Algoritmos y Estructuras de Datos II

ISFTN151 – Analista en Sistemas

Módulo: Algoritmos y Estructuras de Datos II

Este módulo considera lo abordado en Algoritmos y Estructuras de Datos I. En tal sentido, se propone acercar a los alumnos a los distintos **paradigmas** que ofrece la programación, **profundizando en aspectos de la programación orientada a objetos sus componentes y técnicas**. Particularmente, el paradigma de programación orientado a objetos les brinda a los futuros profesionales, técnicos que combinan la abstracción, modularización, encapsulamiento, polimorfismo y herencia. Así, la combinación de estos aspectos promueve una forma de resolver problemas, no abstracta, sino más cercana a la realidad.

Capacidades profesionales

Se espera que al finalizar el cursado del módulo los estudiantes sean capaces de:

- Conocer los distintos paradigmas de la programación.
- Conocer y utilizar el paradigma de objetos, sus características, ventajas y aplicaciones dentro del desarrollo de sistemas.
- Adquirir técnicas de resolución de problemas reales.
- Diseñar aplicaciones con frameworks orientado a objetos.

Contenidos:

Conceptos y paradigmas de lenguajes de programación. Comparación entre paradigmas. Paradigma funcional. Paradigma lógico. **Paradigma orientado a objetos**. Clases y objetos. Subclases. Atributos. Métodos. Recursividad. Modificadores de visibilidad. Encapsulación. Sobrecarga de métodos. Concepto de acoplamiento. Herencia. Sobreescritura. Clases abstractas y concretas. Cardinalidad. Atributos y comportamiento. **Diseño UML**. Diagrama de clases. Relaciones entre clases: herencia, asociación, composición y agregación. Diagrama de Secuencia. Diagramas de Interacción y casos de Uso. **Patrones de Diseño**: introducción, definición, descripción, catálogo, utilidad, selección y usos de un patrón. Patrones creacionales, estructurales y de comportamiento. Mover aspectos entre objetos. Simplificación de invocación de métodos. Manipulación de la generalización. **Patrones creacionales** (Abstract Factory, Singleton). **Patrones estructurales** (Composite, Decorador, Adapter, Proxy. **Patrones de comportamiento**: Observer, State, Strategy, Template Method, Command). Refactoring: **Introducción, utilidad y técnicas de aplicación del Refactoring**. Catálogo de refactoring. Refactoring hacia patrones. Unificación de interfaces con el patrón adaptador. Remplazar lógica condicional con el patrón estrategia. Remplazar estados condicionales con el patrón estado. Reemplazar notificaciones con el patrón observador. Mover código embebido al patrón decorador. **Aplicaciones con frameworks orientado a objetos**. Introducción a Frameworks. Reutilización de software vs. Reutilización de diseño. Clasificación de frameworks según su propósito **Testing**. Tipos de tests: de unidad, de integración, de aceptación. **Metodología de desarrollo ágil TDD** (Test Driven Development). Relación entre refactoring y testing.

Algoritmos y Estructuras de Datos II

ISFTN151 – Analista en Sistemas

AyED II

Diseño de Software
Arquitectura y Código
Limpio

Temario…?

Iremos descubriendo el Temario a medida que avance la Introducción, finalmente habemos de haber razonado este.

"Sumo - Mejor no hablar (de ciertas cosas)"…

Esta Diapositiva es Intencionalmente dejada sin contenido, iremos Induciendo y descubriendo los Temas y al Final veremos entonces,... el por que de la Selección del Contenido...

AyED II

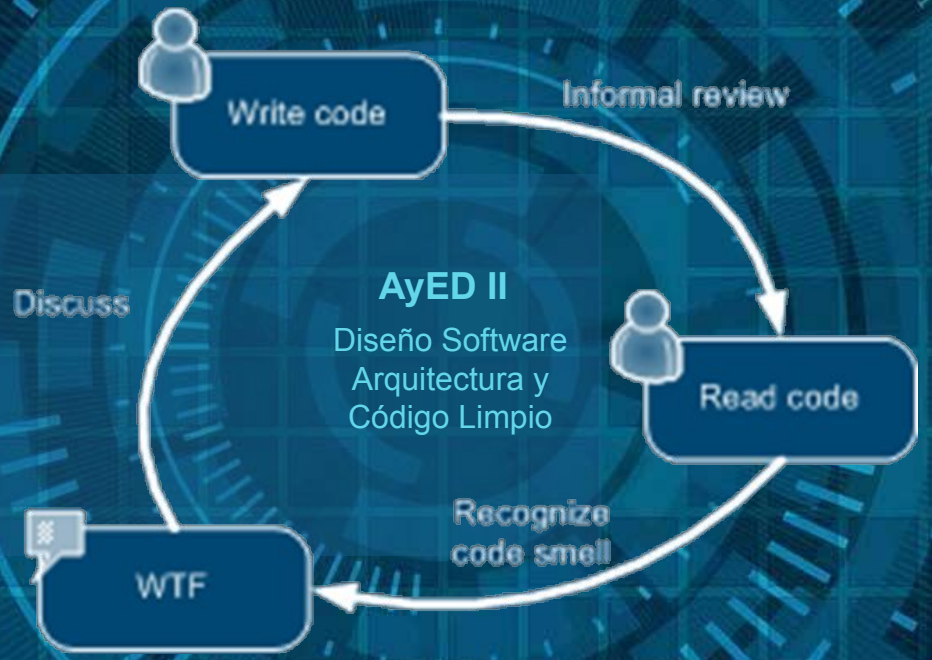
Diseño Software
Arquitectura y
Código Limpio

Unidad 0

Arquitectura y Código Limpio

Una vision sobre el Desarrollo de Software desde una Perspectiva Diferente, Modelos, Procesos, Técnicas, Estatégias, Dinámicas, conceptos… Hacia una Evolución Personal y Profesional sobre como constriur Codigo Límpio.

Escribimos Código, pero….



Cómo Medimos la Calidad del Código?

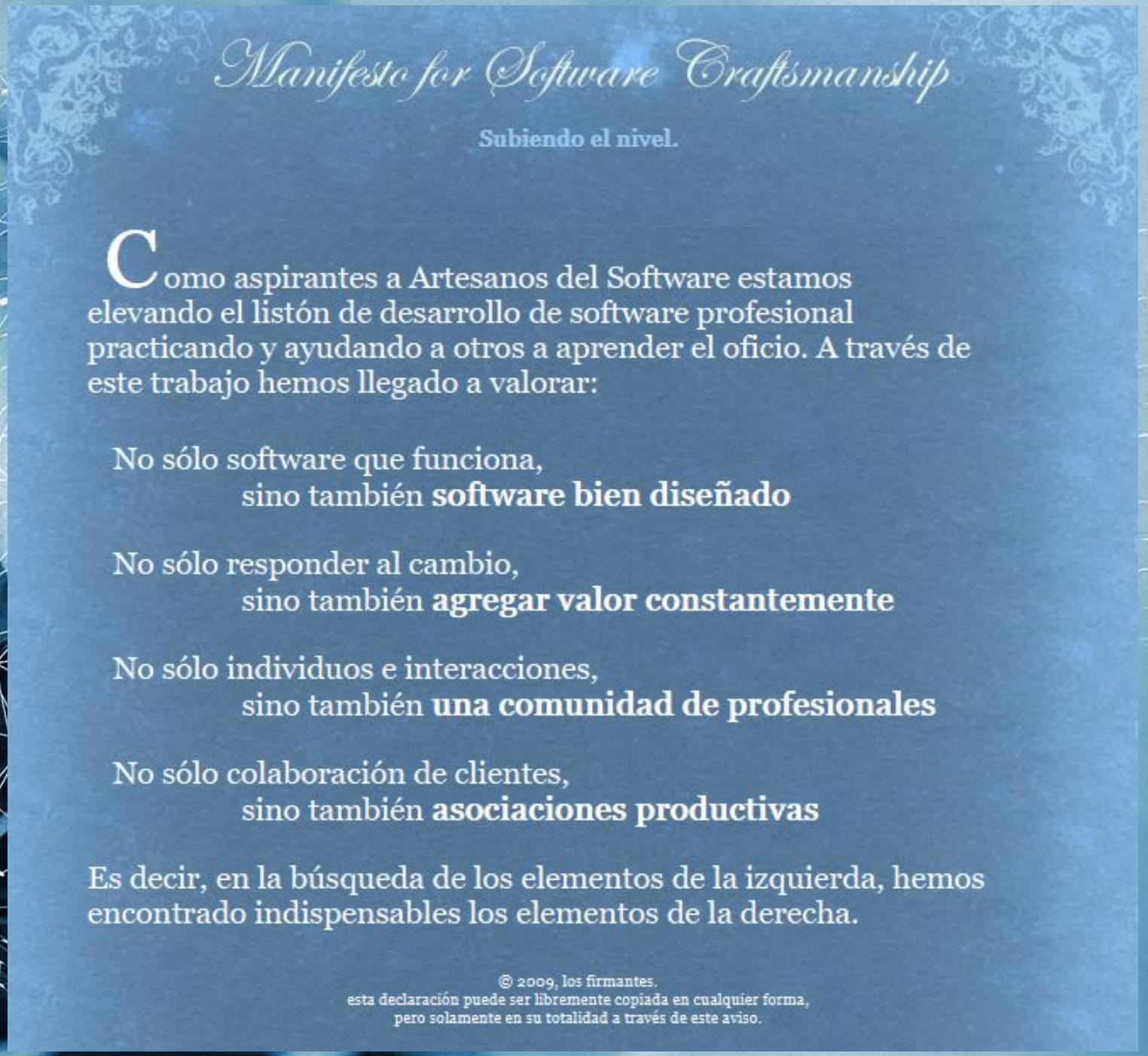
Robert C. Martin también nos habla de la regla del **Boy Scout**, aquella que dice que **“Hay que dejar el código mejor de cómo lo encontraste”**. Esto quiere decir que si tenemos la capacidad de mejorar algo pues debemos hacerlo.

The Software Craftsman

Artesanía de software es un enfoque del desarrollo de software que enfatiza las habilidades de producir código de los propios desarrolladores de software o programadores

Artesanía de software

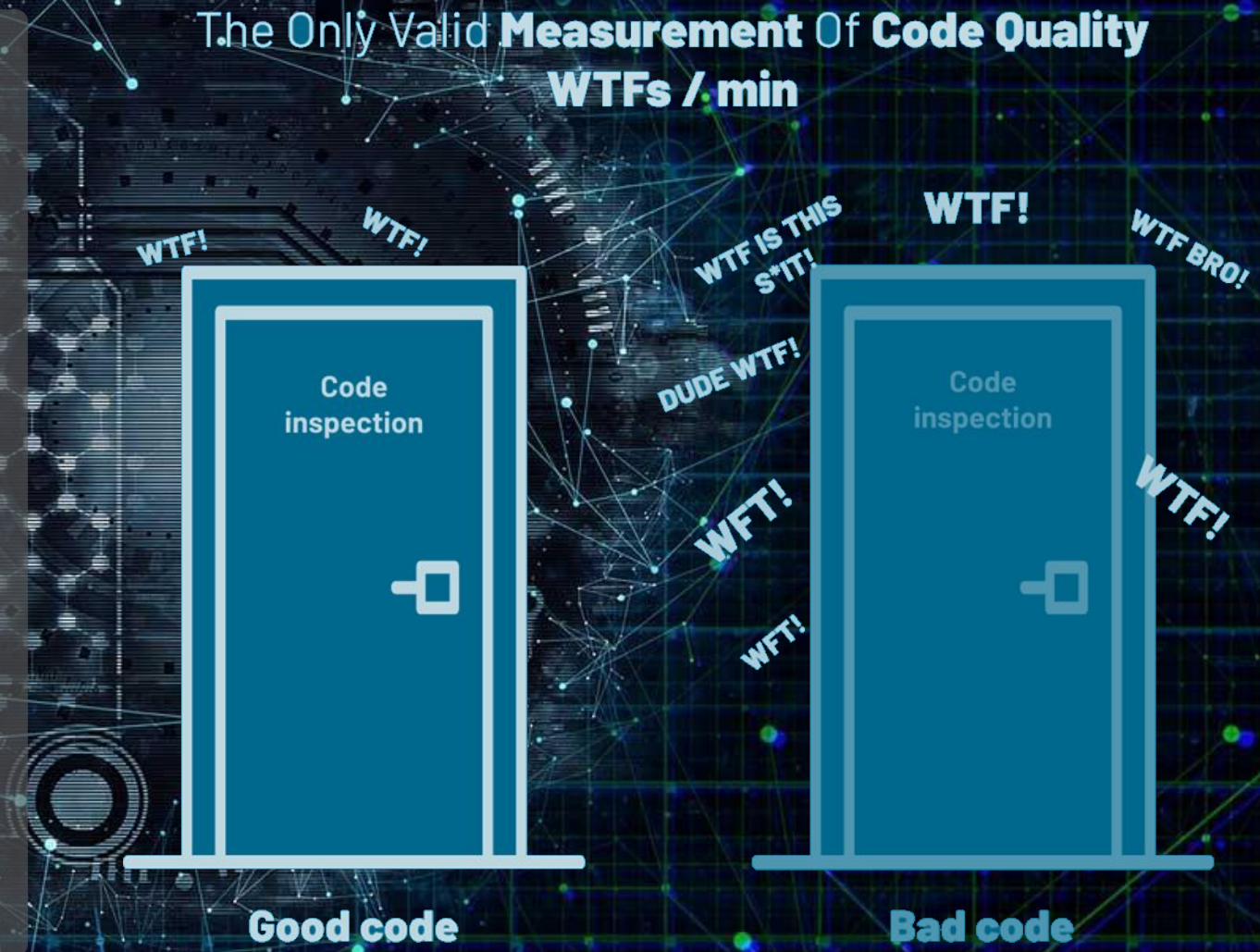
Artesanía de software es un enfoque del desarrollo de software que enfatiza las habilidades de producir código de los propios desarrolladores de software o programadores. Es una respuesta de estos a los males percibidos en las prácticas establecidas de la industria, entre otros la priorización de las preocupaciones financieras sobre la responsabilidad del desarrollador. Históricamente, los programadores han sido animados a verse a sí mismos como practicantes de un análisis estadístico bien definido y con rigor matemático de un enfoque científico con teorías de computación. Esto cambió a un enfoque ingenieril con connotaciones de precisión, previsibilidad, medición, mitigación de riesgos y profesionalismo. La práctica de la ingeniería condujo a llamados a licenciamiento, certificación y a un cuerpo codificado de conocimientos como mecanismos de difusión del conocimiento de la ingeniería y maduración del campo de aplicación.
El Manifiesto Ágil (Agile Manifesto), con su énfasis en "individuos e interacciones por encima de procesos y herramientas" cuestionó algunos de estos supuestos. El manifiesto de la Artesanía de software se extiende y desafía más las suposiciones del Manifiesto Ágil, haciendo una metáfora entre el desarrollo de software moderno y el modelo gremial de la Europa medieval.



