Patrones y Prácticas de Diseño en PAE

Patrones arquitectónicos

Clean Architecture

El proyecto sigue Clean Architecture con tres anillos:

1. **Entities** (MachineData/RutaPAEData)
   - Modelos de negocio
   - Independientes de frameworks
   - Lógica de negocio local

1. **Use Cases** (MachineDomain/RutaPAEDomain)
   - Coordinación de flujos
   - Implementación de reglas
   - Orquestación

1. **Frameworks & Drivers** (Machine/RutaPAE)
   - UI
   - Bases de datos
   - APIs externas

Hexagonal Architecture (Ports & Adapters)

- **Puerto**: `P2PManager`, `Hardware`, `Camera2Service`
- **Adaptador**: `DirectLink`, `ScaleManager`, `Camera2Service`
- El core (dominio) no depende de estos

Patrones de diseño

State Pattern

Ubicación: `MachineDomain/state/StateManager.kt`

```
Estado: interfaz común
├── WaitingForWeight
├── CaptureImages
├── CaptureFace
├── ComparingWeights
├── GenerateEmbedding
├── VerifyInDatabase
├── SaveDelivery
└── WaitForWeightRemoved
```

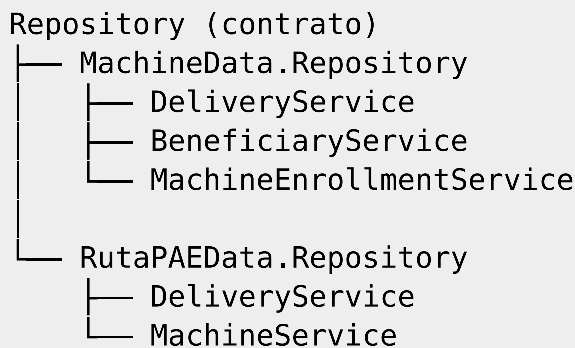
```
StateManager: gestor que ejecuta transiciones
```

Ventajas:

- Flujo lineal y predecible
- Fácil agregar nuevos estados
- Manejo de errores centralizado

Repository Pattern

Ubicación: `MachineData/repository/`, `RutaPAEData/repository/`



```
Repository (contrato)
├── MachineData.Repository
│   ├── DeliveryService
│   ├── BeneficiaryService
│   └── MachineEnrollmentService
│
└── RutaPAEData.Repository
    ├── DeliveryService
    └── MachineService
```

Ventajas:

- Abstrae acceso a datos
- Facilita testing (mock)
- Cambiar BD sin afectar lógica

Observer Pattern

Implementación: Kotlin `Flow` y Emisores

```
// Machine: emite cambios de estado
StateNameEmitter.emit("CapturingFace")

// RutaPAE: emite progreso de sincronización
DeliverySyncUiEmitter.update(estado)

// UI observa
LaunchedEffect {
    StateNameEmitter.collect { nombre ->
        // actualizar UI
    }
}
```

Ventajas:

- Desacoplamiento entre productores y consumidores
- Reactividad
- Múltiples observadores

Factory Pattern

StateManager: crea y ejecuta estados

```
val states = listOf(
    WaitingForWeight(),
    CaptureImages(),
    CaptureFace(),
    // ...
)

while (isRunning) {
    val state = workflow[currentIndex] // factory
    state.run(next, retry)
}
```

Ventajas:

- Creación centralizada
- Configuración flexible
- Fácil agregar variantes

Singleton Pattern

DomainManager en RutaPAE:

```
object DomainManager : Initializer, Closeable {
    var p2pManager: P2PManager? = null

    fun init(context: Context): Boolean {
        if (initialized) return true
        // ...
    }
}
```

Ventajas:

- Instancia única global
- Control centralizado
- Thread-safe

Builder Pattern (implicit)

Contract: construcción de modelos

```
data class P2PMachineState(
    val id: String = "",
    val name: String = "",
    val unsyncedDeliveries: Long = 0,
    // ...
)

// Creación flexible
P2PMachineState(
    id = "123",
    name = "Máquina 1"
)
```

Principios SOLID

Single Responsibility

```
StateManager: solo orquesta máquina de estados
DeliveryService: solo gestiona entregas
BeneficiaryService: solo gestiona beneficiarios
```

Open/Closed

```
// Abierto para extensión
interface State {
    suspend fun run(next: () -> Unit, retry: (String) -> Unit)
}

// Cerrado para modificación
class NewState : State { ... }
```

Liskov Substitution

```
// Cualquier State puede ocupar el lugar de otro
val states: List<State> = listOf(
    WaitingForWeight(),
    CaptureImages(),
    // ...
)
```

