Programación orientada a objetos. Segundo parcial 2º semestre 2008.

Martes 4 de noviembre de 2008.

Al igual que las demás instancias de evaluación este examen será calificado de acuerdo a la escala de letras vigente: D, R, B, MB, BMB, S o D, R, B, MB, S según la generación de cada alumno. Por eso las preguntas no tienen puntos.

Numeren las hojas.
Gracias y mucha suerte.

1.

Sea el siguiente código:

```
class Auto{}

class Ticket {
   private readonly Auto auto;
   public Auto Auto { get { return auto; } }
   public Ticket(Auto auto) {
      this.auto = auto;
   }
}
class Estacionamiento {
   /* código que debes completar */
   public Boolean EsValido(Ticket ticket) { /* código que debes completar */ }
   public Ticket Guardar(Auto auto, Fecha fechaActual) { /* código que debes completar */ }
   public Auto Sacar(Ticket ticket, Fecha fechaActual) { /* código que debes completar */ }
}
```

Y las siguientes condiciones:

a) No se puede guardar el mismo auto dos veces en el estacionamiento.
b) Solo se puede sacar un auto del estacionamiento si el ticket es válido para el estacionamiento.
c) Una vez guardado el auto, el estacionamiento entrega un ticket válido.
d) Al sacar el auto del estacionamiento, el ticket deja de ser válido.
e) El auto del ticket corresponde con el auto sacado del estacionamiento.
f) El auto asociado a un ticket siempre es distinto de nulo
g) No se puede guardar un auto nulo en el estacionamiento.

1.1. Identifica que condiciones corresponden a precondiciones, postcondiciones o invariantes.
1.2. Agrega el código necesario para que la clase Estacionamiento cumpla las precondiciones, postcondiciones e invariantes identificadas. Si no usas algun parámetro para hacerlo, no te preocupes, los necesitarás más adelante.
1.3. Programa el código necesario para verificar las condiciones usando Debug.Assert().

2.
Dada la siguiente clase:

```
class Tarifa {
   public Double Costo( Fecha desde, Fecha hasta) {
      /* código que mira la cara del cliente */
   }
}
```

2. Debes calcular el costo de guardar el auto en el estacionamiento cuando el auto se saca y marcarlo en el Ticket. ¿Utilizarías herencia o composición y delegación para reusar el código de Tarifa? ¿Por qué? Implementa la solución completa, reusando Tarifa y realizando todas las modificaciones que necesites en la clases del ejercicio 1.

3.
3.

Sea el siguiente código:

```
class Usuario                    { /* implementación completa */}
class AutorizadorUsuarios        { /* implementación completa y segura */ }
class Directorio                 { /* implementación completa */ }
class ObservadorNuevoRecurso {
   private ServicioDirectorio servicio;
   public ObservadorNuevoRecurso(ServicioDirectorio servicio) {
      this.servicio = servicio;
   }
   /* Invocado cuando el usuario desea agregar un Recurso */
   public void AgregarRecurso(String padre, String relativo) {
      servicio.AgregarRecurso(padre + "," + relativo);
   }
}
class ServicioDirectorio {
   private AutorizadorUsuarios autorizador = new AutorizadorUsuarios();
   private Usuario usuario;
   public void Login(Usuario usuario) {
      this.usuario = usuario;
   }
   public void AgregarRecurso(String nombre) {
      Directorio d = GetDirectorio(NombrePadre(nombre));
      if (d != null && autorizador.PuedeAgregarRecurso(usuario, d)) {
         d.AgregarRecurso(NombreHijo(nombre));
      }
   }
   public void AgregarUsuario(Usuario u) {
      autorizador.Agregar(u);
   }
   public void QuitarUsuario(Usuario u) {
      autorizador.Quitar(u);
   }
   public Directorio GetDirectorio(String absoluto) { /* */ }
   private String NombrePadre(String absoluto) { /* */ }
   private String NombreHijo(String absoluto) { /* */ }
}
```

Critíca el código anterior basándote en:

3.1. ISP (Principio de Segregación de Interfaces)

3.2. DIP (Principio de Inversión de Dependencias)

4.