The Software Craftsman – Medida de Calidad de Código

¿Qué puerta representa tu código?
¿Qué puerta representa a su equipo o su empresa?
¿Por qué estamos en esa habitación?
¿Es esto solo una revisión de código normal o hemos encontrado un flujo de problemas horribles poco después de salir en vivo?
¿Estamos depurando en pánico, estudiando detenidamente el código?
que pensamos que funcionó?
¿Los clientes se van en masa y los gerentes están respirando nuestros cuellos?
¿Cómo podemos asegurarnos de terminar detrás de la puerta correcta cuando las cosas se ponen?
¿difícil?

La respuesta es: artesanía.



The Software Craftsman – Medida de Calidad de Código

Código Limpio?
La Respuesta es la Artesania: Dividido en **Conocimiento y Trabajo**

La respuesta es: **ARTESANIA**
El aprendizaje de la artesanía consta de dos partes: **Conocimiento y Trabajo.**
Debes ganar el **conocimiento** de los principios, patrones, prácticas y heurísticas que un artesano conoce, y también debe modelar ese conocimiento en sus dedos, ojos e intestinos , con un **trabajo** duro y practicando…

Puedo enseñarte la física de andar en bicicleta. De hecho, las matemáticas clásicas son relativamente sencillo. Gravedad, fricción, momento angular, centro de masa, etc. adelante, se puede demostrar con menos de una página llena de ecuaciones. Dadas esas fórmulas yo podría demostrarle que andar en bicicleta es práctico y brindarle todos los conocimientos que necesita necesario para que funcione. Y todavía te caerías la primera vez que te subiste a esa bicicleta.

La codificación no es diferente. Podríamos escribir todos los principios de "sentirse bien" de la limpieza y luego confiar en ti para hacer el trabajo (en otras palabras, dejarte caer cuando te subas bicicleta), pero entonces, ¿qué clase de profesores nos convertirían y qué clase de estudiante eso te haría?
No. No es así como va a funcionar esta Cátedra, será Trabajo Duro y Practicando..

Why Writing Clean Code? (What About Clean Architecture?)

Aprender a escribir código limpio es un trabajo duro.
Requiere algo más que el conocimiento de principios y patrones.

Debes sudar por eso.
Debes practicarlo tú mismo y observar tú mismo fallas.
Debes ver a otros practicarlo y fallar.
Debes verlos tropezar y vuelve sobre sus pasos.
Debes verlos agonizar por las decisiones
Debes ver el precio que pagan tomando esas decisiones de la manera incorrecta

Dios está en los detalles, dijo el arquitecto Ludwig mies van der Rohe. Esta cita recuerda argumentos contemporáneos sobre el papel de la arquitectura en el desarrollo de software, y en particular en el mundo ágil.
Y sí, mies van der Rohe estuvo atento a la utilidad y a las formas atemporales de edificios que subyacen a la gran arquitectura. Por otro lado, también seleccionó personalmente cada pomo de cada casa que diseñó. ¿Por qué?
Porque las pequeñas cosas importan. La atención a los detalles es un factor aún más crítico.
La base de profesionalismo que cualquier gran visión requiere.
Primero, es a través de la práctica en las pequeñas para que los profesionales adquieran competencia y confianza para la práctica en las grandes.
Segundo, el pedacito más pequeño de construcción descuidada, de la puerta que no cierra herméticamente o el leve baldosas torcidas en el suelo, o incluso el escritorio desordenado, disipa por completo el encanto de la conjunto más grande.

De eso se trata el código limpio.


Diseño - Ludwig mies van der Rohe

Why Writing Clean Code? (What About Clean Architecture?)

Sin embargo, la arquitectura es solo una **metáfora** del desarrollo de software y, en particular, de esa parte del software que entrega el producto inicial en el mismo sentido que un arquitecto entrega un edificio impecable.

En estos días de Scrum y Agile, la atención se centra en la rapidez llevar el producto al mercado.
Queremos que la fábrica funcione a la máxima velocidad para producir software.

Pero estas son fábricas humanas: codificadores que piensan y sienten que trabajan a partir de una cartera de productos o historia de usuario para crear un producto.

"La metáfora de la fabricación cobra cada vez más fuerza en gente que esta pensando…"

Los aspectos de producción de la fabricación de automóviles de una línea de montaje Japonesa inspiran gran parte de Scrum. Sin embargo, incluso en la industria automotriz, "*la mayor parte del trabajo no reside en la fabricación, sino en mantenimiento (o su evitación)*…" En software, el 80% o más de lo que hacemos se llama curiosamente "Mantenimiento": el acto de reparar. En lugar de adoptar el enfoque occidental típico de producir buen software, deberíamos pensar más como reparadores de viviendas en el edificio industria, o mecánica automotriz en el campo automotriz.

…" En software, el 80% o más de lo que hacemos se llama curiosamente "Mantenimiento": el acto de reparar.

Comparación con El Mantenimiento Productivo Total (TPM).

Aproximadamente en 1951, surgió un enfoque de calidad llamado Mantenimiento Productivo Total (TPM).
En la escena japonesa se centra en el mantenimiento más que en la producción. Uno de los pilares principales de TPM son el conjunto de los llamados principios 5S. 5S es un conjunto de "disciplinas“. Estos principios de las 5S están de hecho en los cimientos de Lean, otra palabra de moda en la escena occidental, y una cada vez más prominente palabra de moda en los círculos del software. Estos principios no son una opción. La buena práctica del software requiere tal disciplina: concentración, presencia de ánimo, y pensar. No siempre se trata solo de hacer, de impulsar el equipo de la fábrica para producir a la velocidad óptima. La filosofía 5S comprende estos conceptos:

• **Seiri u organización** (piense en "sort" en inglés). Saber dónde están las cosas: usar enfoques como la denominación adecuada, es crucial. ¿Crees que nombrar identificadores no es importante?.

• **Seiton, o tidiness** (piense en "sistematizar" en inglés). Hay un viejo dicho americano: Un lugar para cada cosa y cada cosa en su lugar. Un fragmento de código debe estar donde espera encontrarlo y, si no es así, debe volver a factorizarlo para llegar allí.

• **Seiso, o limpieza** (piense en "brillar" en inglés): mantenga el lugar de trabajo libre de colgar alambres, grasa, desechos y desperdicios. ¿Qué dicen los autores aquí sobre tirar basura código con comentarios y líneas de código comentadas que capturan el historial o los deseos del futuro? Deshazte de ellos.

• **Seiketsu, o estandarización**: el grupo acuerda cómo mantener limpio el lugar de trabajo. ¿Crees que este libro dice algo sobre tener un estilo de codificación consistente y un conjunto de prácticas dentro del grupo? ¿De dónde provienen esos estándares?

• **Shutsuke o disciplina (autodisciplina).** Esto significa tener la disciplina para seguir las prácticas y reflexionar con frecuencia sobre el trabajo de uno y estar dispuesto a cambiar.

Por que profesionalismo responsable ?

Aquí, finalmente nos dirigimos a las raíces del profesionalismo responsable en una profesión que debe preocuparse por el ciclo de Vida de un producto. Dado que mantenemos automóviles y otras máquinas bajo TPM, Trabajar en la avería el mantenimiento, a la espera de que aparezcan errores, es la excepción.

En cambio, subimos un nivel: inspeccione las máquinas todos los días y repare las piezas de desgaste antes de que se rompan, o haga lo equivalente al proverbial cambio de aceite de 10,000 millas para prevenir el desgaste. ***En código, refactorizar sin piedad.***
Puede mejorar un nivel más, ya que el movimiento TPM innovó hace más de 50 años: construya máquinas que sean más fáciles de mantener en primer lugar. ***Haciendo su código legible es tan importante como hacerlo ejecutable***.
La máxima práctica, introducido en los círculos de TPM alrededor de 1960, se centra en la introducción de máquinas completamente nuevas o reemplazando los viejos. Como nos advierte Fred Brooks, probablemente deberíamos rehacer el software importante desde cero cada siete años aproximadamente para barrer el cruft rastrero. Quizás deberíamos actualizar la constante de tiempo de Brooks a un orden de semanas, días u horas en lugar de años. Ahí es donde reside el detalle.
Código limpio Honra las raíces profundas de la sabiduría debajo de nuestra cultura más amplia, o nuestra cultura como lo fue antes, o debería ser y puede ser con atención al detalle.
Incluso en la gran literatura arquitectónica encontramos huellas que se remontan a estos supuestos detalles. Piense en los pomos de las puertas de mies van der Rohe. Eso es seiri. Eso es estar atento a cada nombre de variable. Debe nombrar una variable con el mismo cuidado con el que nombrar un hijo primogénito.
Como todo propietario sabe, ese cuidado y refinamiento continuo nunca terminan. El arquitecto Christopher Alexander, padre de los ***patrones y los lenguajes de patrones***, considera cada acto de diseño en sí mismo como un pequeño acto local de reparación. Y ve la artesanía de fina estructura que será competencia exclusiva del arquitecto; las formas más grandes se pueden dejar a los patrones y su aplicación por parte de los habitantes.

CLEAN CODE

Código Límpio

¿El arte del código limpio?

Supongamos que cree que el código desordenado es un impedimento importante. Digamos que acepta que la única forma de hacerlo rápido es mantener limpio el código. Entonces debes preguntarte: **"¿Cómo ¿Escribo código limpio? "** No sirve de nada intentar escribir código limpio si no sabes lo que es significa que el código esté limpio! La mala noticia es que escribir código limpio es muy parecido a pintar una imagen. La mayor parte de nosotros saber cuando un cuadro está bien o mal pintado. Pero poder reconocer el buen arte de malo no significa que sepamos pintar. También ser capaz de reconocer código limpio de código sucio no significa que sepamos cómo escribir código limpio.

Escribir código limpio requiere el uso disciplinado de una miríada de pequeñas técnicas aplicadas a través de un sentido de "limpieza" cuidadosamente adquirido. Este **"sentido del código"** es la clave. Algunos de nosotros nacemos con eso. Algunos tenemos que luchar para adquirirlo. No solo nos deja ver si el código es bueno o malo, pero también nos muestra la estrategia para aplicar nuestra disciplina para transformar código malo en código limpio. Un programador sin "sentido de código" puede mirar un módulo desordenado y reconocer el lío pero no tengo idea de qué hacer al respecto. Un programador con "sentido de código" se verá en un módulo desordenado y ver opciones y variaciones. El "sentido del código" ayudará al programador elegir la mejor variación y guiarlo para trazar una secuencia de comportamiento preservando las transformaciones para ir de aquí para allá.

En resumen, un programador que escribe código limpio es un artista que puede tomar una pantalla en blanco a través de una serie de transformaciones hasta convertirse en un sistema elegantemente codificado.

¿Algunas definiciones sobre Qué es el código limpio?

Bjarne Stroustrup

Probablemente haya tantas definiciones como programadores. Les propongo a algunos programadores bien conocidos y profundamente experimentados lo que pensaban. Bjarne Stroustrup, inventor de C ++ y autor de ***The C++ Programming Language :***

…Me gusta que mi código sea elegante y eficiente. la lógica debe ser sencilla para hacerlo difícil para que los errores se oculten, las dependencias mínimas para mantenimiento sencillo, manejo de errores completo según una estrategia articulada y desempeño cerca de lo óptimo para no tentar personas para hacer el código desordenado sin principios optimizaciones.
El código limpio hace una cosa bien.

¿Qué es el código limpio?

Bjarne Stroustrup, inventor de C ++

Bjarne piensa que el código limpio es agradable para leer. Leerlo debería hacerte sonreír como una caja de música bien elaborada o bien diseñada coche lo haría.
Bjarne también menciona la eficiencia, “dos veces”. Quizás esto no debería sorprendernos al venir del inventor de C ++; pero creo que hay más que el puro deseo de velocidad. Los ciclos desperdiciados no son elegantes, no agradan. Y ahora fíjense en la palabra que usa Bjarne para describir la consecuencia de esa falta de elegancia. Utiliza la palabra "tentar". Hay un profundo de verdad aquí. ¡El código incorrecto tienta al desorden a crecer! Cuando otros cambian un código incorrecto, tienden a empeorarlo.

Los pragmáticos Dave Thomas y Andy Hunt dijeron esto de una manera diferente. Usaron la metáfora de ventanas rotas. Un edificio con ventanas rotas parece que a nadie le importa eso. Entonces otras personas dejan de preocuparse. Permiten que se rompan más ventanas. Finalmente los rompen activamente. Despojan la fachada con grafitis y permiten que se acumule la basura. Una ventana rota inicia el proceso hacia la descomposición.

Bjarne también menciona que la gestión de errores debe estar completa. Esto va a la disciplina de prestar atención a los detalles. El manejo abreviado de errores es solo una forma en que los programadores pasar por alto los detalles. Las pérdidas de memoria son otra, las condiciones de carrera son otra.
Inconsistente nombrar a otro. El resultado es que el código limpio muestra mucha atención al detalle.
Bjarne cierra con la afirmación de que el código limpio hace una cosa bien. No es casualidad que hay tantos principios de diseño de software que se pueden reducir a este simple amonestación. Escritor tras escritor ha intentado comunicar este pensamiento. El código incorrecto intenta hacer demasiado, tiene intenciones confusas y ambigüedad de propósito.

“…El código limpio está enfocado. Cada función, cada clase, cada módulo expone una actitud resuelta que permanece completamente sin distracciones y sin contaminación por los detalles circundantes.”

Bjarne Stroustrup
inventor de C ++

¿Qué es el código limpio?

