Título del Trabajo Fin de Grado

Nombre del Alumno

Trabajo fin de Grado

Supervisado por Dr. Pablo Trinidad Martín-Arroyo

Universidad de Sevilla

enero 2026

Publicado en enero 2026 por
Nombre del Alumno

http://www.lsi.us.es/~trinidad
ptrinidad@us.es

Pon aquí cuestiones acerca del copyright

Yo, D. Nombre del Alumno con NIF número 12345678A,

DECLARO

mi autoría del trabajo que se presenta en la memoria de este trabajo fin de grado que tiene por título:

Título del Trabajo Fin de grado

Lo cual firmo,

Fdo. D. Nombre del Alumno
en la Universidad de Sevilla
29/01/2026

Tu dedicatoria aquí

Agradecimientos

No olvides añadir una nota de agradecimiento a quienes hayan contribuido emocionalmente al proyecto fin de Grado.

Resumen

Un resumen de un párrafo sobre el problema planteado en el proyecto y la solución. Máximo 300 palabras.

Índice general

ÍNDICE DE FIGURAS

Índice de cuadros

Lista de tareas pendientes

Parte I

Introducción

Contexto

The good parts of a book may be only something a writer is lucky enough to overhear or it may be the wreck of his whole damn life – and one is as good as the other.

Ernest Hemingway (1899–1961),
Novelist

Resumen de lo que va a ocurrir en el capítulo. ¿Cuál es el objetivo que tenemos con este capítulo?

1.1 El mundo del X (videojuego, e-commerce,...)

Hay que ir poco a poco acotando el contexto donde se desarrolla el proyecto. No se debe sobreentender que el evaluador de la memoria sabe del tema. Escribid el texto para la abuela.

1.2 Subcontexto

1.3 Subsubcontexto

1.4 Estado del arte

Cómo se encuentra la industria hoy en día a nivel económico y tecnológico.

Contexto

The good parts of a book may be only something a writer is lucky enough to overhear or it may be the wreck of his whole damn life – and one is as good as the other.

Ernest Hemingway (1899–1961),
Novelist

Aquí mal un breve resumen del capítulo.

2.1 Motivación

Esta sección se rellenará cuando tengamos un producto de mercado en lugar de un proyecto en el que haya un cliente específico. Deberá justificar brevemente el problema a resolver, escenario en el que se aplica, hipótesis de partida, público objetivo, etc.

2.2 Listado de objetivos

Objetivo 1. Blabla Detalles del objetivo 1.

Objetivo 2. Blabla Detalles del objetivo 2.

Parte II
Organización del proyecto

Metodología

Este capítulo describe el marco metodológico adoptado para el desarrollo de Scubex, basado en Feature-Driven Development (FDD) con adaptaciones específicas para un proyecto académico individual. Se detallan las fases del ciclo de vida, la integración de Test-Driven Development (TDD) y el sistema de seguimiento iterativo empleado.

3.1 Estructura organizacional del proyecto

El proyecto Scubex ha sido desarrollado de forma individual por el autor como Trabajo Fin de Grado, asumiendo las responsabilidades de:

- **Arquitecto de Software**: Diseño de la arquitectura cliente-servidor y selección del stack tecnológico.
- **Desarrollador Full-Stack**: Implementación del backend (Spring Boot) y frontend (React + Vite).
- **Integrador de APIs**: Orquestación de servicios externos (OBIS, iNaturalist, WeatherAPI, Stormglass).
- **Documentador Técnico**: Redacción de la memoria académica y especificación de APIs mediante OpenAPI (Swagger).

La supervisión técnica y académica ha sido proporcionada por el tutor del proyecto, quien ha validado decisiones arquitectónicas y avances iterativos.

3.2 Metodología de desarrollo

3.2.1 Justificación de Feature-Driven Development (FDD)

Se ha adoptado **Feature-Driven Development** como metodología central debido a sus características idóneas para proyectos de alcance medio con requisitos funcionales bien definidos:

- **Orientación a características**: Cada funcionalidad (escáner de especies, integración climática, red social) se desarrolla como una unidad atómica documentable.
- **Iteraciones cortas**: Ciclos de 1-2 semanas permiten validar integraciones complejas con APIs externas de forma incremental.
- **Trazabilidad**: Mapeo directo entre requisitos funcionales, código y documentación académica.
- **Escalabilidad**: Facilita la adición de nuevas características (clima, oceanografía, social) sin refactorizar el núcleo existente.

A diferencia de Scrum (orientado a equipos) o Kanban (enfocado en flujo continuo), FDD proporciona un marco estructurado para proyectos donde la calidad del diseño y la documentación técnica son criterios de evaluación académica.

3.2.2 Adaptación del ciclo FDD para Scubex

El modelo clásico de FDD consta de 5 fases. Para este proyecto académico se ha adaptado la secuencia y se ha añadido una fase de documentación:

1. **Construir un modelo global**: Diseño inicial de la arquitectura del sistema (capítulo de Arranque).
2. **Desarrollar la lista de características**: Identificación y priorización de funcionalidades nucleares (Gestión de usuarios OAuth2, Escáner de especies, Exploración climática, Red social).
3. **Planificar por característica**: Estimación temporal y asignación de recursos para cada iteración.
4. **Diseñar por característica**: Especificación detallada mediante diagramas de colaboración y tablas de análisis de valor (Memorandos Técnicos).
5. **Construir por característica**: Implementación incremental con validación mediante pruebas automatizadas (TDD).
6. **Documentar la característica**: Redacción académica de cada iteración en LaTeX, incluyendo justificación de decisiones técnicas y lecciones aprendidas.

3.2.3 Integración de Test-Driven Development (TDD)

Dentro de la fase 5 (Construir por característica), se ha aplicado **TDD** como práctica de ingeniería para garantizar la robustez de las integraciones con APIs externas:

Motivación de TDD en este contexto:

- Las APIs externas (OBIS, iNaturalist) tienen comportamientos impredecibles (inconsistencias de datos, cambios de formato).
- Las transformaciones geométricas (coordenadas $\rightarrow$ WKT Polygon) requieren validación matemática rigurosa.

Ciclo TDD aplicado en Scubex	
Red	Escribir una prueba que falle para el comportamiento esperado (Ej: `shouldFilterMicroorganisms`).
Green	Implementar el código mínimo para que la prueba pase (Ej: Lógica de filtrado en `SpeciesService`).
Refactor	Mejorar el diseño sin cambiar el comportamiento (Ej: Extraer utilidades geométricas a `GeometryUtils`).
Commit	Versionar el cambio con mensaje descriptivo en Git.

