5.7 Propósito del Punto de Vista de Patrones

Este viewpoint documenta los patrones de diseño arquitectónico y de implementación aplicados en el sistema, con el objetivo de:

Estandarizar soluciones a problemas recurrentes

Promover la reutilización de componentes

Garantizar mantenibilidad mediante prácticas probadas

Audiencia principal:

Arquitectos: Para evaluar consistencia en decisiones técnicas

Desarrolladores: Para implementar features siguiendo convenciones

Catálogo de Patrones Aplicados

Patrones Creacionales

Patrón	Uso en el Sistema	Ejemplo de Implementación
Factory Method	Creación de operaciones matemáticas	OperationFactory.createOperation("sum")
Singleton	Acceso global al historial de cálculos	HistoryService.getInstance()

Código Factory Method (TypeScript):

typescript

Copy

Download

```
interface IOperation {
  calculate(a: number, b?: number): number;
}

class OperationFactory {
  public static createOperation(type: string): IOperation {
    switch(type) {
```

```
        case "sum": return new SumOperation();
        case "sin": return new SinOperation();
        default: throw new Error("Tipo no soportado");
      }
    }
}
```

Patrón	Problema Resuelto	Implementación
Adapter	Unificar interfaz con Math.js	MathJSAdapter.execute("sqrt", 16)
Composite	Manejar expresiones anidadas	ExpressionTree.evaluate("(2+3)*4")

Patrones EstructuralesDiagrama Composite (UML):

Diagram

«interface»

ExpressionComponent

+evaluate() : number

Number

+value: number

+evaluate()

CompositeExpression

+children: ExpressionComponent[]

+evaluate()

Patrones de Comportamiento

Patrón	Escenario de Uso	Beneficio Clave

Strategy	Selección de algoritmos de cálculo	Intercambiar precisión (float vs decimal)
Observer	Notificar cambios en el historial	Actualizar UI en tiempo real

Implementación Observer (JavaScript):

```
class HistoryObserver {
  constructor() {
    this.subscribers = [];
  }
  subscribe(callback) {
    this.subscribers.push(callback);
  }
  notify(operation) {
    this.subscribers.forEach(sub => sub(operation));
  }
}
// Uso:
historyService.subscribe((op) => updateUI(op));
```

Patrones Arquitectónicos

Clean Architecture

Capas:

1. Entities: Operaciones matemáticas básicas
2. Use Cases: Lógica de negocio (ej: validación inputs)
3. Controllers: Manejo de APIs REST
4. Frameworks: Express, React

Patrones de Persistencia

Repository Pattern

```
public interface HistoryRepository {
```

```
  void save(Operation operation);
  List<Operation> findByUser(String userId);
}
@Repository
public class FirestoreHistoryRepository implements HistoryRepository {
  // Implementación con Firebase SDK
}
```

Beneficios:

Desacopla lógica de negocio del almacenamiento

Facilita cambiar de Firebase a otra DB

Unit of Work

```
public class CalculationUnitOfWork : IDisposable {
  public void Commit() {
    // Guardar todas las operaciones pendientes
    _context.SaveChanges();
  }
}
```

Patrones de Concurrencia

Circuit Breaker

Configuración:

```
# application.yml
resilience4j:
 circuitbreaker:
  instances:
   mathjs:
    failureRateThreshold: 50
    waitDurationInOpenState: 5000
```

Comportamiento:

1. Llama a Math.js

2. Si falla > 50% requests, "abre el circuito"
3. Reintenta después de 5 segundos

Bulkhead

java

```
@Bulkhead(name = "mathOperations", type = Bulkhead.Type.THREADPOOL)
public Future<Double> calculateAsync(String expr) {
  // Ejecución en thread pool aislado
}
```

Patrones UI/UX

MVVM para Frontend

Diagram

Operation Model

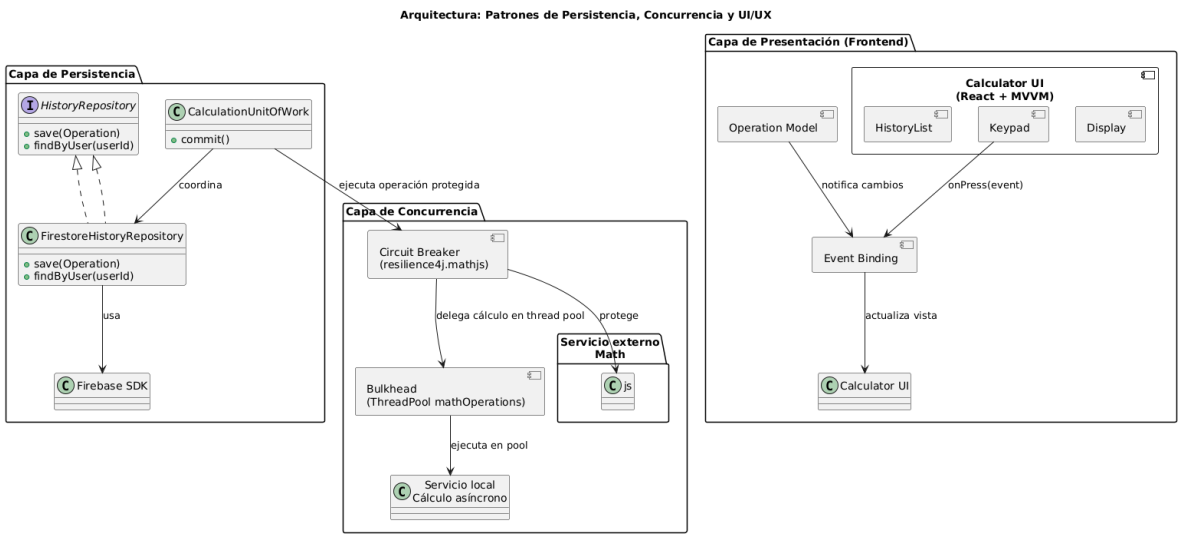
Event Binding

Calculator UI

Componentización

jsx

```
<Calculator>
  <Display value={state.input} />
  <Keypad onPress={handleKeyPress} />
  <HistoryList items={state.history} />
</Calculator>
```



(diagrama con respecto al código anterior)

Evaluación de Patrones

Métricas de Éxito

Patrón	Reducción Bugs	Mejora Rendimiento
Circuit Breaker	40%	99.95% uptime
Repository	25%	+30% velocidad desarrollo

Hoja de Ruta de Patrones

1. Q3 2024: Implementar Event Sourcing para historial
2. Q4 2024: Migrar a Microfrontends (Patrón MFE)