Grady Booch, author of ***Object Oriented Analysis and Design with Applications and UML***

“El código limpio es simple y directo. Código limpio se lee como una prosa bien escrita. Código limpio nunca oscurece la intención del diseñador, sino que está lleno de abstracciones nítidas y líneas sencillas de control…”

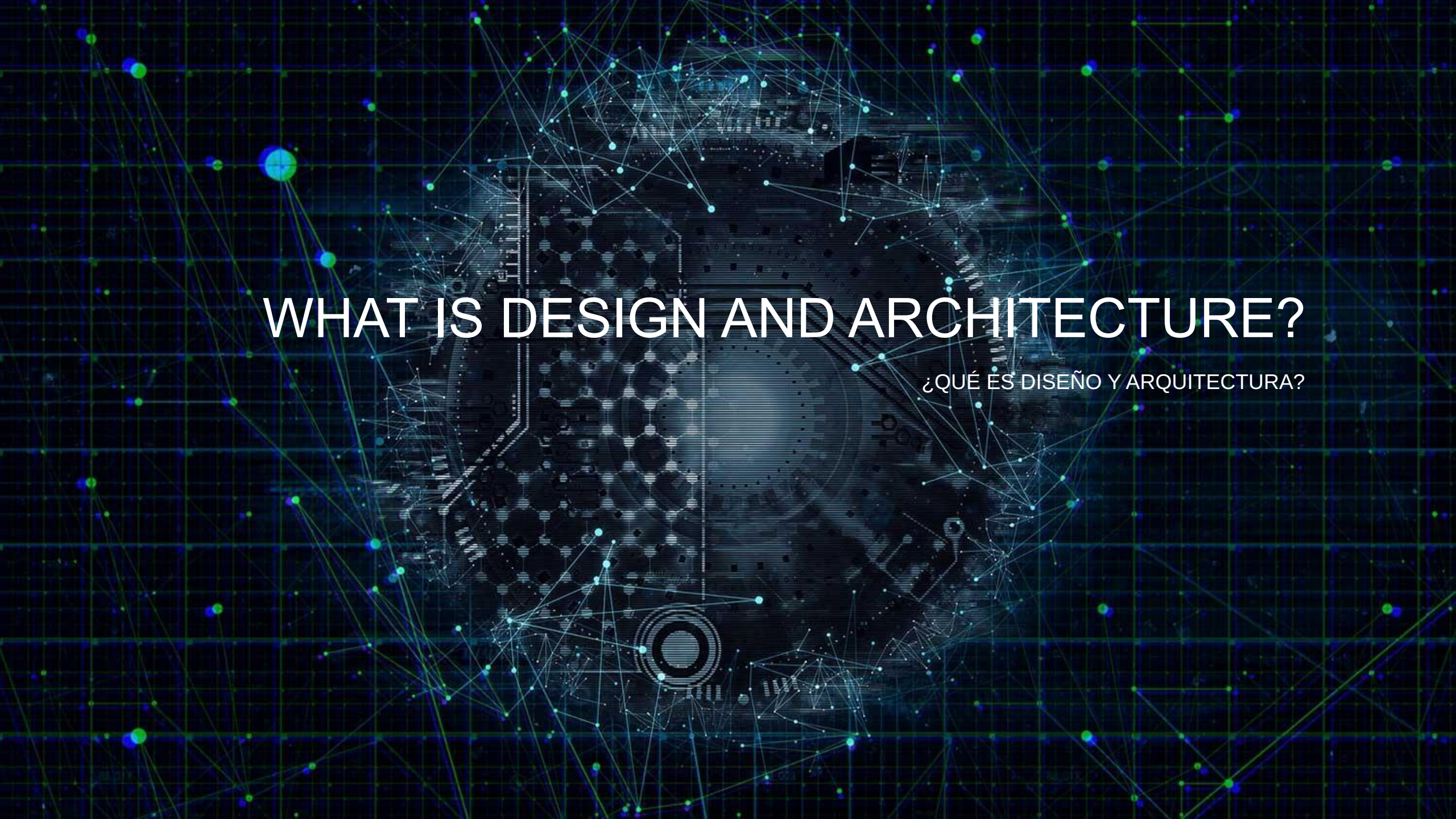
Grady hace algunos de los mismos puntos que Bjarne, pero adopta una perspectiva de legibilidad. Yo especialmente como su opinión de que el código limpio debería leer como prosa bien escrita. Piense en un
muy buen libro que has leído. Recuerda como las palabras desaparecieron para ser reemplazadas por imágenes! Fue como ver una película, ¿no? ¡Mejor! Viste a los personajes, tu escuchaste los sonidos, experimentaste el patetismo y el humor.
Leer código limpio nunca será como leer El señor de los anillos. Aún así, el literario la metáfora no es mala. Como una buena novela, el código limpio debería exponer claramente las tensiones en el problema a resolver. Debería llevar esas tensiones a un clímax y luego darle al lector ese “¡Ajá! ¡Por supuesto!" a medida que los problemas y tensiones se resuelven en la revelación de una solución obvia.
Me parece que el uso de Grady de la frase *"abstracción nítida"* es un oxímoron fascinante.
Después de todo, la palabra "crujiente" es casi un sinónimo de "hormigón". El Wiki tiene la siguiente definición de "crujiente": enérgicamente decisivo y práctico, sin dudarlo o detalles innecesarios. A pesar de esta aparente yuxtaposición de significado, las palabras llevar un mensaje poderoso. Nuestro código debe ser práctico en lugar de especulativo. Debe contener solo lo necesario. Nuestros lectores deberían percibir que hemos sido decisivos al momento de la Codificación.

¿Qué es el código limpio?

"Big" Dave Thomas, founder of OTI, godfather of the Eclipse strategy

"El código limpio se puede leer y mejorar con un desarrollador que no sea su autor original. Tiene pruebas unitarias y de aceptación. Tiene significado nombres. Proporciona una forma en lugar de muchas formas de hacer una cosa. Tiene dependencias mínimas, que se definen explícitamente y proporciona una API clara y mínima. El código debe ser alfabetizados ya que, dependiendo del idioma, no todos la información necesaria se puede expresar claramente solo en código. …"

Big Dave comparte el deseo de Grady por la legibilidad, pero con un giro importante. Dave afirma que
El código limpio facilita que otras personas lo mejoren. Esto puede parecer obvio, pero no ser exagerado. Después de todo, existe una diferencia entre el código que es fácil de leer y código que es fácil de cambiar.
¡Dave relaciona la limpieza con las pruebas! Hace diez años esto habría levantado muchas cejas.
Pero la disciplina del desarrollo basado en pruebas ha tenido un profundo impacto en nuestra industria y se ha convertido en una de nuestras disciplinas más fundamentales. Dave tiene razón. Código, sin pruebas, no está limpio. No importa lo elegante que sea, no importa lo legible y accesible que sea, si no tiene pruebas, será inmundo.
Dave usa la palabra mínimo dos veces. Aparentemente, valora el código que es pequeño, más bien que el código que es grande. De hecho, este ha sido un estribillo común en toda la literatura sobre software.
desde su concepción. Cuanto más pequeño, mejor.
Dave también dice que el código debe ser alfabetizado. Esta es una referencia suave a la alfabetización de Knuth programación.4 El resultado es que el código debe estar compuesto de tal forma que
es legible por humanos.



WHAT IS DESIGN AND ARCHITECTURE?

¿QUÉ ES DISEÑO Y ARQUITECTURA?

¿QUÉ ES DISEÑO Y ARQUITECTURA?

Ha habido mucha confusión sobre el diseño y la arquitectura a lo largo de los años.

¿Qué es el diseño? ¿Qué es la arquitectura? ¿Cuáles son las diferencias entre los dos?
Afiirma Robert Martin que no hay diferencia entre ellos. Ninguno en absoluto. La palabra **“Arquitectura"** se utiliza a menudo en el contexto de algo en un nivel alto que está divorciado de los detalles del nivel inferior, mientras que **“Diseño"** parece más a menudo implicar estructuras y decisiones en un nivel inferior.

Pero este uso no tiene sentido cuando se mira lo que hace un arquitecto real. Piense en el arquitecto que diseñó mi nueva casa. ¿Esta casa tiene anarquitectura? Claro que lo hace. ¿Y cuál es esa arquitectura? Bueno, es la forma de la casa, la apariencia exterior, las elevaciones y la distribución de los espacios y habitaciones. Pero al mirar los diagramas que produjo mi arquitecto, veo una inmensa cantidad de detalles de bajo nivel. Veo dónde se colocarán cada tomacorriente, interruptor de luz y luz. Veo qué interruptores controlan qué luces. Veo dónde está colocado el horno y el tamaño y la ubicación del calentador de agua y la bomba de sumidero. Veo descripciones detalladas de cómo se construirán las paredes, los techos y los cimientos.

En resumen, veo todos los pequeños detalles que respaldan todas las decisiones de alto nivel. También veo que esos detalles de bajo nivel y decisiones de alto nivel son parte del diseño completo de la casa, y lo mismo ocurre con el diseño de software. Los detalles de bajo nivel y la estructura de alto nivel son todos parte de un mismo todo. Forman un tejido continuo que define la forma del sistema. No puedes tener uno sin el otro; de hecho, ninguna línea divisoria clara los separa. Simplemente hay un continuo de decisiones desde los niveles más altos hasta los más bajos.

¿QUÉ ES DISEÑO Y ARQUITECTURA?

Hace casi 2600 años, Esopo contó la historia de la tortuga y la liebre. La moraleja de esa historia se ha expresado muchas veces de muchas formas diferentes:

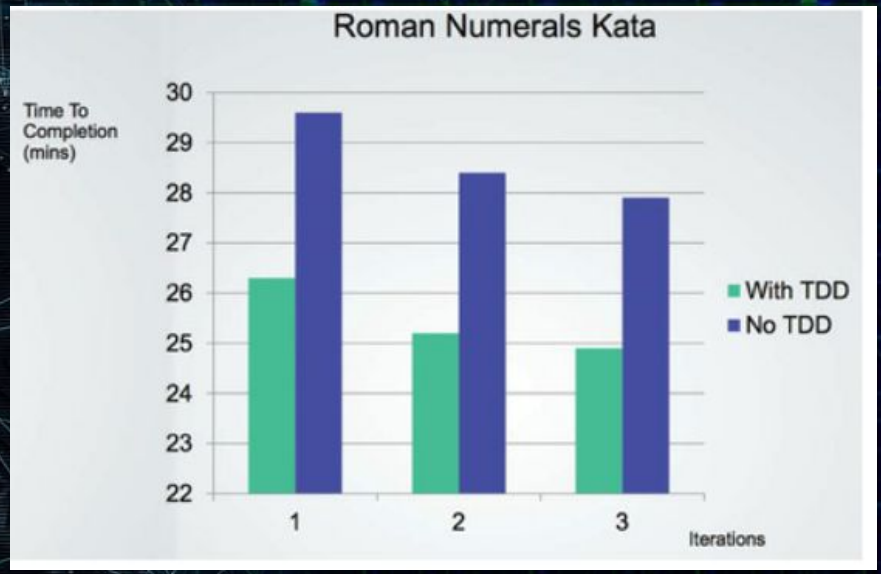
• “Lento y constante gana la carrera”.
• “La carrera no es para los rápidos, ni la batalla para los fuertes”.
• “Cuanto más prisa, menos velocidad ”.

La historia misma ilustra la estupidez del exceso de confianza. La Liebre, tan confiada en su velocidad intrínseca, no se toma la carrera en serio, y por eso duerme mientras la Tortuga cruza la línea de meta.

Los desarrolladores modernos están en una carrera similar y exhiben un exceso de confianza similar. Oh, no duermen, ni mucho menos. La mayoría de los desarrolladores modernos trabajan duro***. Pero aparte de su cerebro sí duerme, la parte que sabe que el código bueno, limpio y bien diseñado es importante***. Estos desarrolladores compran una mentira familiar: “Podemos limpiarlo más tarde; ¡solo tenemos que llegar al mercado primero! " Por supuesto, las cosas nunca se arreglan más tarde, porque las presiones del mercado nunca disminuyen. Llegar al mercado primero simplemente significa que ahora tienes ahorrados competidores detrás de ti, y tienes que mantenerte por delante de ellos corriendo lo más rápido que puedas, por lo que los desarrolladores nunca cambian de modo. No pueden volver atrás y limpiar las cosas porque tienen que hacer la siguiente función, y la siguiente, y la siguiente, y la siguiente. Y así crece el desorden y la productividad continúa su enfoque asintótico hacia cero.

¿QUÉ ES DISEÑO Y ARQUITECTURA?

Así como la liebre estaba demasiado confiada en su velocidad, los desarrolladores confían demasiado en su capacidad para seguir siendo productivos**. Pero el lío progresivo de código que mina su productividad nunca duerme ni cede.** Si se le da su camino, reducirá la productividad a cero en cuestión de meses. La mentira más grande que los desarrolladores aceptan es la noción de que escribir código desordenado hace que se vayan rápido a corto plazo y simplemente los ralentiza a largo plazo. Aceptar esta mentira exhibir el exceso de confianza de la liebre en su capacidad para cambiar de modo de hacer líos a limpiar líos en algún momento en el futuro, pero también cometen un simple error de hecho. **El hecho es que hacer líos es siempre más lento que mantenerse limpio, sin importar la escala de tiempo que esté usando.**
Considere los resultados de un notable experimento realizado por Jason Gorman representado en la Figura debajo. Jason realizó esta prueba durante un período de seis días. Cada día completaba un programa sencillo para convertir números enteros en números romanos. Sabía que su trabajo estaba completo cuando pasó su conjunto predefinido de pruebas de aceptación. Cada día la tarea tomaba un poco menos de 30 minutos. Jason usó una disciplina de limpieza conocida llamada desarrollo impulsado por pruebas (TDD) en el primer, tercer y quinto día. En los otros tres días, escribió el código sin esa disciplina.



Kata (型 o 形) (‘forma’) es una palabra japonesa que describe lo que en un inicio se consideró una serie, forma o secuencia de movimientos establecidos que se pueden practicar tanto en solitario como en parejas.
Una (**Code Kata**) **kata de código** en programación es un ejercicio dirigido a que los programadores desarrollen sus habilidades a base de práctica y repetición. Probablemente fue Dave Thomas, coautor del libro *The Pragmatic Programmer*,[1] quien acuñó el término, en un guiño al concepto japonés de kata de las artes marciales.

ARQUITECTURAS DE SOFTWARE

Algo sobre Diseño y Arquitectura

Arquitectura de software y sus beneficios

¿Qué es la arquitectura?

La RAE define la arquitectura como:

1. *Arte de proyectar y construir edificios.*
2. *Diseño de una construcción.*

Aplicándolo al mundo del desarrollo de software podríamos redefinir arquitectura de software como:

1. *Arte de proyectar y construir aplicaciones informáticas.*
2. *Diseño de una aplicación informática.*

La definición que le damos a arquitectura viene del mundo de la construcción pero viene a significar lo mismo en nuestro mundo, solo que nosotros no construimos edificios, ***construimos software.***

Arquitectura de software y sus beneficios

Qué es una arquitectura de software?

Una arquitectura de software define la forma de trabajar en un sistema, como construir nuevos módulos, pero también debe dejar intuir el tipo de aplicación que describe.

Tal como comenta Uncle Bob, si mostráramos un dibujo arquitectónico de una iglesia o de un piso, simplemente con ver la forma que tiene ese dibujo podemos intuir que tipo de edificio está proyectando. Así pues, si observamos nuestro dibujo arquitectónico de software deberíamos de poder intuir qué tipo de aplicación va a ser construida. No es lo mismo una aplicación que controla un hospital que una aplicación de un cajero automático, cada una tendría un dibujo arquitectónico distinto.

Sin embargo, el dibujo arquitectónico en la construcción no deja claro los materiales con los que está hecha, así mismo en el dibujo arquitectónico de nuestro sistema no deberíamos dejar escapar detalles de nuestra implementación.

