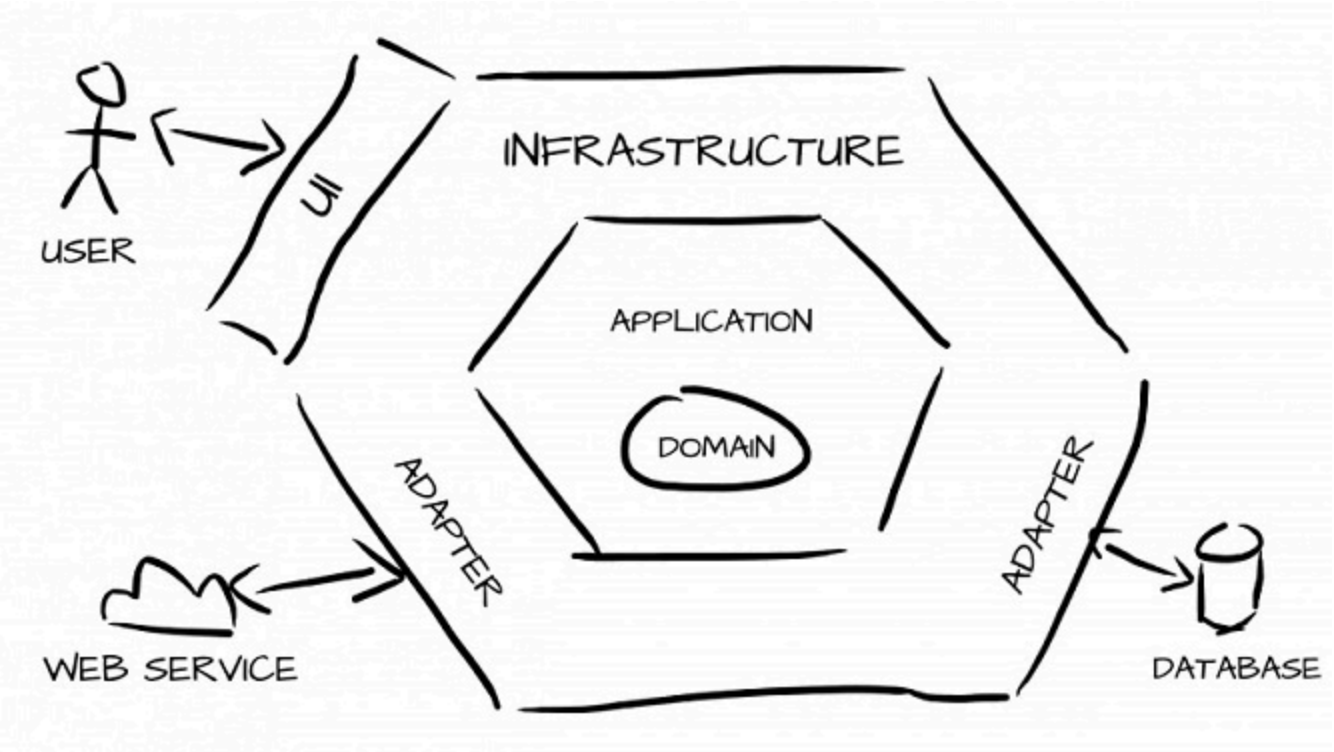
Arquitectura limpia

Introducción a los patrones Repositorio y Servicio en Java

Aplicaciones Distribuidas

Curso 2025/26



Arquitectura limpia

Es una propuesta de **Robert C. Martin** que combina varios estilos arquitectónicos (ej. Arquitectura Hexagonal).

