Diseño de Software

DS-23-07-PD **Ángel Otero Barreiros**

Pedro Garcia Lorenzo

Gestión de Billetes:

Principios de diseño:

Principios de responsabilidad única :Este principio se observa en las clases que implementan la interfaz Properties o la interfaz Operations.Properties asume la única responsabilidad de buscar por cada clase que la implementa la función "sort",que se busca a si misma en una lista de Tickets.Operations realiza la misma función pero en vez de implementar el método "sort" implementa a "logic", que se encarga de realizar las funciones binarias AND o OR. Gracias a esto existe una gran cohesión, se puede observar como la función delega en las interfaces la ejecución del programa. El método "filter" de la clase Api es un ejemplo, delega la ejecución en la interfaz Operations.

Principio abierto cerrado:Se puede observar en todas las clases que implementan las 2 interfazes (clases abstractas)y en la clase Ticket, podrías añadir mas implementaciones y esto no cambiaría nada del funcionamiento principal.

Principio de Liskov: La gran mayoría de la funcionalidad de nuestro programa utiliza este principio como base. La el método "logic" de la interfaz Operations requiere una o varias implementaciones de la interfaz Properties para poder ser ejecutada., lo mismo pasa con el método "filter" de la clase Api, requiere por parámetro una implementación de la interfaz Operations y alguna de Properties en vez de necesitar una implementación en concreto.

Principio de inversión de la dependencia:Al igual que el principio de Liskov se observa en los métodos "logic" y "filter" ya comentados anteriormente, en ellos no se pide una implementación concreta de Properties o Operations en vez de algo en concreto.

Principio de segregación de interfaces:Nuestro código tiene dos interfaces para dos funcionalidades distintas en vez de una interfaz que una esas dos funcionalidades lo que seria peor.

El tipo de herencia utilizada en este programa es herencia por especialización ya que las subclases sobrescriben y ofrecen una versión especializada de la superclase.

Principio encapsula lo que varía:La clase Ticket y sus parámetros son invariables mientras que la Api tiene parámetros variables , por lo que se puede observar los aspectos que varían y que no varían de nuestro programa.

Bajo acoplamiento-Al tener todos los atributos de Ticket encapsulados en implementaciones de la interfaz Properties estos solo dependen de la interfaz, al contrario que la clase Ticket, que depende eso si, de varias clases.

Principio de mínimo conocimiento:En todos nuestros métodos se cumplen las condiciones de este principio,en los métodos solo se mandan mensajes a si mismo a un parámetro del método a un atributo del objeto o a un objeto creado en el método.

Patrones de Diseño:

Patrón Estrategia:La interfaz Operations es un claro ejemplo del patrón de Estrategia , la clase API(juego el rol de contexto) delega la operación en el objeto Operations.(juega el rol de Estrategia) Operations declara una interfaz común para todas las operaciones que queramos implementar.Las clases OR o AND (juegan el rol de estrategia concreta)implementan la operación utilizando el interfaz definido en Operations.

Patrón Inmutable:Todos los objetos que forman parte del objeto Ticket y el objeto Ticket en si forman parte de este patrón.Todas estas clases son publicas y finales , sus atributos son finales y privados ,inicializados en el constructor, sus métodos de lectura no devuelven referencias a objetos mutables.

Instancia única:La clase AND y la clase OR son clases que realizan un algoritmo por lo que no tendría ningún sentido que se pudieran crear más de una clase.

Planificador de Tareas:

Principios de Diseño

Principio de responsabilidad única:Al igual que en el programa anterior , este también tenemos una interfaz que asume la única responsabilidad de ejecutar el algoritmo setteado en el API. Gracias a este principio la cohesión del programa es mayor . Como se observa en el método "work" de la clase API que delega la ejecución en una de las implementaciones de esta interfaz.

Principio de abierto cerrado:La interfaz de Dependency es un ejemplo , se podría añadir mas implementaciones y no haría variar su funcionamiento.

Principio de Liskov:En la clase API existe un método "setDependency" que requiere una implementación de la interfaz Denpendency y se le proporciona para después ser utilizada en el método "work".