Así pues, considero el dibujo arquitectónico en un proyecto de software la propia estructura de módulos y carpetas o paquetes en el caso de Java o cualquier añadido que ayude a expresar la intención de nuestro sistema sin expresar el cómo está hecha.

Arquitecturas limpias

Uncle Bob además define una serie de “arquitecturas limpias” que tienen una serie de objetivos en común:

1. Independiente de los frameworks. La existencia de esta forma de construir cosas en el sistema no depende de un framework.

2. Testable. Tu arquitectura hace que tu código pueda ser testado.

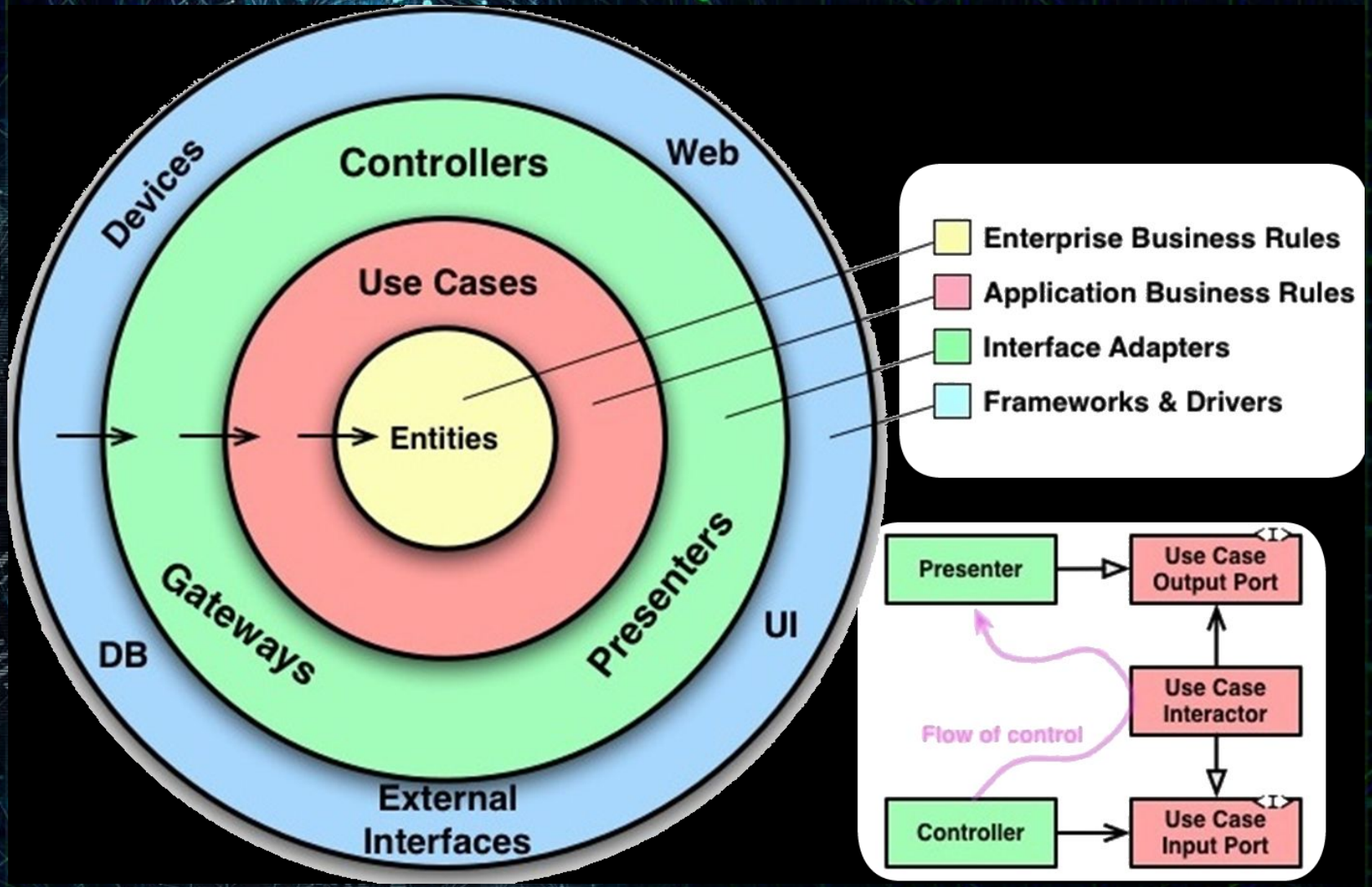
3. Independiente de la UI. Tus reglas de negocio no se ven alteradas por un requerimiento de UI, cuando desarrollas una funcionalidad nueva es la UI la que se adapta a tus reglas de negocio y nunca al revés.

4. Independiente de la base de datos. Puedes cambiar el motor de persistencia ya que tus reglas de negocio no son dependientes de la implementación concreta de la base de datos sino que es la base de datos la que se adapta a estas reglas.

5. Independiente de cualquier componente externo. Se aplica la misma regla descrita en la base de datos pero relacionada a componentes externos así como integraciones con otros sistemas, librerías, etc…
Si una arquitectura de software cumple estos objetivos podría entrar en el grupo de arquitecturas limpias.

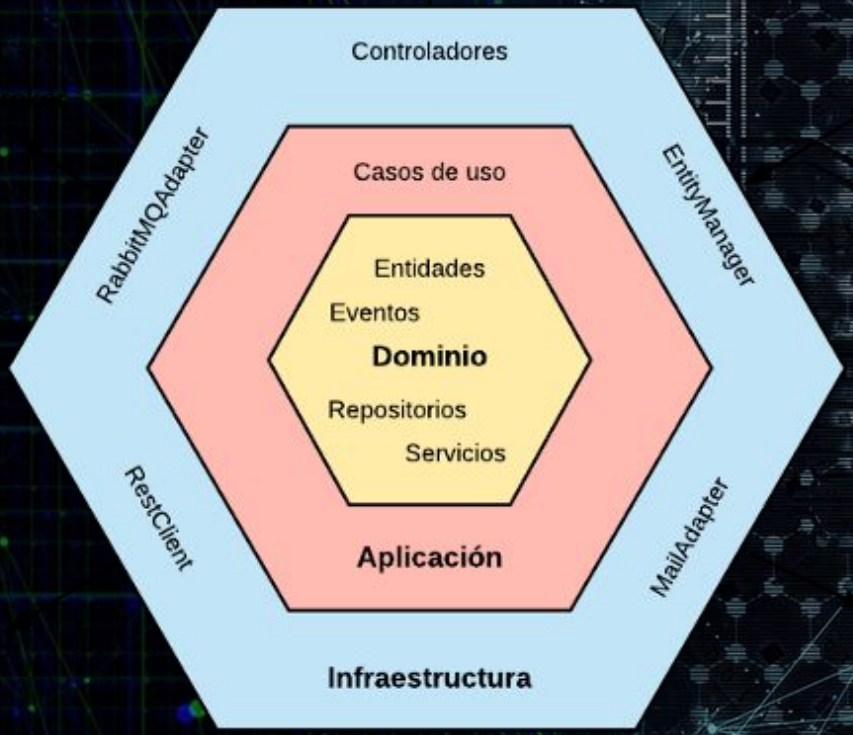
Arquitectura Limpia.

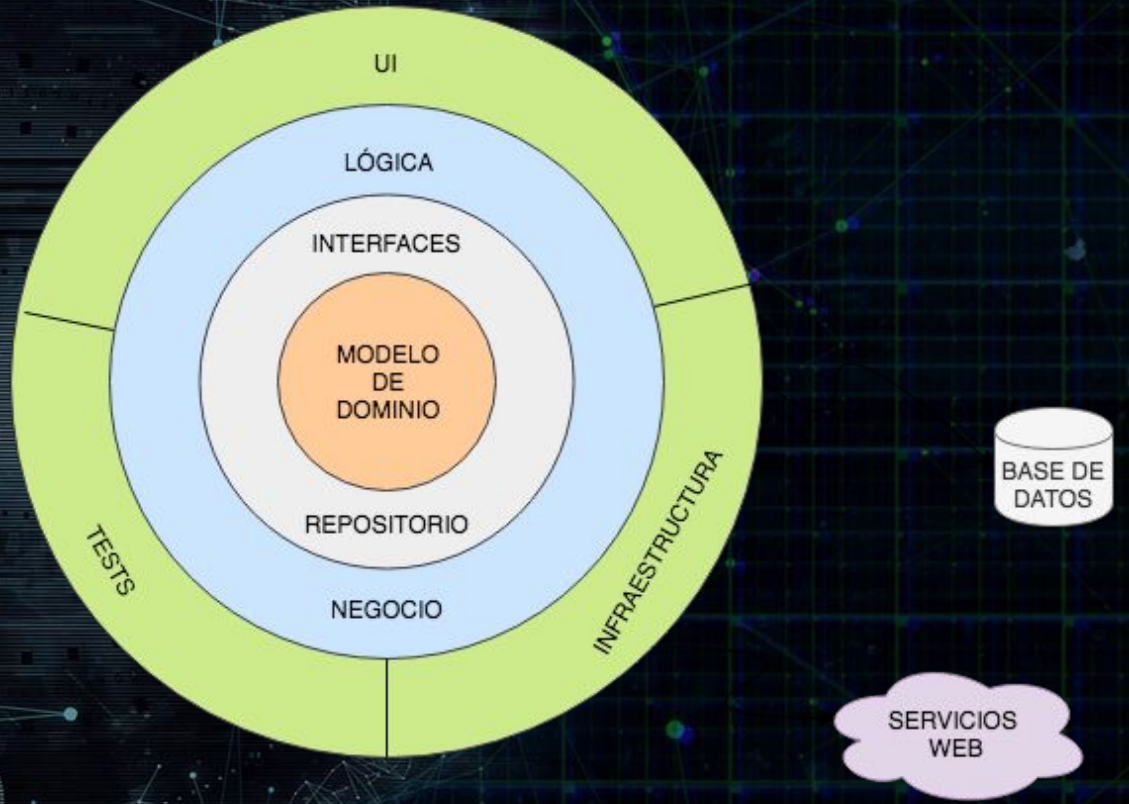
En su libro Clean Code, Robert C. Martin se atrevió a hablar de Clean Architecture. No en vano, ya había sido uno de los responsables del punto 11 del manifiesto ágil:
“The best architectures, requirements, and designs emerge from self-organizing teams.“.
Se trata de buscar la independencia en el desarrollo, entre **casos de uso**, **desacoplamiento** entre capas, o gestión de duplicidades. También nos habla de las fronteras que delimitan las partes en las que dividir la arquitectura y cómo se cruzan desde el punto de vista técnico: **comunicación de hilos**, procesos o servicios. Todo ello compartiendo **principios** ya tratados en la arquitectura hexagonal de Alistair Cockburn: entidades, controladores, puertos, adaptadores, etc.
De este tipo de arquitecturas a las que Robert C. Martin llama "Limpias" muchos aprendimos que la base de datos, como elemento externo a nuestra aplicación, no debe estar **acoplada y** por tanto no puede ser la base de nuestra aplicación. O lo que es lo mismo: no podemos empezar diseñándola en primer lugar tal como nos enseñaban en los 90. Incluso el framework, que también es algo externo a nuestra aplicación, debe ser algo con lo que evitar acoplarse.



En definitiva, una arquitectura limpia es también una arquitectura en capas. Esto significa que los diversos elementos de uso de su sistema están categorizados y tienen un lugar específico donde estar, según la categoría que les hayamos asignado.

Clean Architecture





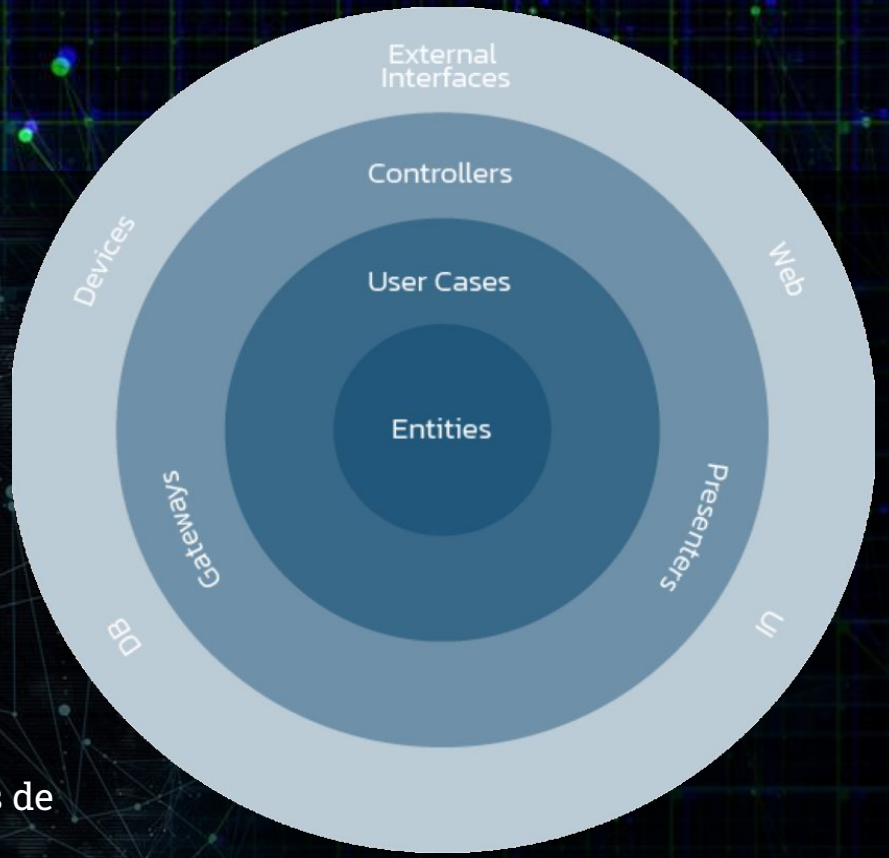
Arquitectura Hexagonal

O el ***patrón puertos y adaptadores***

Arquitectura Onion


ARQUITECTURAS DE SOFTWARE
Layers -Onion – Hexagonal – Clean +
Arquitecturas Limpias

Clean Architecture - Capas



A partir de este esquema se ve que hay **4 capas**, aunque internamente cada una podría dividirse en todas las que puede ser necesario.

En la capa interna tenemos las **entidades**, que vendría a ser el modelo de negocio, las funciones básicas o lo que sea que represente la lógica del negocio (el dominio, vaya).

En la capa inmediatamente superior están los **casos de uso**, que no es otra cosa que la lógica propia de la aplicación. Aquí también están lo que serían los puertos en una arquitectura hexagonal, que en este caso se llaman **Use Case Input Port** (si son primarios) y **Use Case Output Port** (si son secundarios), y la implementación de los de entrada está en lo que se denomina **Use Case Iteractor**.

