📄 Entrega 01: Modelado del Dominio y Materialización en Código

Curso: Programación Avanzada
Unidad: Arquitecturas Empresariales y Modelado de Negocio
Tiempo estimado: Semanas 1 a 5
Equipo: Máximo 3 integrantes

1. Objetivo de la Entrega

Diseñar y materializar en código el núcleo del **Sistema de Triage y Gestión de Solicitudes Académicas**. Los estudiantes deben demostrar su capacidad para:

- Analizar el dominio del problema e identificar sus reglas de negocio.
- Diseñar un modelo de dominio coherente usando UML.
- Traducir ese modelo a código Java expresivo y bien estructurado.
- Validar las reglas de negocio mediante pruebas unitarias.

> **Esta entrega combina artefactos de análisis/diseño con código funcional probado**, coherente con lo trabajado en las guías 01 a 05.1.

2. Alcance Funcional

RF	Descripción	Cobertura
RF-01	Registro de solicitudes	Diseño + Código
RF-02	Clasificación de solicitudes	Diseño + Código
RF-03	Priorización de solicitudes	Diseño + Código
RF-04	Gestión del ciclo de vida (estados)	Diseño + Código

RF	Descripción	Cobertura
RF-06	Historial auditable	Diseño + Código
RF-08	Cierre de solicitudes (reglas)	Diseño + Código
RF-11	Independencia de IA en el diseño	Diseño
RF-13	Definición de roles	Diseño + Código

> 📌 Los RF restantes (API REST, asignación operativa de responsables, consultas, IA) quedan para entregas posteriores, pero el diseño debe contemplar su existencia futura.

3. Entregables

A. Componente de Análisis y Diseño

A.1 — Glosario de Lenguaje Ubicuo

Documento que defina los términos clave del dominio. Para cada concepto incluir:

Campo	Descripción
Nombre	Término del dominio
Definición	Significado en el contexto del sistema
Clasificación	Entidad, Value Object, Agregado o Servicio de Dominio

> 💡 Referirse a lo trabajado en las guías **01** (lenguaje compartido) y **04** (identificación de conceptos).

A.2 — Modelo de Dominio (Diagrama de Clases UML)

Representación visual del modelo de **objetos de negocio** (no es un diagrama E-R de base de datos):

- **Entidades**: `Solicitud` , `Usuario` (y subtipos si aplica: `Estudiante` , `Funcionario`)

- **Value Objects**: `Email`, `Prioridad`, `TipoSolicitud`, `EstadoSolicitud`, `CodigoSolicitud`, y otros identificados
- **Agregados**: Identificar la raíz de agregado (`Solicitud`) y sus componentes internos (`EventoHistorial`, etc.)
- **Relaciones**: Con cardinalidad correcta (ej. una `Solicitud` tiene N registros en su `Historial`)
- **Atributos**: Tipos de datos y modificadores de acceso

A.3 — Diagrama de Estados (Ciclo de Vida de la Solicitud) (OPCIONAL)

Según el RF-04, modelar el flujo de estados de la solicitud:

- **Estados obligatorios**: `REGISTRADA` → `CLASIFICADA` → `EN_ATENCION` → `ATENDIDA` → `CERRADA`
- **Transiciones permitidas**: Qué acciones disparan cada cambio de estado
- **Restricciones**: Transiciones prohibidas (ej. no se puede cerrar si no ha sido atendida)
- **Reglas de cierre** según RF-08

A.4 — Documento de Reglas de Negocio

Para cada regla identificada, documentar:

- ¿Qué acción regula?
- ¿Qué condición debe cumplirse?
- ¿A qué RF está asociada?
- ¿Dónde vive en el código? (agregado, value object o servicio de dominio)

B. Componente de Materialización en Código

B.1 — Proyecto Spring Boot configurado

- Creado con **Spring Initializr** (Spring Boot 4.x, Gradle, Java 25)
- Estructura de paquetes **orientada al dominio**:

```
co.edu.uniquindio.proyecto
├── domain/
│   ├── entity/
│   ├── valueobject/
│   ├── service/        (si aplica)
│   └── exception/
├── application/        (vacío por ahora)
└── infrastructure/     (vacío por ahora)
```

- Repositorio Git configurado en GitHub

B.2 — Entidades del Dominio

Implementar en Java las entidades identificadas en el modelo:

- Con **identidad propia** (IDs tipados como Value Objects)
- Con **comportamiento** que exprese las reglas del dominio (modelos ricos, no anémicos)
- **Sin setters públicos**: el estado cambia a través de métodos de negocio
- Con **nombres del lenguaje ubicuo**

B.3 — Value Objects

Implementar los value objects identificados:

