Arquitectura Hexagonal

Codely Tv Pro

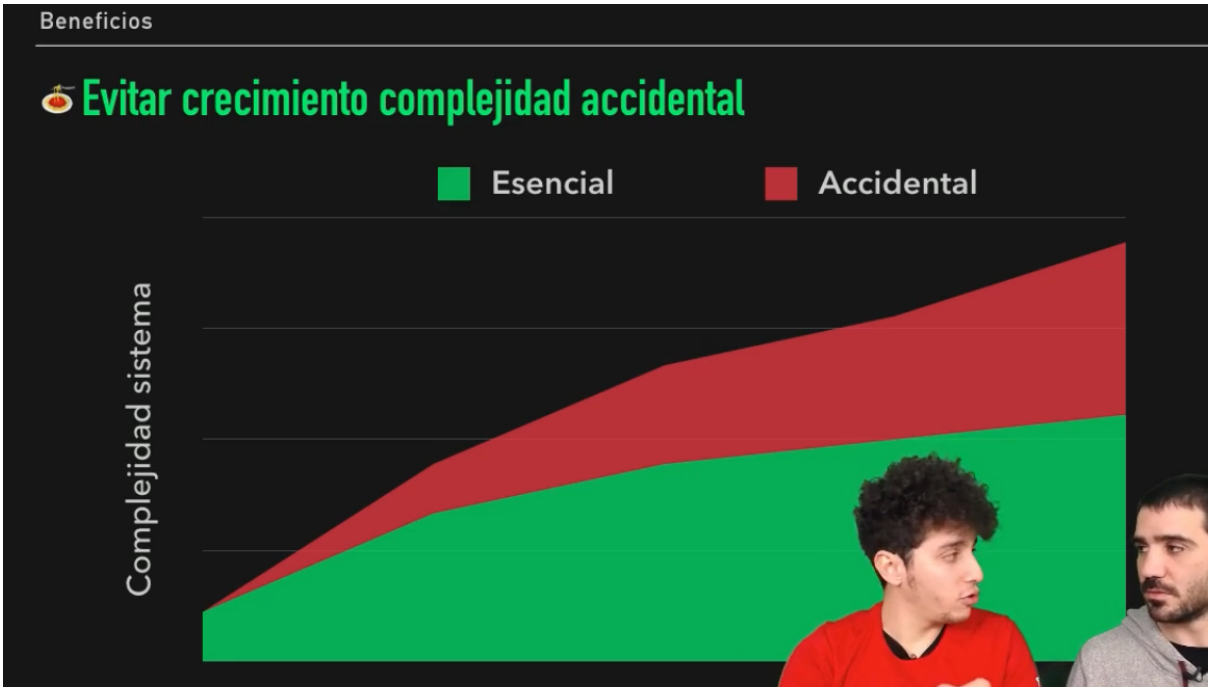
1. Qué es la Arquitectura de Software 💎

1.1. Qué es la Arquitectura de Software 💎

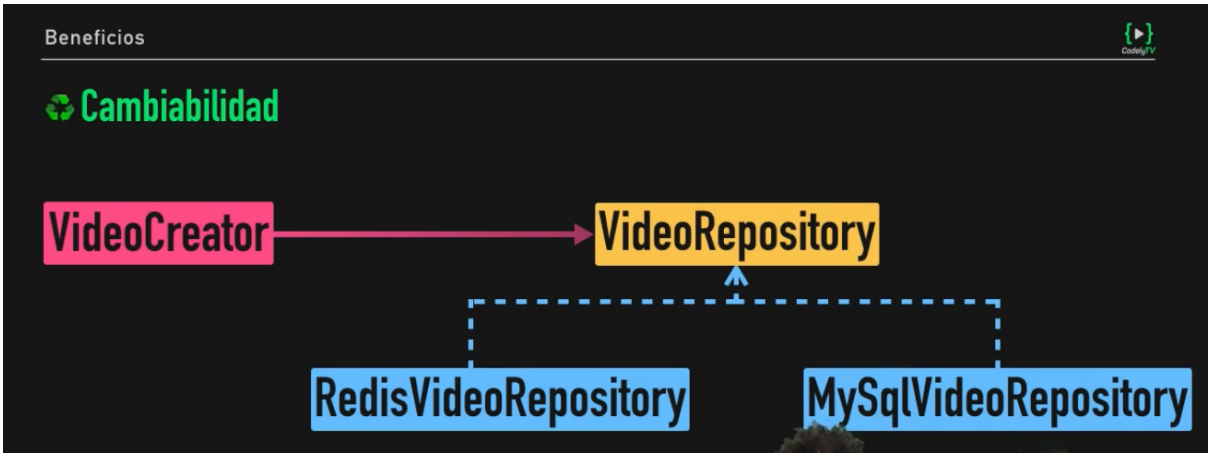
- Reglas autoimpuestas al definir cómo diseñar software.
  - DDD dentro de la parte táctica se apoya en la Arquitectura hexagonal.
  - Macro design: Arquitectura Hexagonal
  - Micro design: Cómo modelar el dominio con el Patrón Repository, Validador, etc.
  - Romper el Rol del arquitecto de software