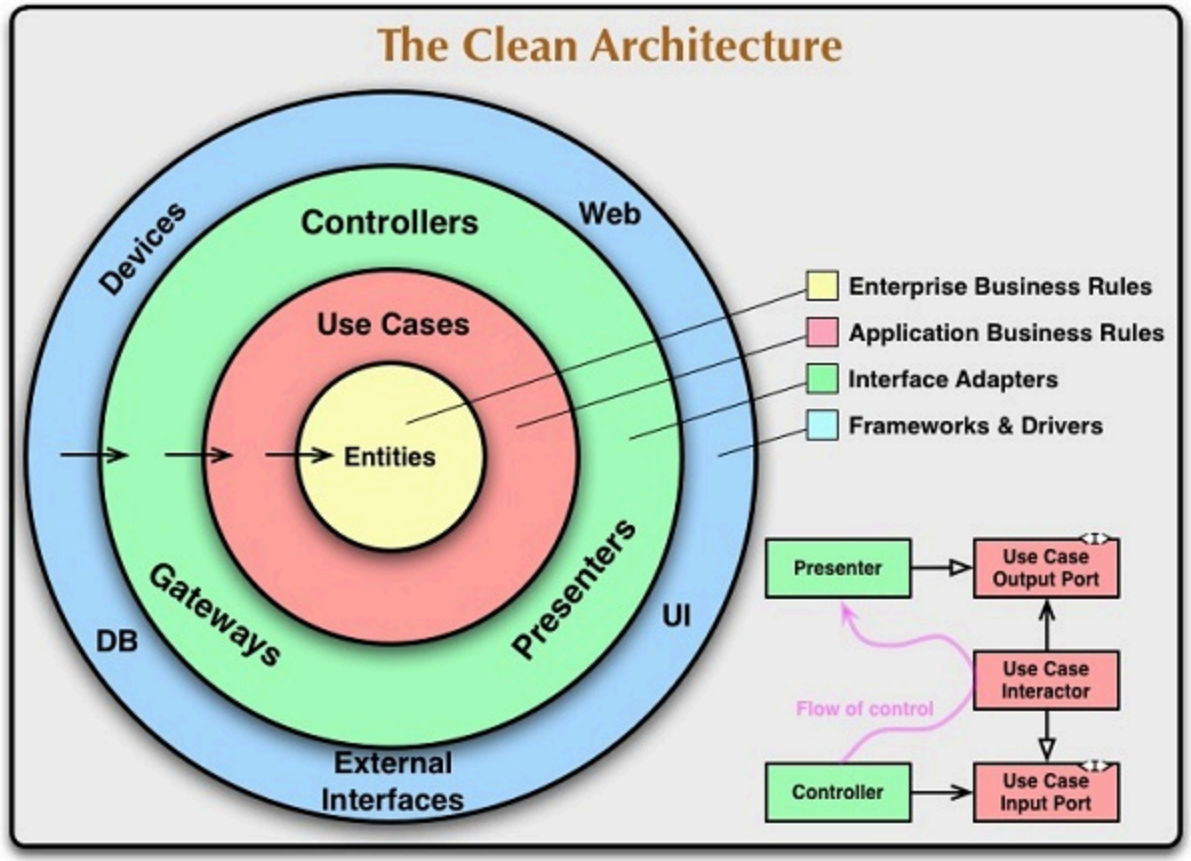
Se caracteriza por:

- No depende de ningún framework (ej. Spring).
- El dominio de la aplicación es independiente de base de datos, interfaz de usuario y servicios externos.
- Facilita las pruebas.