Por encima de esto tenemos la capa de lo que serían los **adaptadores**: controladores, presentadores, acceso a terceros...

Y por encima está una **última capa** que ya no forma parte del sistema **backend** como tal y que son los dispositivos con los que nos comunicamos, la base de datos, la interfaz de usuario que nos llama, etc.

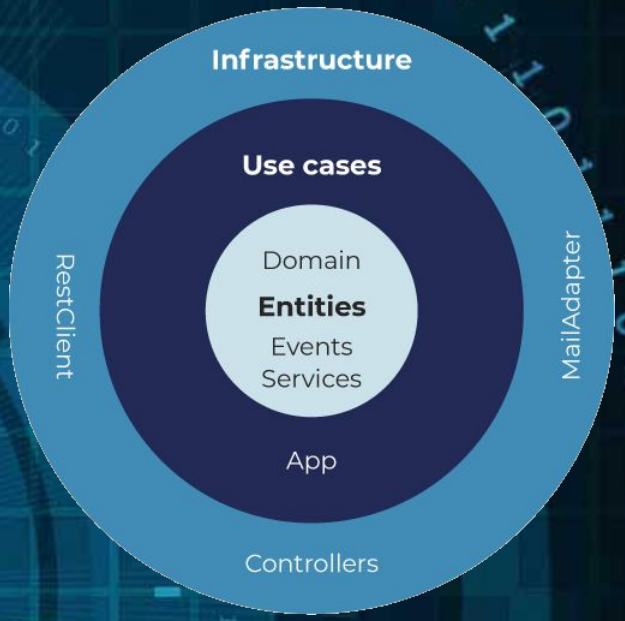
Como vemos, nada nuevo bajo el sol. Simplemente es una **arquitectura hexagonal** con otros nombres y en la que se ha definido un poco más la separación interna en, al menos, dos capas. Sin embargo, aunque no se haga de manera explícita este mismo planteamiento de separar el dominio y construir sobre él emana de forma natural de *Ports and Adapters* si queremos implementarlo como hemos comentado arriba.

Veamos en mas detalle ->

1. Enterprise Business Rules (ENTIDADES - Reglas comerciales empresariales)

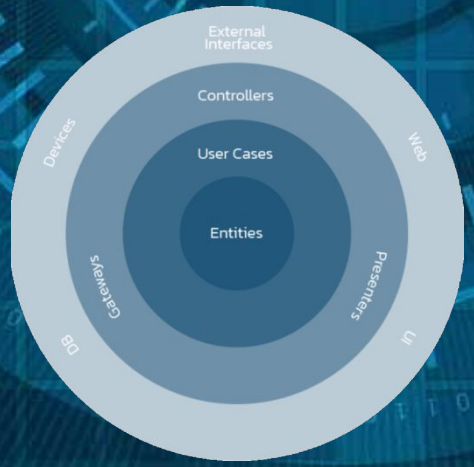
ENTIDADES

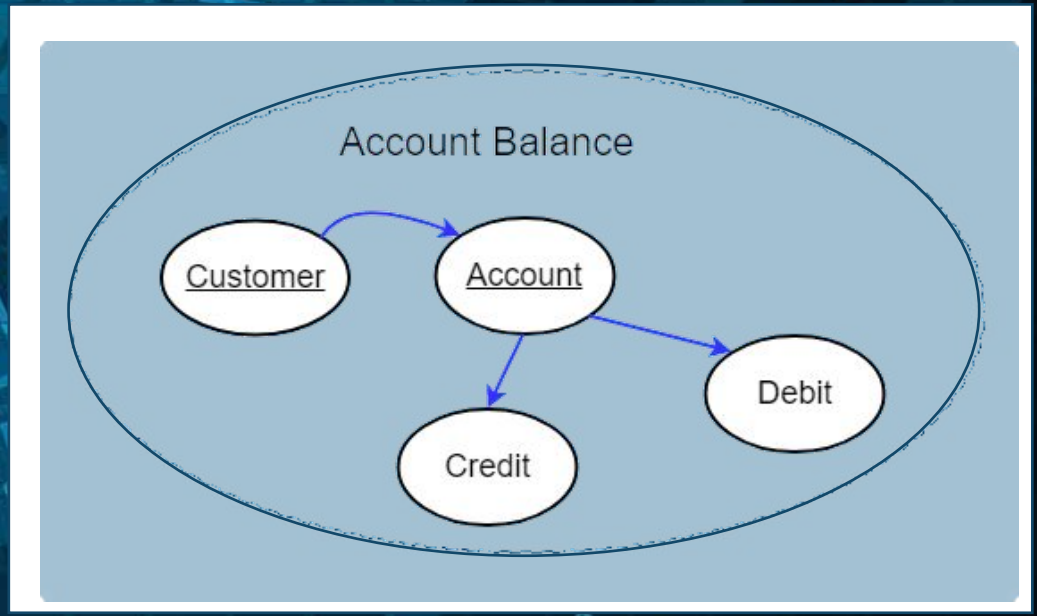
Comenzando con la capa de reglas empresariales, estamos hablando de agregados, entidades, objetos de valor y otros patrones de un dominio rico. En nuestro contexto limitado específico tenemos al cliente y la cuenta como raíces agregadas, también las transacciones de crédito / débito como entidades y, por último, pero no menos importante, tenemos el nombre, el número de persona y la cantidad como objetos de valor.



En pocas palabras, los componentes anteriores son las entidades comerciales que encapsulan campos y previenen cambios o comportamientos inesperados, estos componentes mantienen el estado de la aplicación de la manera más confiable. El artesano del software Vaugn Vernon escribió las reglas para diseñar agregados efectivos

"Se diseñan las clases para que las propiedades con conjuntos privados o conjuntos protegidos para evitar cambios de estado inesperados de varios clientes a lo largo de los casos de uso (evitamos conjuntos públicos cuando es posible)".





2. Application Business Rules (Conceptual …Casos de Uso)

Un caso de uso es una acción que un usuario o agente externo realiza en nuestro sistema. Se nombran siempre con un verbo + nombre. Si estuviéramos desarrollando una aplicación en la que se venden productos, ListarProductos sería un nombre válido así como ComprarProducto.
Una manera común de identificar la intención de nuestro sistema es modelarlo en base a sus casos de uso. Son la forma de acceder a las reglas de negocio de nuestra aplicación, por lo tanto podríamos decir que representan la parte pública del dominio de nuestra aplicación. Al ser la parte pública hacen de entrada/salida de datos al dominio.
Los casos de uso nos ayudan a expresar con un lenguaje más natural las acciones posibles que nuestro sistema puede realizar y de esa forma listan las funcionalidades del mismo. Un listado de los casos de uso ordenados por funcionalidad nos ayudará a saber de qué trata la aplicación con la que estamos trabajando. Por ejemplo, en Java, si distribuimos los casos de uso separados por paquete de funcionalidad, nos dará esa visión de dibujo arquitectónico de la que hablábamos al inicio del post.

Para hacer la acción hablarán con elementos internos de nuestro dominio tales como servicios u objetos de modelo ricos (DDD domain Driven Design) para que mediante la colaboración de los mismos resuelvan la acción.
Estos casos de uso **son pues el elemento de entrada de nuestra aplicación y controlan la secuencia de pasos de los elementos internos con los que colaboran**. Para resolver un caso de uso como `ComprarProducto` quizá en nuestro sistema los datos de entrada tengan que pasar por un objeto validador y en el caso de que los datos sean válidos guardar ese objeto en persistencia, por lo que esos dos pasos estarían correlados en el caso de uso.

Nota: Veremos UML en las Proximas Unidades

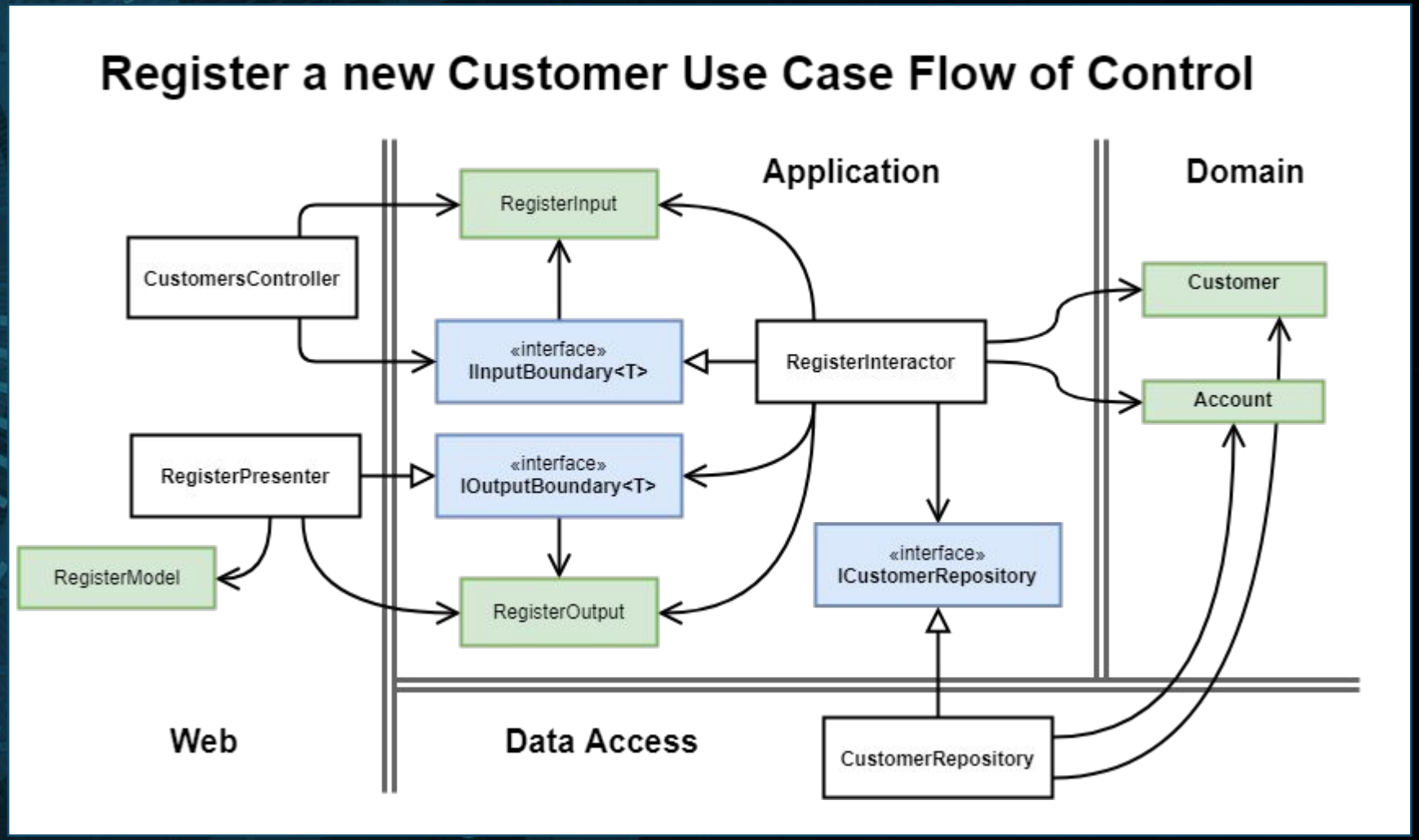
2. Application Business Rules (Casos de Uso – Aplicación de Reglas de Negocio)

Pasemos a la capa de reglas comerciales de la aplicación que contiene los casos de uso de nuestro contexto limitado. Como dijo el tío Bob en su libro Clean Architecture:

"Así como los planos para una casa o una biblioteca gritan sobre los casos de uso de esos edificios, la arquitectura de una aplicación de software debería gritar sobre los casos de uso de la aplicación."

Entonces, nuestras implementaciones de casos de uso son módulos de primera clase en la raíz de esta capa. La forma de un caso de uso es un objeto Interactor que recibe una entrada, realiza un trabajo y luego pasa la salida a través de la instancia actual del presentador como se muestra en la siguiente figura:

Un caso de uso es una acción que un usuario o agente externo realiza en nuestro sistema. Se nombran siempre con un verbo + nombre. Si estuviéramos desarrollando una aplicación en la que se venden productos, `ListarProductos` sería un nombre válido así como `ComprarProducto`.

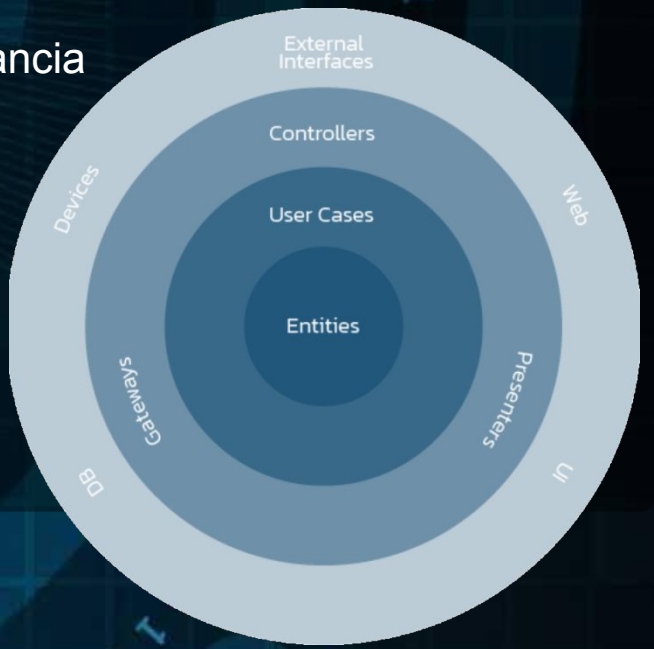


3. Interface Adapters(ENTIDADES – Controladores Gateways Presentadores.)

Ahora avanzamos a la siguiente capa, en la Capa de Adaptadores de Interfaz traducimos la entrada del Usuario de una manera que los Interactores entiendan, es una buena práctica no reutilizar entidades en esta capa porque crea acoplamiento, el front-end tiene frameworks, otras formas de crear sus estructuras de datos, presentación diferente para cada campo y reglas de validación.