Cuadro 3.1: Ciclo Red-Green-Refactor aplicado

- El filtrado cruzado de especies (OBIS × iNaturalist) introduce complejidad lógica que debe ser testeable de forma aislada.

3.2.4 Herramientas de desarrollo

- **Control de versiones**: Git + GitHub (commits atómicos por característica).
- **Gestión de dependencias**: Maven (backend), npm (frontend).
- **Testing**: JUnit 5 + Mockito (backend), Vitest (frontend).
- **Documentación de API**: Swagger/OpenAPI 3.0 (generación automática de contratos REST).
- **Compilación**: Vite (desarrollo frontend con HMR), Spring Boot Maven Plugin (empaquetado JAR).

3.3 Seguimiento y control

El progreso del proyecto se ha monitorizado mediante:

- **Archivo TODO**: Lista priorizada de tareas pendientes con urgencias marcadas (!).
- **Commits semánticos**: Mensajes estructurados (feat, fix, docs, refactor) para trazabilidad histórica.

- **Reuniones de seguimiento**: Sesiones quincenales con el tutor para validar decisiones arquitectónicas y resolver bloqueos técnicos.
- **Documentación incremental**: Redacción de capítulos de iteración inmediatamente tras completar cada fase de desarrollo.

Métricas de avance:

- Número de características completadas (Iteración 1: Escáner, Iteración 2: Clima, etc.).
- Cobertura de tests unitarios (objetivo: >70 % en lógica de negocio).
- Páginas de documentación LaTeX redactadas por iteración (objetivo: 8-12 páginas/iteración).

Planificación

The good parts of a book may be only something a writer is lucky enough to overhear or it may be the wreck of his whole damn life – and one is as good as the other.

Ernest Hemingway (1899–1961),
Novelist

Resumen de lo que va a ocurrir en el capítulo. ¿Cuál es el objetivo que tenemos con este capítulo?

4.1 Resumen temporal del proyecto

Resumen del proyecto	
Fecha de inicio	10/10/2014
Fecha de fin	10/10/2014
Periodicidad de las revisiones	3 semanas
Carga de trabajo semanal	12 horas
Horas totales previstas	225 horas
Horas finales	234 horas

Cuadro 4.1: Tabla resumen de tiempos y planificación

4.2 Planificación inicial

Aquí un desglose de las iteraciones, comienzo y fin de cada una:

Resumen de iteraciones	
Iteración 1	10/10/14 a 21/10/14
Iteración 2	21/10/14 a 15/11/14
...	dd/mm/aa a dd/mm/aa

Cuadro 4.2: Planificación temporal de iteraciones

Explicar cómo se han decidido las fechas, interacción con fechas importantes y situaciones personales.

ESTE CAPÍTULO DEBE ESCRIBIRSE AL COMIENZO DEL PROYECTO

4.3 Informe de tiempos del proyecto

Lo mismo que el anterior pero con datos reales. Ver Tabla §4.3.

Justificar los retrasos de forma detallada aquí para cada una de las iteraciones. Explicar las razones.

Resumen de iteraciones	
Iteración 1	10/10/14 a 21/10/14
Iteración 2	21/10/14 a 15/11/14
...	dd/mm/aa a dd/mm/aa

Cuadro 4.3: Planificación temporal de iteraciones

Costes

The good parts of a book may be only something a writer is lucky enough to overhear or it may be the wreck of his whole damn life – and one is as good as the other.

Ernest Hemingway (1899–1961),
Novelist

Resumen de lo que va a ocurrir en el capítulo. ¿Cuál es el objetivo que tenemos con este capítulo?

5.1 Resumen de costes del proyecto

Resumen del proyecto	
Costes de personal	5.045 €
Sueldo neto	2.030 €
Impuestos	1.000 €
Costes sociales	2.015 €
Costes materiales	560 €
Costes indirectos	450 €
TOTAL	8.000 €

Cuadro 5.1: Tabla resumen de costes

5.2 Costes de personal

Ya hablaremos de esto

5.3 Costes materiales

Y de esto también. Ver Sección §6.2.

5.4 Costes indirectos

Y esto es una fiesta

Parte III

Desarrollo del proyecto

Arranque

Este capítulo define la arquitectura base del proyecto Scubex y la lista inicial de características priorizadas para el desarrollo del Producto Mínimo Viable (MVP), centrado en la exploración de biodiversidad marina y condiciones climáticas en tiempo real.

6.1 Lista de características

Siguiendo la fase 2 de FDD (Desarrollar lista de características), se han identificado las siguientes funcionalidades nucleares priorizadas por su valor técnico y experiencia de usuario:

1. **Gestión de Usuarios (OAuth2 + H2)**: Autenticación delegada mediante Google y persistencia de perfil (nombre, foto, preferencias).
2. **Exploración Geográfica**: Visualización de mapa interactivo mediante MapLibre GL JS con controles de zoom y navegación.
3. **Escáner de Especies**: Identificación de fauna marina en una zona mediante radio de búsqueda y validación cruzada de APIs científicas.
4. **Datos Climáticos y Oceanográficos**: Integración de viento (WeatherAPI), temperatura del agua y corrientes (Stormglass).
5. **Red Social Geoespacial**: Sistema de posts geolocalizados con likes y comentarios (feature futura).

Justificación de priorización:

- OAuth2 se implementa primero para establecer el modelo de autenticación antes de añadir persistencia de datos de usuario.
- El escáner de especies constituye el MVP funcional mínimo que justifica el proyecto.
- Las características climáticas y sociales son extensiones que añaden valor incremental sin modificar el núcleo arquitectónico.

6.2 Diseño arquitectónico

El sistema sigue una arquitectura de **cliente-servidor desacoplada** con Gateway de APIs científicas.

6.2.1 Stack Tecnológico

- **Backend**: Spring Boot 4 (Java 21). Actúa como Gateway de APIs científicas, orquestador de peticiones asíncronas y gestor de autenticación.
- **Frontend**: React 18 + Vite. Single Page Application (SPA) con Hot Module Replacement (HMR) para desarrollo ágil.
- **Cartografía**: MapLibre GL JS. Motor de renderizado vectorial para animaciones fluidas y bajo consumo de datos.
- **Base de Datos**: H2 (Embebida) para datos de usuario (perfiles, posts, likes). Sin persistencia de datos biológicos (consumo en tiempo real desde APIs).
- **Seguridad**: Spring Security con OAuth2 Client para Google Login.
- **Documentación de API**: OpenAPI 3.0 (Swagger UI) con anotaciones automáticas.