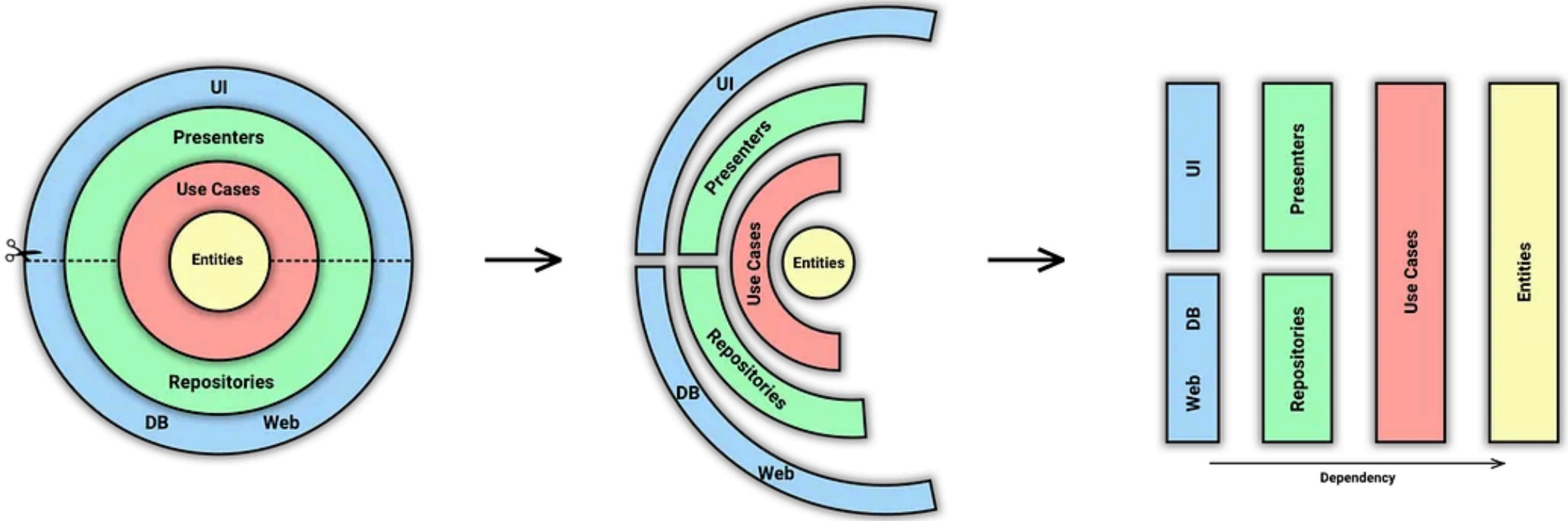
Fuente

Arquitectura limpia

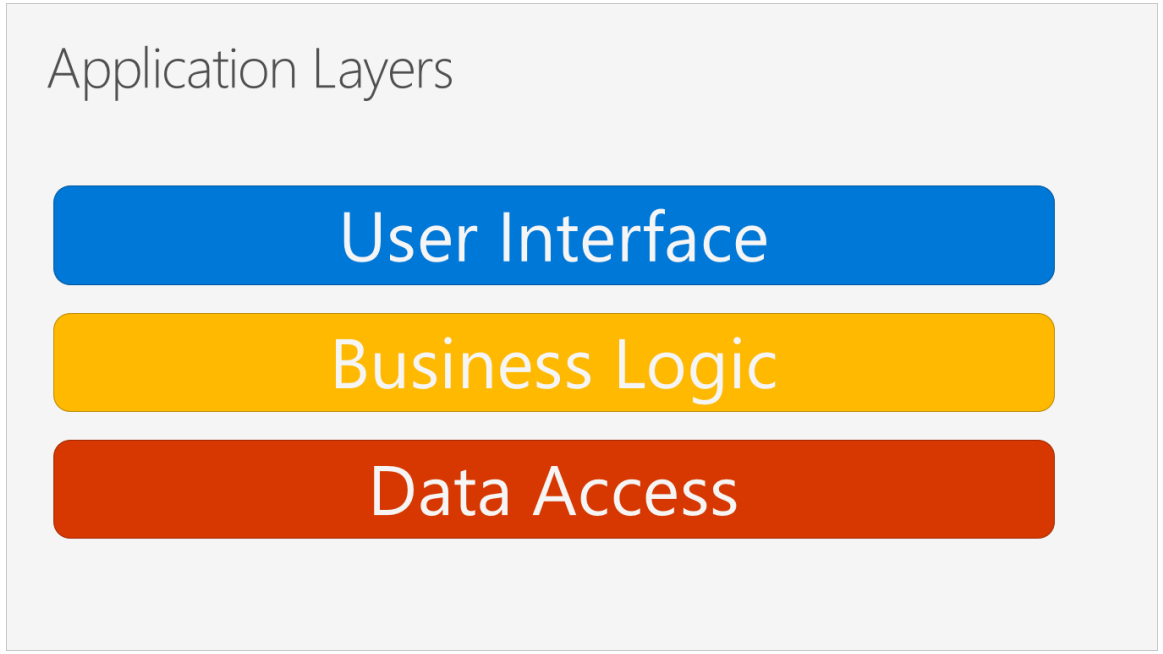
Las dependencias van de fuera hacia dentro.