- Beneficios de la AH
  - Mantenibilidad: Código mucho más mantenible.
  - Cambiabilidad
  - Testing: Facilita el testing
  - Simplicidad

  - Evitar crecimiento complejidad accidental
    - Complejidad del sistema: Es toda la complejidad de la app
      - Esencial: La complejidad core del propio sistema
      - Accidental:
        - Decisiones que tomas al escribir código
        - Tener un código muy acoplado que no permite implementar rápido y fácil una nueva feature.



- Cambiabilidad
  - Ejemplo de DB, ahora puedes migrar a redis sin problema porque el código no está acoplado ya que usa una Interfaz VideoRepository. La implementación de los Repository no están acoplados a VideoCreator.



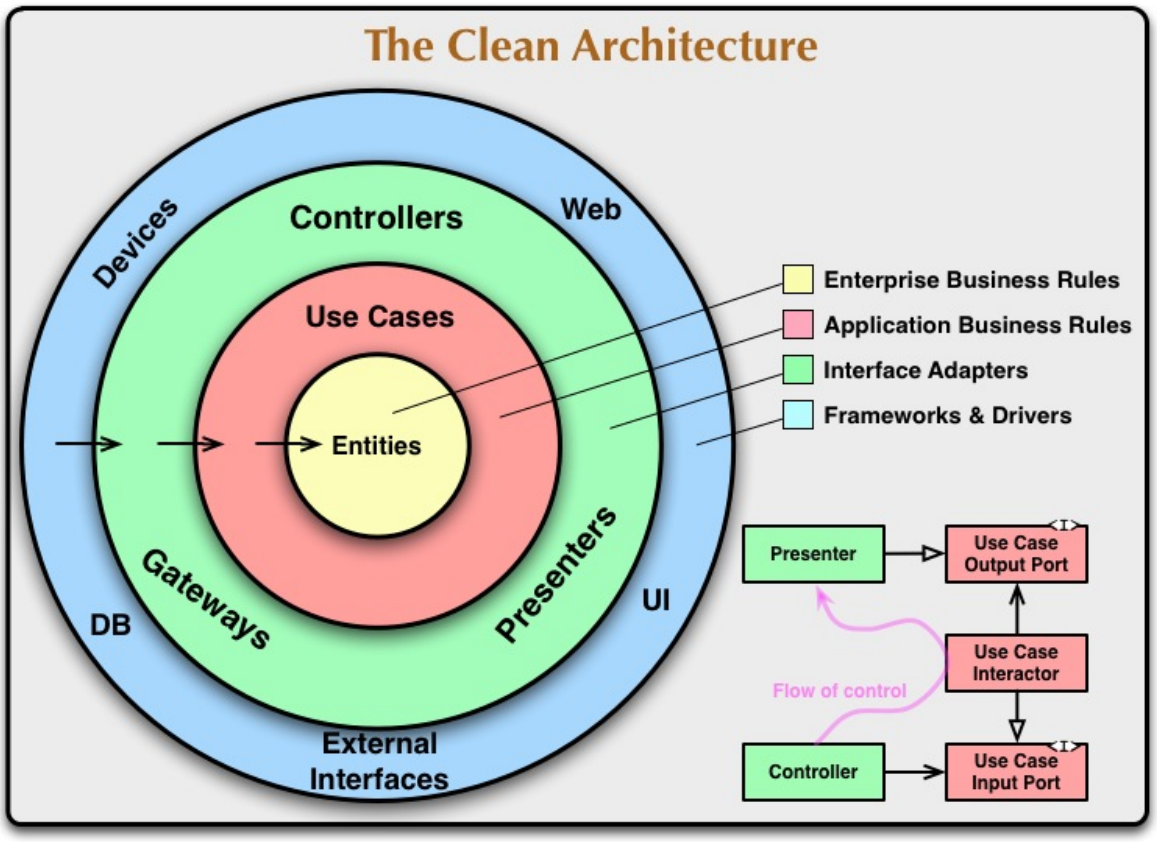
- Testing
  - Podemos mockear la Interface y facilitar el testing

```php
protected function shouldSaveVideo(Video $video)
{
    $this->mock(VideoRepository::class)
        ->shouldReceive('save')
        ->with(similarTo($video))
        ->once()
        ->andReturnNull();
}
```

