear_reserva(self, estudiante: Estudiante, tutor: Tutor, fecha_hora: str) -> Reserva:
        disponibilidad = self.disp_repo.find_free(tutor.id, fecha_hora)
        if not disponibilidad:
            raise DomainError("No hay disponibilidad LIBRE del tutor para esa fecha/hora.")

        for r in self.reserva_repo.list_by_tutor(tutor.id):
            if r.fecha_hora == fecha_hora and r.estado not in (EstadoReserva.CANCELADA, EstadoReserva.EXPIRADA):
                raise DomainError("Ya existe una reserva activa del tutor en esa fecha/hora.")

        reserva = Reserva(
            id=str(uuid4()),
            tutor_id=tutor.id,
```

```

        estudiante_id=estudiante.id,
        fecha_hora=fecha_hora,
        estado=EstadoReserva.CREADA,
    )
    self.reserva_repo.add(reserva)
    self.disp_repo.set_state(disponibilidad.id, EstadoDisponibilidad.OCUPADA)
    return reserva

def cancelar_reserva(self, reserva_id: str) -> Reserva:
    reserva = self.reserva_repo.get(reserva_id)
    if reserva.estado not in (EstadoReserva.CREADA, EstadoReserva.CONFIRMADA):
        raise DomainError("Solo se puede cancelar una reserva si está CREADA o CONFIRMADA.")

    reserva_cancelada = replace(reserva, estado=EstadoReserva.CANCELADA)
    self.reserva_repo.update(reserva_cancelada)
    return reserva_cancelada

```

Explicación del código:

- **DomainError**: Excepción personalizada para violaciones de reglas de negocio
- **Validaciones OCL implementadas**: Las precondiciones se convierten en if statements
- **uuid4()**: Genera IDs únicos automáticamente
- **Transacionalidad**: La operación completa se ejecuta o falla
- **Estado consistente**: Actualiza tanto la reserva como la disponibilidad

Mapeo OCL → Python:

- `tutor.disponibilidades->exists(...)` → `disp_repo.find_free()`
- `tutor.reservas->forAll(...)` → bucle for con validación
- `pre CancelarSoloSiActiva` → `if reserva.estado not in (...)`

5.5 Archivo: tests/test_reservas.py

Crea el archivo `python/tests/test_reservas.py`:

```

import unittest

from tutorias.domain import Disponibilidad, EstadoDisponibilidad, EstadoReserva, Estudiante, Tutor
from tutorias.repositories import InMemoryDisponibilidadRepo, InMemoryReservaRepo
from tutorias.services import DomainError, ReservaService

class TestReservaService(unittest.TestCase):
    def setUp(self) -> None:
        self.tutor = Tutor(id="t1", nombre="Carlos Tutor", especialidad="Arquitectura")
        self.estudiante = Estudiante(id="e1", nombre="Ana Estudiante", carnet="AB2026")
        self.disp_repo = InMemoryDisponibilidadRepo()
        self.res_repo = InMemoryReservaRepo()
        self.service = ReservaService(self.disp_repo, self.res_repo)
        self.disp_repo.add(
            Disponibilidad(
                id="d1",
                tutor_id=self.tutor.id,
                fecha_hora="2026-01-20T10:00",
                estado=EstadoDisponibilidad.LIBRE,
            )
        )

    def test_crear_reserva_ok(self):
        r = self.service.crear_reserva(self.estudiante, self.tutor, "2026-01-20T10:00")
        self.assertEqual(r.estado, EstadoReserva.CREADA)
        self.assertEqual(r.fecha_hora, "2026-01-20T10:00")

    def test_crear_reserva_falla_sin_disponibilidad(self):
        with self.assertRaises(DomainError):
            self.service.crear_reserva(self.estudiante, self.tutor, "2026-01-20T11:00")

    def test_crear_reserva_falla_doble_reserva_activa(self):
        self.service.crear_reserva(self.estudiante, self.tutor, "2026-01-20T10:00")
        with self.assertRaises(DomainError):
            self.service.crear_reserva(self.estudiante, self.tutor, "2026-01-20T10:00")

    def test_cancelar_reserva_ok(self):
        r = self.service.crear_reserva(self.estudiante, self.tutor, "2026-01-20T10:00")
        r2 = self.service.cancelar_reserva(r.id)
        self.assertEqual(r2.estado, EstadoReserva.CANCELADA)

    def test_cancelar_reserva_falla_si_no_activa(self):
        r = self.service.crear_reserva(self.estudiante, self.tutor, "2026-01-20T10:00")
        r2 = self.service.cancelar_reserva(r.id)
        with self.assertRaises(DomainError):
            self.service.cancelar_reserva(r2.id)

