Desarrollo de Software
TERM202215
Prof. Carlos Alonzo
Primer Examen Parcial
10-12-2021 **Nombre:** _____

El examen es individual. Lea con detenimiento y tranquilidad cada uno de los enunciados. Sus respuestas deben enviarlas en la Evaluación de Módulo 7. Existirá un apartado por cada pregunta del parcial.

1 Pregunta 1 (4 puntos)

A continuación se enuncian varias afirmaciones, usted debe indicar si es Verdadera o si es Falsa. Se aplica factor de corrección, es decir, una respuesta mala anula una buena. En otras palabras, no deje sus respuestas a la suerte.

1. Los Agregados (*Aggregates*) en *Domain-driven Design* (DDD) comparten el mismo comportamiento que las composiciones de clases en UML, en las que el objeto principal consiste o posee todos los objetos secundarios, y cuando el objeto principal se elimina, todos los objetos secundarios también deben eliminarse, porque ya no tiene sentido que esos objetos existan. El objeto padre de un agregado se llama *Aggregate Root*.
   ☐ Verdadero ☐ Falso

2. Las Entidades en DDD se encargan de validar que nuestro modelo sea consistente y se cumplan las invariantes. Pero, por otro lado, los *Value Objects* solo permiten encapsular el uso de tipos primitivos y no validan correctitud del modelo de dominio.
   ☐ Verdadero ☐ Falso

3. Los Comandos y los Eventos de Dominio no están ligados a ningún lenguaje de programación o tecnología. También describen muy bien el comportamiento del sistema, utilizando el Lenguaje Ubicuo y expresando la intención de los usuarios del sistema.
   ☐ Verdadero ☐ Falso

4. Los servicios de dominio en DDD se encargan de lidiar con los detalles de implementación, como por ejemplo el acceso a Base de datos, aplicando el Principio de Inversión de Dependencias de SOLID.
   ☐ Verdadero ☐ Falso

Desarrollo de Software
Primer Examen Parcial

2 Pregunta 2 (5 puntos)

El autor Craig Larman en su libro *UML y Patrones* expone en la página 318 lo siguiente: *"Merece la pena definir dos puntos de cambios [en el diseño de software]: a)* ***Punto de variación****: variaciones en el sistema actual, existente o en los requisitos. b)* ***Punto de evolución****: puntos especulativos de variación que podrían aparecer en el futuro en el sistema."*

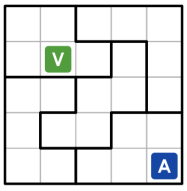

Figure 1: Tricolor

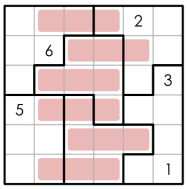

Figure 2: Consecutivos

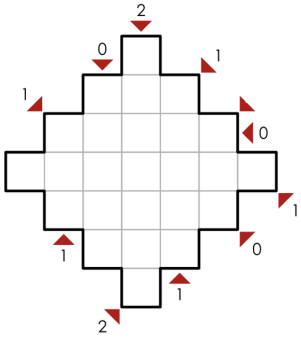

Figure 3: Círculos

Esta afirmación de Larman se relaciona con el **Principio *SOLID* OCP (Principio Abierto Cerrado)**, es decir, construir software que sea extensible y flexible para incorporar nuevos cambios o funcionalidades, incluso en tiempo de ejecución, sin tener que modificar el código fuente existente.

Pongamos en práctica estos principios. Considere los tres (3) juegos que se ven a la izquierda.

Tricolor: Cada casilla del arreglo de 5x5 debe estar pintada de un color: rojo, verde o azul. En cada fila, en cada columna y en cada región demarcada de cinco casillas debe hacer solo una casilla pintada de rojo y solo una casilla pintada de verde. Las casillas pintadas de rojo no pueden tocarse ni horizontal, ni vertical ni diagonalmente. Lo mismo se cumple para las casillas pintadas de verde.

