Mejores Prácticas en PAE

Convenciones de código

Nomenclatura

```
// ✅ Bueno

// Services: servicio de negocio
class DeliveryService { }
class MachineService { }

// Managers: coordinadores
class StateManager { }
class P2PManager { }

// Emitters: publicadores de eventos
object StateNameEmitter { }
object DeliverySyncUiEmitter { }

// Repositories: acceso a datos
interface Repository { }

// ViewModel: state holder para UI
class LaboratoryViewModel { }

// Data classes: entidades
data class Delivery(...)
data class Beneficiary(...)

// Enums: conjuntos finitos
enum class P2PMachineStatus { Active, Inactive, Error }

// Companions: funciones/propiedades de clase
companion object {
    fun create(): StateManager { }
}

// ❌ Malo
class DeliveryLogic { }           // vago
class D { }                        // demasiado corto
class BeneficiaryRepoImpl { }      // sufijo "Impl"
val d = Delivery()                // variable de 1 letra
```

Paquetes

```
co.ada.paemachine/
├── screens/                        # Composables de pantalla
│   ├── LaboratoryScreen.kt
│   ├── ShiftSelectionScreen.kt
│   └── ...
├── viewmodels/                     # State holders
│   └── LaboratoryViewModel.kt
├── navigation/                     # Routing
│   └── AppNavigation.kt
├── di/                             # Dependency injection
│   └── AppModule.kt
└── ...

co.ada.domain/
├── state/                          # State machine
├── services/                       # Business logic
├── hardware/                       # Interfaces hardware
├── emitters/                       # Observable streams
└── ...

co.ada.data/
├── entities/                       # DB models
├── repository/                     # Data access
├── services/                       # CRUD helpers
└── ...
```

Archivos

- **Un archivo por clase/interface**: exceptions, no hay helpers files
- **Máximo 500 líneas por archivo**: si es más grande, dividir
- **Nombrar archivos por clase**: `StateManager.kt` not `States.kt`

Patrones y anti-patrones

Patterns a usar

1. Repository Pattern

```
// ✅ Bueno: abstracción clara
interface Repository {
    val deliveries: Table<Delivery>
    fun getDelivery(id: Long): Delivery?
}

// Uso
```

```
val delivery = repository.getDelivery(1)
```

2. State Pattern

```
// ✅ Bueno: flujo predecible
interface State {
    suspend fun run(next: () -> Unit, retry: (String) -> Unit)
}

// Transiciones claras
WaitingForWeight → CaptureImages → CaptureFace
```

3. Dependency Injection (DI)

```
// ✅ Bueno: inyección en constructor
class StateManager(
    val repository: Repository,
    val hardware: Hardware
) { }

// Fácil testear
val testManager = StateManager(mockRepository, mockHardware)
```

4. Extension Functions

```
// ✅ Bueno: funciones de utilidad
fun String.toDeliveryId(): Long {
    return this.toLong()
}

fun Float.isValidWeight(): Boolean {
    return this > 0.5f && this < 100f
}

// Uso limpio
"123".toDeliveryId()
weight.isValidWeight()
```

5. Sealed Classes

```
// ✅ Bueno: tipos discriminados
sealed class SyncResult {
    data class Success(val count: Int) : SyncResult()
    data class Failure(val error: String) : SyncResult()
    object InProgress : SyncResult()
```

```
}

// Pattern matching exhaustivo
when (result) {
    is SyncResult.Success -> {}
    is SyncResult.Failure -> {}
    is SyncResult.InProgress -> {}
    // compiler fuerza todos los casos
}
```

Anti-patterns a evitar

1. God Objects

```
// Malo: responsabilidades múltiples
class Machine {
    fun captureDelivery() { }
    fun syncWithRutaPAE() { }
    fun configureHardware() { }
    fun generateReports() { }
    // ... 50 métodos más
}

// Bueno: responsabilidad única
class StateManager {
    fun cycle(): StateResult { }  // solo orquesta estados
}

class P2PManager {
    fun syncDeliveries() { }      // solo P2P
}
```

2. Null hell

```
// Malo: nullable everywhere
fun getDelivery(id: Long): Delivery? {
    val repo = getRepository()
    if (repo == null) return null
    val delivery = repo.find(id)
    if (delivery == null) return null
    // ...
    return delivery
}

// Bueno: null-safety
fun getDelivery(id: Long): Delivery = repository.getDelivery(id)
```

```
    ?: throw IllegalArgumentException("Delivery not found: $id")

// o usar Optional pattern
sealed class OptDelivery {
    data class Found(val delivery: Delivery) : OptDelivery()
    object NotFound : OptDelivery()
}
```