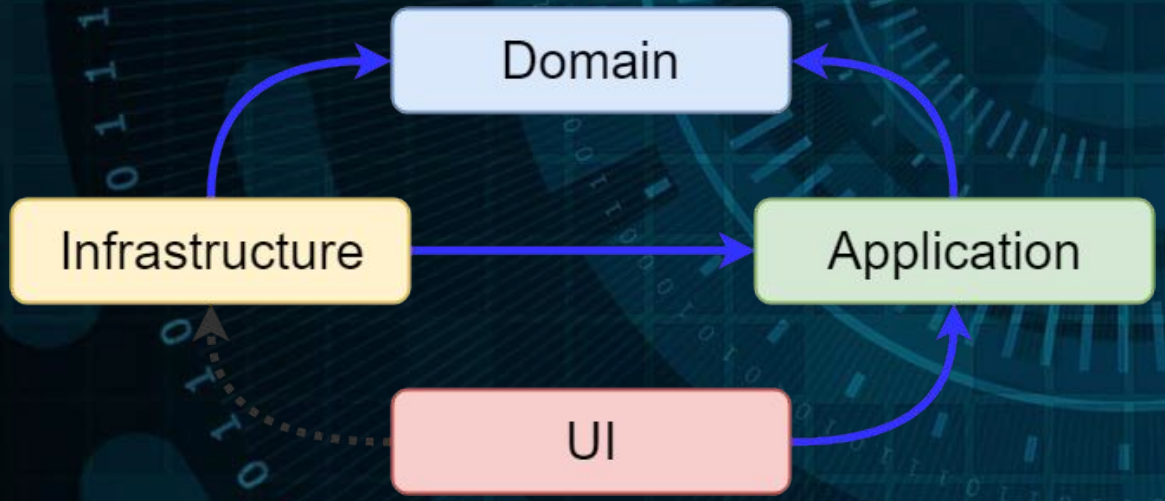
Un ejemplo de componentes para cada caso de uso:

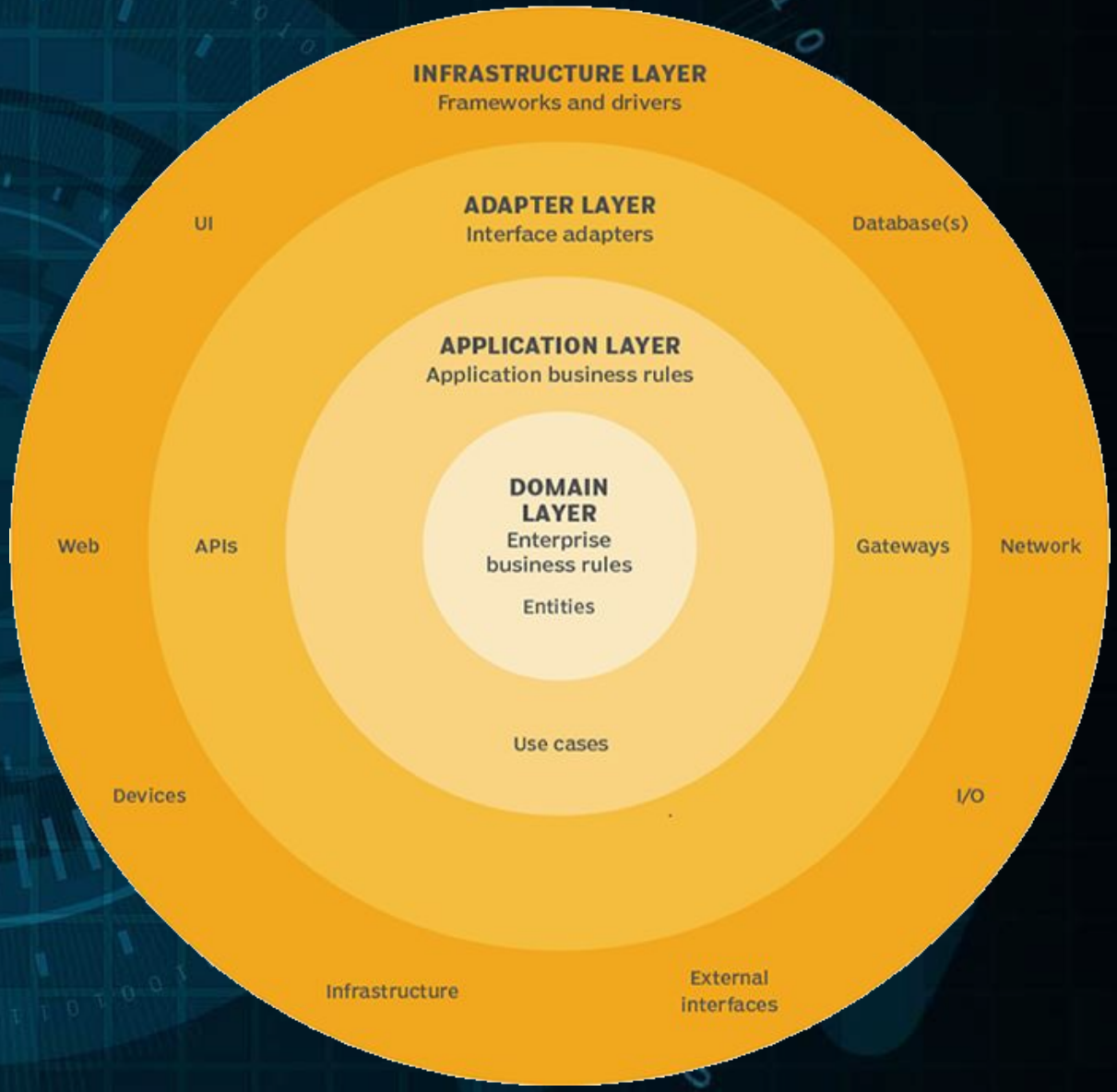
- **Solicitud:** una estructura de datos para la entrada del usuario.
- **Un controlador** con una acción: este componente recibe la entrada del usuario, llama al Interactor apropiado que realiza un procesamiento y luego pasa la salida a través de la instancia del Presentador.
- **Presentador:** convierte la Salida al Modelo.
- **Modelo:** esta es la estructura de datos de retorno para aplicaciones MVC.

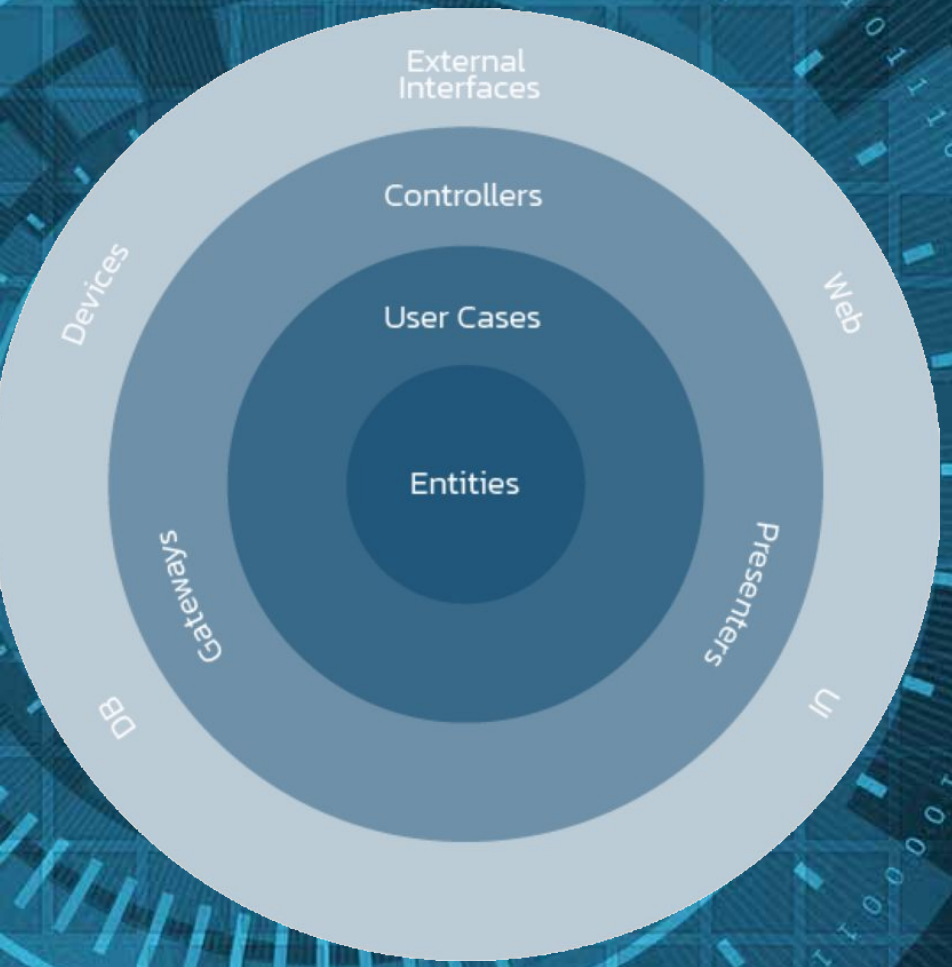


4. Framework y Drivers

Nuestra capa más externa son los Frameworks & Drivers que implementan el acceso a la base de datos, el marco de inyección de dependencia (DI), el serializador JSON y cosas específicas de tecnología.



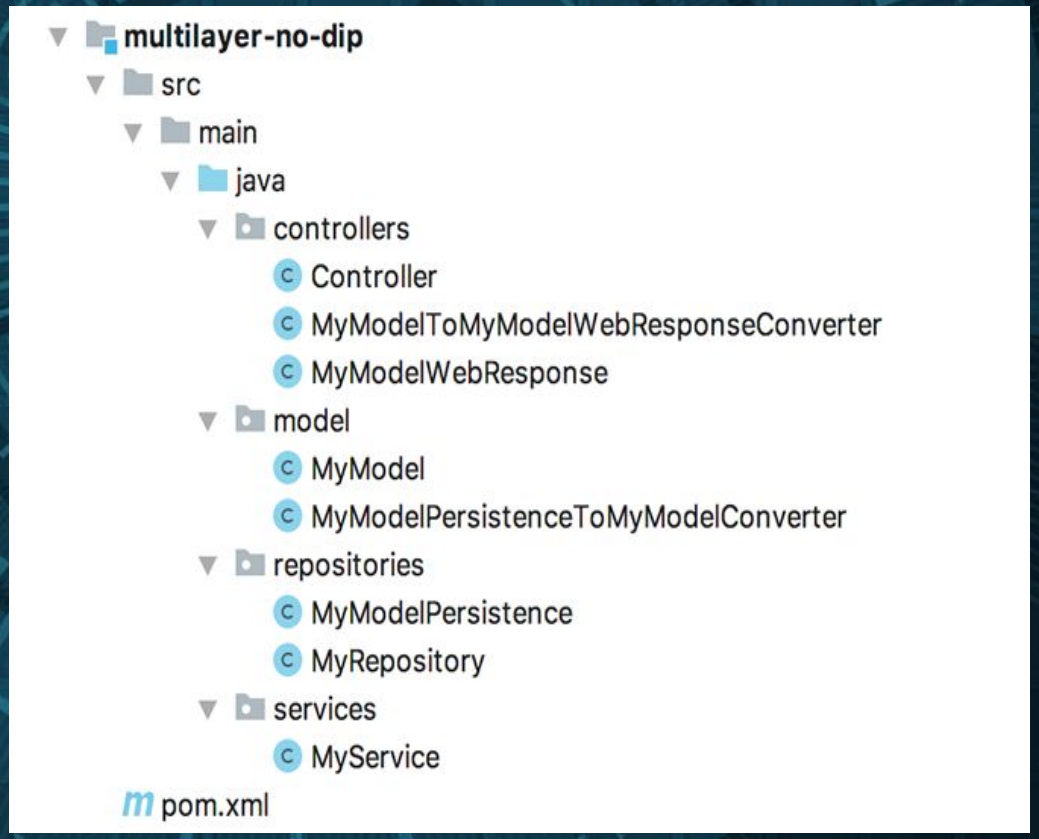




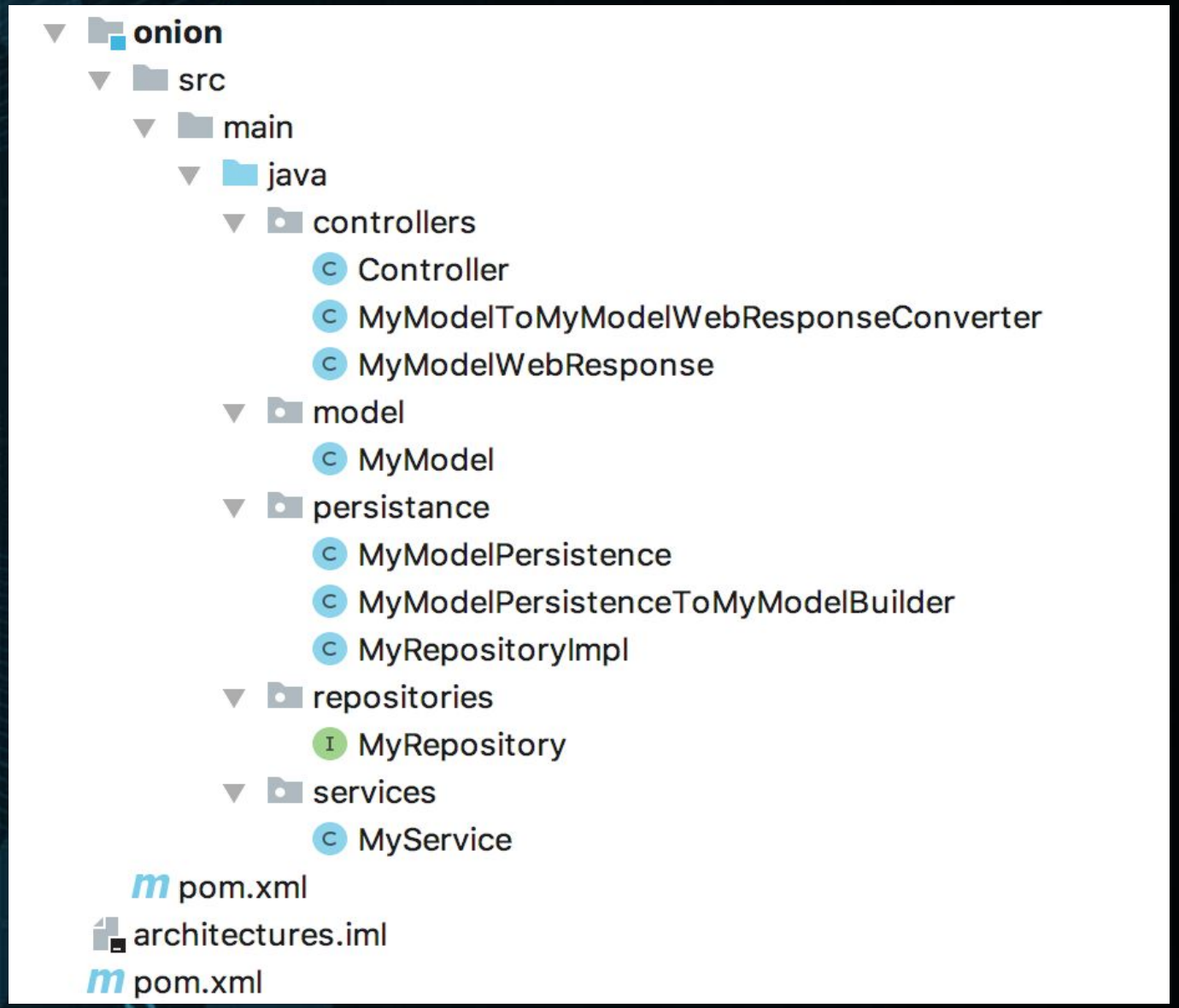
Como se ve todo esto en Código?

