Final 1

Ejercicio 1

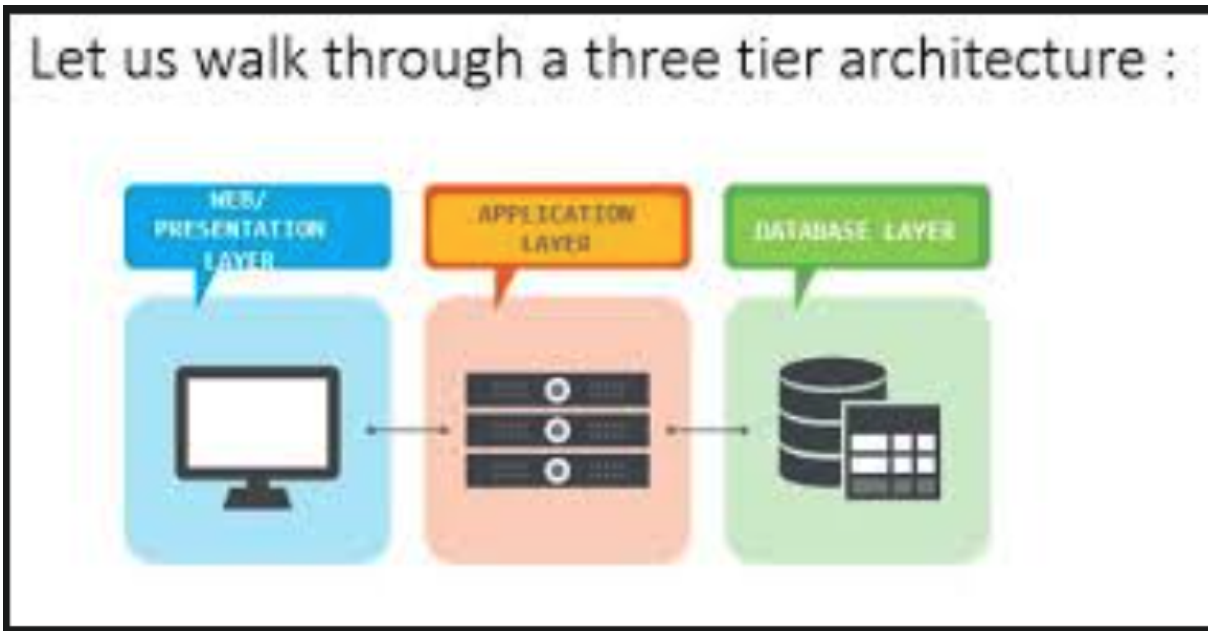
a) ¿Podría el patrón de arquitectura Model View Controller ser considerado un patrón de arquitectura 3TIER? ¿Podría el patrón de arquitectura Model View Controller ser considerado un patrón de arquitectura 3Layer? Si, no, ¿por qué?

~~Cuando se habla de *tier* se hace referencia a una separación física de los componentes, mientras que cuando se habla de *layer* se refiere a una separación lógica, es decir, la estructura interna del código divide el mismo en áreas de funcionalidad (típicamente presentación, reglas de negocio y acceso de datos). Por supuesto, nada impide, y a menudo así se hace, que cada layer se guarde en un tier, es decir, que se realice una separación física que coincida con la lógica, pero también podría por ejemplo, compilarse los tres layers en un único tier que fuese un monolito por ejemplo.~~

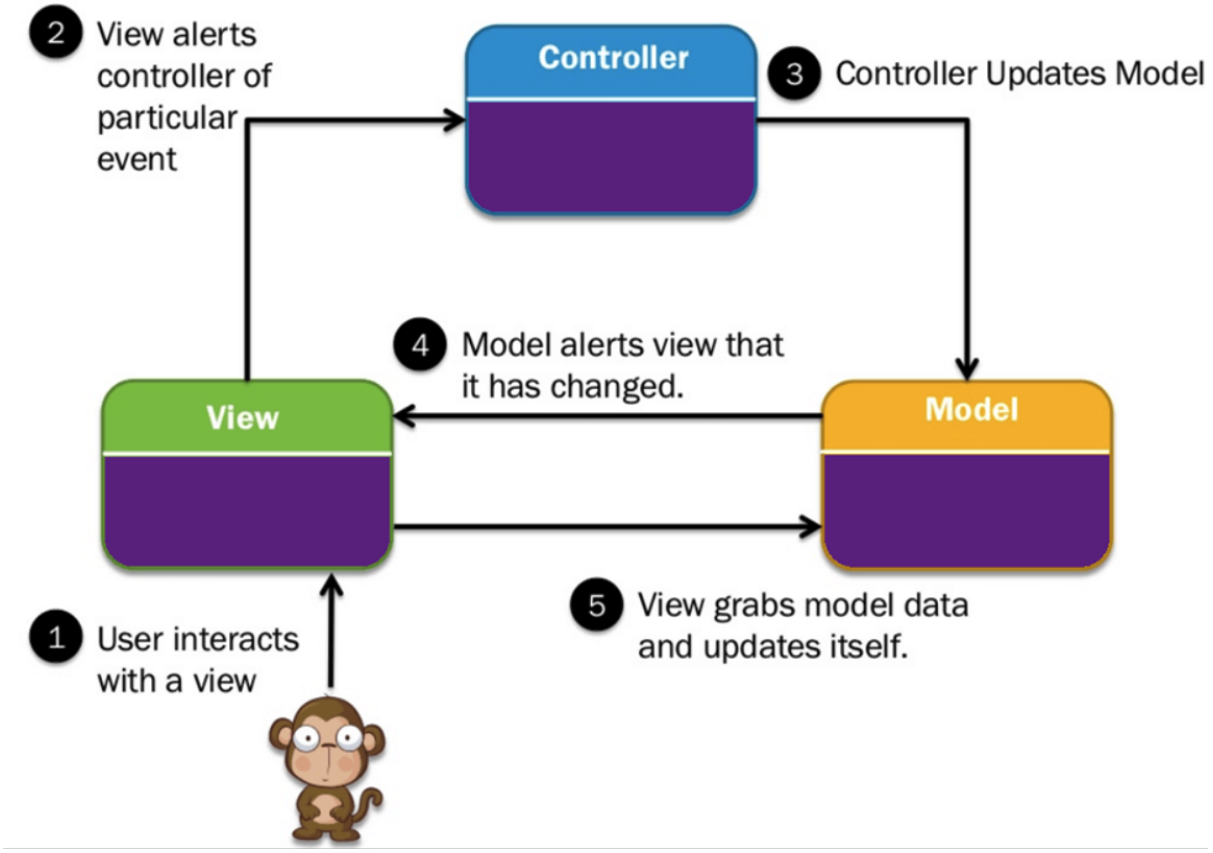
~~MVC no tiene nada que ver con lo anterior. A grandes rasgos, se trata de un patrón que permite estructurar internamente la capa de presentación, de forma que se separa la interfaz de usuario (view) del modelo con la lógica de negocio (model) y la lógica que sirve para interactuar con el modelo (controller).~~

En un principio podría parecer similar el patrón 3 tier con el concepto de MVC, pero topológicamente estos son distintos. Una regla fundamental en la arquitectura tier3 es que el client tier nunca se comunica directamente con el data tier, en la arquitectura tier3 toda comunicación debe pasar por el middle tier. Conceptualmente sería una arquitectura lineal. En cambio MVC es una arquitectura triangular, la vista envía updates al controller, el controller actualiza el modelo y la vista se actualiza con el modelo.