if __name__ == "__main__":
    unittest.main()

```

Explicación del código:

- **setUp()**: Ejecutado antes de cada test, prepara el entorno
- **Fixtures**: Datos de prueba reutilizables (tutor, estudiante, disponibilidad)

- **assertRaises()**: Verifica que se lance la excepción esperada
- **Tests específicos**: Cada test valida una regla de negocio específica
- **Naming convention**: `test_[accion]_[escenario]_[resultado]`

Cobertura de casos:

1. **Caso feliz**: Crear reserva con disponibilidad
2. **Sin disponibilidad**: Falla según precondición OCL
3. **Doble reserva**: Falla según invariante OCL
4. **Cancelar válida**: Transición de estado permitida
5. **Cancelar inválida**: Violación de precondición OCL

5.6 Ejecutar las Pruebas

Opción 1: Con pytest (recomendado)

```
cd python/
pip install pytest
python -m pytest tests/test_reservas.py -v
```

Opción 2: Con unittest

```
cd python/
python -m unittest tests/test_reservas.py -v
```

Opción 3: Directo

```
cd python/
python tests/test_reservas.py
```

Salida esperada:

```
test_crear_reserva_ok ... ok
test_crear_reserva_falla_sin_disponibilidad ... ok
test_crear_reserva_falla_doble_reserva_activa ... ok
test_cancelar_reserva_ok ... ok
test_cancelar_reserva_falla_si_no_activa ... ok
```

Ran 5 tests in 0.003s

OK

Éxito: Si ves 5 tests OK, todas las reglas de negocio OCL están correctamente implementadas y validadas.

Checklist - Implementación Python:

- 4 archivos Python creados sin errores de sintaxis
- Enums y dataclasses implementados correctamente
- 2 repositorios in-memory funcionando
- ReservaService con validaciones OCL implementadas
- 5 pruebas automatizadas pasando (100% éxito)
- DomainError lanzado correctamente en violaciones

6. TRAZABILIDAD UML → OCL → CÓDIGO

Tabla de Trazabilidad Completa

Elemento UML	Restricción OCL	Implementación Python	Prueba
Clase Reserva.estado:	inv EstadoValido: Set{...}- class EstadoReserva(str, >includes(self.estado) Enum)		
EstadoReserva			<code>test_crear_reserva_ok</code>
Asociación Tutor[1] ↔ Reserva[*]	inv TieneTutorYEstudiante: not self.tutor.ocliIsUndefined() inv NoDobleReservaMismoHorario: for r in reservas: if ...->isUnique(r r.fecha_hora == fecha_hora... r.fechaHora) Invariante No Duplicados	reserva.tutor_id: str (required)	<code>test_crear_reserva_ok</code>
Secuencia "crear reserva"	pre CrearReservaValida: tutor.disponibilidades->exists(...)	disp_repo.find_free(tutor.id, fecha_hora)	<code>test_crear_reserva_falla_sin_disponibilidad</code>
Estado CREADA → CONFIRMADA	Transición válida en máquina de estados	<i>(no implementado en código Python básico)</i>	-

Elemento UML	Restricción OCL	Implementación Python	Prueba
Estado CREADA → CANCELADA	Transición válida en máquina de estados	service.cancelar_reserva()	test_cancelar_reserva_ok
Estado CANCELADA → X	Transición inválida	pre CancelarSoloSiActiva → validación en Python	test_cancelar_reserva_falla_si_no_activa
Postcondición crear	post ReservaQuedaCreada: result.estado = EstadoReserva::CREADA	estado=EstadoReserva.CREADA en constructor	test_crear_reserva_ok

Mapeo de Conceptos OCL → Python

OCCL	Python Equivalente
self.atributo	objeto.atributo
self.rol->size()	len(lista_objetos)
colección->exists(x condicion)	any(condicion for x in colección)
colección->forAll(x condicion)	all(condicion for x in colección)
colección->select(x condicion)	[x for x in colección if condicion]
colección->isUnique(x expresion)	len(set(expresion for x in colección)) == len(colección)
not objeto.oclIsUndefined()	objeto is not None
EstadoEnum::VALOR	EstadoEnum.VALOR

Lecciones Aprendidas

✓ Lo que funcionó bien:

- OCL como especificación:** Las restricciones OCL sirvieron como "tests de aceptación" precisos
- Validación temprana:** USE detectó inconsistencias antes de programar
- Trazabilidad directa:** Cada invariante OCL se mapea a código Python específico
- Tests como documentación:** Las pruebas explican el comportamiento esperado

⚠ Desafíos encontrados:

- Sintaxis OCL:** Curva de aprendizaje para operaciones de colección
- USE tool:** Algunos errores de sintaxis requieren experiencia para depurar