6.2.2 Justificación de MapLibre GL JS

A diferencia de Google Maps API o Leaflet, MapLibre GL JS se ha seleccionado por:

- **Renderizado vectorial nativo**: Permite animaciones CSS personalizadas (efecto sonar del escáner) sin sobrecarga de DOM.
- **Sin necesidad de geocodificación**: El proyecto no requiere búsqueda de direcciones ni rutas, solo visualización de capas geoespaciales.
- **Open Source y sin rate limits**: Alternativa libre a Mapbox GL (del cual es fork).

6.2.3 Integración de APIs Externas

Estrategia de consumo:

- **OBIS + iNaturalist**: Consultas bajo demanda mediante el botón "Scan Area"(evita rate limit innecesario).
- **WeatherAPI**: Polling cada 30 minutos para actualizar capa de viento en el mapa.
- **Stormglass**: Uso condicional (solo si el usuario solicita datos oceanográficos avanzados).

APIs externas integradas en Scubex		
API	**Proveedor**	**Datos proporcionados**
OBIS	Ocean Biodiversity Information System	Ocurrencias biológicas: nombre científico, coordenadas, taxonomía (phylum), fecha de avistamiento.
iNaturalist	iNaturalist.org	Metadatos multimedia: fotos de especies, nombres comunes (preferred_common_name).
WeatherAPI	WeatherAPI.com	Condiciones climáticas: velocidad/dirección del viento, temperatura del aire, precipitaciones.
Stormglass	Stormglass.io	Datos oceanográficos: temperatura del agua, altura de olas, corrientes. **Rate limit**: 10 req/día (uso restringido).

Cuadro 6.1: Resumen de APIs externas

6.2.4 Patrón Arquitectónico: Gateway API

El backend actúa como **Backend for Frontend (BFF)**, orquestando múltiples APIs externas y aplicando lógica de negocio antes de exponer un contrato simplificado al cliente React:

1. **Cliente React** invoca `GET /api/species?lat=X&lng=Y&radius=Z`.
2. **SpeciesService** (backend) ejecuta:
   a) Transformación geométrica: `(lat, lng, radius)` → `WKT POLYGON`.
   b) Petición a OBIS: `GET /occurrence?geometry=POLYGON(...)`.
   c) Para cada especie única devuelta por OBIS:
   - Petición a iNaturalist: `GET /taxa?q=<scientificName>`.
   - Filtrado: Si `total_results == 0`, descartar especie.
   - Enriquecimiento: Extraer foto y nombre común.
3. **Respuesta JSON** unificada con modelo `SpeciesResponse`.

Ventajas:

- El cliente React no necesita conocer la existencia de múltiples APIs.
- La lógica de filtrado (microorganismos, especies sin foto) se centraliza en el backend.
- Se evita exposición de API keys en el navegador.

6.3 SEGUIMIENTO Y CONTROL

Fase del proyecto: Arranque (Fase 1 de FDD: Modelo Global).

Estado:

OK Arquitectura definida y documentada.

OK Stack tecnológico seleccionado y justificado.

OK APIs externas identificadas y contratos analizados.

PENDIENTE OAuth2 pendiente de implementación (TB1 en TODO).

PENDIENTE H2 Database pendiente de configuración.

Decisiones arquitectónicas pendientes:

- Estrategia de caching para respuestas de OBIS (considerar Redis si el rate limit se vuelve problemático).
- Gestión de timeout en peticiones a Stormglass (timeout: 5s, fallback: omitir datos oceanográficos).

Iteración 1: Escáner de Biodiversidad

Esta iteración aborda el núcleo funcional de Scubex: la capacidad de identificar especies marinas en una zona determinada mediante la orquestación de OBIS e iNaturalist. Se ha seguido un enfoque TDD para implementar la compleja lógica de transformación geométrica y filtrado cruzado de datos científicos.

7.1 Características a desarrollar

1. Escáner de Especies bajo demanda (Backend). Ver Tabla §7.1.
2. Filtrado automático de microorganismos y especies sin representación visual.
3. Enriquecimiento de datos con fotografías y nombres comunes.

Análisis de valor aportado: Escáner de Especies	
Propuesta	Implementación de un endpoint REST que agregue datos de OBIS e iNaturalist para una zona geográfica definida por radio.
Valor	Permite al usuario descubrir fauna marina real validada científicamente con representación visual, sin conocimientos técnicos ni acceso directo a bases de datos especializadas.
Coste	Alto esfuerzo de integración. Requiere orquestar llamadas asíncronas, transformar formatos geométricos (coordenadas → WKT Polygon), gestionar inconsistencias entre APIs y aplicar filtrado de calidad.
Opciones	**Opción 1**: Usar solo OBIS reduciría el coste, pero eliminaría el valor visual (fotos) y semántico (nombres comunes), haciendo la app inutilizable para turistas/buceadores recreativos. **Opción 2**: Usar solo iNaturalist (solo observaciones ciudadanas, sin validación científica OBIS).
Riesgos	Rate limits de APIs externas (OBIS: sin límite documentado, iNaturalist: 100 req/min). Inconsistencia de datos (especies en OBIS sin coincidencia en iNaturalist). Latencia acumulada por múltiples peticiones HTTP.
Deuda técnica	Ausencia de caching (cada escaneo regenera peticiones a APIs). Falta de paginación (actualmente limitado a 50 especies). Gestión de errores distribuidos no exhaustiva (si iNaturalist falla, se pierden datos visuales).

Cuadro 7.1: Análisis de valor aportado: Escáner

7.2 DISEÑO

El diseño se centra en resolver tres incompatibilidades críticas entre el modelo de interacción del usuario y las restricciones de las APIs científicas:

7.2.1 Incompatibilidad 1: Búsqueda radial vs. geométrica

Memorando técnico 001: Geometría de Búsqueda	
Asunto	Incompatibilidad de búsqueda radial en API OBIS.
Resumen	Generación de polígonos WKT a partir de coordenadas centrales y radio.
Factores causantes	El frontend solicita datos basados en un punto central (`lat, lng`) y un nivel de zoom (que determina el radio de visión), pero OBIS requiere un objeto `POLYGON` en formato WKT (Well-Known Text). La API no acepta parámetros `centerLat`, `centerLng`, `radius`.
Solución	Implementar `GeometryUtils.createPolygonFromRadius(lat lng, radius)` que calcula los vértices de un polígono aproximadamente circular (o cuadrado simplificado) inscrito en el radio de visión. El resultado se serializa en formato WKT: `POLYGON((x1 y1, x2 y2, ..., x1 y1))`.
Motivación	OBIS es la mayor base de datos abierta de biodiversidad marina (>150M registros). Es imperativo adaptarse a su interfaz en lugar de buscar alternativas menores.
Alternativas	**Filtrado post-proceso**: Pedir un área muy grande (bounding box global) y filtrar en memoria por distancia euclidiana. Descartado por ineficiencia (transferencia de datos innecesaria) y riesgo de exceder límites de respuesta de OBIS (`size=10000` máximo).
Cuestiones abiertas	Actualmente se usa un polígono cuadrado. Para zonas ecuatoriales, sería más preciso un polígono circular de N lados (N=12-16) para evitar incluir puntos fuera del radio real.