- **Inmutables** (preferiblemente usando `record` de Java)
- Con **auto-validación** en el constructor
- Con **igualdad por valor** (`equals` / `hashCode`)
- Mínimo: `Email` , `Prioridad` , `TipoSolicitud` , `EstadoSolicitud` , `CodigoSolicitud`

B.4 — Agregado Raíz: `Solicitud`

La clase `Solicitud` como agregado raíz debe:

- Controlar el acceso a sus componentes internos (historial)
- Implementar las transiciones de estado con validación

- Proteger las siguientes invariantes:
  - ✅ Toda acción relevante se registra en historial
  - ✅ No se puede modificar historial directamente (solo a través de la raíz)
  - ✅ Las transiciones de estado son válidas
  - ✅ Una solicitud cerrada no puede modificarse
  - ✅ Estado e historial están siempre sincronizados
- Exponer métodos de negocio como: `clasificar(...)`, `priorizar(...)`, `atender(...)`, `cerrar(...)`

B.5 — Servicios de Dominio (si aplica)

Si se identifican reglas que involucran más de una entidad o que no pertenecen naturalmente a un solo agregado, implementarlas en el paquete `domain.service`.

B.6 — Pruebas Unitarias del Dominio

Siguiendo lo trabajado en la guía **05.1**:

Tipo de prueba	Qué valida
Value Objects	Validación en construcción, inmutabilidad, igualdad por valor
Entidades	Cambios de estado controlados, reglas en métodos de comportamiento
Agregado	Invariantes protegidas, transiciones válidas e inválidas
Servicios de Dominio	Lógica de coordinación, con Mockito si requieren dependencias

Requisitos:

- Patrón **AAA** (Arrange, Act, Assert)
- **100% de las reglas de negocio** identificadas deben tener prueba asociada
- Todas las pruebas deben estar **en verde** ✅

4. Formato de Entrega

1. **Repositorio en GitHub** — uno por grupo (máximo 3 integrantes)
2. `README.md` con:
   - Nombre de los integrantes
   - Descripción breve del proyecto basada en el contexto institucional
   - Instrucciones para compilar y ejecutar las pruebas
3. **Directorio** `/docs` con:
   - Diagrama de clases UML (`.png` o `.pdf`)
   - Diagrama de estados (`.png` o `.pdf`)
   - Glosario de lenguaje ubicuo (`.md` o `.pdf`)
   - Documento de reglas de negocio (`.md` o `.pdf`)
4. **Código fuente** en `src/main/java` — paquete `domain` completo
5. **Pruebas unitarias** en `src/test/java` — todas en verde

5. Criterios de Evaluación

Criterio	Peso	Descripción
Modelo de Dominio (UML)	25%	Entidades, VOs y agregados reflejan los requisitos. Relaciones y cardinalidad correctas.
Diagrama de Estados	5%	Cubre el ciclo de vida completo. Transiciones válidas e inválidas claras. Sin estados huérfanos.
Glosario y Reglas	10%	Lenguaje ubicuo definido. Reglas documentadas y trazables a los RFs.
Materialización del Dominio	35%	Código limpio y expresivo. Reglas encapsuladas en el dominio. Sin dependencias de infraestructura. VOs inmutables y auto-validados.
Pruebas Unitarias	20%	Cubren VOs, entidades y agregado. Validan camino feliz e invariantes violadas. Todas en verde.
Calidad General	10%	Repositorio organizado, README profesional, código documentado, uso del lenguaje del dominio.

> **La nota de la entrega se ve afectada por la sustentación oral del trabajo, la cual se califica de 0 a 1 y se multiplica por la nota obtenida en la entrega.**

6. Trazabilidad: Guías → Entregables

Guía	Competencia adquirida	Se evalúa en
01 — Introducción a la Programación Empresarial	Dominio, reglas, lenguaje compartido	Glosario, Reglas de Negocio
02 — Instalación de Entorno de Desarrollo	Herramientas configuradas	Proyecto funcional
03 — Proyecto Spring Boot / Gradle	Proyecto base con estructura DDD	Estructura de paquetes (B.1)
04 — Modelado del Dominio I	Entidades, Value Objects	UML (A.2), Código (B.2 y B.3)
05 — Modelado del Dominio II	Agregados, invariantes, servicios	UML (A.2), Código (B.4 y B.5)
05.1 — Pruebas Unitarias del Dominio	Pruebas AAA, JUnit 5, Mockito	Pruebas (B.6)

💡 Tip

> "Recuerden que el **RF-11** dice que el sistema debe funcionar sin IA. En su diseño de clases, asegúrense de que la 'Sugerencia de IA' sea un componente que se pueda 'enchufar' o 'desenchufar' sin romper la lógica de registro de la solicitud."

> ⚠️ **El dominio construido en esta entrega será la base sobre la cual se construirán las capas de aplicación, persistencia y presentación en las entregas posteriores. Un dominio bien diseñado facilita enormemente el trabajo futuro.**