- Mapeo no trivial:** `isUnique()` y `forAll()` requieren implementación cuidadosa
- Estados:** La máquina de estados no se mapea directamente a código simple

⌚ Mejores prácticas identificadas:

- Empezar simple:** Cars → University → Reservas → Tutorías (complejidad incremental)
- Validar frecuentemente:** Ejecutar check en USE después de cada cambio
- Pruebas primero:** Escribir tests que fallen, luego implementar
- Excepciones específicas:** DomainError hace debug más fácil que RuntimeError genérico

7. ENTREGABLES Y RÚBRICA

7.1 Lista de Entregables

Documentación y Diagramas:

- Diagrama de Casos de Uso** (PDF exportado de Draw.io)
- Diagrama de Clases Completo** (PDF exportado de Draw.io)
- Diagrama de Secuencia "Crear Reserva"** (PDF exportado de Draw.io)
- Diagrama de Estados de Reserva** (PDF exportado de Draw.io)

Modelos OCL:

- Cars.use** - Modelo de primer contacto
- University.use** - Modelo con association class
- Reservas.use** - Modelo simplificado del sistema
- Tutorias.use** - Modelo completo con pre/post condiciones

Scripts de Validación:

- Cars.cmd** - Script de pruebas para Cars
- University.cmd** - Script de pruebas para University
- Reservas.cmd** - Script de pruebas para Reservas

Implementación Python:

- domain.py** - Entidades del dominio con enums y dataclasses
- repositories.py** - Repositorios in-memory
- services.py** - Lógica de negocio con validaciones OCL
- test_reservas.py** - Suite de pruebas automatizadas

Evidencias:

- Capturas de USE** mostrando violaciones detectadas y corregidas
- Reporte de ejecución de pruebas** (captura o archivo de texto)
- README.md** con instrucciones de instalación y ejecución
- Tabla de trazabilidad UML → OCL → Código** (formato PDF o Markdown)

7.2 Rúbrica de Evaluación (100 puntos)

Criterio	Puntos	Excelente (90-100%)	Bueno (70-89%)	Suficiente (60-69%)	Insuficiente (<60%)
Modelado UML	30				
- Casos de uso	10	4 actores + 9 casos + include/extend correctos	4 actores + 7-8 casos + relaciones	3 actores + 6 casos básicos	Menos de 6 casos o errores conceptuales
- Clases	10	9 clases completas + asociaciones + multiplicidades	7-8 clases + la mayoría de asociaciones	6 clases + asociaciones básicas	Menos de 6 clases o errores graves
- Secuencia y Estados	10	Ambos diagramas completos y correctos	1 diagrama completo + 1 básico	1 diagrama completo	Diagramas incompletos o incorrectos
OCL y USE	30				
- Archivos .use correctos	10	4 archivos compilando sin errores	3 archivos compilando	2 archivos compilando	1 o ningún archivo compilando
- Restricciones OCL	10	6+ restricciones (inv/pre/post) correctas	4-5 restricciones correctas	3 restricciones básicas	Menos de 3 restricciones
- Validación en USE	10	Evidencia de violaciones detectadas y corregidas + consultas	Violaciones detectadas	USE ejecutado básicamente	No hay evidencia de ejecución
Implementación Python	30				
- Dominio (domain.py)	10	Enums + dataclasses + type hints completos	Clases correctas + algunos type hints	Clases básicas sin type hints	Errores de sintaxis o estructura
- Servicios (services.py)	10	Validaciones OCL implementadas + excepciones	Algunas validaciones + DomainError	Lógica básica sin validaciones	No implementado o errores graves
- Pruebas (test_reservas.py)	10	5/5 tests pasando + buena cobertura	4/5 tests pasando	3/5 tests pasando	Menos de 3 tests o no ejecutan
Documentación y Trazabilidad	10				
- Trazabilidad	5	Tabla completa UML → OCL → Código	Tabla parcial pero útil	Algunos mapeos identificados	No hay trazabilidad
- README e instrucciones	5	Instrucciones claras + comandos + estructura	Instrucciones básicas	README mínimo	No hay documentación

7.3 Criterios de Aceptación

Para considerar el laboratorio **APROBADO**, debe cumplir:

Criterios mínimos obligatorios:

- Compilación exitosa:** Todos los archivos .use cargan sin errores en USE
- Violaciones detectadas:** Al menos 1 violación de invariante detectada y documentada
- Tests pasando:** Mínimo 3 de 5 pruebas Python ejecutándose exitosamente
- Código Python funcional:** Al menos