Consecutivos: Ubique uno de los dígitos 1,2,3,4,5 y 6 en cada casilla vacía del arreglo, de tal manera que en cada fila, en cada columna y en cada región demarcada de seis casillas no se repita dígito. Las ternas de casillas vecinas sombreadas deben contener dígitos consecutivos, no necesariamente en orden. Además, no es posible formar más ternas de casillas sombreadas vecinas con dígitos consecutivos en ninguna fila.

Círculos: En este arreglo escalonado de 25 casillas se encuentran algunos círculos diseminados en su interior de tal forma que ningún círculo es vecino a otro ni horizontal, ni vertical, ni diagonalmente. Los dígitos que aparecen fuera del arreglo indican cuántos círculos se encuentran en la dirección indicada por cada flecha.

ATENCIÓN: Usted no debe resolver los juego o puzzles de arriba, solo debe modelar lo que se le pide a continuación. No hace falta resolver los puzzles para diseñar ni programar su software.

Desarrollo de Software
Primer Examen Parcial

[4 puntos] Usted debe construir el **Diagrama de clases UML** de su diseño tomando en consideración los siguientes lineamientos (no es opcional, es obligatorio cumplir con todos estos lineamientos):

1. Usted debe iniciar su diseño con estas 4 abstracciones o clases: Ficha, Casilla, Restricción y Juego.

2. La **Ficha** corresponde, según sea el juego, a un número, un color o un círculo.

3. La **Casilla** esta en relación de composición con una **Ficha** y también conoce cuáles son sus casillas vecinas.

4. La abstracción **Restricción** es una *clase abstracta* que representa cualquier restricción o regla en los juegos. Es evidente que una **Restricción** tiene una relación de agregación con las casillas que forman parte de la restricción o regla.

5. Su diseño debe contemplar la aplicación del **patrón de diseño Observador (*Observer*** de la Pandilla de los Cinco (GoF - del libro *Design Patterns* de Erich Gramma). Se debe aplicar este patrón cuando una casilla se le asigna una ficha (o cuando se elimina de ella una ficha), ya que todas las restricciones que observan esa casilla deben ser notificadas por el cambio que ocurre en la casilla. El patrón Observador se puede revisar en detalle en este link (en castellano) *Observer Pattern*.

6. La abstracción **Juego** solo se relaciona con objetos del tipo **Ficha**, **Casilla** y **Restricción**. Esta abstracción no sabe de detalles particulares. Por ejemplo, no sabe si las fichas son números, círculos o colores, no sabe nada sobre la disposición de las casillas.

7. **Esta prohibido modelar una abstracción Tablero. No hace falta.**

8. Su diseño debe cumplir con la cualidad de **Extensibilidad**, usando el **Principio Abierto Cerrado de SOLID**.

[1 punto] ¿ Cuántos objetos del tipo **Restricción** deben existir para construir el juego **Consecutivos** ?

[BONUS: 2 puntos extras]. De seguro utilizará Polimorfimo de Subtipos o Herencia en su diseño. Si logra mejorar su diseño aplicando correctamente Polimorfismo Paramétrico o Programación genérica se puede ganar 2 puntos extras

Desarrollo de Software
Primer Examen Parcial

3 Pregunta 3 (5 puntos)

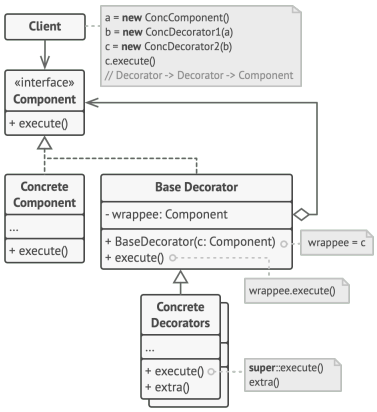

Figure 4: Decorador

Uno de los objetivos principales del curso de Desarrollo de Software es trabajar con **Arquitecturas Altamente Desacopladas**. Y esto se logra construyendo código altamente desacoplado (**Bajo Acoplamiento (*Loose Coupling*)**). Tal como expone Mark Seemann en su libro *Dependency Inyection*, la manera de construir software **mantenible** es mediante el **Bajo Acoplamiento**. Un códgio con bajo acoplamiento es **extensible** y, por ende, **mantenible**. Los *Patrones de Inyección de Dependencias (DI)* permiten generar código con bajo acoplamiento. En clase estudiamos el patrón de DI llamado ***Constructor Injection*** que consiste en definir las dependencias de una abstracción mediante el pase de paramétros en el constructor de la abstracción.