3 tier:



MVC



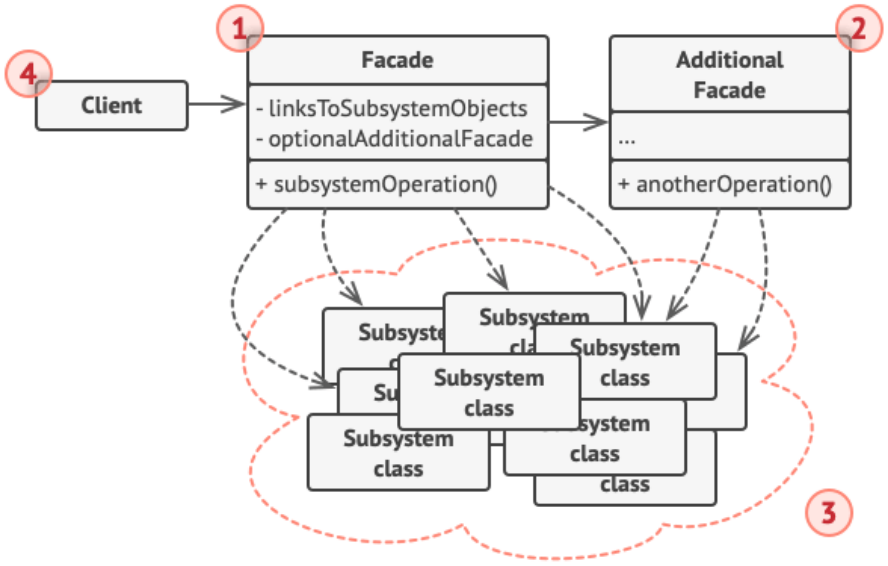
No hay capas lógicas o jerarquías (tiers) en MVC.

b) ¿Cuál de los componentes del patrón MVC es más reusable? ¿por qué?

El componente más reusable es el **Modelo** ya que sólo contiene lógica de negocio, por lo tanto si se quiere sacar ese componente y utilizarlo con otro controllers y vistas que respeten las firmas de las interfaces este podrá reutilizarse sin problema, por ejemplo uno podría ser para la versión web y otro para la versión desktop.

c) El patrón de diseño de la Facade encapsula las clases que definen las interfaces que se simplifica por su uso. Indique si es Verdadero o Falso.

Verdadero. Facade brinda una interfaz simple que encapsula un conjunto de entidades o un subsistema complejo, a través de este encapsulamiento simplifica el uso / acceso hacia el lado cliente. Por ejemplo un Convertidor de video, por detrás debe ser complejo ya que debe manejar la compresión, el audio, el codec, etc. Pero al cliente sólo le interesa la función de convertir un video, en este caso todo el subsistema se encapsula mediante facade y la interfaz solo expone el método que quiere utilizar el cliente.



d) El patrón de diseño Facade introduce nuevas interfaces. Indique si es Verdadero o Falso.

Verdadero. Ya que de esta forma permite cumplir con el principio SOLID de interface segregation, evitando tener firmas innecesarias en la interfaz y utilizando las que tengan sentido, incluso se podrían utilizar múltiples facades para distintas partes del sistema.

e) Cuando se utiliza el patrón de diseño Facade, las clases que implementan a éste son conocidas por el subsistema al cual controla el acceso. Indique si es Verdadero o Falso.

Falso. La idea del patrón es aislar al cliente del subsistema y que ellos se comuniquen a través de la fachada justamente, desconociendo completamente cómo esté implementado por detrás.

f) ¿Qué patrón de diseño es utilizado conjuntamente con el patrón facade en un gran porcentaje de los diseños de software? ¿Por qué? ¿Qué función cumple?

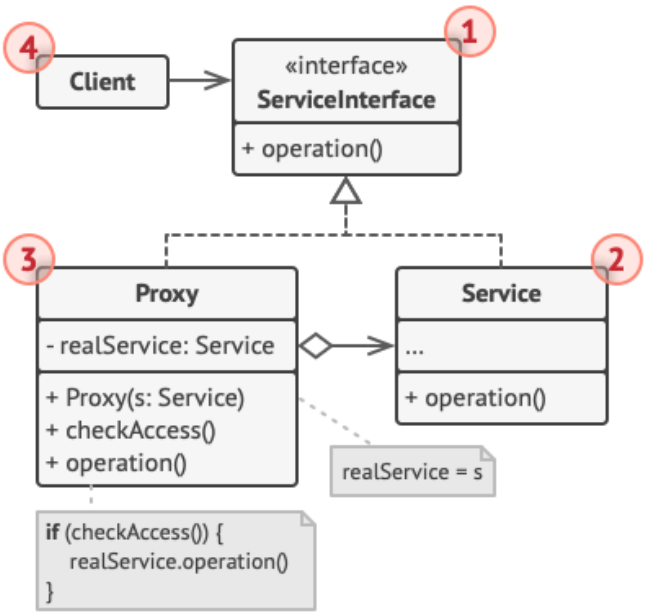
Se suele usar el patrón de **Singleton** en conjunto con Facade dado que, en general, solo se necesita una instancia de este último objeto y el patrón Singleton permite asegurar esto. De esta forma, resulta mucho más sencillo llamar a la instancia de la clase Facade

Respuesta basada en los resueltos:

Este patrón se utiliza en conjunto con el patrón de arquitectura Layer, siendo el Facade la puerta de entrada / interfase de una capa a otra ya que el facade encapsula todo el subsistema y simplemente expone una interfaz para su uso.