- Simplicidad
  - Código simétrico.
  - Predecible.

2. Qué es la Arquitectura Hexagonal ⬢

- Parte de las Clean Architectures
  - Las Clean Architectures son una serie de capas en nuestra app con una regla de dependencias.
  - La regla de dependencia va ***de afuera hacia dentro***.
    - Si las capas internas No conocen nada de las Externas, podremos cambiar cosas en las capas externas SIN que las capas internas se enteren.
      - Podremos cambiar Controllers SIN que los Casos de Uso se enteren.
      - Si tenemos que cambiar el Framework en los Controllers, pues se hace sin que las capas internas se enteren.
    - Las Capas Externas SOLO conocen a las Capas Inmediatamente cercanas (de adentro).

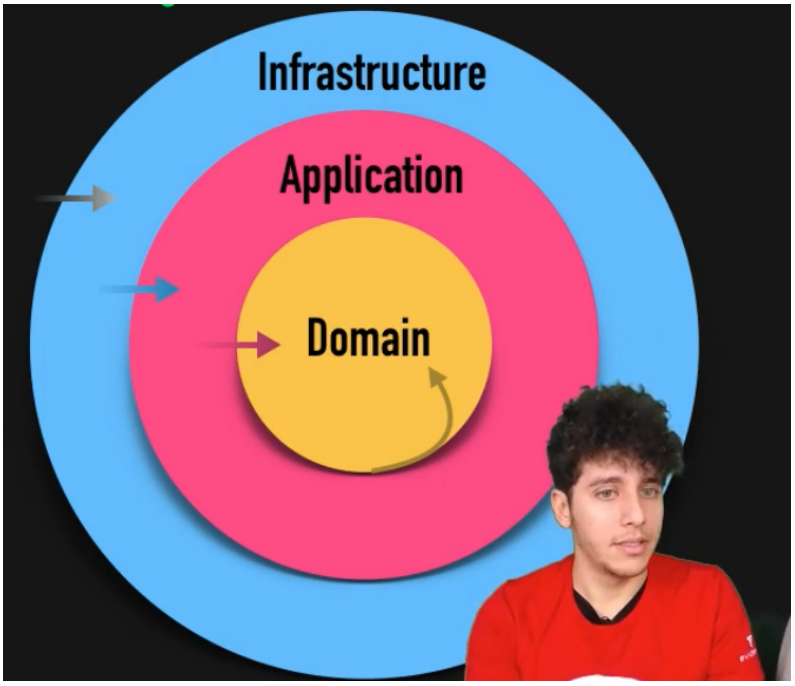


  - Dominio: Es todo lo que nos importa.
    - Son cosas que están en nuestro contexto a nivel de modelos/Entity.

- Son Reglas de Negocio que solo van a variar por criterios propios.
    - Decisiones extremas NO implican cambio en el Dominio.
        - Si cambia la forma de conectarse a Redis, pues esto NO cambia el Dominio.
        - Pero, si decidimos que los primeros videos de todos los cursos sean gratuitos sin registro, eso es Lógica de Negocio que SI afecta el Dominio.
        - El cambio en el Dominio es por decisiones internas de la Lógica de Negocio.
- Si en nuestra empresa tenemos videos, en nuestro dominio tenemos videos.