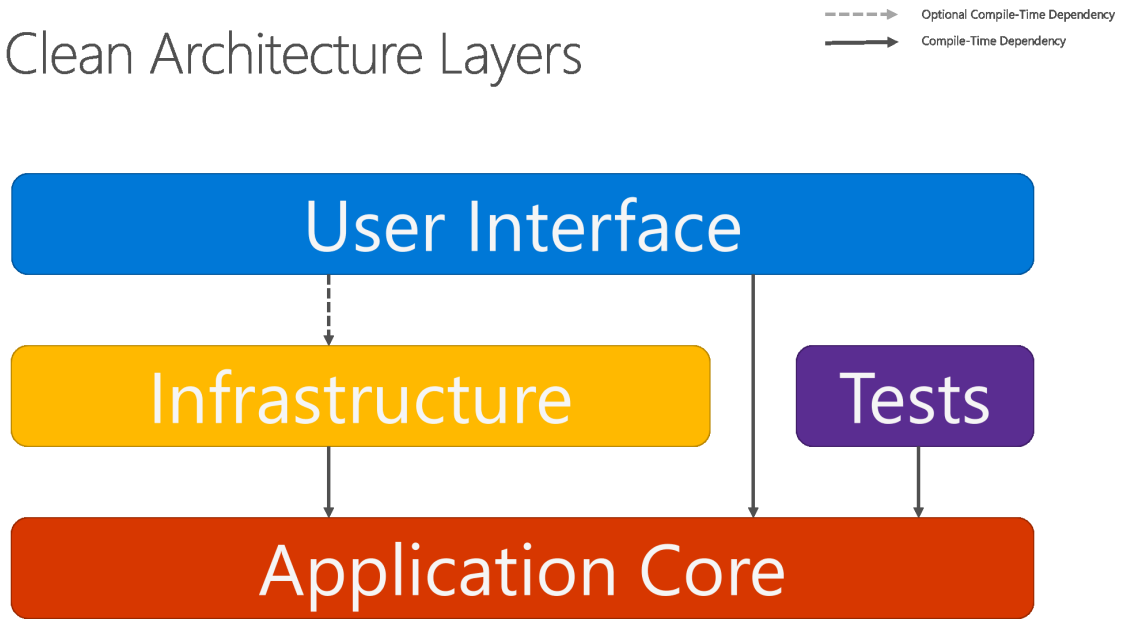


Arquitectura limpia



Arquitectura 3-capas vs limpia

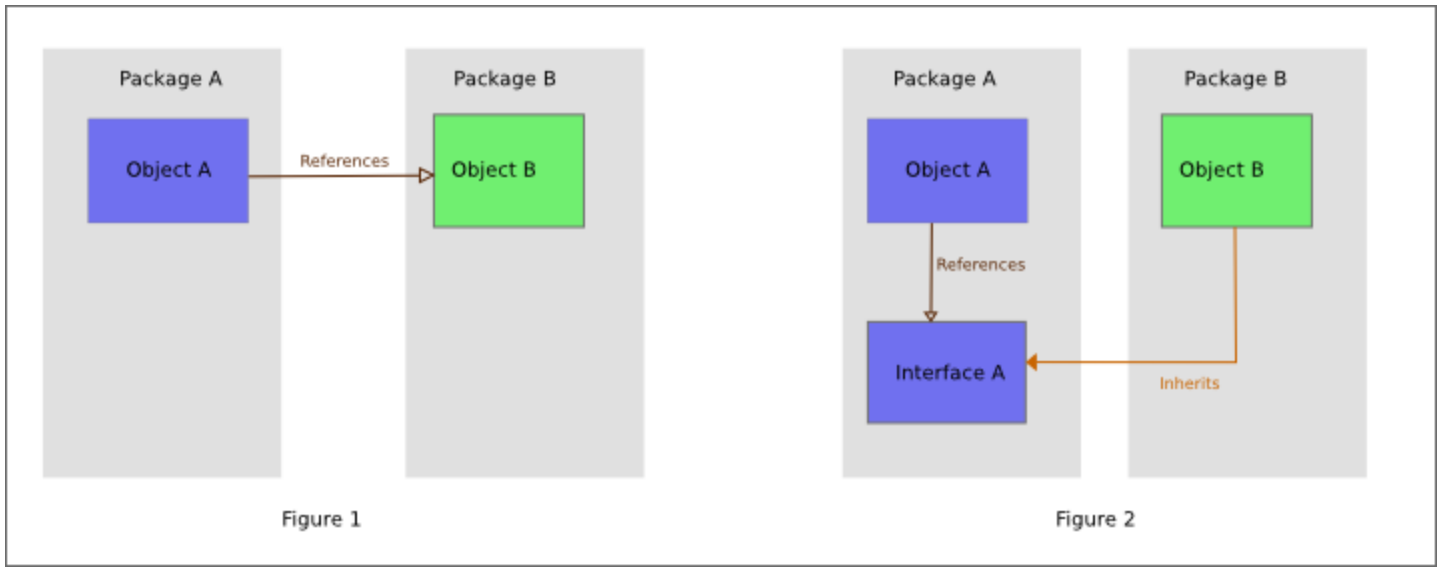




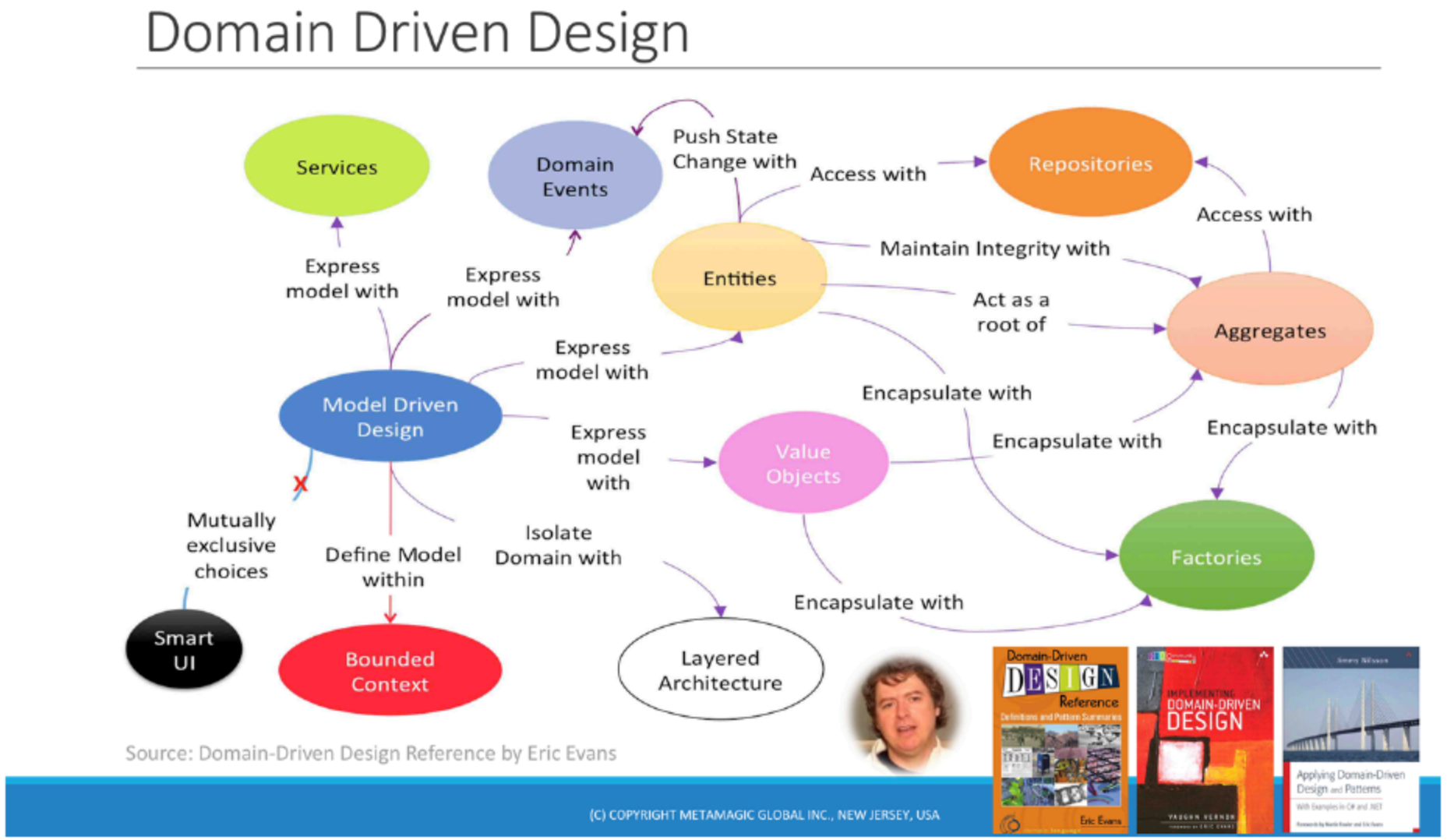
Inversión de dependencias

Para desacoplar los módulos se aplica el principio de Inversión de dependencias:

- Se introducen interfaces para establecer dependencias con abstracciones (ej. sistema de persistencia).
- Las implementaciones de las interfaces concretan los detalles (ej. acceso a datos).



Domain Driven Design



Patrón Repositorio

- El patrón Repositorio tiene como propósito ofrecer una abstracción del sistema de persistencia (inversión de dependencias).
- Un repositorio representa una **colección de objetos** persistente.
- Favorece la implementación de pruebas unitarias, permitiendo utilizar técnicas de *mock* o implementaciones en memoria.