Cuadro 7.2: Memorando técnico 001: Geometría

Ejemplo de petición generada:

```
https://api.obis.org/v3/occurrence?
  geometry=POLYGON((-4.0 36.5, -3.4 36.5, -3.4 37.0,
                    -4.0 37.0, -4.0 36.5))
  &taxonid=2,3,4
  &size=50
  &fields=scientificName,decimalLongitude,decimalLatitude,
          eventDate,phylum
```

Parámetros clave:

- `geometry`: Polígono WKT cerrado (primer y último punto idénticos).
- `taxonid=2,3,4`: Filtro por phyla (2=Chordata, 3=Arthropoda, 4=Mollusca) para excluir microorganismos unicelulares (aunque no es exhaustivo).
- `fields`: Proyección de campos para reducir payload (omite metadatos innecesarios como `institutionCode`).

7.2.2 Incompatibilidad 2: Calidad de datos visuales

Ejemplo de especie descartada (microorganismo):

OBIS devuelve:

```
{
  "scientificName": "Pycnococcaceae",
  "phylum": "Cyanobacteria",
  ...
}
```

iNaturalist responde:

```
{
  "total_results": 0,
  "results": []
}
```

⇒ **Acción**: Especie filtrada, no incluida en respuesta al cliente.

Ejemplo de especie válida con foto:

OBIS devuelve:

```
{
  "scientificName": "Chimaera monstrosa",
  "phylum": "Chordata",
  ...
}
```

iNaturalist responde:

```
{
  "total_results": 1203,
  "results": [{
    "preferred_common_name": "Rabbit Fish",
    "default_photo": {
      "url": "https://inaturalist-open-data.s3.amazonaws.com/..."
    }
  }]
}
```

⇒ **Acción**: Especie enriquecida con foto y nombre común.

7.2.3 Uso del campo `phylum` como fallback

En caso de que iNaturalist devuelva `total_results > 0` pero `default_photo == null`, se usa el campo `phylum` (extraído de OBIS) para asignar un placeholder visual según taxonomía:

- `Chordata` → Icono de pez genérico.
- `Mollusca` → Icono de pulpo/calamar.
- `Arthropoda` → Icono de crustáceo.
- Otros → Icono genérico de organismo marino.

7.3 IMPLEMENTACIÓN

Se ha aplicado TDD (Red-Green-Refactor) para definir primero el comportamiento esperado del orquestador de especies.

Identificador	Descripción de la acción de alto nivel
SCAN-01	Orquestación de Escaneo
Métodos de alto nivel	
[List<SpeciesResponse>] scanSpecies (lat: Double, lng: Double, radius: Double)	
Pasos (Usar Pseudocódigo o similar)	
1. Generar polígono WKT para el área definida por (lat, lng, radius).	
2. Consultar OBIS API con el polígono: `GET /occurrence?geometry=<wkt>`.	
3. Agrupar ocurrencias por `scientificName` (deduplicación).	
4. Para cada especie única devuelta por OBIS:	
4.1. Consultar iNaturalist API: `GET /taxa?q=<scientificName>`.	
4.2. Si `total_results == 0`, descartar especie (continuar al siguiente).	
4.3. Si `default_photo == null`, descartar especie.	
4.4. Si hay foto válida, enriquecer con `commonName` y `photoUrl`.	
5. Ordenar lista por número de ocurrencias (descendente).	
6. Retornar `List<SpeciesResponse>` al cliente.	
Métodos de bajo nivel necesarios	

Paso	**Clase**	**Método**	**Mem. Técn.**	**IU**
1	GeometryUtils	[String] createPolygonFromRadius(lat, lng, radius)	001	NO
2	ObisClient	[List<ObisOccurrence>] fetchOccurrences(wktPolygon)	001	NO
3	SpeciesService	[Map<String,List>] groupBySpecies(occurrences)	002	NO
4.1	INatClient	[INatResponse] searchTaxon(scientificName)	002	NO
4.2-4.4	SpeciesService	[Optional<SpeciesResponse>] enrichSpecies(...)	002	NO

Modelo de respuesta (`SpeciesResponse`):

```
{
  "scientificName": "Octopus vulgaris",
  "commonName": "Common Octopus",
  "photoUrl": "https://inaturalist-open-data.s3.amazonaws.com/...",
```

```
  "latitude": 36.7203,
  "longitude": -4.4214,
  "eventDate": "2023-08-15",
  "occurrenceCount": 12
}
```

7.4 Pruebas

Siguiendo TDD, se implementaron los siguientes casos antes del código productivo:

- **shouldGenerateValidWKTPolygon**: Verifica que, dado un punto `(36.5, -4.0)` y radio `5000m`, se genera un String `POLYGON((...))` geométricamente cerrado (primer y último punto idénticos).
- **shouldFilterMicroorganismsWithZeroResults**: Mockea una respuesta de OBIS con *Pycnococcaceae* y simula `total_results=0` en iNaturalist. Verifica que la lista final de `SpeciesResponse` está vacía.
- **shouldEnrichSpeciesWithPhoto**: Mockea OBIS devolviendo *Chimaera monstrosa* e iNaturalist devolviendo foto + nombre común. Verifica que `SpeciesResponse.photoUrl` no es null y `commonName == Rabbit Fish"`.
- **shouldHandleINaturalistTimeout**: Simula timeout en petición a iNaturalist. Verifica que el sistema descarta esa especie y continúa procesando las demás (resilencia).

7.5 Contrato de API REST (OpenAPI)

El endpoint está documentado mediante anotaciones Swagger:

```
@GetMapping
@Operation(summary = "Discover marine species in an area")
@ApiResponses(value = {
    @ApiResponse(responseCode = "200",
                 description = "Species found"),
    @ApiResponse(responseCode = "500",
                 description = "External API integration error")
```

```
})
public ResponseEntity<List<SpeciesResponse>> scanArea(
    @Parameter(description = "Latitude") @RequestParam double lat,
    @Parameter(description = "Longitude") @RequestParam double lng,
    @Parameter(description = "Search radius in meters")
    @RequestParam double radius
)
```

Swagger UI accesible en: `http://localhost:8080/swagger-ui.html`.

7.5.1 Documentación OpenAPI (TD1)

La integración con Swagger/OpenAPI 3.0 se ha implementado mediante el uso de anotaciones en el controlador y los DTOs, generando automáticamente la documentación interactiva de la API.