- **Arquitectura Hexagonal**
    - NO se representa gráficamente con un Hexágono.
    - Puerto y ADtadores

    - *Capas de la Arquitectura Hexagonal*
        - Infrastructure → Application → Domain
            - Regla de dependencia: Afuera hacia Adentro.

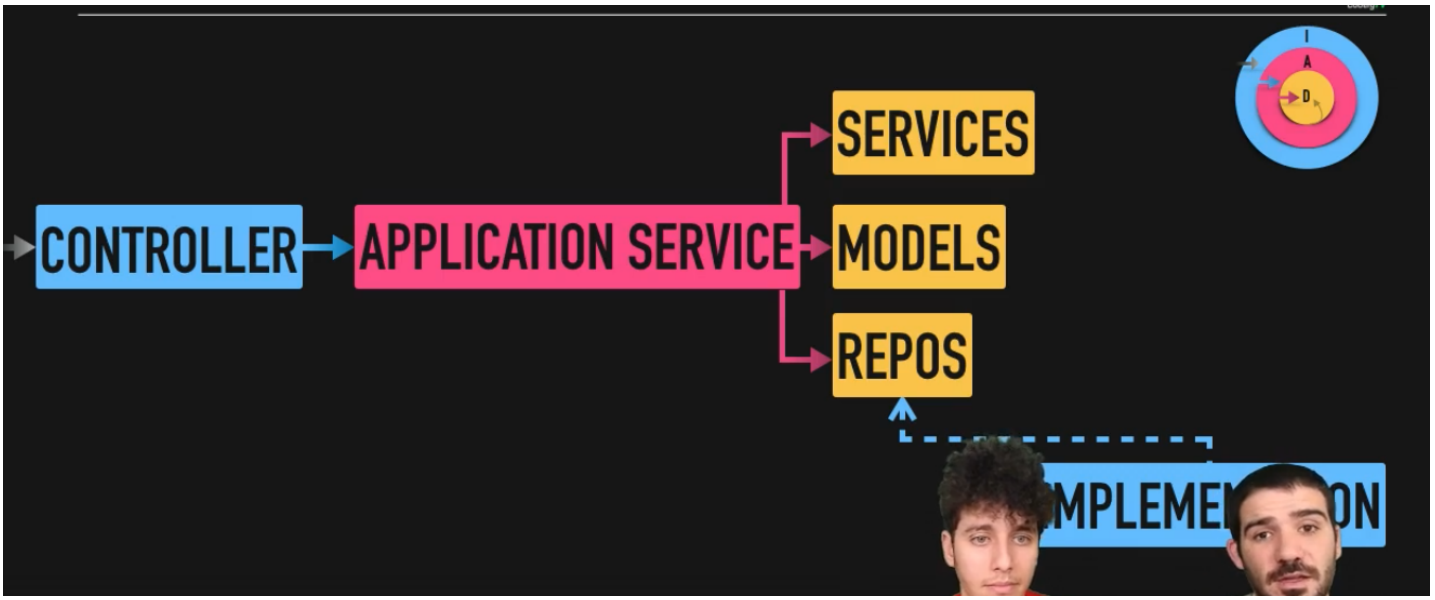


- Infrastructure: Se aplica mediante Interfaces
  - En AH vamos a utilizar Interfaces para Desacoplar el Dominio de la Infraestructura.
  - Todo lo que cambie un estado externo.
  - Cosas que pueden variar por decisiones externas.

- Application
  - los ***Casos de Uso/Servicios de Aplicación/Application Service*** (Features de la app: Se debe poder publicar un video, se debe poder crear un sub, comentar un video, etc.)