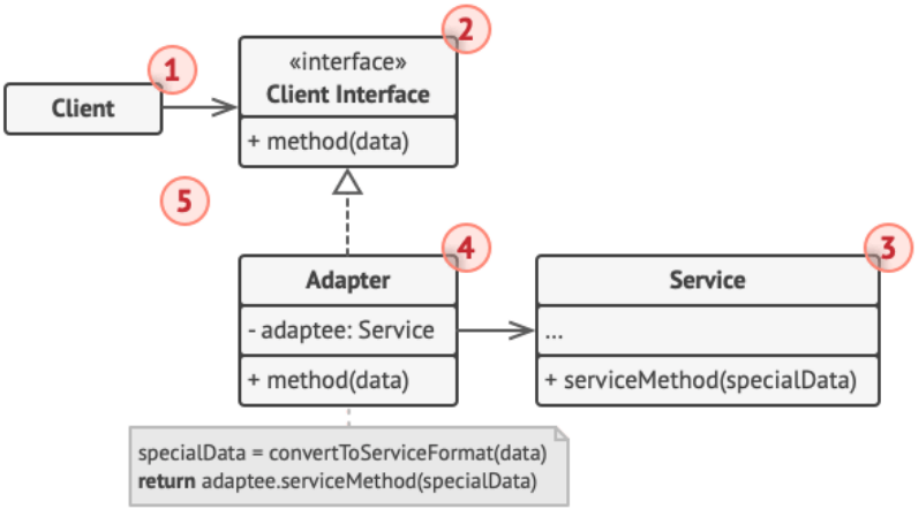
g) Para controlar el acceso a un subsistema cual de los patrones de diseño utilizaría, marque solo con una cruz:
facade adapter mediador proxy

Proxy, ya que el proxy expone una clase con los mismos métodos que la clase a la que está encapsulando (a vistas del cliente sería la clase original) y el proxy antes de delegar las llamadas a la clase original puede implementar su lógica de control de acceso, por ejemplo esto podría ocurrir para un rate limiter a algún servicio externo que no soporte más de un umbral de accesos.

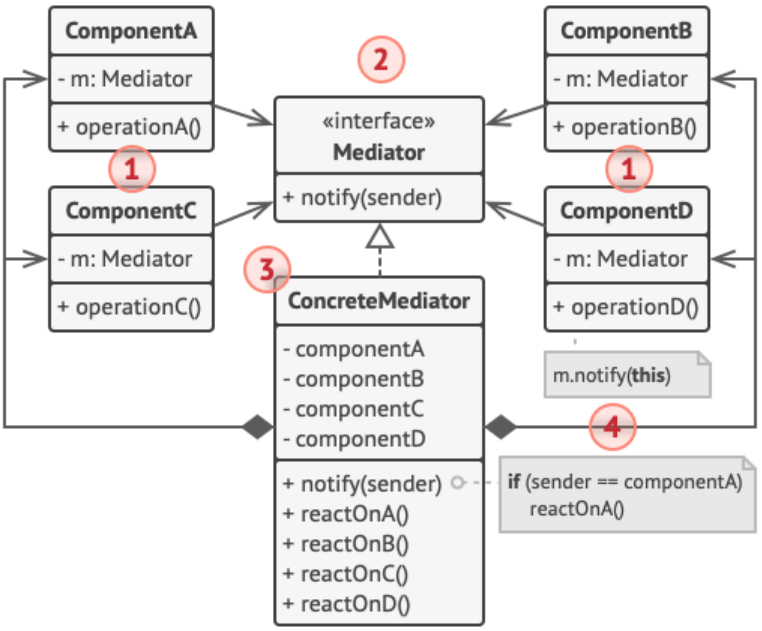


A modo informativo pongo los otros patrones:

- **Adapter**: La idea es que el cliente necesita comunicarse con el servicio y estos “hablan idiomas distintos”, ejemplo el cliente maneja data XMl y el servicio en JSON, por lo que el adapter entra en acción, hace literalmente de adaptador entre el cliente y el servicio, le expone al cliente el idioma que habla el e internamente realiza la lógica para poder comunicarse con el servicio



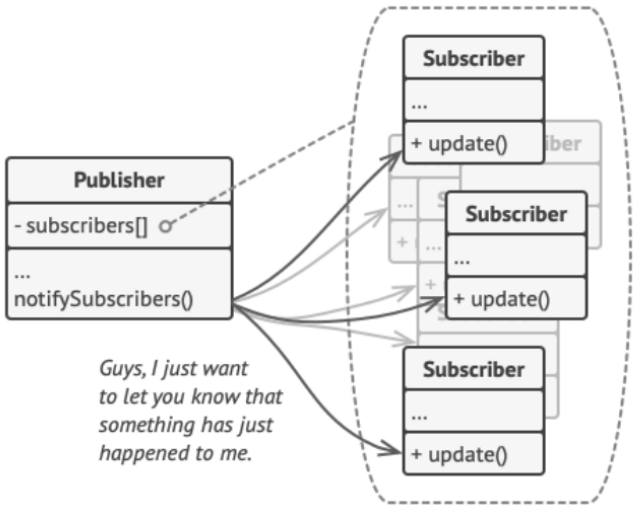
- **Mediator**: La idea es que cuando hay muchas dependencias complejas, simplificar estas poniendo una clase mediadora y que todas las clases hablen con el mediador. Se puede pensar como la analogía de que en un aeropuerto los aviones hablan con una torre de control y no con otros aviones.

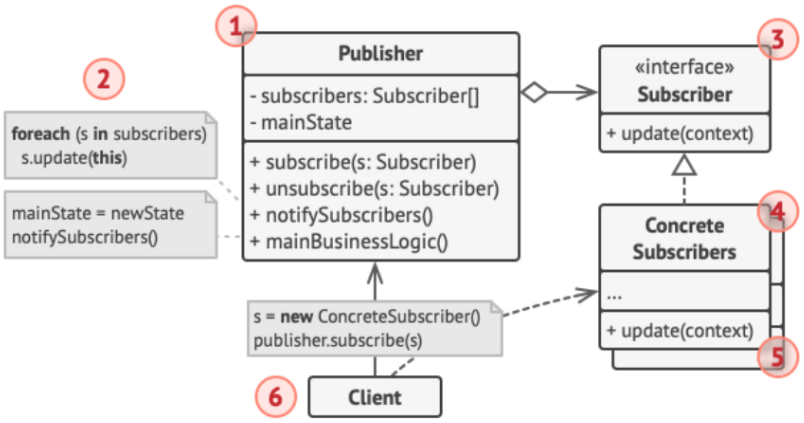


h) ¿Por qué el método de actualización de la interfaz de Observadores incluye una referencia al objeto observado? ¿Tiene el observer saber qué objeto se está observando?

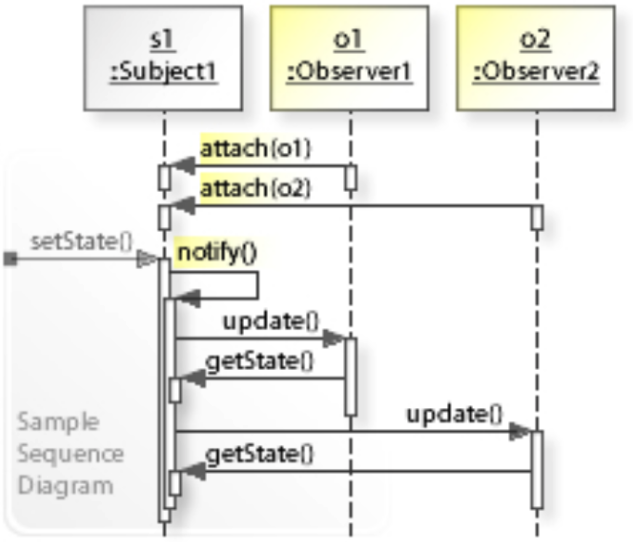
i)
```
interface Observer {
        void update (Observable model, Object object);
}
```

Observer: Es un patrón de publisher - subscriber, básicamente los subscribers quieren enterarse de novedades de algo, entonces se suscriben a ese algo y cuando hay alguna novedad el publisher se encarga de notificar a los subscribers.





