Clase 1- Introduccion Teoria

Introduccion a heuristicas:
Malos olores de diseño:

- Envidia de atributos: Un objeto que pide datos a otros objetos para hacer algo el mismo (por ejemplo un calculo). Para evitarlo: La tarea la debe hacer el objeto que tiene los datos que se necesitan, es decir, se le delega la responsabilidad a la otra clase.
- Clase Dios: Una clase que resuelve todo y las demas están todas anemicas. Una clase asi no cumple con el principio de "Una sola responsabilidad", ademas, posiblemente haya envidia de atributos. Para evitarlo: Ver que objetos objetos podria hacer aparecer que se pueda encargar de alguna de las responsabilidades.
- Codigo dupliciado: Para evitarlo: preguntarse si se puede generalizar el comportamiento en una clase y heredarla, tambien si se puede llevar a otro objeto y reutilizarlo.
- Clase larga: tengo una clase larga en comparacion al resto, para evitarlo: Identificar dentro del metodo largo, partes que podria considerarse comportamientos individuales. Llevar cada parte a un nuevo metodo y cuando necesite llevar a cabo alguno de esos comportamientos, enviar mensaje a this.
- Objetos que conocen el id de otro: Nunca relacionar objetos por medio de claves o ids. Para evitarlo: Cuando un objeto se relaciona con otro, lo hace con una referencia, nunca conoce su id.
- Variables de instancia que en realidad deberian ser temporales: Si una variable de instancia deja de tener sentido en algun momento de la vida del objeto, entonces es probable que sea temporal o que sea responsabilidad de otro. Para evitarlo: Pensar realmente si esa variable es un atributo del objeto que lo acompaña siempre.
- Romper encapsulamiento: Romper el encapsulamiento de un objeto es muy malo, nos hace perdes la gran mayoria de las ventajas de la POO. No solo con tener las variables publicas rompemos el encapsulamiento, sino que al momento de tener getters y setters tenemso que tener en cuenta sobre que variables, porque los metodos mencionaos anteriormente permiten a otros que modifiquen lo que quieran. Para evitarlo: Solo agregar setters y getters cuando sea necesario. Nunca modificar una coleccion que no sea de nuestra clase, delegar las tareas a los que tienen la informacion que se necesita..
  -Entre otras heuristicas, estas me parecieron las mas importantes.

Clase 1 Teorica:
Terminos mas importantes

🔷**Refactoring:** Transformacion de codigo que **preserva** el comportamiento pero mejora el diseño, estructura y mantenibilidad.

ejemplo: Evitar variables publicas y utilizar getters y setters en su lugar.
La idea de refactoring es que se haga lo mismo (en cuanto a comportamiento y output), pero de una forma mejor en cuanto a diseño.

DETALLE IMPORTANTE: Refactoring no agrega funcionalidad. Si no que cambia el diseño.

🔷**Code Smell:** Indicador de posibles problemas en el diseño del codigo. Destaca areas en las que el codigo podria estar violando principios de diseño de fundamentos.
Indicadores de que el codigo podria mejorarse porque es dificil de entender, mantener o extender. No significa que el codigo no funcione, sino que su calidad es baja o que podria traer problemas a futuro.
Ejemplo: Metodos demasiados largos, clases con muchas responsabilidades,nombres pocos descriptivos
Los code smell, **no son errores** pero futuros desarrollos pueden ser:

- Mas dificiles
- Demandar mas horas de trabajo.

Velocity: Cantidad de tareas (funcionalidades) / periodo de tiempo

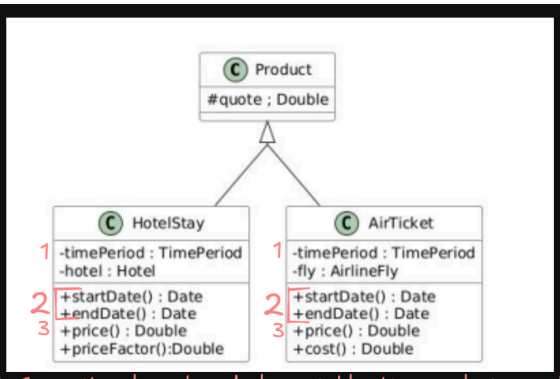
Deuda tecnica: Costo de rehacer una tarea que tiempo atras fue resuelta con una solucion rapida y desprolija.

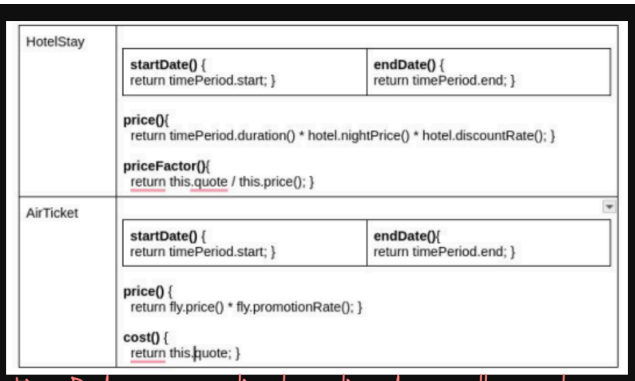
✅ Conclusion: Refactoring y code Smell NO SON BUGS ! Son aspectos del codigo que afectan su calidad y mantenibilidad, pero no su funcionalidad.

💡 **Relación entre ambos:**
Si se detecta un code small (indicador de diseño deficiente que puede traer problemas a futuro), podemos solucionarlo realizando un refactoring (resuelve el code smell con buen diseño/practicas, pero no agrega funcionalidad, no altera el comportamiento, solo lo plantea con buenas practicas).

Por cada code smell, se puede hacer un refactoring, pero un refactoring no esta ligado si o si a code smell.

Ejercicio de Teoria:





1 Se repite el nombre de la variable, tienen el mismo objetivo. Podemos generalizarlo realizando un pull up en la superclase
2 Se repiten metodos, tienen el mismo objetivo y el mismo codigo. Se generaliza realizando un pull up en la superclase.
3 Se repiten los metodos, tienen el mismo objetivo (calcular precio), pero distinto codigo, podemos resolverlo de esta forma:
- Declarar en la superclase, un metodo abstracto Price que retorne un Double

aclaracion:
en plantUml , el ▭ representa a una variable de instancia privada

Resolviendo el code smell nro 3 de la primera alternativa