Por otro lado, existe el patrón de diseño *Decorator* que permite agregar responsabilidades adicionales o nuevas funcionalidades a un objeto o abstracción de manera dinámica, sin tener que modificar el código del objeto o abstracción. Es decir, nuevamente cumplimos con el **Principio SOLID Abierto Cerrado**. El patrón Decorador se puede revisar en detalle en este link (en castellano) *Decorator Pattern*. La *Figura 4* es un diagrama tomado de ese link.

(a) **Planteamiento del problema**.

1. Steve es un programador junior con poca experiencia en patrones de software. Hace seis meses, Steve construyó una aplicación similar a Instagram y en la primera versión se tenían las siguientes funcionalidades: publicar una foto, compartir una foto con otros, guardar una foto en favoritos, comentar una foto, dar *like* a una foto. Las fotos tienen asociado un usuario que es el propietario (*Owner*).
2. Steve tenía claro que todas estas acciones corresponden a *Comandos* porque generan un cambio de estado en las entidades de su modelo de dominio. Él creó una *interface* llamada ***IAction*** con la cual modela toda acción de tipo comando que ejecuta un usuario de la aplicación, es decir, cada comando implementa la *interface IAction*. Por ende, cada acción tiene asociado un usuario (que será el usuario que ejecuta la acción).
3. Al cabo de tres (3) meses, le piden a Steve que pueda agregar notificaciones a los propietarios de las fotos, cuando un usuario ejecuta cualquiera de los siguientes comandos: un *like*, envia un mensaje o comparte la foto. Steve tuvo que modificar

el código de cada una de las clases que representan cada comando. Es decir, su código no es extensible. De seguro, en el futuro, aparecerán nuevos requerimientos o cambios y Steve no quiere tener que modificar su código de nuevo.

4. Dos (2) meses después, le piden a Steve que pueda agregar trazabilidad cuando un usuario ejecuta cualquier comando, es decir, que se pueda registrar todo lo que hace un usuario en la aplicación. Nuevamente, Steve tuvo que cambiar el código de cada clase que representa un comando.
5. Steve pide ayuda para poner fin a esta historia: necesita aplicar correctamente patrones de software para que su código sea extensible y mantenible en el tiempo.
6. Además, que su código de la capa de dominio está acoplado a detalles de infraestructura, porque desde sus clases hace uso directo de librerías para el envío de notificaciones y para le registro de trazabilidad.

(b) **[5 puntos]** Usted debe construir el **Diagrama de clases UML** que solucione el problema descrito, utilizando el patrón DI de *Constructor Injection* y el patrón de diseño *Decorator*. Centremos nuestra atención en construir un diseño limpio y fácil de mantener frente a requisitos inconsistentes o cambiantes en el tiempo.
[Tip: Steve pensó bien al crear la interface IAction, usted debe definir decoradores para cada funcionalidad nueva que le pidieron en el tiempo.]

ATENCIÓN: Para que su respuesta sea válida debe detallar los métodos de sus clases en el diagrama UML con el pseudo-código mínimo necesario para justificar el uso de los patrones solicitados.