OJO: Acá en el diagrama update() recibe un contexto desde el publisher, el modelo que plantea la pregunta no recibe ningún arg y en lugar de eso los subscribers se encargan de hacer un get del estado del "publisher" o más bien subject (subject es lo que el observer observa).



Se incluye la referencia al objeto observado para que cuando el Sujeto avise que hay una novedad el observer pueda efectivamente consultar por el nuevo estado del objeto observado por ejemplo con un getState o consultando la información específica que necesite.

Pregunta 2

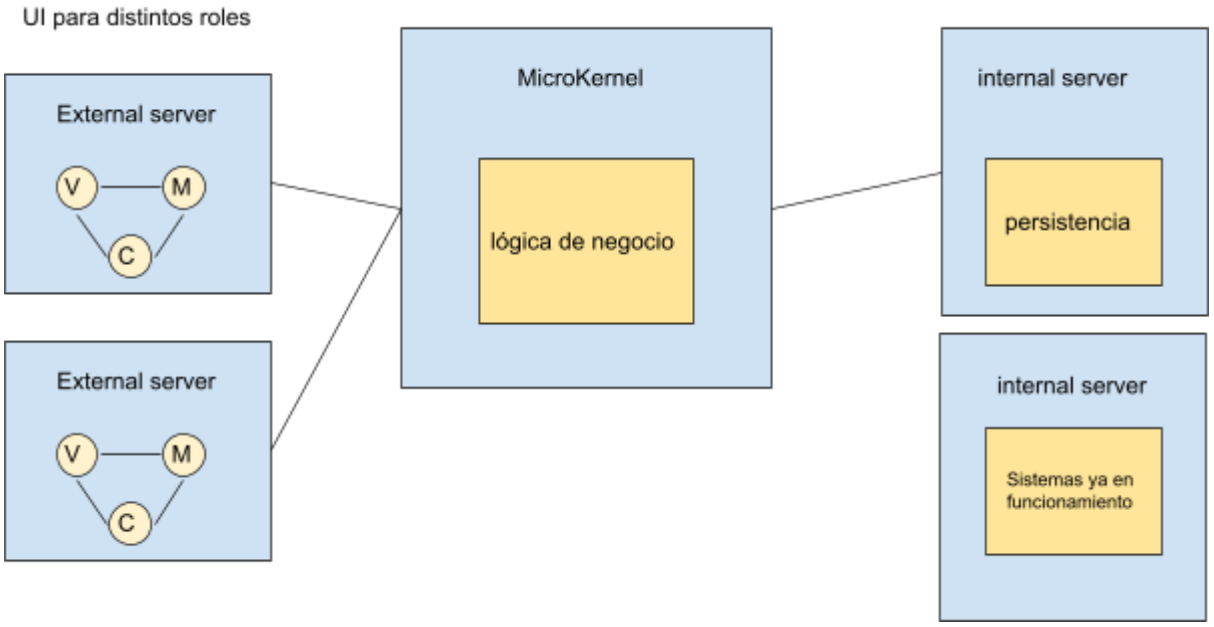
Usted es consultado para definir la arquitectura de un producto de software que debe entre otras cosas interactuar con sistemas ya en funcionamiento. Estos sistemas externos fueron definidos y pertenecen a un grupo de tecnologías determinadas.

Además y por necesidades de negocio se ha decidido lanzarlo al mercado con los requerimientos con que se cuenta a la fecha. Sin embargo, se estima que deberán generar una segunda versión el próximo año con funcionalidades específicas para nuevos roles. Estos deberán se incorporados ya que por falta de tiempo y conocimiento de este aspecto del negocio no estarán incluídas en esta primera versión.

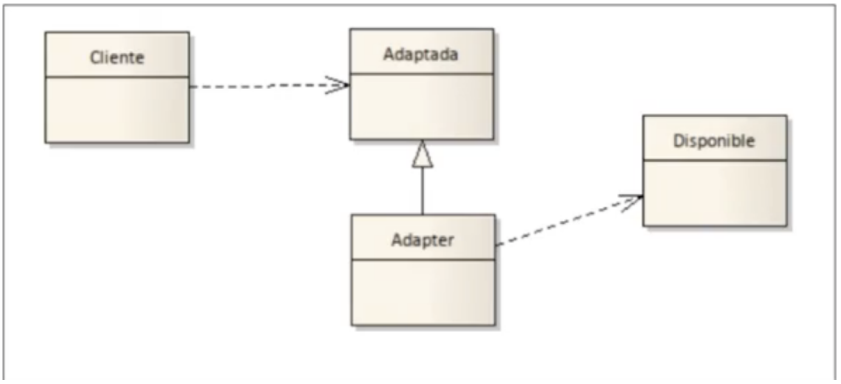
Qué criterios/patrones de diseño propondría utilizar y que esperaría lograr con ello. Incluya un diagrama simple de paquetes o clases.

- Hipótesis:
  - Se trabaja con Bases de Datos
  - Cada rol trabaja con distintas interfaces
- Arquitectura:
  - **Microkernel**: nuestra aplicación deberá interactuar con sistemas externos, por lo que esta arquitectura se ajusta a la situación siendo los sistemas externos y las bases de datos el internal server y las interfaces de usuarios para los múltiples roles los external server. Además, esta arquitectura nos va

a permitir la funcionalidad necesaria para lanzar una nueva versión con más funcionalidades ya que si queremos complejizar el modelo de negocio se podría usar una arquitectura layer para simplificar el modelo de negocio.
    - **MVC:** nos permite definir distintas interfaces para los distintos roles.