Escribe claramente las modificaciones necesarias del código del ejercicio 3 para que éste:

4.1. Cumpla ISP, en caso que no lo cumpla (por más que sea una violación menor).

4.2. Cumpla DIP, en caso que no lo cumpla (por más que sea una violación menor).

5.

¿Cuáles son las características de un programa modular? ¿Cuál es el efecto del acoplamiento en un programa?

6.

Dado el siguiente código:

```
interface IAuto {
   void Avanzar();
}
abstract class Auto: IAuto {
   public void Avanzar() { }
   public abstract Boolean TieneCombustible();
}
class AutoMovimientoPerpetuo: Auto {
```

```
    public override Boolean TieneCombustible() { return true; }
    public virtual void Empujar(Double fuerza) { }
}
```

6.1. ¿En base a que se determina el método a ejecutar al usar encadenamiento estático? ¿Y al usar dinámico?

6.2. Dada la siguiente inicialización,

```
Auto auto = new AutoMovimientoPerpetuo();
IAuto iauto = auto;
AutoMovimientoPerpetuo automp = (AutoMovimientoPerpetuo) auto;
```

Indica el tipo de encadenamiento usado en cada caso:

a) iauto.Avanzar();
b) auto.Avanzar();
c) auto.TieneCombustible();
d) automp.TieneCombustible();
e) automp.Empujar(100.0);

7.

Sea el siguiente código:

```
public class Char {
    public virtual Boolean IsCrLf() { /* código altamente trivial */ }
    /* más código trivial */
}
public class KeyPressEventArgs : EventArgs {
    public virtual  Char KeyChar { get { /* get */ } set { /* set */ } }
}
public class Edit {
    private String text;
    public virtual String Text {
        get { return text; }
        set { SetText(value); }
    }
    protected virtual void SetText(String value) {
        text = value;
    }
    protected virtual void KeyPressed(KeyPressEventArgs e) {
        /* código indescifrable pero correcto */
    }
}
public class Calculator {
    public virtual Double Calculate(String expression) {
        /* código altamente inteligente */
    }
}
public class CalculatorEdit {
    public void Calculate() {
        /* COMPLETAR */
    }
    /* COMPLETAR */
}
```

La clase `Edit` representa un control de edición de un framework de interfaz de usuario. La propiedad `Text` representa el texto mostrado en el control. Cuando el usuario escribe en el control el framework usa la propiedad `Text` para cambiar el contenido del control. También el programador puede usar la propiedad `Text` para cambiar el contenido del control por código.

El método `void KeyPressed(KeyPressEventArgs e)` es usado por el framework cada vez que el usuario oprime una tecla en el control. La tecla oprimida está disponible en la propiedad `Char`

`KeyChar` del argumento `e` de tipo `KeyPressEventArgs`. Un método interesante de la clase `Char` para lo que viene a continuación es `Boolean IsCrLf()` que permite determinar si el carácter correspondiente a una tecla oprimida es el retorno de carro.

La clase `Calculator` tiene la responsabilidad de calcular el resultado de una expresión aritmética simple. Por ejemplo `Calculate("2+2")` retornaría 4. Si la expresión aritmética no es válida el método levanta una excepción del tipo `ArgumentException`.

7.1. Termina de programar la clase `CalculatorEdit` usando herencia y composición y delegación. La clase `CalculatorEdit` debe tener por lo menos los mismos métodos que la clase `Edit`. El método `void Calculate()` debe remplazar el contenido del control por el resultado de la expresión en él contenida si la expresión es válida y no hacer nada en caso contrario.

7.2. Termina de programar la siguiente clase usado herencia (puedes usar la clase programada en la parte 7.1). En los controles `AutoCalculatorEdit` el usuario puede oprimir el retorno de carro para reemplazar el contenido del control por el resultado de la expresión en él contenida si la expresión es válida y no hacer nada en caso contrario. La clase `AutoCalculatorEdit` debe tener por lo menos los mismos métodos que la clase `CalculatorEdit`.

```
public class AutoCalculatorEdit {
    /* COMPLETAR */
}
```