Principio de inversión de la dependencia:La clase en vez de depender de 3 atributos de tipo Dependency tiene un único atributo que engloba a estos tres, en nuestro programa sería la interfaz Depencency, que luego es setteada en alguno de sus implementaciones.El tipo de herencia utilizada para estas implementaciones es una herencia por especialización que las subclases sobrescriben y ofrecen una versión especializada de la superclase.

Bajo acoplamiento:Excluyendo a la clase Graph , las demás clases tienen un bajo acoplamiento ya que los cambios realizados en las clases que implementan la interfaz Dependency no hacen variar a otras clases.La clase Graph tendrían un alto acoplamiento ya que si es modificada las demás clases del programa se verían modificadas.

Principio de mínimo conocimento:La clase graph solo sabe sobre si misma, al igual que las clases que implementan la interfaz Dependency.La clase API si necesita saber sobre otras clases ya que si no , no podríamos elegir un tipo de algoritmo.

Principio dont repeat yourselve: Al hacer los implementaciones de los algoritmos utilizé un conjunto de operaciones privadas en cada implementación , y para seguir este principio las que se repetían las pasamos a la clase Graph , como eliminar una clave de todo el hashmap o calcular la longitud de un nodo.

Patrones de Diseño:

Estrategia:La interfaz El patrón estrategia se basa en tres roles principalmente , uno que hace de contexto , en nusetro caso la clase API, otro que juega el papel de estrategia, en nuestro caso la interfaz Dependency y otro juega el papel de estrategia concreta, en nuestro caso las implementaciones de esta interfaz.Este principio ayuda a tener unidas en una misma interfaz las diferentes formas de recorrer el grafo por lo que es mucho mas sencillo elegir una e implementar nuevas formas de recorrerlo.