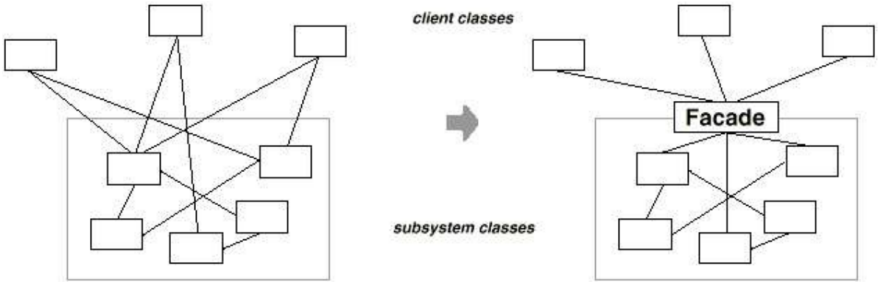


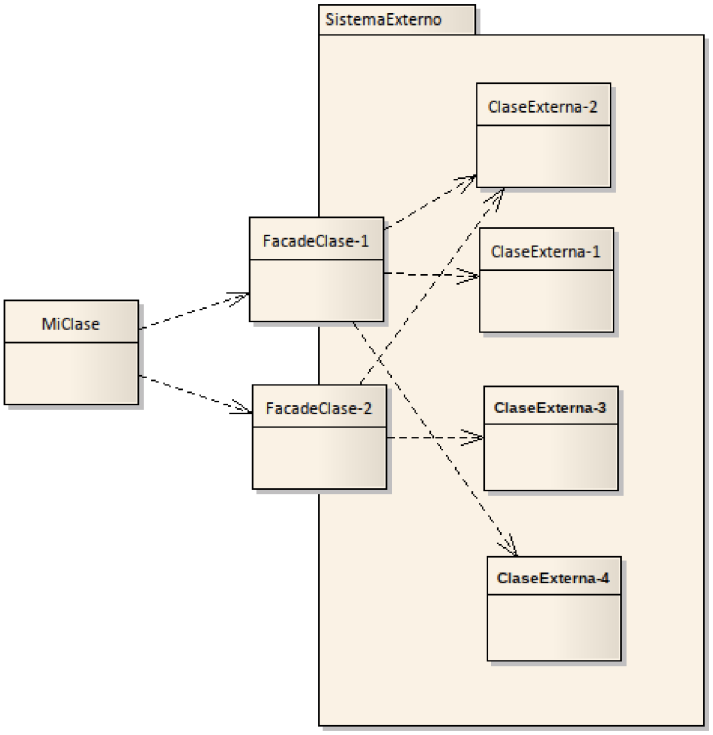
- Patrones de diseño posibles:
    - **Adapter:** para que el producto a diseñar pueda interactuar con los sistemas ya definidos.



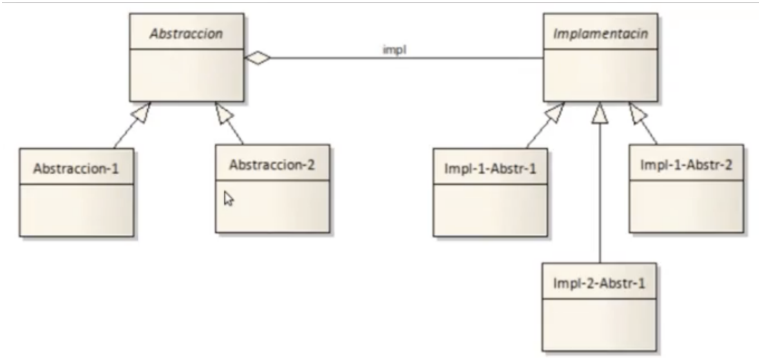
    - **Facade** para evitar tener métodos que no se utilicen, de forma tal que si los sistemas son muy complejos habrá una interfaz simple que contenga solo los métodos a utilizar y así, abstraernos de la complejidad de los sistemas ya

existentes.



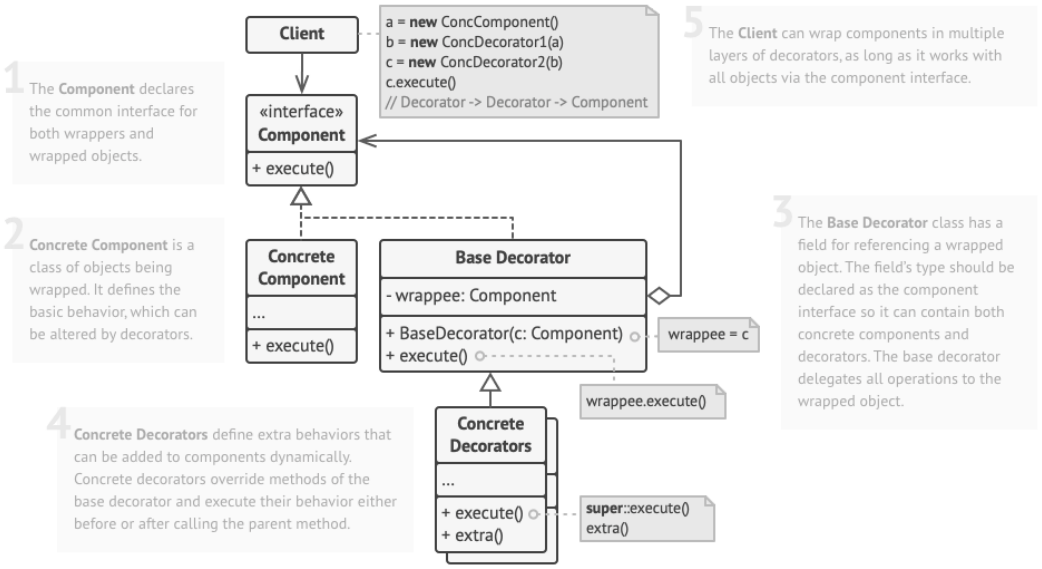


- [Dani C.] **Bridge**: este patrón nos permite separar la abstracción de la implementación. Proporciona una interfaz estable a los diferentes roles permitiendo que cambien las clases de implementación. Además, permite incorporar nuevas implementaciones a medida que evoluciona el sistema, con lo cual será importante para la proyección que ya tenemos de generar una segunda versión.



- [Dani B] **Decorator:** Permite agregar dinámicamente distintos comportamientos/features (decorators) a un componente en runtime. En este

caso podríamos tener un usuario como componente y los distintos roles que se quieren implementar podrían ser distintos decorators. De esta manera extendiendo funcionalidades de forma simple.



