Programa de Ingeniería de Sistemas y Computación
Universidad del Quindío

Curso: Programación Avanzada
Guía: 05
Título: Modelado del Dominio II – Agregados y Reglas de Negocio
Duración estimada: 120 minutos
Docente: Christian Andrés Candela

OBJETIVO

En la guía anterior identificamos:

- entidades,
- value objects,
- y reglas de negocio del dominio.

En esta guía daremos un paso clave:

> pasar de **conceptos identificados** a **comportamiento explícito en el dominio**.

El objetivo **no es introducir infraestructura**, sino aprender a:

- proteger reglas de negocio,
- agrupar correctamente responsabilidades,
- y entender por qué ciertas reglas viven juntas.

CONTEXTUALIZACIÓN TEÓRICA

¿Qué es un Agregado?

Un **Agregado** es un conjunto de objetos del dominio (entidades y value objects) que:

- se tratan como una sola unidad,
- protegen reglas de negocio importantes (invariantes),
- y controlan cómo se modifica su estado.

Cada agregado tiene:

- una **raíz de agregado**, que controla el acceso a los objetos del agregado,
- y uno o más objetos que dependen de ella.

> El agregado existe para **evitar estados inválidos** del dominio.

Referencia: Evans, E. (2003). *Domain-Driven Design*, Capítulo 6: "Aggregates"

Raíz de Agregado (Aggregate Root)

La entidad principal del agregado que actúa como punto de entrada. Solo la raíz puede ser referenciada desde fuera del agregado.

Invariante

Regla de negocio que debe ser **siempre verdadera**. Los agregados garantizan que sus invariantes se mantengan.

Ejemplo: "Una solicitud cerrada no puede cambiar de estado"

La Solicitud como Agregado

En nuestro dominio, la **Solicitud** es un claro candidato a ser un agregado.

¿Por qué?

- tiene identidad propia,
- cambia de estado,
- concentra reglas importantes,
- y coordina otros conceptos (estado, prioridad, tipo).

Todas las modificaciones relevantes a una solicitud deben pasar por su **raíz de agregado**.

¿Por qué Agregados?

Problema: Sin agregados, las operaciones pueden violar invariantes

```
// ❌ Problema: Acceso directo sin control
class SolicitudService {
    void cambiarEstado(String solicitudId, EstadoSolicitud nuevo) {
        Solicitud s = repo.findById(solicitudId);
        EventoHistorial evento = new EventoHistorial(...);

        // Se puede crear historial sin actualizar solicitud
        historialRepo.save(evento);

        // O actualizar solicitud sin registrar en historial
        s.setEstado(nuevo);  // ⚠️ Inconsistencia!
    }
}
```

Ejemplo de materialización del agregado (dominio puro)

Antes de pensar en frameworks, veamos cómo se expresa el agregado **solo con Java**.

```
public class Solicitud {

    private final SolicitudId id;
    private EstadoSolicitud estado;
    private Prioridad prioridad;
    private TipoSolicitud tipo;
    private Usuario responsable;

    public void asignarResponsable(Usuario responsable) {
        if (this.estado == EstadoSolicitud.CERRADA) {
            throw new ReglaDominioException("No se puede asignar responsable a una soli
        }
        this.responsable = responsable;
    }

    public void cerrar() {
        if (this.responsable == null) {
            throw new ReglaDominioException("No se puede cerrar una solicitud sin respo
        }
        if( this.estado != EstadoSolicitud.ATENDIDA) {
            throw new ReglaDominioException("No se puede cerrar una solicitud que no es
        }
        this.estado = EstadoSolicitud.CERRADA;
    }
}
```

Observa que:

- las reglas viven dentro del dominio,
- el estado no se cambia libremente,
- y el agregado protege su consistencia.

Principios de Agregados

1. **Límite de consistencia:** Todo dentro del agregado debe ser consistente
2. **Raíz única:** Solo una entidad es la raíz
3. **Acceso controlado:** Solo la raíz es accesible desde fuera
4. **Transacciones:** Un agregado = una transacción
5. **Referencias:** Agregados externos se referencian solo por ID

Servicio de Dominio

Algunas reglas:

- no pertenecen claramente a una sola entidad,
- coordinan varios objetos del dominio,
- o representan una operación del negocio.

Para estos casos usamos se hace necesario crear un componente que contiene dicha lógica de negocio y son denominados **Servicios de Dominio**.