Interface Segregation

```
// Interfaces pequeñas y específicas
```

```
interface Initializer {
    fun init(context: Context): Boolean
}

interface Closeable {
    fun close(context: Context): Boolean
}

// vs una sola interfaz grande
```

Dependency Inversion

```
// Domain depende de abstracción (Repository)
class StateManager(val repository: Repository)

// Data implementa la abstracción
class RepositoryImpl : Repository
```

Patrones de concurrencia

Coroutines

Machine Domain:

```
// State.run() ejecuta en contexto de corrutina
suspend fun run(next: () -> Unit, retry: (String) -> Unit)
```

RutaPAE Domain:

```
// P2PManager usa corrutinas para operaciones P2P
suspend fun syncDeliveriesFromMachine(machineDatabaseId: Long)
```

StateFlow

```
// P2PManager mantiene state reactivo
val machines: StateFlow<Set<P2PPeer>>
val peers: StateFlow<Set<P2PPeer>>

// UI observa
LaunchedEffect {
    machines.collect { peers ->
        // actualizar UI
    }
}
```

Patrones de persistencia

VectorDB ORM

```
@DataTable("Beneficiary")
data class Beneficiary(
    @VectorColumn(dimensions = 512, distanceMetric = DistanceMetric.COSINE)
    val embedding: FloatArray,
    // ...
)

// Búsqueda vectorial
repository.beneficiaries
    .nearestNeighbors(embeddings, queryVector)
```

Repository + Service

```
// Data layer abstraction
class Repository {
    val beneficiaries: Table<Beneficiary>
    val deliveries: Table<Delivery>
}

// Service expone API
object BeneficiaryService {
    fun findSimilar(embedding: FloatArray): List<Beneficiary>
}
```

Patrones de error handling

Retry en State

```
try {
    captureImage()
    next() // éxito
} catch (e: Exception) {
    retry("Error capturando imagen: ${e.message}")
    // retrocede a estado anterior
}
```

Result Type

```
suspend fun syncDeliveriesFromMachine(): SyncDeliveriesResult {
```

```
    return SyncDeliveriesResult(
        success = true,
        message = "Completado",
        fetched = 10,
        failed = 0
    )
}
```

WorkManager Retry

```
// WorkManager reintentatrabajos automáticamente
Result.retry() // reintentar
Result.failure() // fallar
Result.success() // éxito
```

Patrones de comunicación

Event-driven

Machine:

```
StateNameEmitter.emit(stateName) // event
StateMessageEmitter.emit(message) // event
```

RutaPAE:

```
DeliverySyncUiEmitter.update(state) // event
SyncRunningEmitter.emit(running) // event
```

Request-Reply

P2P:

```
// Machine recibe GET /p2p/machine/deliveries/1/1
// responde con P2PDeliveriesPageResponse
```

Publisher-Subscriber

```
// StateNameEmitter (publicador)
Flow<String> (suscriptores)

// detalles: 1 a N
// desacoplamiento: productor no conoce suscriptores
```

Prácticas de testing (implícitas)

Dependency Injection

```
// Facilita mock
class StateManager(
    val repository: Repository, // inyectable
    val hardware: Hardware      // inyectable
)
```

Separación de responsabilidades

```
// Fácil probar cada servicio
DeliveryService.getAll() // no depende de UI
BeneficiaryService.findSimilar() // no depende de P2P
```

Contrato compartido

```
// Contract proporciona modelos testables
data class P2PMachineState(...) // serializable, comparable
```

Convenciones de código

Nomenclatura

```
// Services
class MachineService { }
class DeliveryService { }

// Managers
class StateManager { }
class P2PManager { }
class ScaleManager { }

// Emitters
object StateNameEmitter { }
object DeliverySyncUiEmitter { }

// Interfaces
interface State { }
interface Repository { }
interface Initializer { }
```

```
// Data classes
data class Delivery(...) { }
data class P2PMachineState(...) { }
```

Estructura de paquetes

```
co.ada.paemachine/
├── screens/        # UI Composable
├── viewmodels/     # State holders
├── di/             # Inyección
└── ...

co.ada.domain/
├── state/          # State pattern
├── services/       # Business logic
├── hardware/       # Hardware abstractions
└── ...

co.ada.data/
├── entities/       # Data models
├── repository/     # Repository pattern
└── services/       # Data access
```

Code style

- **Lenguaje**: Kotlin 100%
- **IDE**: Android Studio
- **Formatter**: Kotlin conventions
- **Compile target**: Java 17
- **Null safety**: Non-null by default

From:
http://wiki.adacsc.co/ - **Wiki**

Permanent link:
http://wiki.adacsc.co/doku.php?id=ada:howto:sicoferp:factory:new-migracion-sicoferp:patterns

Last update: **2026/04/07 20:00**