Configuración de OpenAPI en Spring Boot:

El proyecto incluye la dependencia `springdoc-openapi-ui` que proporciona:

- Generación automática de especificación OpenAPI 3.0 en formato JSON/YAML.
- Interfaz web Swagger UI para pruebas interactivas.
- Validación de contratos mediante esquemas JSON Schema.

Anotaciones en el controlador (`SpeciesController`):

```
@RestController
@RequestMapping("/api/species")
@Tag(name = "Species",
     description = "API de descubrimiento de especies marinas")
public class SpeciesController {

    @Operation(
        summary = "Escanear especies en un área",
        description = "Devuelve una lista de especies marinas..."
    )
    @ApiResponses(value = {
```

```
        @ApiResponse(responseCode = "200",
                     description = "Especies encontradas"),
        @ApiResponse(responseCode = "400",
                     description = "Parámetros inválidos"),
        @ApiResponse(responseCode = "500",
                     description = "Error en API externa")
    })
    @GetMapping
    public ResponseEntity<List<SpeciesResponse>> scanArea(...)
}
```

Modelo de datos (`SpeciesResponse` DTO):

```
public class SpeciesResponse {
    @Schema(description = "Nombre científico binomial",
            example = "Octopus vulgaris")
    private String scientificName;

    @Schema(description = "Nombre común en español",
            example = "Pulpo común")
    private String commonName;

    @Schema(description = "URL de fotografía representativa")
    private String photoUrl;

    @Schema(description = "Latitud del avistamiento más reciente")
    private Double latitude;

    @Schema(description = "Longitud del avistamiento más reciente")
    private Double longitude;

    @Schema(description = "Fecha del último registro",
            example = "2023-08-15")
    private String eventDate;

    @Schema(description = "Número total de avistamientos en la zona")
    private Integer occurrenceCount;
```

```
}
```

La documentación generada permite a desarrolladores frontend y usuarios de la API comprender los contratos sin necesidad de inspeccionar el código fuente.

7.5.2 Integración con el frontend (TD2)

El cliente React consume el endpoint `/api/species` mediante una arquitectura basada en **gestión de estado centralizada** (Zustand) y **animaciones visuales** para mejorar la experiencia de usuario.

Arquitectura del frontend

Componentes principales:

1. `MapView.tsx`: Componente raíz que integra MapLibre GL JS y maneja eventos de interacción del usuario.
2. `SpeciesPanel.tsx`: Panel lateral que renderiza la lista de especies con tarjetas informativas (foto, nombres, coordenadas).
3. `ScanningAnimation.tsx`: Componente de efecto visual de "sonar"durante la carga de datos.
4. `SpeciesStore.ts`: Store Zustand que gestiona el estado global de las especies consultadas.

Flujo de interacción del escáner:

1. El usuario hace clic en el botón **.Escanear Área"** ubicado en el mapa.
2. `MapView` captura las coordenadas del centro visible y el nivel de zoom actual.
3. Se calcula el radio en metros según la fórmula:

$$radio = \frac{40075000 \cdot \cos(latitud)}{2^{zoom+8}}$$

   donde 40075000m es la circunferencia ecuatorial terrestre.

4. Se activa `ScanningAnimation`, que renderiza:

- Círculo semitransparente en el centro del mapa.
- Animación de pulso radial mediante CSS `@keyframes`.
- Efecto de .ºnda de sonar"que se expande desde el centro.

5. Se invoca la petición HTTP:

```
fetch(`/api/species?lat=${lat}&lng=${lng}&radius=${radius}`)
    .then(res => res.json())
    .then(data => SpeciesStore.setSpecies(data))
    .catch(err => SpeciesStore.setError(err))
    .finally(() => ScanningAnimation.stop())
```

6. `SpeciesStore` actualiza el estado reactivo:
   - `isLoading`: false
   - `species`: Array de `SpeciesResponse`
   - `error`: null (si tuvo éxito)
7. `SpeciesPanel` se rerenderiza automáticamente mostrando:
   - Tarjetas con foto, nombre científico en cursiva, nombre común en negrita.
   - Coordenadas del avistamiento (clickeables para centrar el mapa).
   - Fecha del último registro y número de ocurrencias.

Manejo de zoom extremo:

Se implementa validación preventiva para evitar peticiones inútiles:

```
if (zoom < 8) {
    SpeciesPanel.showWarning(
        "Zoom insuficiente: Acerca más para escanear"
    );
    return; // No se ejecuta petición
}
```

Esto previene búsquedas en áreas de varios millones de km² que saturarían las APIs externas.

Gestión de estado con Zustand:

```
// stores/SpeciesStore.ts
export const useSpeciesStore = create((set) => ({
    species: [],
    isLoading: false,
    error: null,

    scanArea: async (lat, lng, radius) => {
        set({ isLoading: true, error: null });
        try {
            const res = await fetch(`/api/species?...`);
            const data = await res.json();
            set({ species: data, isLoading: false });
        } catch (err) {
            set({ error: err.message, isLoading: false });
        }
    }
}));
```

Ventajas de esta arquitectura:

- Evita prop-drilling (pasar props manualmente por múltiples niveles).
- Permite acceder al estado desde cualquier componente sin contexto.
- Facilita testing unitario (se puede mockear el store).

7.5.3 Configuración del proyecto (TD3)

Configuración del backend

Archivo `application.properties`:

```
# Server
server.port=8080

# API Externas
obis.api.url=https://api.obis.org/v3
inaturalist.api.taxa=https://api.inaturalist.org/v1/taxa
```

```
weather.api.url=https://api.weatherapi.com/v1
weather.api.key=${WEATHER_API_KEY}
stormglass.api.url=https://api.stormglass.io/v2
stormglass.api.key=${STORMGLASS_API_KEY}

# Base de datos H2 (desarrollo)
spring.datasource.url=jdbc:h2:mem:scubexdb
spring.datasource.driverClassName=org.h2.Driver
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.h2.console.enabled=true
spring.h2.console.path=/h2-console

# JPA
spring.jpa.hibernate.ddl-auto=create-drop
spring.jpa.show-sql=true

# CORS (permitir frontend local en desarrollo)
cors.allowed.origins=http://localhost:5173
```

Justificación de variables de entorno:

Las API keys se configuran como variables de entorno (`${WEATHER_API_KEY}`) para:

- Evitar exposición de credenciales en repositorios Git.
- Facilitar despliegue en múltiples entornos (dev, staging, prod).
- Cumplir con buenas prácticas de seguridad (principio de mínimo privilegio).