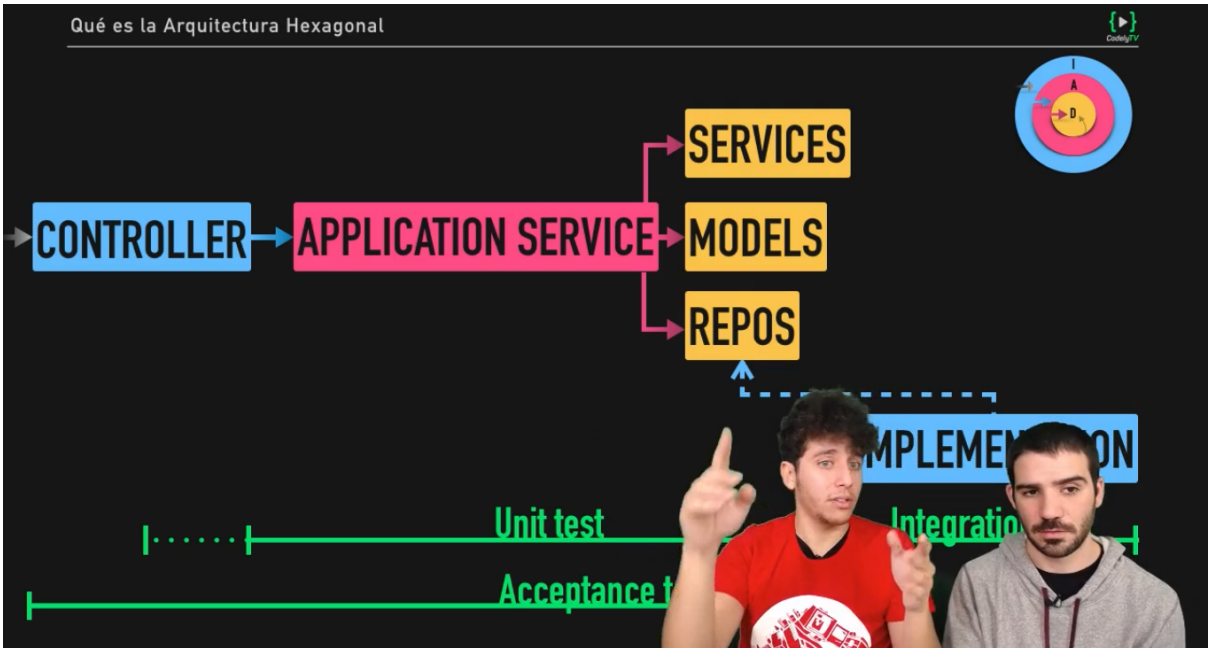
- Domain
  - Nuestras reglas de negocio.
  - Tiene interfaces de dominio.

▪ Una request a un endpoint:
  - Repos es la Interface que solo conoce el Dominio, y para interactuar con persistencia DB se tiene las Implementaciones de estas Interface. Ejemplo, la Implementación de MySQL, en la que ya se tiene código o sentencias de MySQL.



  - Al final estamos aplicando el Dependency Inversion Principle de SOLID.

- Rompe el acoplamiento metiendo un elemento que está en la capa de Dominio a la que SÍ puede acceder la Capa de Aplicación, que sería la Interface Repository.
- Testing:
  - Los Unit Test atacaran a la unidad entendida como Unidad Lógica de Casos de Uso



3. ¡Nuestro primer puerto y adaptador! Patrón Repository 🥓

- Formas de W
  - Active Record:
    - Código acoplado.
    - La infraestructura conoce del Dominio
      - En la infraestructura hay .save() para guardar en DB

  - Data Mapper:
    - Tienes un fichero que mapea de Entidad a DB. Un fichero de infraestructura.

  - DAO: NO usar porque crece demasiado

    - Usar Repository Pattern en su lugar.
        - Se abstrae con Interfaces

- Patrón repository
    - Retorna Colecciones de Componentes YA Instanciados
    - Las excepciones son de dominio
    - El Repositorio NO lanza Excepciones
        - Si no lo encuentra devuelve un null/optional

- Todos nuestro Casos de Uso solo van a tener Imports de Dominio

4. Servicios de infraestructura y estructura de directorios 🎯

- Los ***Repositorios*** no dejan de ser un servicio de Infraestructura
- En Infraestructura da igual tener Imports que No sean nuestros
    - Aqui si hacemos uso de Librerías Externas
    -
    - s
        - s
            - s

- s
    - s

- s
  - s
    - s

- s
  - s
  - s
    - s
      - s

- g
  - s
- g
  - s
    - n
- g

5. Servicios de Aplicación vs. Servicios de Dominio 🤼

-
  - Si necesitamos reutilizar Lógica la Extraemos a un Servicio de Dominio.