Ejercicio 3

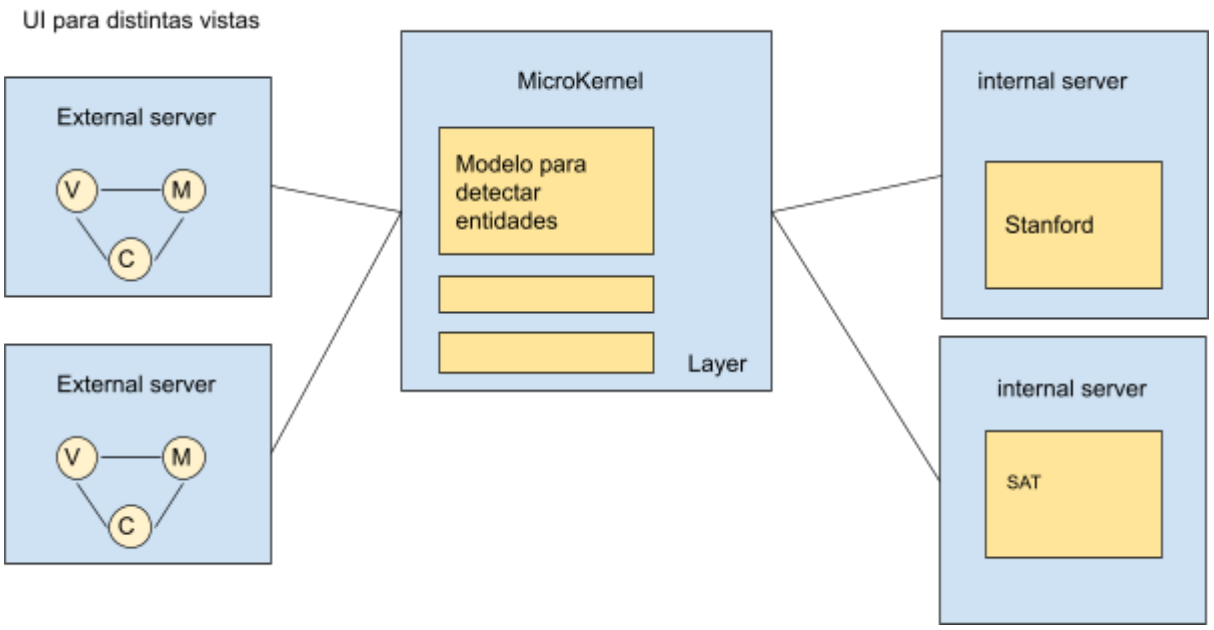
Pregunta 3 -

Para un sistema de asistencia en el desarrollo de software por computadora se piensa desarrollar las siguientes funcionalidades: detección de entidades relacionadas y construcción automática del modelo de dominio usando patrones de colaboración a partir de lenguaje natural expresado por texto, validación de la asignación de las reglas de negocio a las entidades del modelo según la estrategia definida en clase y validación de la consistencia lógica entre reglas de negocio (detectar contradicciones). El sistema debe presentar una interfaz de edición del texto de entrada que define el dominio del problema a resolver y vistas gráficas para cada uno de los resultados de salida mencionados. Para procesar el texto de entrada y validar su correcta expresión según el estándard SBVR (Semantics of business vocabulary and rules specification) se piensa utilizar el parser de la Universidad de Stanford implementado en Java. No obstante el procesamiento de la salida de este parser deberemos implementarlo en el proyecto, así como la construcción del modelo con las entidades y relaciones detectadas y la validación de la asignación de las reglas.
Para la validación de la consistencia lógica se usará un SAT Solver (Problema de satisfacibilidad booleana) implementado en Java que operará sobre un modelo lógico construido a partir del modelo de dominio, esta funcionalidad debe construirse. El sistema deberá poder extenderse agregando otras funcionalidades en forma continua como por ejemplo la generación automática de código u otras. Es decir que deberá evolucionar con frecuencia. El funcionamiento se pensó interactivo, casi en tiempo real. El uso debe ser lo más flexible posible en el sentido de que el usuario elegirá solo construir el modelo de dominio, o además asignar las reglas y validar su asignación o solo validar la consistencia lógica de las reglas sin ver ningún modelo o todos los anteriores. El sistema será utilizado por un analista con escasos conocimientos de programación. Se está analizando la posibilidad de implementar cada aspecto del problema que resuelve con diferentes tecnologías. ¿Cuál sería su elección? ¿Por qué? Si es necesario haga supuestos para fundamentar sus decisiones. Ayudese con un diagrama de contexto para explicar su propuesta y definir las distintas partes de la arquitectura.

- *El sistema debe presentar una interfaz gráfica de edición de texto de entrada (...) y vistas gráficas para cada uno de los resultados de salida mencionados* -> **MVC** para mostrar distintas vistas de los resultados OK
- *Para procesar el texto y validar su expresión se usa un parser (...) el procesamiento de la salida de este parser deberemos implementarlo en el proyecto así como la construcción del modelo con las entidades y relaciones detectadas (...)* -> **Layer** para obtener abstracciones de cada capa y que las mismas se puedan reutilizar para construir otros subproductos

- Una vez definido el modelo deben cargarse las reglas, que serán validadas a partir de una arquitectura de Pipe que elimine los que no pasen las validaciones subsecuentemente. (supongo que el procesamiento del resultado del parser debe filtrado por distintos filtros, en este caso, las validaciones).
- *Deberá evolucionar con frecuencia* -> **Microkernel** ya que se trata de un sistema que va a evolucionar con el tiempo y esta arquitectura se adapta a los cambios. En este caso, el internal server se encontrarían el parser de la Universidad de Stanford y el SAT Solver, en el external server se encontrarán los modelos MVC para cada una de las vistas y en el Micro Kernel, la lógica de negocio separada en distintas

layers en donde se aplicarán los distintos filtros según el diseño de Pipe Filter para las validaciones.



Final 2

Pregunta 1

Pregunta 1 -

Usted es asignado en su nueva empresa al mantenmiento de un sistema Enterprise del cual depende gran parte del negocio de la empresa. Debe aprenderlo ya que además de mantenerlo deberá incorporar nuevas funcionalidades. Para implementar estos cambios debe conocer el impacto (grande, chico, nulo) que tendrán en cada parte del sistema a extender. Qué métrica utilizaría para obtener una visión primera del sistema y qué aportaría la métrica que usted proponga.

Como primera instancia utilizaría métricas para medidas de tamaño, complejidad, acoplamiento y utilización del mecanismo de herencia (Overview Pyramid). Para poder determinar el impacto de los cambios en las distintas partes del sistema.

Para tamaño y complejidad nos servirá:

1. **Cyclo**: número de decisiones por línea de código.

2. **Loc**: número de líneas de código por método.

3. **Nom**: número de método por clase.

4. **Noc**: número de clases.

5. **Nop**: número de paquetes.

Para poder comparar el tamaño del proyecto usaría la razón entre cyclo y loc, entre loc y nom, entre nom y noc y entre noc y nop.

A partir de esto se pueden obtener métricas de que tan grande es el sistema. Para acoplamiento, se puede pueden usar:

1. **Calls**: número de invocaciones a cualquier método de una clase desde diferentes métodos de otras.

2. **Fanout**: número de clases dependientes (uso, invocaciones).

Para poder comparar el acoplamiento, se pueden usar las siguientes razones entre Fanout y calls y entre calls y nom.

Para Medidas de herencia, se pueden usar:

1. **ANDC**: número promedio de clases derivadas. Sumatoria de número de clases por número de herencias por clases dividido NOC.

2. **AHH**: Promedio de altura de herencia. Sumatoria de número de clases root por número de niveles de herencia dividido NOC.

Adicionalmente, se podrían considerar métricas polimétricas, aquellas que brindan un medio simple y poderoso de visualización de la complejidad. Gracias a ella podríamos encontrar las "god classes" que rompen el principio de única responsabilidad.

Pregunta 2 (repetida)

Pregunta 3

1 -Cuál es la diferencia entre un paradigma/lenguaje de programación **declarativo** vs uno **imperativo**? La programación funcional, a qué grupo pertenece? Por qué?
2 - De un ejemplo de programación declarativa y el mismo en programación imperativa. Marque la esencia, en el código/pseudo código que presente, que represente el mecanismo de su razonamiento al enfrentar el problema de resolución.
3 - En el desarrollo de un sistema ¿utilizaría ambos? ¿para qué? ¿en qué forma?