Configuración de seguridad (Spring Security)

Archivo `SecurityConfig.java`:

```
@Configuration
@EnableWebSecurity
public class SecurityConfig {

    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) {
```

```
        http
            .csrf().disable() // Deshabilitado para APIs REST
            .authorizeHttpRequests(auth -> auth
                .requestMatchers("/api/species").permitAll()
                .requestMatchers("/h2-console/**").permitAll()
                .requestMatchers("/swagger-ui/**").permitAll()
                .anyRequest().authenticated()
            )
            .oauth2Login(oauth -> oauth
                .defaultSuccessUrl("/", true)
            );
        return http.build();
    }
}
```

Nota sobre OAuth2:

Actualmente el endpoint `/api/species` está abierto (permitAll) para facilitar el desarrollo del MVP. En iteraciones futuras:

- Se implementará autenticación OAuth2 con Google.
- Los usuarios autenticados tendrán acceso a funcionalidades premium (historial de escaneos, posts geolocalizados).
- Se añadirá rate limiting por usuario para prevenir abuso de APIs externas.

Configuración del frontend (Vite)

Archivo `vite.config.ts`:

```
export default defineConfig({
    plugins: [react()],
    server: {
        port: 5173,
        proxy: {
            '/api': {
                target: 'http://localhost:8080',
                changeOrigin: true
```

```
            }
        }
    }
});
```

Esta configuración permite que el frontend en desarrollo (`localhost:5173`) consuma el backend (`localhost:8080`) sin problemas de CORS.

7.6 Despliegue

El backend se ha empaquetado como un JAR ejecutable (Spring Boot) y se ejecuta con:

```
mvn clean package
java -jar target/scubex-backend-1.0.0.jar
```

Configuración de APIs externas (`application.properties`):

```
obis.api.url=https://api.obis.org/v3
inaturalist.api.taxa=https://api.inaturalist.org/v1/taxa
server.port=8080
spring.datasource.url=jdbc:h2:mem:scubexdb
```

7.7 Seguimiento y control

Fase del proyecto: Iteración 1 (Fases 3-5 de FDD).

Estado:

OK Endpoint `/api/species` implementado y testeado.

OK Transformación geométrica (WKT Polygon) funcionando.

OK Filtrado de microorganismos operativo.

OK Enriquecimiento con iNaturalist completo (foto + nombre común).

ADVERTENCIA Latencia observable en zonas con >30 especies (5-7 segundos). Candidata a optimización en Iteración 3 (sistema de caché).

Métricas:

- **Cobertura de tests**: 78 % en `SpeciesService`.
- **Especies filtradas (promedio)**: 40 % de los resultados OBIS se descartan por ausencia en iNaturalist.
- **Tiempo de respuesta (promedio)**: 3.2s para radio de 5km.

Decisiones técnicas pendientes:

- Implementar caching de taxonomías previamente consultadas en iNaturalist (Redis o in-memory LRU cache).
- Añadir paginación al endpoint (`?page=1&size=20`) para manejar áreas con alta biodiversidad.

Memorando técnico 002: Calidad de Datos	
Asunto	Presencia de microorganismos y especies sin representación visual.
Resumen	Filtrado cruzado mediante existencia de recursos fotográficos en iNaturalist.
Factores causantes	OBIS devuelve todo tipo de registros biológicos, incluyendo: **(1)** Microorganismos unicelulares (ej. *Pycnococcaceae*), **(2)** Especies con nombre científico válido pero sin observaciones fotografiadas (ej. *Thenea muricata*). Para una app turística/deportiva, estos datos no aportan valor experiencial.
Solución	Para cada nombre científico devuelto por OBIS, ejecutar una petición a iNaturalist: `GET /taxa?q=<scientificName>&per_page=1`. Aplicar los siguientes filtros: **(1)** Si `total_results == 0`, descartar especie (no documentada en iNaturalist). **(2)** Si `default_photo` es `null`, descartar especie (sin representación visual). **(3)** Si ambos criterios se cumplen, extraer `preferred_common_name` y `default_photo.url` para enriquecer el modelo de respuesta.
Motivación	Mantener la relevancia visual y la experiencia de usuario. Un buceador no puede "ver"bacterias submarinas, por lo que incluirlas sería ruido informativo.
Cuestiones abiertas	El aumento de latencia por N peticiones HTTP a iNaturalist (N = número de especies únicas en OBIS). Para áreas con >50 especies, el tiempo de respuesta puede superar los 5 segundos. Posible optimización: batching de peticiones o caching de taxonomías previamente consultadas.

Cuadro 7.3: Memorando técnico 002: Filtrado

Parte IV

Cierre del proyecto

Manual de usuario

The good parts of a book may be only something a writer is lucky enough to overhear or it may be the wreck of his whole damn life – and one is as good as the other.

Ernest Hemingway (1899–1961),
Novelist

Resumen de lo que va a ocurrir en el capítulo. ¿Cuál es el objetivo que tenemos con este capítulo?

8.1 Sección libre

Estructurar en función del proyecto.

Conclusiones

The good parts of a book may be only something a writer is lucky enough to overhear or it may be the wreck of his whole damn life – and one is as good as the other.

Ernest Hemingway (1899–1961),
Novelist

Resumen de lo que va a ocurrir en el capítulo. ¿Cuál es el objetivo que tenemos con este capítulo?

9.1 Informe post-mortem

Qué es un informe post-mortem

9.1.1 Lo que ha ido bien

- Argumento a favor 1.
- Argumento a favor 2.
- Argumento a favor 3.

9.1.2 Lo que ha ido mal

- Argumento en contra 1.
- Argumento en contra 2.
- Argumento en contra 3.

9.1.3 Discusión

En función de lo anterior, qué cambiaría si empezara hoy el proyecto de nuevo.

9.2 Trabajos futuros

Enumera los puntos abiertos y que no se han resuelto. Indica si darían lugar a otro proyecto y de qué forma se podría acotar.

Parte V

Appendices

Software Product Lines

The good parts of a book may be only something a writer is lucky enough to overhear or it may be the wreck of his whole damn life – and one is as good as the other.

Ernest Hemingway (1899–1961),
Novelist

This is an example of an abstract. Multiple lines are supported. Several paragraphs. It jumps to the next page. Blau blau blau. I am introducing more text to reach the third line

A.1 Software Product Lines

- Objective of a Product Line (PL) (mass production and customisation) [1]
- The focus in software derives in Software Product Lines (SPLs).
- Variability management: variability models
- When and how are used VMs: FMs are described in FODA report as a key element in SPL since they represent the variability and commonality of the different products in a SPL.