  - Los Servicios de Aplicación representan una barrera transaccional tanto a nivel de DB como a nivel de Eventos
  - El Servicio de Aplicación (y NO el de Dominio) va a ser quien Cierre las Transacciones a nivel de DB y quien finalmente publique los Eventos de Dominio que se hayan ido produciendo.
    - *Transacciones y eventos a publicar y cerrar en el Servicio de Aplicación.*

    - ***Instanciar*** a los Servicios de Dominio (o se inyectan) SIN necesidad de implementar una interfaz que nos abstraiga de ese servicio de dominio porque es nuestro.

        - Los Servicios de Aplicación son esos puntos de entrada a la app que coordinan la llamada a distintos elementos del Dominio

- s
    - s
    - s
        - s
            - s

6. Modelando nuestro dominio y publicando eventos 🔑

- Value Object
    - Encapsular la semántica de Dominio en Objetos
        - Esto nos permite validar desde nuestro dominio que sea consistente ese objeto.
            - E.g.: Siempre que se construya una videoUrl, que esta sea una URL válida.
            - Con esto nos olvidamos de los IF repetidos por todo el código ya que aquí estamos validando.

- El Puerto es la Interfaz que defino
- El Adapter de ese puerto es la Implementación concreta
    - s

        - s

- s
  - s
  - s
    - s
      - s

- s
  - s
  - s
    - s
      - s

7. Testing capa de aplicación y dominio ⛱

- Las cosas de Infraestructura son las que desde nuestros tests unitarios Mockeamos para asegurarnos que son llamados.
- En estos tests Mockeamos la Infraestructura
  - En estos test testeamos tanto la capa de Aplicación como la de Dominio, pese a que se empiece por la capa de aplicación.
  - Aquí Mockeamos la Infraestructura.
    - s
      - s

8. Testing capa de infraestructura 🏖️

- Aquí testeamos la Infraestructura.
- Aquí testeamos los Repositorios.
  - Debemos testear los repositorios para asegurarnos que toda la lógica que existe por detrás funciona y hace lo que tiene que hacer.
  - s
    - s
      - s