¿Y qué pinta tiene todo esto en Código?

Se muestran a continuación una serie de **esqueletos** muy muy básicos de las arquitecturas que hemos comentado.
Son proyectos **Java** creados con *Maven*, pero no deberíais tener ningún tipo de dificultad para entenderlos puestos que lo que nos interesa es la estructura.
Por supuesto en un sistema real más grande es posible que adopten otra forma y tengan más paqueterías o niveles. O que se esté siguiendo DDD (*Domain Driven Development*) para construirlo y sea dentro de cada dominio donde se pueda observar el tipo de arquitectura que sigue.



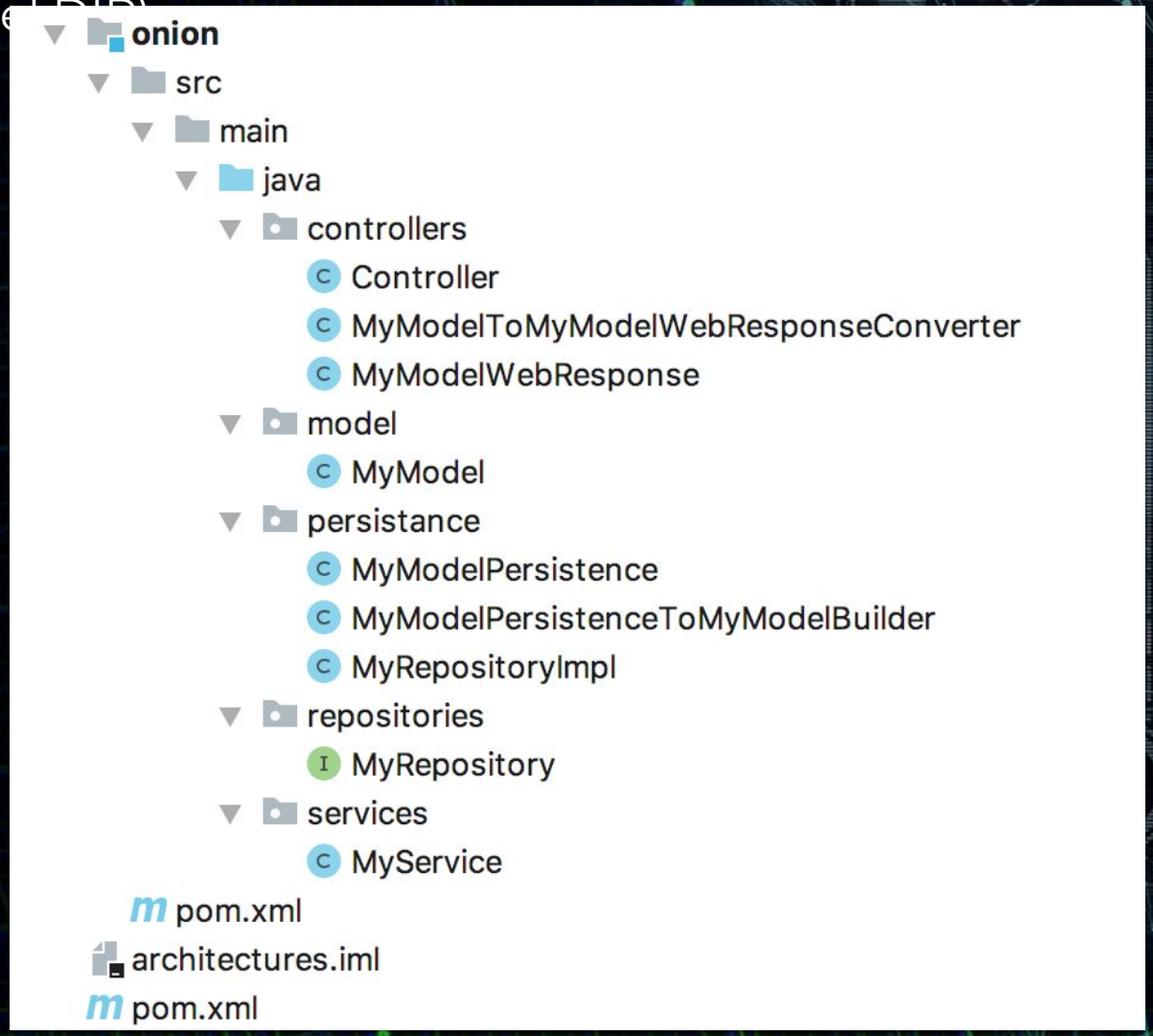
Arquitectura multicapa sin seguir el DIP



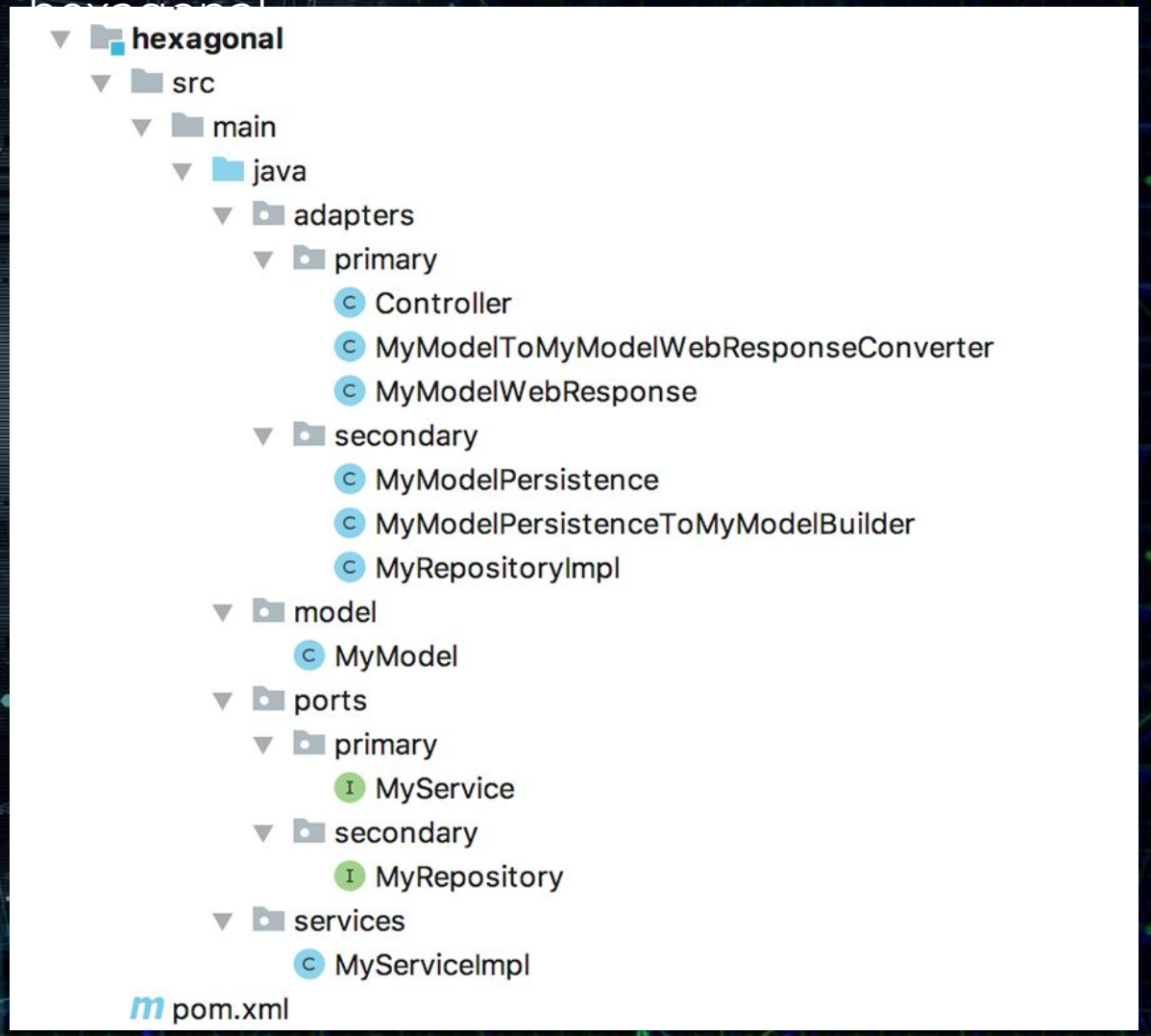
Onion (arquitectura multicapa que sigue el DIP)

Independiente de agentes externos

Onion (arquitectura multicapa que sigue el DIP)



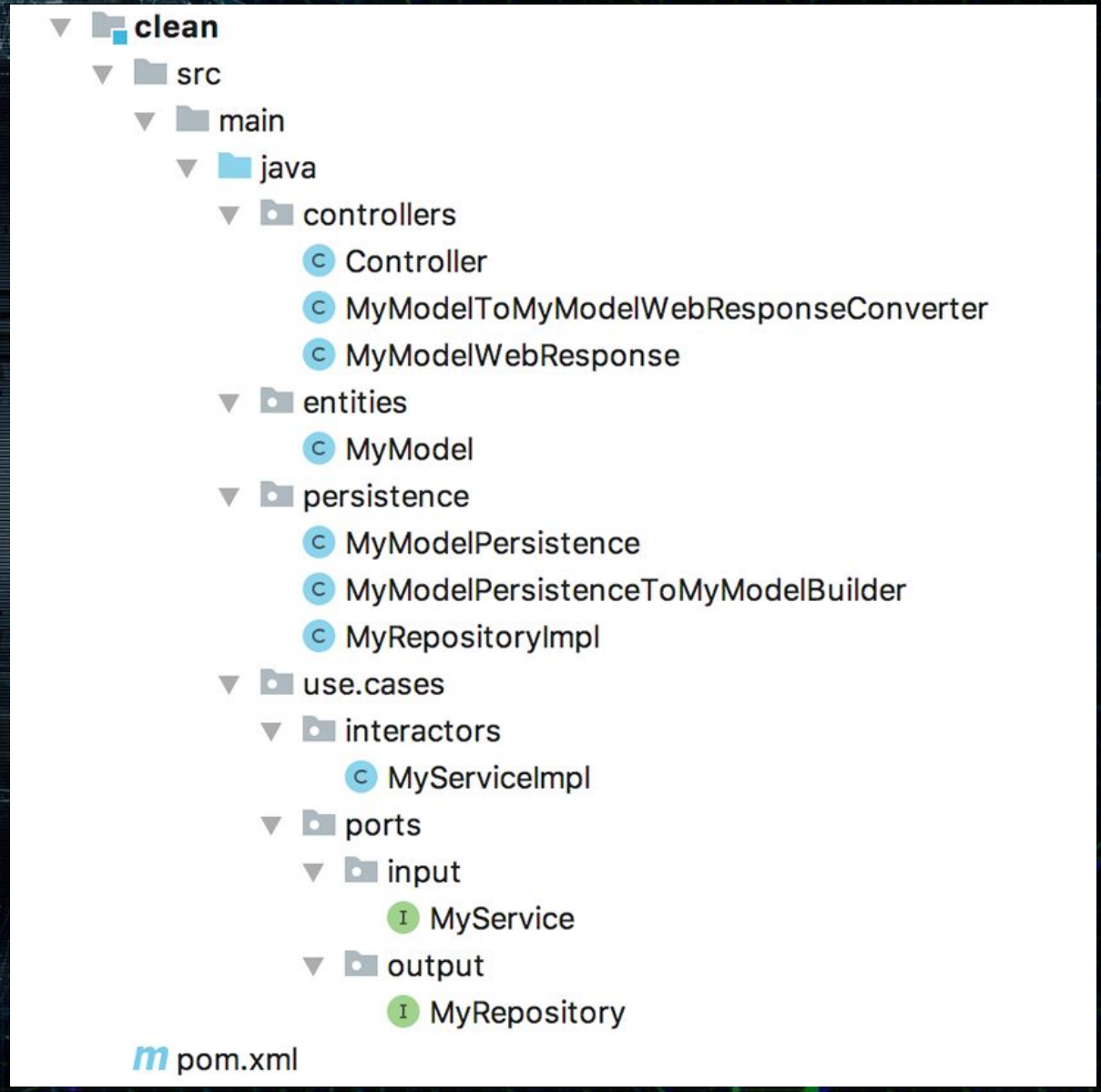
Arquitectura hexagonal



Independiente de agentes externos

Clean
architecture

El primer ejemplo es un caso aparte porque no está siguiendo el mismo principio que las demás. Pero dejando eso aparte , podemosver que realmente **no son tan distintas** unas arquitecturas de otras. En las tres la conversión
entre *MyModel* y *MyModelPersistence* se hace en el propio repositorio para que no afecte al resto de la aplicación. Y lo mismo ocurre en el controlador al transformar a *MyModelWebResponse*. Además en los tres también estamos programando contra interfaces y luego inyectando la implementación (DIP). Lo único que cambia realmente son los nombres de la paquetería y, ligeramente, dónde está cada uno.



Resumen y Conclusiones Preliminares.

Noten que Solo hemos hablado sobre cómo separar nuestro software por **capas** que el desarrollo sea más organizado, sencillo y, principalmente, mantenible. Inicialmente sin preocuparnos excesivamente de las **dependencias**. Esto tiene perfecto sentido para **aplicaciones pequeñas** que implementan un **CRUD** y poco más. Si al final todo lo que haces es consumir un dato, mostrarlo por pantalla y encima el sistema no tiene un volumen inmenso, ¿para qué vamos a complicarnos la vida montando algo que no nos va a aportar demasiado pero que puede ser más difícil de entender?

Cada **herramienta** tiene su uso, y estas arquitecturas no son una excepción.

Posteriormente hemos dado varias **alternativas** para además construir toda la aplicación centrada sobre nuestro dominio, lo cual tiene bastante más sentido y es conveniente si estamos en un desarrollo medianamente grande. Estos, desde hace tiempo se han empezado a convertir en un estándar. Lo que es importante, y a mi me interesa, es que entendán el **motivo** de lo que están haciendo y las **ventajas** que aporta. Noten que todas ellas esencialmente son lo mismo pero con distinto nombre. En cualquier caso, particularmente muchos prefieren la arquitectura hexagonal. Dado que aporta un plus de **semántica** a toda esta nube de capas y ayuda a que en la arquitectura y la paquetería queden muy remarcadas las fronteras de nuestro sistema con el exterior. Y además los nombres son mas representativos.