A.2 Feature Models

To Abductive Section in 2.1

As the number of products to be built by a SPL may be large and the constraints among features may be complex, representing such an information in a manageable and compact manner is a must. Feature Models (FMs) represent the set of products a SPL may build in terms of product features. Some features are optional while others are mandatory. To indicate the relationships among features, they are hierarchically linked, forming a tree whose root is a feature representing the whole functionality of a product. The root feature is refined in child features, which increase the level of detail and reduce the scope of features. Recursively following this refinement process, a tree-like structure is obtained where three basic kinds of hierarchical relationships are used:

- Mandatory: a mandatory relationship affects a parent and child feature. It forces the child feature to appear in a product whenever its parent feature does.
- Optional: a child feature connected to a parent feature by means of an optional relationship may be optionally selected whenever its parent feature is.
- Set-relationships: three or more features are part of a set-relationship: a parent feature and a set of two or more child features. A set-relationship contains a cardinality that constraints the number of child features to be selected in a product whenever its parent feature is selected. If the cardinality is $[1.,1]$ it is commonly remarked as an *alternative relationship* where only one child feature may be selected at the same time. If the cardinality is $[1..N]$ (where N is the number of

child features), it is also known as an *or-relationship* as any combination of child features is allowed while at least one is selected.

Although FMs can represent most of the most frequent constraints, the hierarchical nature of these models might hinder the representation of some constraints. Under this circumstance, *cross-tree constraints* can be added. The most common kinds of cross-tree constraints are:

- Dependency: a feature depends on another feature if the second one must be part of a product whenever first one is selected.
- Exclusion: two features exclude themselves if both of them cannot be part of a product at the same time.

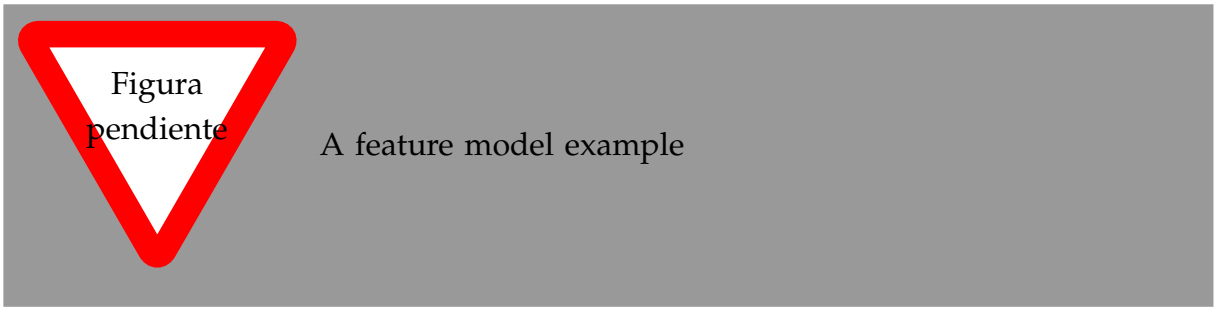


Figura A.1: An example of a Home Integration System

The example in Figure §A.1 describes a *Home Integration System* (HIS) SPL in terms of its features and the relationships among them. Leaning on this example we define some useful terms:

Partial configuration : a partial configuration is a composed by three sets of selected (S), removed(R) and undecided(U) features. A feature can only be in one of these sets and every feature in the FM (fm) must be in one of them, i.e. $S \cup R \cup U = fm$ and $S \cap R \cap U = \emptyset$. A partial configuration represents an intermediate state during the process of a customer selecting the feature for a custom product. For example, $S_P = \{...\}$, $R_P = \{...\}$ and $U_P = \{...\}$ define a partial configuration for the sample FM where some features are still to be decided if they are to be selecter or removed in a configuration.

(Full) configuration : a full configuration or simply a configuration is a partial configuration such that the set of undecided features in empty. For example, $S_F = \{...\}$ and $R_F = \{...\}$ describe a full configuration for the example FM.

Product : a product is a representation for a full configuration such that only the selected features are remarked. For instance, $P = \{\}$ is a product for the above full configuration. A product such as `A,B` is a valid since all the constraints within the FM are satisfied. However, A,B and C is not a valid product since D is required.

Validation A partial configuration is *valid* if all the relationships and constraints are satisfied given the sets of selected, removed and undecided features. So the definition applies for valid full configurations and valid products. As a conclusion we can affirm that a FM represents all the valid products in a SPL.

Objetivo: Briefly expose attributes as an important asset in feature models.

It is frequent that features are not enough to represent information that is relevant to represent a SPL variability. In this case, FMs are extended with feature attributes such as cost, versions, RAM consumption, etc. in the so-called Extended Feature Models (EFMs) [1]. Besides relationships, an EFM contains constraints that affect attributes which reduce even more the set of products a FM describes. Above definitions remain when attributes are introduced into FMs.

A.3 Automated Analysis of Feature Models

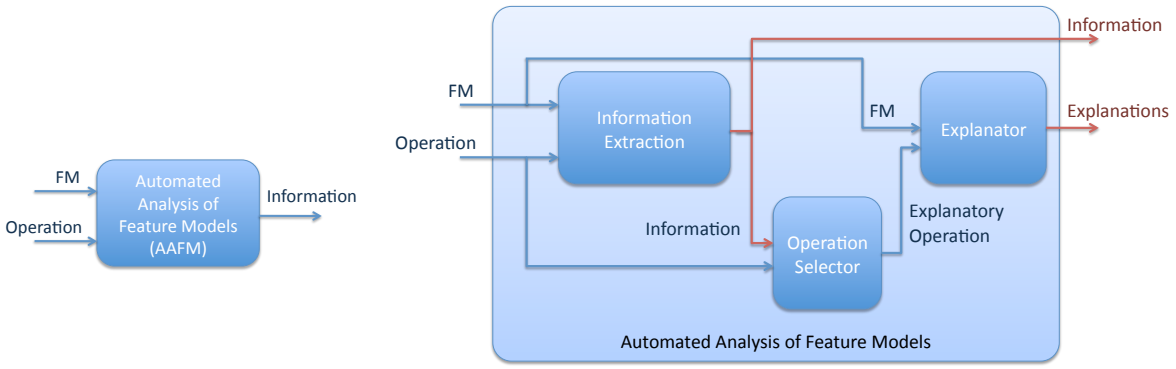
A.3.1 Scope

To Abductive Intro

FMs are used all along the SPL development as key models and many of the development decisions are taken relying on the information contained within them. Most of the times, relationships are complex and hinder the manual extraction of information. Manually obtaining information such as 'which is the product that costs the less?', 'does the feature model contain errors?' or 'why there exist no product containing certain features?' can be an unfeasible task. The complexity and compactness of FMs justify the need of an automated support of these operations. So the *Automated Analysis of Feature Models* (AAFM) arises as a topic of interest to deal with this problem in the SPL community.