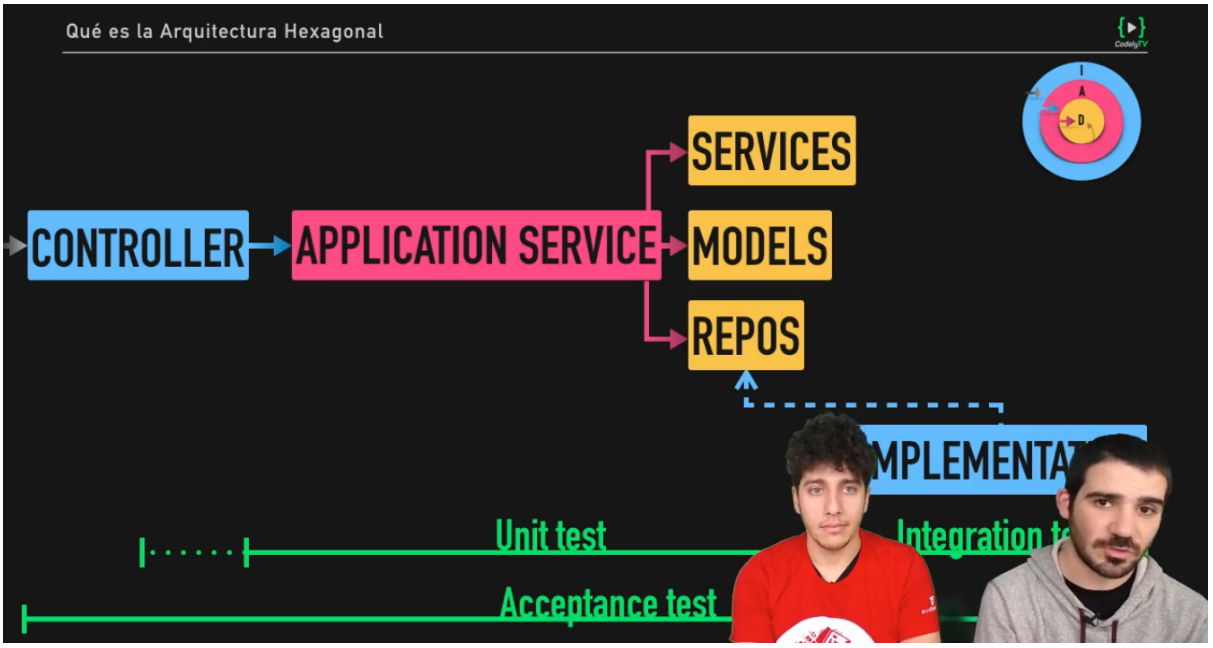
- s
  - s

    - s
        - s
        - s

- s
    - s
    - s
        - s
        - s

9. Resumen 👩‍🎨

- La AH nos ayuda a evitar que crezca desmedidamente la complejidad accidental.
- Capas de la AH
- Estrategias de testing



- s
  - s
  - s
    - s
    - s

- s
  - s
  - s
    - s
    - s

10. ¿Inyectar Servicios de Dominio?

- Puedes Inyectar Servicios de Dominio en lugar de Instanciarlos (new) SIEMPRE Y CUANDO el equipo sea lo suficientemente maduro y domine las reglas de que se inyecta y que no se inyecta.

- Para el testing, aunque se estén inyectando servicios de dominio, lo que vamos a mockear siempre serán cosas de las cuales vamos a tener más de una implementación y están en infraestructura. Por lo tanto NUNCA vamos a hacer un mock de los servicios de dominio. Debemos hacer un mock de la cosa interna con la que interactúe nuestro servicio de dominio.

11. ¿Y si nos quitamos la capa de Aplicación?

- Si quieres seguir AH (Arquitectura Hexagonal) NO puedes quitarte la capa de Aplicación, porque la AH distingue esas 3 capas por definición.

- Si ya NO aplicas AH y te quitas la Capa de Aplicación puedes aún tener una Clean Architecture ya que separas la en dos capas: Dominio e Infraestructura.
  -
  - s
    - s
    - s

- s
  - s
  - s
    - s
    - s

- s
  - s
  - s
    - s
    - s

11.1. j

- g
  - s
- g
  - s
    - n
- g

12. g

12.1. j

- g
  - s
- g
  - s
    - n
- g

12.2. j

- g
  - s
- g
  - s
    - n
- g

12.3.j

- g
  - s
- g
  - s
    - n
- g

12.4.j

- g
  - s
- g
  - s
    - n
- g

12.5.j

- g
  - s
- g
  - s
    - n
- g

12.6.j

- g
  - s
- g
  - s
    - n
- g

12.7. j

- g
  - s
- g
  - s
    - n
- g

12.8. j

- g
  - s
- g
  - s
    - n
- g

12.9. j

- g
  - s
- g
  - s
    - n
- g

13. g

13.1. j

- g
  - s
- g
  - s
    - n
- g

13.2. j

- g
  - s
- g
  - s
    - n
- g

13.3. j

- g
  - s
- g
  - s
    - n
- g

13.4. j

- g
  - s
- g

        - s
            - n
- g

13.5. j

- g
    - s
- g
    - s
        - n
- g

13.6. j

- g
    - s
- g
    - s
        - n
- g

13.7. j

- g
    - s
- g
    - s
        - n
- g

13.8.j

- g
  - s
- g
  - s
    - n
- g

13.9.j

- g
  - s
- g
  - s
    - n
- g

14. g

14.1. j

- g
  - s
- g
  - s
    - n
- g

14.2.j

- g
  - s
- g
  - s
    - n
- g

14.3.j

- g
  - s
- g
  - s
    - n
- g

14.4.j

- g
  - s
- g
  - s
    - n
- g

14.5.j

- g
  - s
- g
  - s
    - n
- g

14.6.j

- g
  - s
- g
  - s
    - n
- g

14.7. j

- g
  - s
- g
  - s
    - n
- g

14.8.j

- g
  - s
- g
  - s
    - n
- g

14.9.j

- g
  - s

- g
  - s
    - n
- g

14.10. j

- g
  - s
- g
  - s
    - n
- g

14.11. j

- g
  - s
- g
  - s
    - n
- g

14.12. j

- g
  - s
- g
  - s
    - n
- g

14.13. j

- g
  - s
- g
  - s
    - n
- g

14.14. j

- g
  - s
- g
  - s
    - n
- g

14.15. j

- g
  - s
- g
  - s
    - n
- g