- Las clases que implementan el repositorio encapsulan la lógica de acceso a las fuentes de datos.
- Introducido originalmente por Martin Fowler en el libro Patterns of Enterprise Application Architecture.

Patrón Repositorio

Definición de Martin Fowler:

A repository performs the tasks of an **intermediary between the domain model layers and data mapping**, acting in a similar way to a set of domain objects in memory.
[...] Conceptually, a repository encapsulates a **set of objects** stored in the database and operations that can be performed on them
[...]
Repositories, also, support the purpose of separating, clearly and in one direction, the dependency between the work domain and the data allocation or mapping.

Interfaz Repositorio

Declara las operaciones que permiten gestionar una colección o almacén de objetos.

```
public interface Repositorio <T, K> {

    K add(T entity) throws RepositorioException;

    void update(T entity) throws RepositorioException, EntidadNoEncontrada;

    void delete(T entity) throws RepositorioException, EntidadNoEncontrada;

    T getById(K id) throws RepositorioException, EntidadNoEncontrada;

    List<T> getAll() throws RepositorioException;

    List<K> getIds()throws RepositorioException;
}
```

Interfaz Repositorio

Características de la definición:

- Es un tipo genérico. El parámetro `T` representa el tipo de datos que se va almacenar en el repositorio (*entidad*).
- El repositorio proporciona un identificador a las entidades que se añaden al almacén. El parámetro `K` representa el tipo del identificador.
- Por ejemplo, un repositorio para gestionar *encuestas* con identificadores alfanuméricos será declarado como `Repositorio<Encuesta, String>`.