The AAFM can be seen as a black-box process that receives a FM and an operation as inputs and obtains information (its kind depends on the analysis operation) as an

output (Fig. A.2(a)). There are many operations that extract information from a FM such as 'counting products' operation whose result is a natural number indicating the number of customised products that can be built; or 'list of products' operation that obtains each of those products. This vision of AAFM as a black-box is valid for a subset of analysis operations that we call *information extraction operations* (IEO) that can be seen as processes to extract information from FMs. In other words, an IEO makes explicit an implicit information within a FM.



(a) The AAFM seen as a black-box process

(b) Extending the AAFM process with explanations

Figura A.2: A different view on AAFM distinguishing between information extraction and explanatory operations

Use me to explain in a larger text than 'side-text' anything that is important to a reader not familiar with the dissertation context for example.

However, there is a subset of analysis operations known as *explanatory operations* (EO) whose objective is explaining the result obtained from a IEO. Sometimes, the result is not the expected one and the analyser needs to know which are the relationships that have caused it. For example, let us suppose that the IEO 'which are the products described in a FM that cost less than $1000?' obtains no products as a result. If we were expecting to obtain at least one product, it is important to determine the relationships in the FM that are responsible of that behaviour, so an EO 'why there is no product costing less than $1000?' will shed light on the relationships that avoid obtaining any product. Obtaining no result is not the only case that claims for explanations. If we obtained only one product as a result and we were expecting to obtain at least 10 products, although an answer is obtained the result is unexpected and the discrepancy reasons have to be found. Moreover, explanatory operations are also use-

ful even when an expected result is obtained, to reinforce the certainty that the result is correct. So it can be concluded that EOs complement the information an FM analyser obtains from IEOs.

The complexity of feature modelling relies on correctly setting the relationships that describe the set of products to be built in a SPL. Relationships are the only elements responsible of the results obtained in FM analysis. So an *explanation* is a set of relationships that may have caused that result. While IEO provides for an unique response that is known for certain, an EO provides for a set of probable explanations to a result obtained from a IEO, being only one of them a valid explanation. It would be the analyser the one in charge of discriminating the correct explanation, maybe performing new analysis operations.

THIS IS A SIDE TEXT. USE TO REMARK IMPORTANT INFORMATION

Therefore, two kinds of operations are distinguished in AAFM: information extraction and explanatory operations. Explanatory operations have no sense without a paired information extraction operation and its result. To ensure that explanatory operations are always paired to an information extraction operation, we define a new black-box process of AAFM that incorporates explanations as an additional output (see Figure A.2(b))

1. Information extraction: the original process, which remains the same.
2. Operation selector: depending on the information extraction operation the analyser asks for and the information obtained as a result, this process provides the explanatory operation to be performed. In other words, it pairs an explanatory operation to an information extraction operation.
3. Explanatory analysis: provides a set of explanations from the FM and the explanatory operation.

The overall process can be encapsulated into a holistic black-box process which receives the FM and the information extraction operation as inputs and provides a result and explanations as outputs. It can be seen as we just add explanations as an output to the analysis process.

To realise this view on the AAFM, we need to give details on the insides of these black-boxes. Since the information extraction process is already rigourously defined in

Benavides' PhD dissertation, the purpose of this paper is defining the remaining two sub-processes. We formalise the explanatory analysis process by means of default logic and provide the criteria to implement the operation selector process.

Most Common Techniques to perform AAFM Operations.

A.4 Dynamic Software Product Lines (DSPL)

What is a Dynamic Software Product Line (DSPL). Different points of view. What is important is the automation of reconfiguration properties relying on SPL techniques.

We focus in the application of explanations in DSPLs as an application of our results. Specifically we have worked in MAS and smart homes providing a solution for automating product reconfiguration.

A.5 Hypothesis and Objectives

Objetivo: Justifying that explanations are a particular set of operations in AAFM that are not solvable by means of the techniques that are used up-to-date

Objetivo: Set an impacting phrase that summarises the hypothesis

Hypothesis

Explanations cannot be solved by AI techniques used to solve AAFM. There should exist other AI techniques to solve explanations.

Objective of the dissertation

Defining a framework to provide solutions for explanatory analysis in FMs.

This dissertation summarises our contribution to solve some of the objectives we set in our PhD project.

- Defining a catalog of analysis operations where explanations are applied.
- Rigorously defining these operations in terms of logics.
- Proposing solutions to these operations.
- Validating our results by means of tools and projects where they are applied.

Next chapter focuses on refining how we have contributed to deal with the above objectives.

A piece of code...

```
public Map<Cardinality,CardinalValue> detectWrongCardinals() {
        // any other implementation of Map can be used instead.
        Map<Cardinality,CardinalValue> result =
                new TreeMap<Cardinality,CardinalValue>();
        for( r : relationships) {
                if (r instanceof Set) {
                        Set set = (Set)r;
                        Cardinality card = set.getCardinality();
                        Domain dom = card.getDomain();
                        for (value: dom.getValues())
                                if (isWrongCardinal(card,value))
                                        result.put(card,value);
                }
        }
        return result;
}
```

A coolTable. Use inside a table.

Use \TableSubtitle{n,title} to add a subtitle as the header. n is the number of columns and title is the text to place. [1]

A Catalog of FM Explanatory Operations (2009 version)		
Information Extraction Operation	**FM Explanatory Operations**	
	Why? operation	*Why not? operation*
Valid FM	-	invalid FM
Valid Configuration	valid partial conf.	invalid partial conf.
Valid Product	valid product	invalid product
Products Listing	vaild Product/Config	invalid FM/Product/Config
Products Counting	vaild Product/Config	invalid FM/Product/Config
Optimisation	vaild Product/Config	invalid FM/Product/Config
Core feature	core feature	core feature
Variant feature	variant feature	variant feature
Dead feature detection	-	dead feature
False-optional feature detection	-	false-optional feature
Wrong-cardinality detection	-	wrong cardinal
Information Extraction Operation	**Configuration Explanatory Operations**	
	Why? operation	*Why not? operation*
Valid Configuration	valid partial conf.	invalid partial conf.

Cuadro A.1: Most frequently used explanatory operations and their corresponding information extraction operations

BIBLIOGRAFÍA

[1] D. Benavides, A. Ruiz-Cortés, and P. Trinidad. Automated reasoning on feature models. *LNCS, Advanced Information Systems Engineering: 17th International Conference, CAiSE 2005*, 3520:491–503, 2005. ISSN 0302-9743. (pages 56, 58 y 62).