3. Callback hell

```
// ❌ Malo: nesting profundo
repository.getDelivery(id) { delivery ->
    repository.getBeneficiary(delivery.beneficiaryId) { beneficiary ->
        repository.getShift(delivery.shiftId) { shift ->
            // ... lógica aquí
        }
    }
}

// ✅ Bueno: corrutinas
val delivery = repository.getDelivery(id)
val beneficiary = repository.getBeneficiary(delivery.beneficiaryId)
val shift = repository.getShift(delivery.shiftId)
// lógica clara
```

4. Tight coupling

```
// ❌ Malo: acoplamiento fuerte
class StateManager {
    private val db = Database.getInstance()  // hardcoded
    private val api = ApiClient.getInstance() // hardcoded
}

// ✅ Bueno: inyección
class StateManager(
    private val repository: Repository,
    private val api: ApiClient
) { }
```

5. Magic numbers/strings

```
// ❌ Malo
if (weight > 0.5f) { }           // qué es 0.5?
val timeout = 30000              // milliseconds?

// ✅ Bueno
const val MIN_VALID_WEIGHT = 0.5f
```

```
const val API_TIMEOUT_MS = 30000

if (weight > MIN_VALID_WEIGHT) { }
delay(API_TIMEOUT_MS)
```

Manejo de errores

Try-catch o Result?

```
// ✅ Para operaciones IO o fallibles
val result = try {
    api.syncDeliveries()
    Result.Success()
} catch (e: Exception) {
    Result.Failure(e.message)
}

// ✅ Para funciones suspending
suspend fun syncDeliveries(): Result<List<Delivery>> = try {
    Result.Success(repository.deliveries.getAll())
} catch (e: Exception) {
    Result.Failure(e)
}

// ✅ En corrutinas
try {
    val deliveries = repository.deliveries.getAll()
} catch (e: IOException) {
    Logger.e("SYNC", "Network error: ${e.message}")
} catch (e: Exception) {
    Logger.e("SYNC", "Unknown error: ${e.message}")
}
```

Logging de errores

```
// ✅ Bueno: contexto claro
try {
    camera.takeSelfie()
} catch (e: Exception) {
    Logger.e("CAPTURE", "Failed to capture selfie: ${e.message}", e)
    // incluir stack trace en logs
}

// ❌ Malo
try {
    camera.takeSelfie()
```

```
} catch (e: Exception) {
    println("Error!")  // no logger, no contexto
}
```

Concurrencia y Corrutinas

Estructura

```
// ✅ Bueno: scope claro
viewModelScope.launch {  // en ViewModel
    try {
        val result = async { repository.getDelivery(id) }
            .await()
        _state.value = State.Success(result)
    } catch (e: Exception) {
        _state.value = State.Error(e.message)
    }
}

// ❌ Malo: scope global
GlobalScope.launch {  // evitar siempre
    repository.getDelivery(id)
}

// ❌ Malo: no cancela
Thread {
    repository.getDelivery(id)  // no se cancela con UI cleanup
}.start()
```

Suspensión

```
// ✅ Bueno: suspend function
suspend fun getDelivery(id: Long): Delivery = withContext(Dispatchers.IO) {
    repository.getDelivery(id)
}

// ✅ Bueno: usar correct dispatcher
withContext(Dispatchers.Main) {  // UI updates
    _state.value = newState
}

withContext(Dispatchers.IO) {  // DB, network
    repository.save(entity)
}

withContext(Dispatchers.Default) {  // CPU-intensive
    embedding = generateEmbedding(image)
```

```
}
```

Performance

Evitar allocations innecesarias

```
// ❌ Malo: crea lista nueva cada llamada
fun getActiveDeliveries(): List<Delivery> {
    return repository.deliveries
        .filter { it.status == Status.Active }  // O(n)
        .map { it.copy(processed = true) }       // O(n) copies
}

// ✅ Bueno: operación única
fun getActiveDeliveries(): Sequence<Delivery> {
    return repository.deliveries
        .asSequence()
        .filter { it.status == Status.Active }
        .map { it.copy(processed = true) }
}
```