Diagrama de Clases:Gestión de billetes

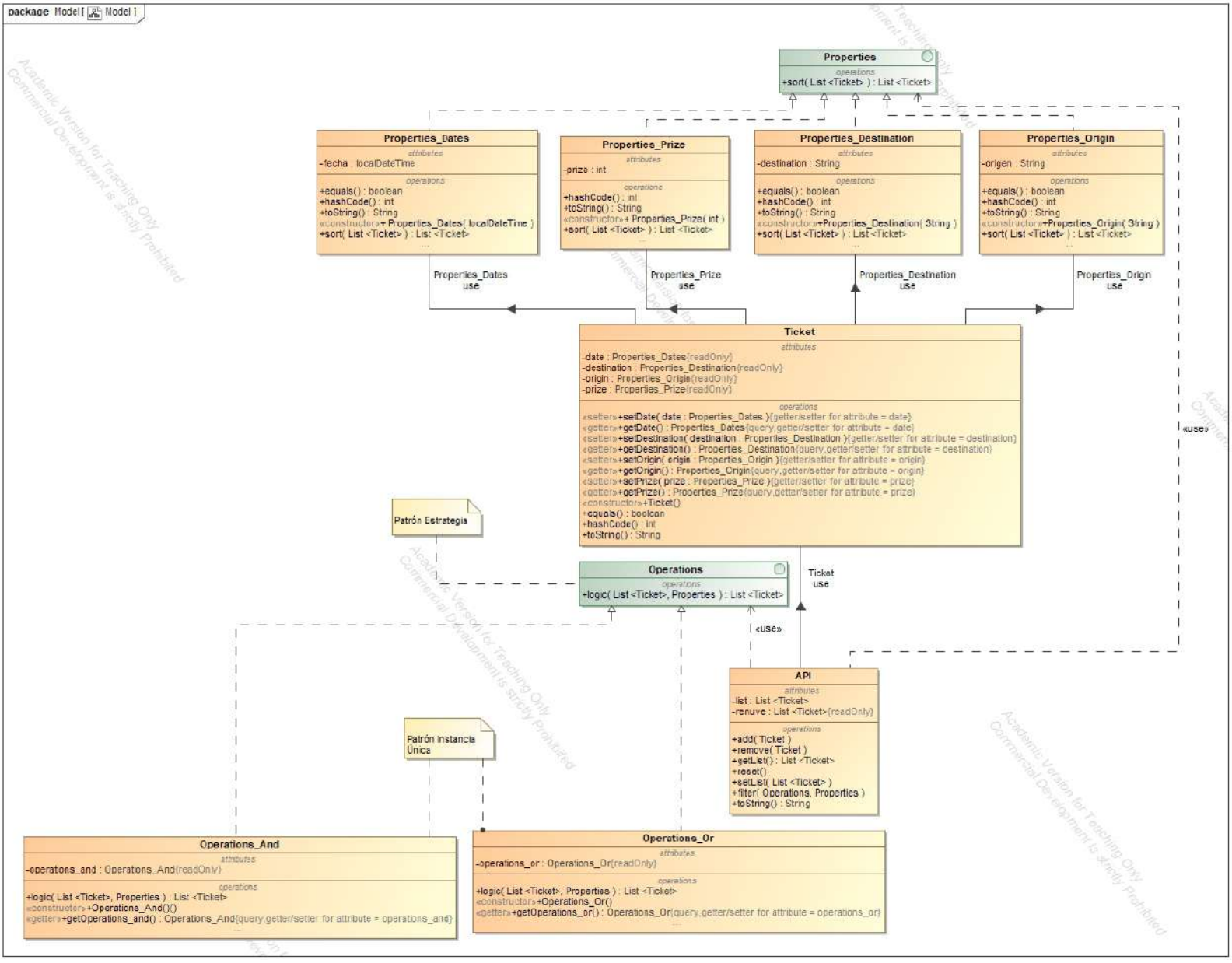

Diagrama dinámico de secuencia:Gestión de billetes

