Sebastián Valencia Calderón
201111578

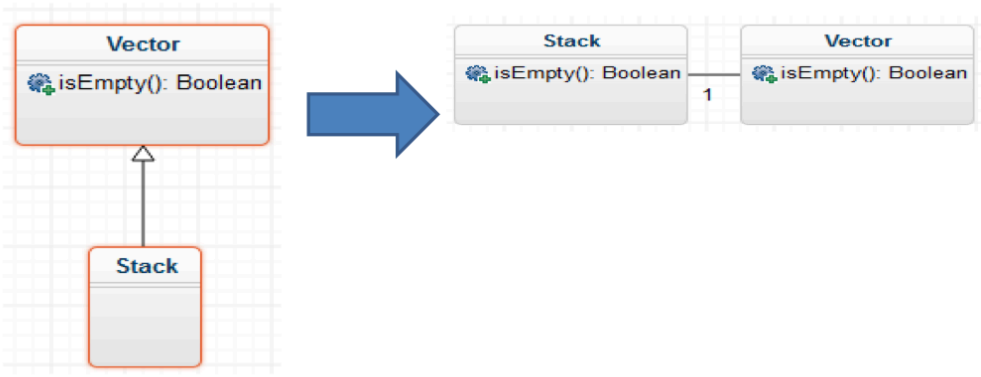
Punto 1.

a) El factor de acoplamiento, es un número de cero a uno inclusive, que mide el acoplamiento de un diseño provisto generalmente en UML. Entre más alto sea éste, más acoplado es el diseño, por lo tanto, más afecta la modificabilidad del código que acoja el contrato del diseño propuesto. En éste caso en particular, el factor de acoplamiento, dado por la fórmula mostrada, compromete el número de asociaciones que no son herencia sobre el total de asociaciones.

$$FA = \frac{Asociaciones\ que\ no\ son\ herencia}{Total\ de\ asociaciones} = \frac{3}{7} = 0.4286$$

En éste caso en particular, se sabe que el acoplamiento es menor al caso donde se tenga un factor de acoplamiento medio, por lo tanto, se considera que se tiene bajo acoplamiento.

b) La táctica de modificabilidad que pudo haberse usado, según el diseño propuesto sobre el diseño de componentes mostrado, es Wrapper, es decir, replicar lógicas expuestas en una relación de herencia a través de clases atomizadas. Un ejemplo claro se muestra en el siguiente gráfico:



En concreto, en el caso de Shiro en particular, se tiene que las asociaciones entre:

- AuthnticatingSecurityManager, Authenticator.
- AuthorizingSecurityManager, Authorizer.
- SessionSecurityManager, SessionManager.

Pudieron haber sido desacopladas de herencias previas incluidas en algún diseño. Esto, claramente reduce el factor de acoplamiento, pues si se considera

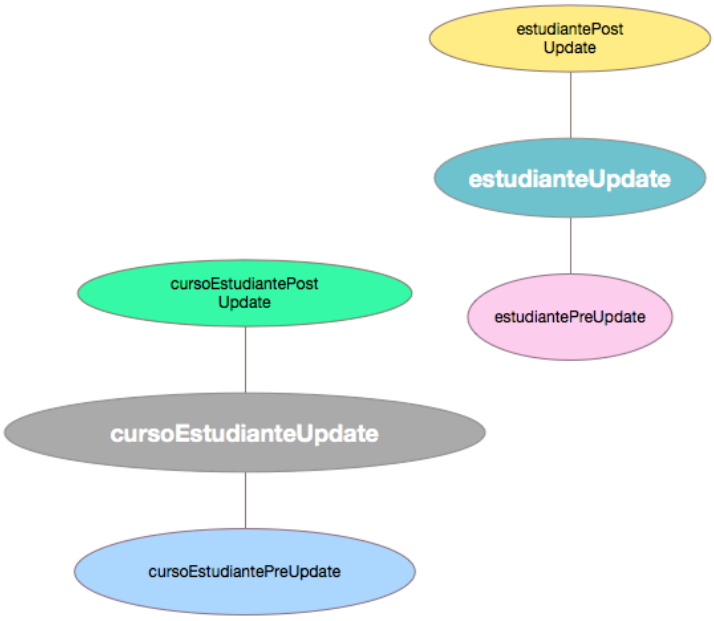
una arquitectura previa que mapee directamente de la arquitectura provista a una potencial previa usando como mapa de ruta el gráfico mostrado, el viejo factor de acoplamiento, habría sido:

$$FA = \frac{Asociaciones\ que\ no\ son\ herencia}{Total\ de\ asociaciones} = \frac{0}{7} = 0$$

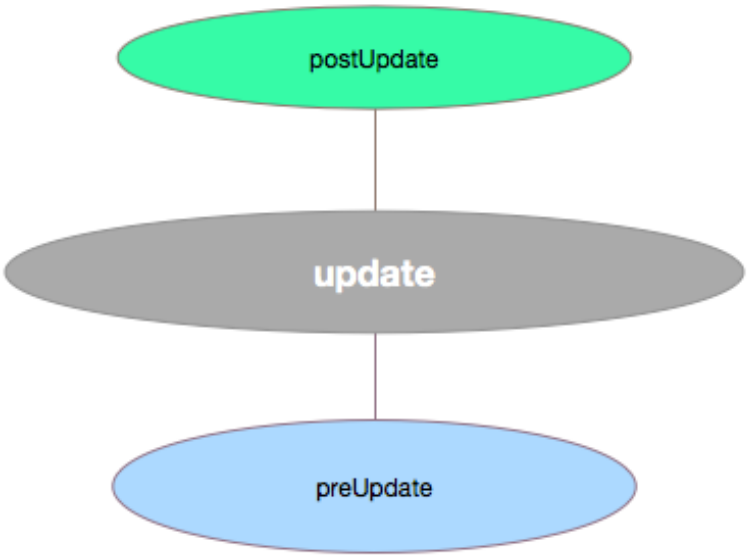
En éste último caso, el factor de acoplamiento no daría pistas claras sobre el acoplamiento del diseño.

Punto 2.

a) El diseño más favorable para afectar favorablemente el atributo de calidad de modificabilidad, es el segundo, pues la responsabilidad de cada clase, es decir, de cada página, se separa haciendo uso del compromiso sugerido por la clase abstracta, por lo tanto, si ha de cambiar el diseño, se cambia la clase abstracta y es fácil darse cuenta de las modificaciones gracias a las alertas del compilador. Por otra parte, el primer diseño, replica código, y el compilador no brinda pistas si se ha olvidado modificar alguna de las clases. La anterior explicación, se concretiza a continuación. Donde se muestran los conjuntos disyuntos de cada clase mostrada (la cantidad de conjuntos disyuntos, es inversamente proporcional con el grado de cohesión de la implementación propuesta).



Conjuntos disyuntos EstudianteSessionBean



Conjuntos disyuntos ManejadorSessionBean

En general, se generaliza la lógica de cada método al acordar el contrato propuesto por la clase abstracta.

Punto 3.

a) En general, no usaría la solución propuesta, pues la única similitud que abstrae el uso de la clase abstracta, es el de los identificadores, los cuales pueden ser sólo una mínima parte de la composición del objeto. Es decir, igual toca realizar la implementación de los demás métodos get y set, por lo que la clase no soluciona la escritura de código. La clase con la responsabilidad de definir los métodos básicos, sería la clase que extienda la clase de la solución propuesta, o puede ser el MappedUser, lo cual no es normal, y no fortalece en nada la solución de los DTOs. El hecho de que el mapeo se haga usando la marca mostrada, hace que toque modificar cada clase hija cada vez que se cambie el padre.

Además, al hacer casting de la entidad, con otra entidad que tenga ésos atributos y ésos métodos, se puede acceder a la lógica adicional provista por la clase con la cual se cambió forzadamente el tipo de un objeto. Lo cual no es seguro, y afecta la modificabilidad.