Ejemplo conceptual:

```
public class AsignacionSolicitudService {

    public void asignar(Solicitud solicitud, Usuario responsable) {
        // Validaciones adicionales o coordinación con otros objetos del dominio
        // ...
        solicitud.asignarResponsable(responsable);
    }
}
```

> Un servicio de dominio **no contiene lógica técnica**, solo lógica del negocio.

PARTE 1: MATERIALIZACIÓN DEL DOMINIO

Materializaremos en Java:
• las entidades,
• los value objects,
• y las reglas de negocio identificadas previamente.

📦 Paquete base del dominio

Todo el código del dominio vivirá durante todo el semestre en el siguiente paquete base:

```
co.edu.uniquindio.solicitudes.domain
```

Estructura sugerida:

```
domain
 ├─ entity
 ├─ valueobject
 ├─ service        (opcional)
 └─ exception      (opcional)
```

> En el paquete exception se definen las excepciones que se utilizan en el dominio.

1. A partir del modelo trabajado en la Guía 04, cree las siguientes clases Java, identificando con claridad si son entidades o value objects:
    - Solicitud
    - Usuario
    - CodigoSolicitud
    - Prioridad
    - EstadoSolicitud
    - Email
    - TipoSolicitud

Recomendaciones:
- Los value objects deben ser inmutables
- No use setters
- Valide reglas simples en los constructores
- Use nombres del lenguaje del dominio

2. Reglas dentro del Agregado

Considere a Solicitud como Agregado Raíz.

Implemente en esta clase métodos que representen reglas del negocio, por ejemplo:
- asignar responsable
- cambiar estado
- cerrar una solicitud

Ejemplo conceptual:

Observe que:
- el estado no se modifica directamente,
- las reglas viven dentro del dominio,
- el agregado protege su coherencia.

Invariantes Garantizadas por el Agregado

1. ✅ **Toda acción se registra en historial**
2. ✅ **No se puede modificar historial directamente** (solo a través de la raíz)
3. ✅ **Las transiciones de estado son válidas**
4. ✅ **Una solicitud cerrada no puede modificarse**
5. ✅ **El estado y el historial están siempre sincronizados**

PARTE 2: SERVICIOS DE DOMINIO

1. Reglas fuera del agregado (si aplica)

Si identifica reglas que:
- involucran más de una entidad, o
- no pertenecen naturalmente a una sola,

pueden modelarse como Servicios de Dominio en el paquete:

domain.service

Ejemplo:
- validaciones cruzadas
- decisiones de negocio que no modifican directamente el estado

Ejemplo NotificadorSolicitudes

```
@Service  // Servicio de Dominio
public class NotificadorSolicitudes {

    private final UsuarioRepository usuarioRepository;

    /**
     * Determina quién debe ser notificado basándose en reglas de negocio
     */
    public List<String> determinarDestinatarios(Solicitud solicitud,
                                                TipoNotificacion tipo) {
        return switch (tipo) {
            case NUEVA_SOLICITUD -> {
                // Notificar a coordinadores activos
            }
            case ASIGNACION -> {
                // Notificar al responsable y al solicitante
            }
            case CAMBIO_ESTADO -> {
                // Notificar al solicitante
            }
            case CIERRE -> {
                // Notificar al solicitante y al responsable
            }
        };
    }
}
```

EVALUACIÓN O RESULTADO

Al finalizar esta actividad debe existir:
- un paquete domain completamente materializado en Java,
- entidades y value objects coherentes con el dominio,
- reglas de negocio expresadas como comportamiento,
- cero dependencias técnicas o de framework.

PRÓXIMA ACTIVIDAD

Para prepararse para la **Guía 06: Diseño de APIs desde el Dominio**, los estudiantes deben:

1. **Completar la implementación** del agregado `Solicitud` con todos los métodos de negocio
2. **Crear tests unitarios** para validar las invariantes
3. **Leer:**
   - Fowler, M. - "Repository Pattern"
   - Spring Data Documentation

REFERENCIAS BIBLIOGRÁFICAS

1. Evans, E. (2003). *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley. Capítulo 6: "Aggregates"
2. Vernon, V. (2013). *Implementing Domain-Driven Design*. Addison-Wesley. Capítulo 10: "Aggregates"
3. Fowler, M. (2002). *Patterns of Enterprise Application Architecture*. Addison-Wesley.
4. Richardson, C. (2018). *Microservices Patterns*. Manning Publications.
5. Spring Data MongoDB - Aggregates: https://docs.spring.io/spring-data/mongodb/reference/