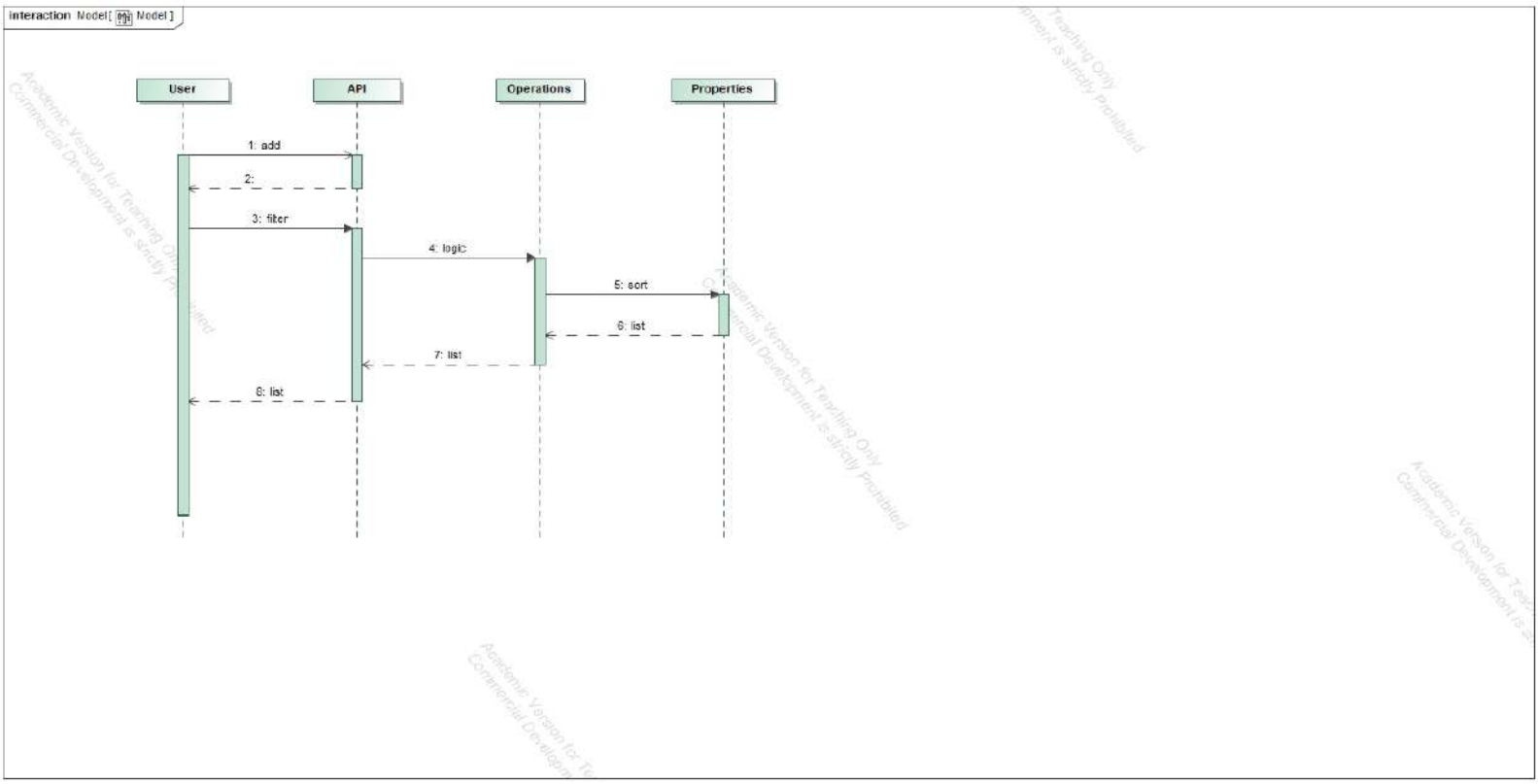

Diagrama de Clases:Planificador de Tareas

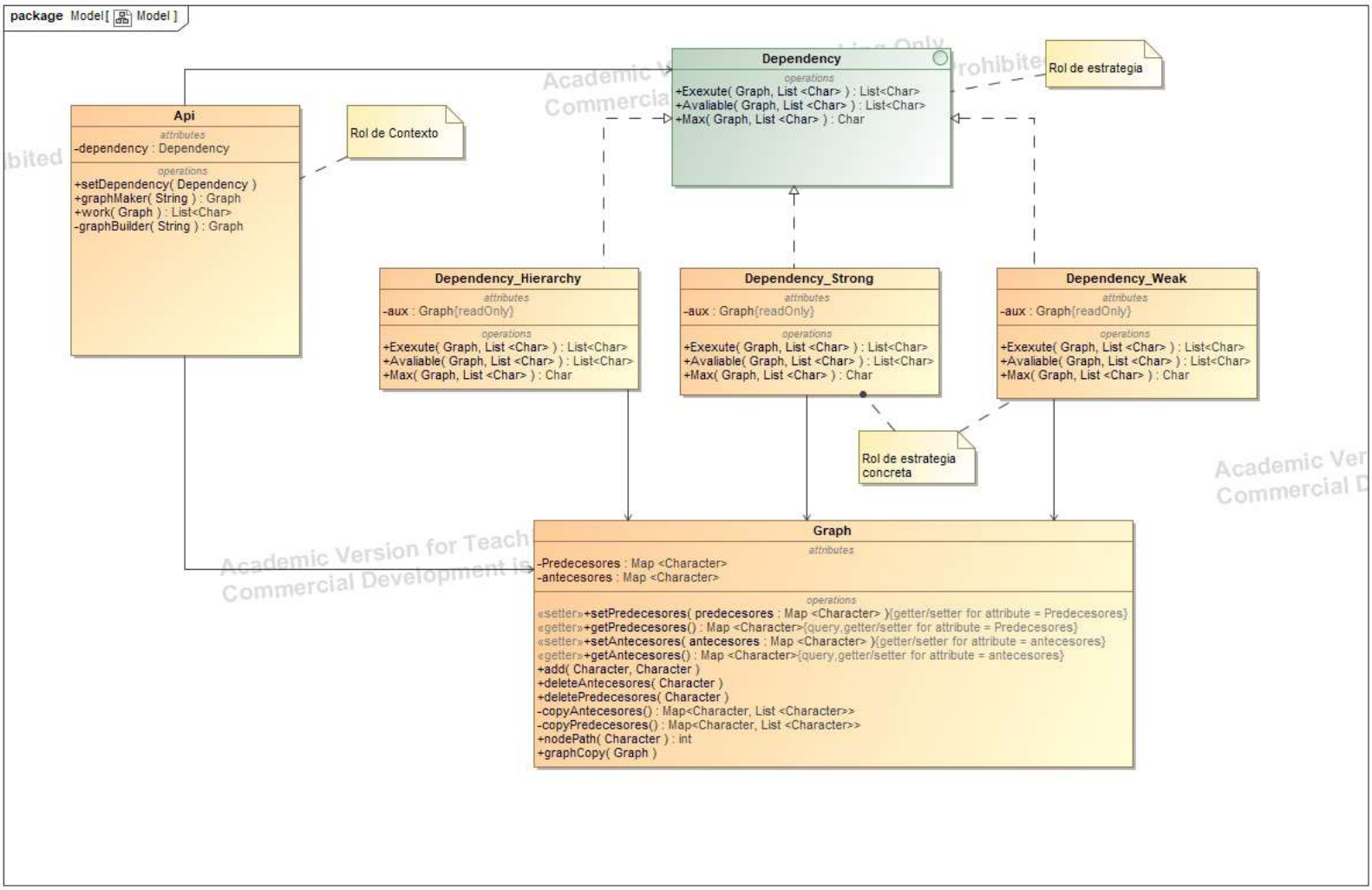