Desarrollo de Software
Primer Examen Parcial

4 Pregunta 4 (6 puntos)

Descripción del Problema (*Domain Problem*) Considere que estamos construyendo un Sistema de Manejo de Tickets de Soporte (*HelpDesk*) para Atención al Cliente:

1. Los Usuarios pueden crear Tickets cuando tienen algún problema.
2. Todo Usuario posee un correo electrónico. El correo debe estar escrito con su @ seguido de un nombre de dominio.
3. Los Agentes son los encargados de dar soporte. Son las personas que atienden los Tickets.
4. Los Agentes tienen Turnos de trabajo. Por ejemplo, pueden trabajar 5 días corrido y 2 de descanso.
5. Tanto los Usuarios como los Agentes pueden enviar Mensajes asociados a un Ticket. Es decir, toda la comunicación entre Usuarios y Agentes se registran como Mensajes asociados al Ticket.
6. A un mensaje se pueden adjuntar varios Archivos.
7. Cada Ticket tiene una prioridad: low, medium, high, or urgent.
8. Un Agente debe ofrecer una solución dentro de un límite de Tiempo Establecido (SLA/Acuerdo de Nivel de Servicio) que se basa en la prioridad del ticket, considerando también su Turno de trabajo.
9. Si el Agente no responde dentro del SLA, el Usuario puede escalar el ticket al Gerente del Agente.
10. El escalamiento de tickets, reduce el límite de tiempo de respuesta del Agente en un 33%.
11. Si el Agente no abrió un ticket escalado dentro del 50% del límite de tiempo de respuesta, se reasigna automáticamente a un Agente diferente.
12. Los Tickets se cierran automáticamente si el Cliente no responde a las preguntas del Agente dentro de los siete días.
13. Los tickets escalados no pueden ser cerrados automáticamente ni por el Agente, solo por el Usuario o el Gerente del Agente.

14. Un Usuario puede reabrir un Ticket cerrado solo si estuvo cerrado en los últimos siete días.

[6 puntos] Vamos a construir el Modelo de Dominio (*Domain Model*) empleando los **Patrones Tácticos** de ***Domain-driven Design***:

1. Identificar todas las **Entidades**, **Value Objects** y **Eventos de Dominio**
2. Identificar los **Servicios de Dominio**
3. Identificar el(los) **Agregado(s)** que pudiesen existir
4. Debe construir su Modelo de Dominio usando un **Diagrama de Clases UML**, utilizando *esteriotipos* para indicar cuáles clases son *Value Object*, *Aggregate Root*, *Aggregate*,*Entity*, *Domain Event*, *Domain Service*. De igual modo, puede representar un agregado dibujando fronteras para identificar las clases que componen el agregado. En la *Figura 5* se muestra un ejemplo de como debe quedar su Diagrama de clases UML con los esteriotipos de DDD
5. Es importante que modele todas las reglas de negocio (*business logic/business rules*) y las invariantes

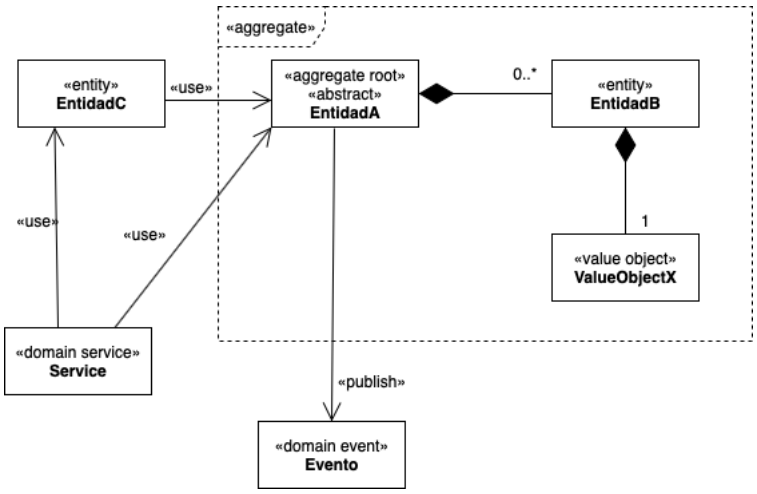

Figure 5: Diagrama de Clases con esteriotipos de DDD

ATENCIÓN: Recuerde que estamos modelando Capa de Dominio, es decir, exclusivamente el Modelo de Dominio. No estamos modelando ningún Detalle de Insfraestructura o de Implementación (Web, Mobile, UI, Databases, etc.)