1.0 El paradigma **declarativo** es un paradigma de programación en donde el programador se enfoca en escribir el qué queremos y no el cómo, es decir el programador no describe el flujo de control, un ejemplo de esto es el lenguaje SQL donde la sintaxis es casi como hablarle a la computadora. En el paradigma **imperativo** por contrario al declarativo escribimos el qué y cómo queremos que se ejecute el programa, es decir, se escribe secuencias de órdenes y algoritmos a modo de controlar el flujo de la ejecución. La programación funcional pertenece al paradigma declarativo porque la idea es representar flujos con funciones, además estos no contienen estado y por lo tanto no se define el flujo para cambiar de estado, el programador escribe lo que quiere que la función devuelva.

1.1: La Programación con Procedimientos o Imperativa consiste en una secuencia de secuencias que se ejecutan en orden y van cambiando el estado del programa del cual forman parte. Ejemplos de este tipo de lenguaje son Pascal y C. El componente básico de estos lenguajes es la secuencia de secuencias que pueden estar agrupadas formando procedimientos que se invocan. Una característica saliente es el escaso grado de encapsulamiento que presentan.

1.2: En los lenguajes **Imperativos** las sentencias que el programador escribe definen “qué” se hace y “cómo se hace”. En los lenguajes **Declarativos** sólo se anuncia “qué” se hace y la implementación queda oculta. (APUNTE)

1.3

Programación declarativa = > definiciones => representación del conocimiento externo al sistema + argumentos => sin estados explícitos

Programación iterativa/imperativa => comandos => representación interna del sistema => con estados explícitos

2:

Declarativo:

```
const arr = [1,2,3];
const total = arr.reduce((e, acum) => e + acum);
console.log(`el total es ${total}`);
```

Imperativo:

```
const arr = [1,2,3];
let total = 0;

for(let i=0; i < arr.length; i++) {
    total += arr[i];
}

console.log(`el total es ${total}`);
```

Observamos que en el declarativo no hay un for o secuencia de pasos que indique como hacer la suma total del array, sino que simplemente se hace el llamado a la función reduce y se le indica la operación que se le quiere hacer al array. Indicamos el qué ! En cambio en el imperativo ponemos explícitamente como queremos que se ejecute.

3. Ambos paradigmas son útiles en la construcción de un proyecto, se podrían utilizar en distintos componentes, cuando se requiere de mayor control sobre el programa y mejor administración de recursos se puede optar por el paradigma imperativo ya que nos permite especificar la implementación. En cambio para implementaciones que el performance no es de alta importancia y quizá es más importante entregar rápido sea más conveniente utilizar un paradigma declarativo.

Final 3

Ejercicio 1

1. Explique en frases cortas el proceso de resolución de un problema de diseño de software utilizando programación orientada a objetos. Indique punto de partida, condiciones necesarias, criterios a aplicar y modo de validación del producto resultante.

~~Punto de partida, contexto del modelo de dominio, si estamos desarrollando un software para un cliente primero necesitamos realizar un relevamiento del problema a intentar resolver y de allí identificar el modelo de dominio, es decir los objetos que serán necesarios para el software, esto se puede documentar en diagramas de clase o de secuencia, como para tener un mayor entendimiento de los objetos y la relación entre ellos.~~
~~Luego debemos elegir las tecnologías a utilizar y una arquitectura de software que se acomode a las necesidades del problema con sus distintos criterios, es decir si tenemos una lógica de negocio compleja probablemente sea necesario utilizar un patrón de arquitectura layers, si necesitamos manejar interfaces de usuario seguro necesitaremos utilizar un patrón MVC, y así con nuestras distintas necesidades. Luego para el desarrollo del software se pueden utilizar distintas metodologías como scrum.~~
~~A modo de validación se pueden escribir test automatizados para garantizar que el software se comporta como nosotros queremos, se pueden hacer distintos tipos de test: unitarios, funcionales, integración de punta a punta. Finalmente una vez construído el software se puede realizar una demostración con el cliente como para validar que la plataforma es lo que efectivamente el cliente desea, caso contrario se toma nota de las mejoras o cambios que se necesitará hacer.~~

Notas clase 1 de Pantaleo

Complejidad = Complejidad solución + complejidad problema

A veces por falta de criterio la complejidad de la solución termina siendo mucho mayor a la del problema y esto es por falta de criterio, la complejidad debería estar solo en el problema. Las soluciones deberían ser simples, lo que es difícil es llegar a estas soluciones simples

⇒ Conocer bien el problema a resolver para no proponer una solución que aporte a la complejidad

Mecanismos para atacar la complejidad desde OOP:

- Descomposición → descubrir conceptos del dominio del problema, que estos van a ser objetos del dominio del problema.
- Abstracción → Pensar, hay que ponerle límites a los objetos que se identificaron anteriormente y me quedó con lo importante o lo necesario (ejemplo un usuario solo necesitamos nombre, DNI, no me interesa el color del pelo o que comió ayer)

Esos son los dos mecanismos más importantes ya que implican un descubrimiento del negocio.

- Establecer jerarquías → inventar, por ejemplo generalizaciones, factura, remito, etc son documentos comerciales. Por lo tanto creamos una relación de herencia, pero el documento comercial no existe en la vida real, es un invento nuestro.

Modelo de dominio se soporta sobre esto. Al modelo de dominio le aplicamos los patrones de colaboración + reglas de negocio y obtenemos modelo de negocio. Una vez que tengamos el modelo de negocio recién ahí podemos realizar un diseño, ya que significa que hemos entendido el problema

Luego se van a escribir tests que traten de romper los objetos implementados, una vez que pasen los tests estos objetos van a estar representando el modelo de negocio deseado.