Diagrama dinámico de secuencia:Planificar de Tareas-Dependency

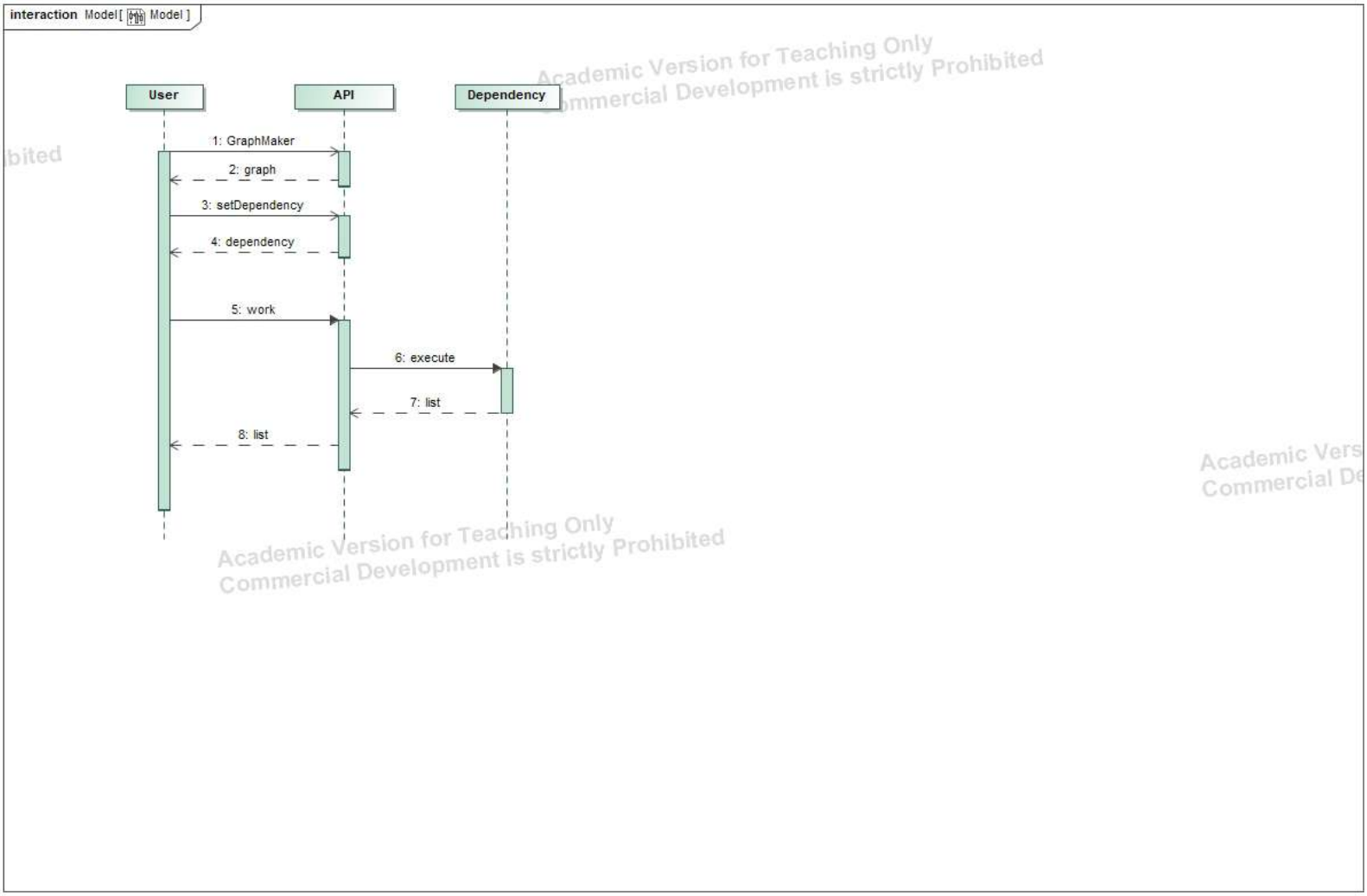