*La **arquitectura** es cómo vuestro sistema se compone de partes más pequeñas que se relacionan entre sí y dependen unas de otras. Así que cuando alguien les pregunte cuál es la de vuestra aplicación no les mencionen el Lenguahe, el framework, la base de datos y demás, al menos hasta la segunda frase….*

*Recordar que **ERES** un **"Diseñador y Desarrollador de Sotfware"** y no un Implementador, que lo que "hagas", esta pensado para Existir, Evolucionar, Modificarse, Ampliarse, que sea Robusto…., y con Calidad….*

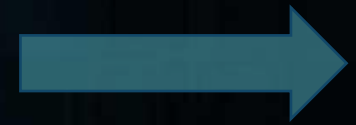
Bienvenidos a:

ALGORITMOS Y ESTRUCTURAS DE DATOS II

Un Camino hacia un Profesional Responsable,
Con Técnicas y Métodos Eficientes, con Capacidad de Resolución de Problemas con Métodos de Ingeniería de Software, Agil Dinámico y Adaptativo en la Construcción de Software Industrial de Calidad.

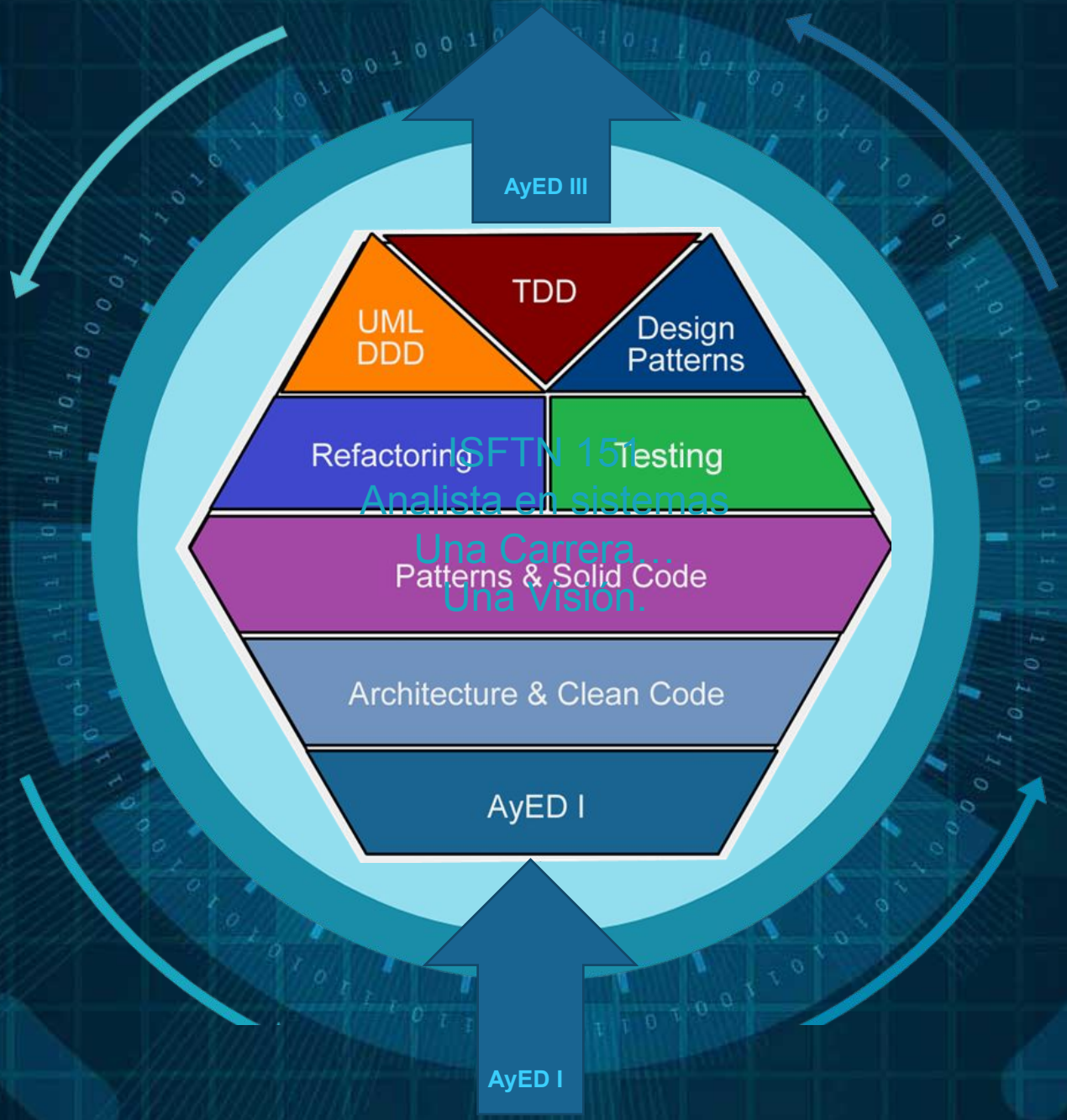
Veamos el Camino Proyectado…

2024.v1



ALGORITMOS Y ESTRUCTURAS DE DATOS II

Diseño de Software, UML, Patrones
Arquitectura y Código Limpio



ALGORITMOS Y ESTRUCTURAS DE DATOS II

Diseño de Software, UML, Patrones , Arquitectura y Código Limpio

Unidad 0 Inducción a las Prácticas Eficientes de Diseño Arquitectura Limpia

Hacia una Evolución Personal y Profesional sobre como constriur Codigo Límpio

Unidad 1 Paradigmas de Programacion

Un Abordaje a los Distintos Paradigmas, pero con eje en la solución de Problemas.

Unidad 2 Python, POO & Frameworks

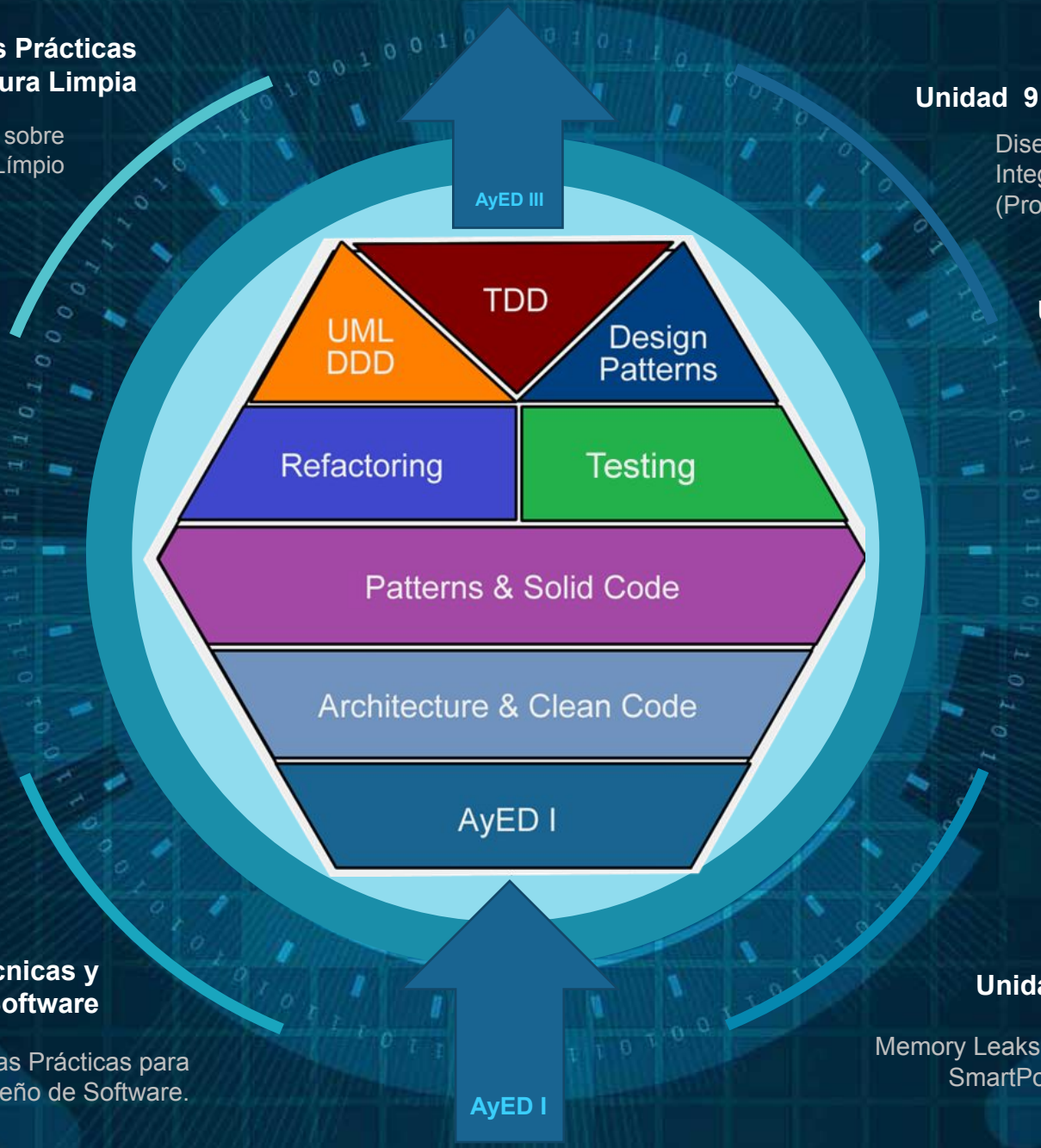
Introduccion al Lenguaje, Funcionalidades, POO, Uso de Frameworks, Interfaces Gráficas con Tkinter, Machine Learning, Simple IA

Unidad 3 Patrones Software & POO Avanzada

Conceptos Avanzados de POO, Patrones de Software Estructurales, Creacionales y Compotamiento.

Unidad 4 S.O.L.I.D. & GRSP, Técnicas y Buenas Prácticas de Diseño Software

Principios SOLID y GRASP. Buenas Prácticas para el Diseño de Software.



Unidad 9 Trabajo Final Integrador

Diseño y Desarrollo de un Trabajo Final que Integre lo visto, No solo en esta Materia. (Proyecto Intercatedras 2022)

Unidad 8 Metodologia TDD

Metodología de Desarrollo basado en Pruebas, ciclo de Desarrollo Agil TDD.

Unidad 7 Implementing Testing

Introducción al concepto de Testing, Implementacion de plan de Testing.

Unidad 6 Refactoring de Software

Aplicación de Tecnicas de Refactorin aplicadas al diseño de un Código Limpio.

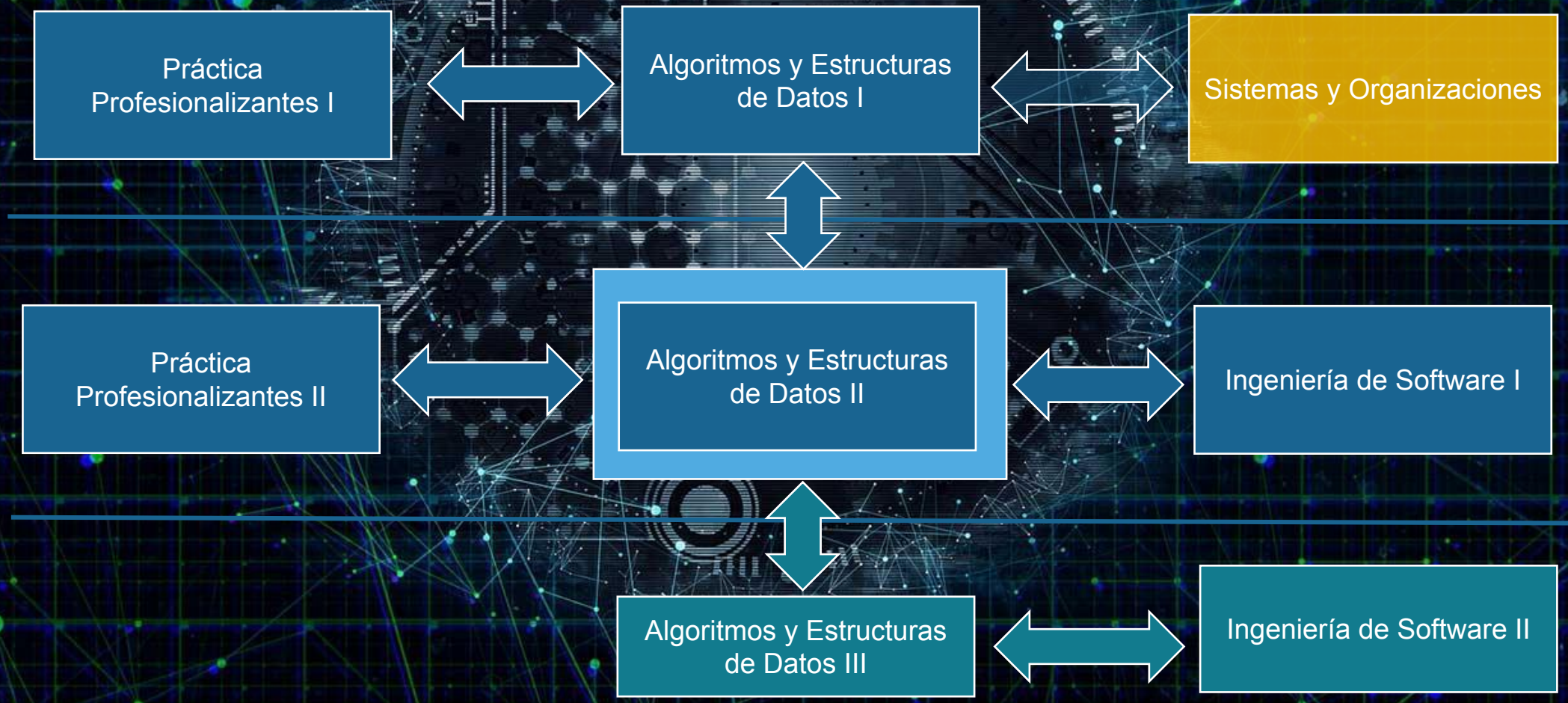
Unidad 5 Memory Leaks & RAII

Memory Leaks, RAII, Free Function, Exceptions, PTR SmartPointers, STL Standard Template Library.

.

ISFTN 151 – Analista de Sistemas - Proyecto Integración InterCátedras 2022

Los Espacios Intercátedras en el Instituto Superior de Formación Técnica Nº 151,
son instancias integradoras de áreas del conocimiento y Practicas Comunes que reúnen a docentes, alumnos, staff y no docentes con el objeto de promover, coordinar y compatibilizar proyectos y actividades Productivo-Académicas, que contribuyan al logro de los objetivos institucionales Impartidos, tendientes a la formación Personal y Profesional de Calidad.



GRACIAS

Unidad 0 – Introducción.

Algoritmos y Estructuras de Datos II

ISFTN151 – Analista en Sistemas