Interfaz Repositorio

Características de la definición (continúa):

- Todas las operaciones pueden notificar la excepción `RepositorioException`: fallo en el sistema de persistencia.
- Las operaciones que requieren localizar una entidad (`update`, `delete`, `getById`) pueden notificar `EntidadNoEncontrada`.

El repositorio genérico se puede concretar, por ejemplo, para gestionar identificadores alfanuméricos:

```
public interface RepositorioString<T>
            extends Repositorio<T, String> {

}
```

Repositorio en memoria

Para hacer **pruebas** a una aplicación resulta útil el uso de un repositorio que almacene la información en memoria.

```
public class RepositorioMemoria<T extends Identificable>
         implements RepositorioString<T> {

        private HashMap<String, T> entidades = new HashMap<>();

        // asigna secuencialmente los identificadores
        private int id = 1;

        @Override
        public String add(T entity) {

                String id = String.valueOf(this.id++);
                entity.setId(id);
                this.entidades.put(id, entity);
                return id;
        }
        //...
```

Repositorio en memoria

- La implementación del repositorio exige a las entidades que implementen la interfaz `Identificable`, es decir, que ofrezcan métodos *get/set* para gestionar el identificador.
- Debido a la ausencia de un sistema de persistencia, este tipo de repositorio no lanza la excepción `RepositorioException`.
- Resulta útil extender un repositorio en memoria para incluir **datos de prueba** de una aplicación:

```
public class RepositorioEncuestasMemoria
   extends RepositorioMemoria<Encuesta> {

        public RepositorioEncuestasMemoria() {
                // Datos iniciales para pruebas
                Encuesta encuesta1 = // ...
                this.add(encuesta1);
        }
}
```

Factoría de repositorios

Las implementaciones de un repositorio se obtienen a través de una factoría:

```
Repositorio<Encuesta, String> repositorio =
     FactoriaRepositorios.getRepositorio(Encuesta.class);
```

- La factoría instancia una clase que implementa el repositorio para la entidad.
- La correspondencia entre la entidad y la implementación del repositorio se establece en un fichero de configuración:

```
// fichero: repositorios.properties

encuestas.modelo.Encuesta=encuestas.repositorio.RepositorioEncuestasMemoria
```

Servicios

Definimos la funcionalidad de los casos de uso en interfaces:

```
public interface IServicioEncuestas {

String crear(String titulo, String instrucciones, LocalDateTime apertura,
    LocalDateTime cierre, List<String> opciones) throws RepositorioException;

boolean haVotado(String id, String usuario)
    throws RepositorioException, EntidadNoEncontrada;

void votar(String id, int opcion, String usuario)
    throws RepositorioException, EntidadNoEncontrada;

// ...
```

Implementamos el servicio utilizando el repositorio obtenido a través de la factoría de repositorios.

Factoría de servicios

- Se utiliza una factoría para obtener la implementación de los servicios de forma análoga a la factoría de los repositorios.
- La diferencia más significativa es que gestiona una instancia única de cada servicio (***singleton***).

```
IServicioEncuestas servicio =
  FactoriaServicios.getServicio(IServicioEncuestas.class);

// Alta de la encuesta

String id = servicio.crear(titulo, ...);
```

Inversión de dependencias

La inversión de dependencias se consigue definiendo interfaces para los repositorios y servicios, y obteniendo su implementación utilizando *reflexión* a través de factorías.

El enlace interfaz-implementación se establece en **ficheros de configuración**:

- `repositorios.properties` : asocia una entidad del dominio con la implementación de su repositorio.
- `servicios.properties` : asocia la interfaz de un servicio con su implementación.

En Maven, los ficheros de configuración residen en la carpeta `src/main/resources` .

Referencias

- Los fragmentos de código de la presentación han sido extraídos del proyecto `encuestas` que está disponible en el repositorio GitHub de la asignatura.
- La propuesta de repositorio será extendida más adelante para incluir un mecanismo de consulta (**patrón Especificación**).