Diagrama dinámico de secuencia-Dependency_Strong

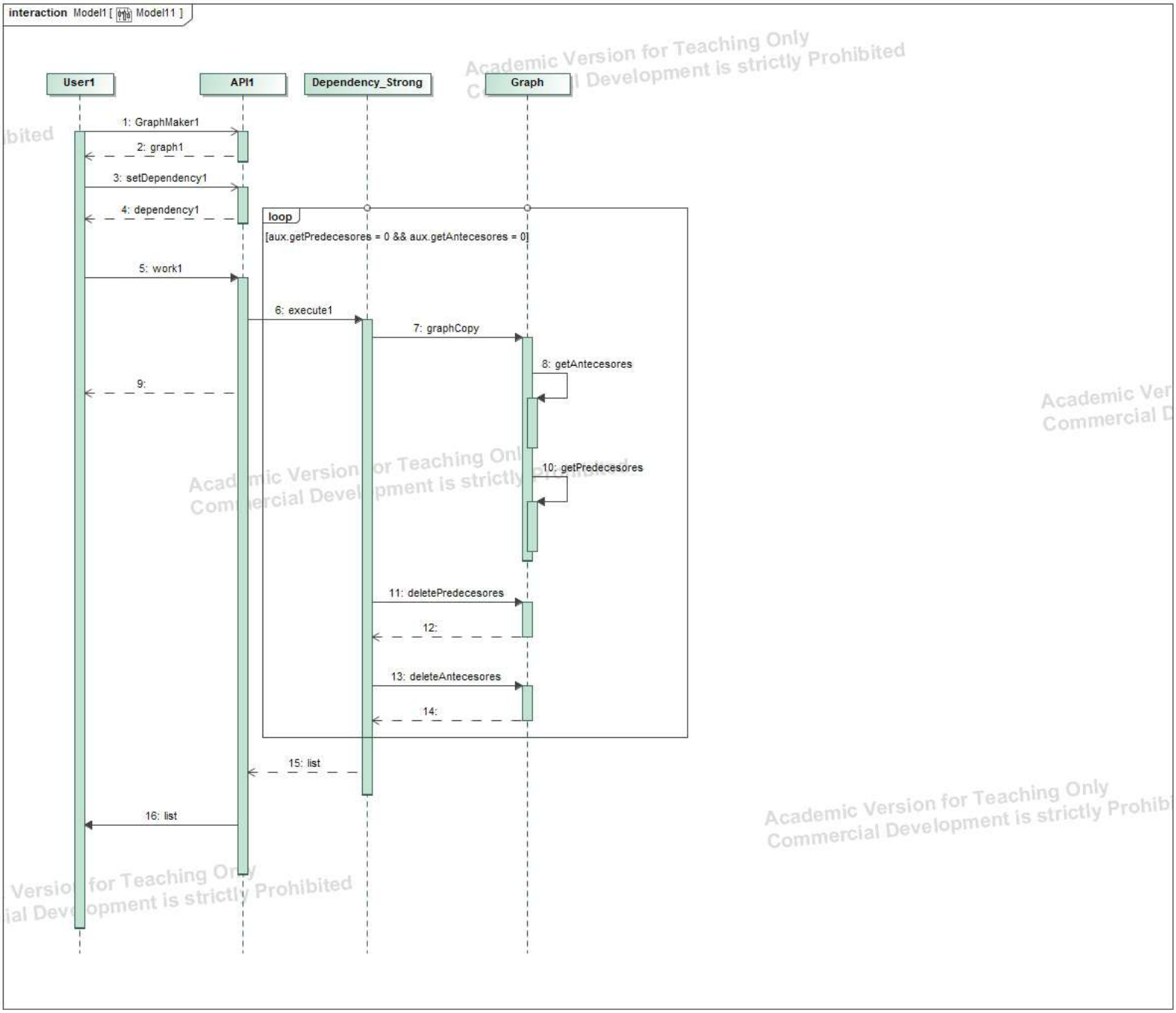

Diagrama dinámico de