<u>Proceso de desarrollo:</u>

1. Modelo de casos de uso / historias de usuario, básicamente es una especificación del problema a resolver. Estos serían nuestros requisitos funcionales.
2. A partir de ahí puedo desarrollar diagrama de secuencia o diagramas de colaboración, esto nos va a aportar para poder construir nuestro modelo de negocio.
3. Arranco modelo de dominio, la idea es realizar descubrimientos en esta etapa y relacionar entidades. Me permite entender en detalle el negocio y sus reglas
4. Luego de tener el modelo de dominio realizo modelo de negocio o análisis, representa la dinámica del modelo de dominio, nos interesa el comportamiento de las entidades. Me permite entender el negocio
5. Con todo lo anterior estoy en condiciones de realizar el diagrama de clases.
6. Las reglas de negocio tienen que ser validadas para que nuestro modelo de negocio tenga un comportamiento adecuado y la validación están llevado a cabo por los objetos que representan los conceptos del dominio del problema

Modelo de casos de uso → diagramas de secuencia → modelo de dominio → modelo de negocio → diagrama de clases → validación de modelo de negocio a través de tests en los objetos

Respuesta armada con las notas

Proceso de desarrollo:

1. Primero lo que queremos es entender bien el problema y su contexto, esto es de suma importancia, para no proponer una solución que aporte a la complejidad de la solución. Para esto podemos partir de los Modelo de casos de uso / historias de usuario, básicamente es la especificación del problema a resolver. Estos serían nuestros requisitos funcionales.
2. A partir de ahí puedo desarrollar diagrama de secuencia o diagramas de colaboración, esto nos va a aportar para poder construir nuestro modelo de negocio.
3. Modelo de dominio, la idea es realizar descubrimientos en esta etapa y relacionar entidades. Me permite entender en detalle el negocio y sus reglas.
4. Luego de tener el modelo de dominio realizo modelo de negocio o análisis, representa la dinámica del modelo de dominio, nos interesa el comportamiento de las entidades. Me permite entender el negocio.
5. El objetivo de estos primeros pasos descritos es realizar un descubrimiento de los conceptos del dominio del problema, que estos van a ser objetos del dominio del problema. Y a su vez definir el límite de la abstracción. Con todo lo anterior estoy en condiciones de realizar el diagrama de clases.
6. Las reglas de negocio tienen que ser validadas para que nuestro modelo de negocio tenga un comportamiento adecuado y la validación es llevada a cabo por los objetos que representan los conceptos del dominio del problema. Para esto se escribirán tests que intenten romper las reglas de negocio y una vez que estos tests ya estén todos pasando, en ese momento podemos asegurar que los objetos diseñados cumplen con el modelo de negocio que planteamos.

El Diseño de Software

Se lo define como el proceso de aplicar ciertas técnicas y principios con el propósito de definir un dispositivo, un proceso o un Sistema, con suficientes detalles como para permitir su interpretación y realización física. La etapa del Diseño del Software encierra cuatro etapas:

1. Trasforma el modelo de dominio de la información, creado durante el análisis, en las estructuras de datos necesarios para implementar el Software.
2. El diseño de los datos.- Define la relación entre cada uno de los elementos estructurales del programa.
3. El Diseño Arquitectónico.- Describe como se comunica el Software consigo mismo, con los sistemas que operan junto con el y con los operadores y usuarios que lo emplean.

4. El Diseño de la Interfaz.
5. El Diseño de procedimientos.

El Diseño del software transforma elementos estructurales de la arquitectura del programa.

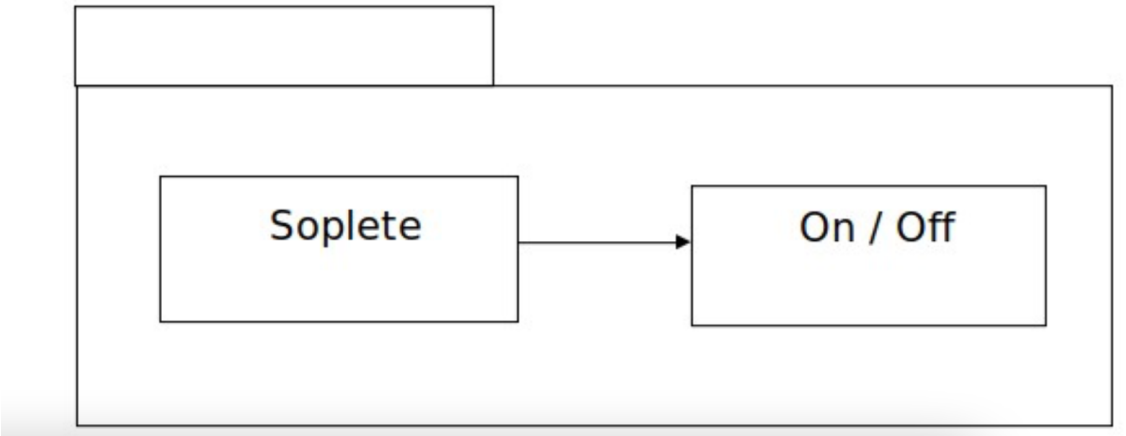
Ejercicio 3

3. Para un sistema en desarrollo se ha definido la arquitectura utilizando los siguientes patrones: MVC, pipe filter, EA y layers en la capa de negocio. Según esta definición indique en qué contexto y qué problemas se buscó resolver en el sistema en cuestión con la utilización de los patrones de arquitectura mencionados.

- **MVC**: se usa inicialmente en sistemas donde se requiere el uso de interfaces de usuario. Surge de la necesidad de crear software más robusto con un ciclo de vida más adecuado, donde se potencie la facilidad de mantenimiento, reutilización del código y la separación de conceptos.
- **EA:** desacoplando las diferentes “tiers” se logra separación de incumbencias y su reuso. Trae como ventajas la reusabilidad y la cambiabilidad. Posiblemente el sistema sea complejo, maneje distintas interfaces de usuario, lógica de negocio y persistencia de datos.
- **Pipe filter:** Se utiliza para procesamiento de datos, no sirve para sistemas críticos ya que no existe un estado global. Cada capa del pipe aplica un filtro o operación a los datos. Esto se podría utilizar para datos como texto o imágenes por ejemplo.
- **Layers**: se centra en una distribución jerárquica de los roles y responsabilidades proporcionando una separación efectiva de las preocupaciones (cada cual se encarga de lo que le corresponde). La separación, además de permitir el desarrollo simultáneo de distintos features de la aplicación, reduce el riesgo y el impacto de los cambios tecnológicos, por lo que también brinda adaptación al cambio.Posiblemente el modelo de dominio sea complejo y haya muchas reglas de negocio, por lo que con el patrón layers se divide el modelo en distintas capas de abstracción, simplificando el problema.

Ejercicio 4

4. Para una simulación se escribió una librería de prueba y ensayo de máquinas herramientas eléctricas. Entre el código de la misma puede verse el correspondiente al diseño que se muestra más abajo. Se trata de un interruptor de tipo on/off y un soplete eléctrico para remover pintura. Ahora ha surgido la necesidad de simular una agujereadora que posee además de este tipo de interruptor uno por contacto, es decir que permanece activado mientras se oprime. Analice el diseño actual en relación con los criterios de buen diseño, modifiquelo si lo considera necesario al momento de agregar la nueva funcionalidad. Justifique cada aseveración y modificación que realice.



El nombre On/Off no es representativo, sería una mejor práctica reemplazarlo por el nombre del objeto correspondiente: Interruptor. Para poder agregar una Agujereadora, se debe agregar a la librería un nuevo Paquete llamado “Agujereadora” que interactúe con Interruptor pero que este último no sepa de la presencia de Agujereadora. Dentro del paquete Interruptor, podremos tener distintos tipos de interruptores: On/Off y por contacto

Criterios del buen diseño: basarnos en abstracciones no en implementaciones, utilizar delegación antes que herencia