Memory leaks en listeners

```
// ❌ Malo: memory leak
scale.addEventListener(object : ScaleEventListener {
    override fun onWeightChanged(weight: Float) {
        // captura 'this' implícitamente
        deliveryService.handleWeight(weight)
    }
})

// ✅ Bueno: remover listener
scale.addEventListener(listener)
// ... después
scale.removeEventListener(listener)

// ✅ Bueno: usar WeakReference
class DeliveryScaleListener : ScaleEventListener {
    private weak var deliveryService: DeliveryService? = null
}
```

Testing

Estructura AAA

```
@Test
fun testCalcSimilarity() {
    // ARRANGE: setup
    val embedding1 = FloatArray(512) { 1f }
    val embedding2 = FloatArray(512) { 1f }

    // ACT: ejecutar
    val similarity = calculateSimilarity(embedding1, embedding2)

    // ASSERT: verificar
    assertEquals(1.0f, similarity, 0.01f)
}
```

Nombres descriptivos

```
// ❌ Malo
fun test1() { }
fun testStateManager() { }

// ✅ Bueno
fun testStateTransitionFromWaitingForWeightToCaptureFaceOnSuccess() { }
fun testBeneficiarySearchReturnsTopResultsWithHighestSimilarity() { }
```

Mock vs Fake

```
// ✅ Mock: verificar llamadas
val repository = mock<Repository>()
`when`(repository.getDelivery(1)).thenReturn(delivery)
verify(repository).getDelivery(1)

// ✅ Fake: implementación simplificada
class FakeRepository : Repository {
    override fun getDelivery(id: Long) = fakeDeliveries[id]
}
```

Documentación

Código auto-documenta

```
// ❌ Malo: requires comment
val w = 0.5f  // minimum weight in kg

// ✅ Bueno: código habla por sí solo
```

```
const val MINIMUM_VALID_WEIGHT_KG = 0.5f
val isValidWeight = weight >= MINIMUM_VALID_WEIGHT_KG
```

KDoc para APIs públicas

```
/**
 * Calculates the similarity between two facial embeddings.
 *
 * @param embedding1 First embedding (512 dimensions)
 * @param embedding2 Second embedding (512 dimensions)
 * @return Similarity score between 0.0 and 1.0
 * @throws IllegalArgumentException if embeddings have wrong size
 *
 * @sample
 * val sim = calculateSimilarity(emb1, emb2)
 * println(sim)  // 0.95
 */
fun calculateSimilarity(embedding1: FloatArray, embedding2: FloatArray):
Float {
    require(embedding1.size == 512) { "embedding1 must have 512 dimensions"
}
    require(embedding2.size == 512) { "embedding2 must have 512 dimensions"
}
    // ...
}
```

Version Control

Commits

```
 Bueno:
commit a3f1e8c
    Add vectorial search to beneficiary lookup
    - Implement HNSW index in VectorDB
    - Add 512-dim embedding support
    - Result: 10ms queries (was 100ms)
    Fixes #123

 Malo:
commit a3f1e8c
    Fix stuff
```

Branches

```
 Esquema
```

```
feature/delivery-state-machine
feature/p2p-sync
bugfix/camera-permissions
hotfix/scale-calibration

 Malo
new-stuff
temporary
test123
```

Code review

Checklist

- [ ] Código compila sin warnings
- [ ] Tests pasan (unit + integration)
- [ ] Coverage mantiene o sube (≥70% para data/domain)
- [ ] No hay hardcoded secrets
- [ ] Nombramiento consistente con proyecto
- [ ] Performance considerado
- [ ] Manejo de errores adecuado
- [ ] Documentation actualizada
- [ ] Cambios no introducen nuevas dependencias innecesarias

Security review

- [ ] No hay datos sensibles en logs
- [ ] Encriptación donde aplique
- [ ] Permisos Android solicitados mínimamente
- [ ] Inyección de SQL/XSS prevenida
- [ ] Certificados pinned si es HTTPS
- [ ] No hardcoded URLs/API keys

Performance review

- [ ] Queries de BD optimizadas
- [ ] No N+1 queries
- [ ] Memoria OK (< 100MB app)
- [ ] Battery impact considerado
- [ ] Corrutinas en dispatcher correcto
- [ ] No main thread blocking

Checklist pre-commit

```
# Format código
./gradlew ktlintFormat

# Lint
./gradlew lint

# Tests
./gradlew test

# Build
./gradlew build

# Code coverage
./gradlew jacocoTestReport

git commit -m "Meaningful message"
```