secuencia-Dependency_Weak

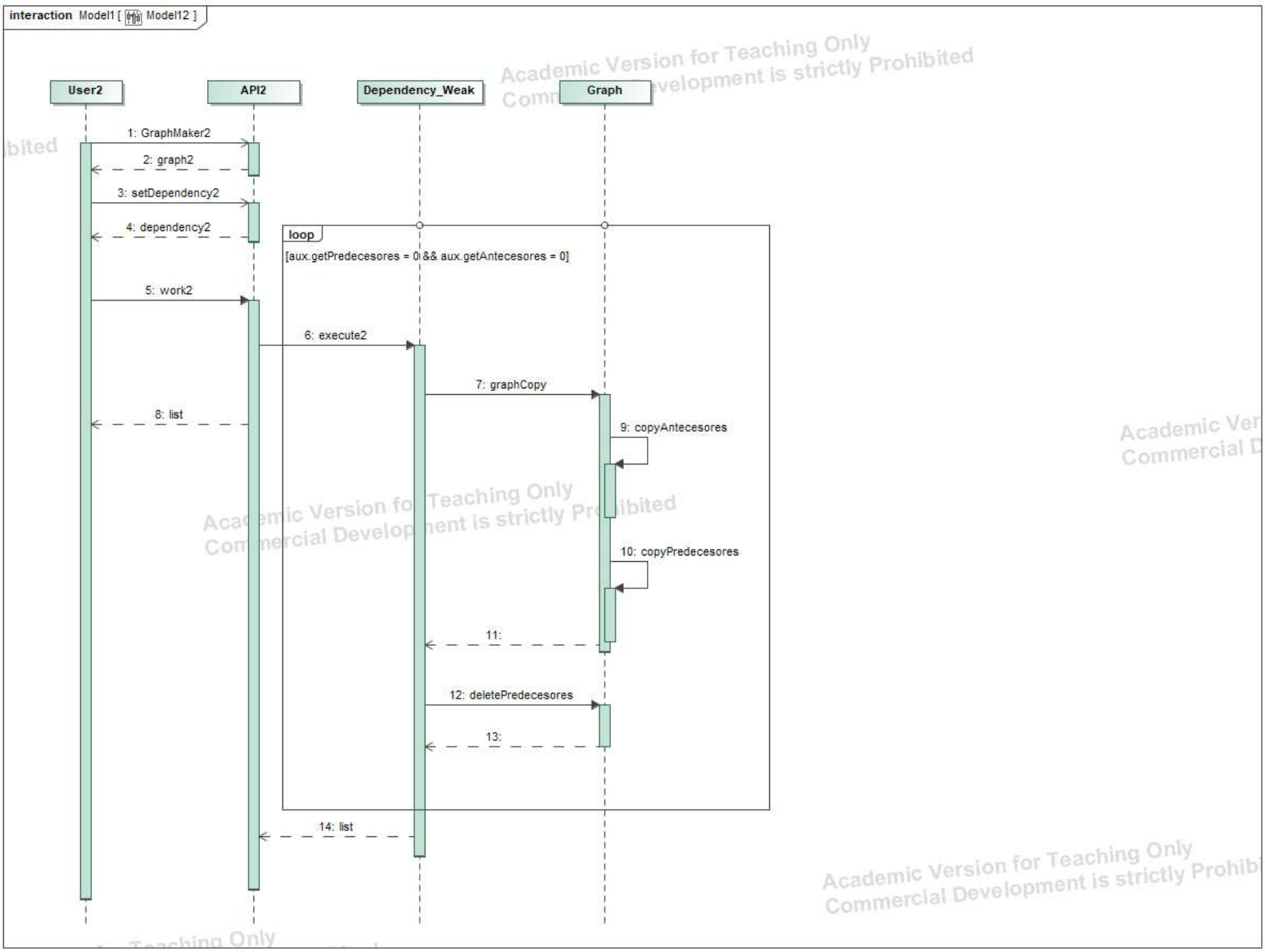

Diagrama dinámico de secuencia-Dependency_Hierarchy