UTN.BA
UNIVERSIDAD TECNOLÓGICA NACIONAL
FACULTAD REGIONAL BUENOS AIRES

Centro de e-Learning

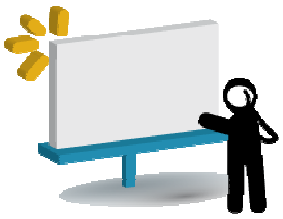
FUNDAMENTOS DE LA PROGRAMACIÓN

Centro de e-Learning SCEU UTN - BA.
Medrano 951 2do piso (1179) // Tel. +54 11 4867 7589 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning

MÓDULO 3 - UNIDAD 12

La programación en el entorno laboral actual

Centro de e-Learning SCEU UTN - BA.
Medrano 951 2do piso (1179) // Tel. +54 11 4867 7589 / Fax +54 11 4032 0148
www.sceu.frba.utn.edu.ar/e-learning



Presentación:

Con esta Unidad damos un cierre al curso, analizando algunos conceptos que serán de utilidad para comprender y aplicar los conocimientos vistos en el curso en un ambiente laboral actual.

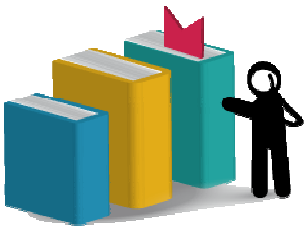
Se verán conceptos como los objetivos de la programación, los patrones de diseño y ciclo de vida de un proyecto, además de analizar y conocer los lenguajes de programación más difundidos en la actualidad.

Finalizamos el curso con un ejercicio integrador final que reúne los principales conceptos de la POO.

Objetivos:

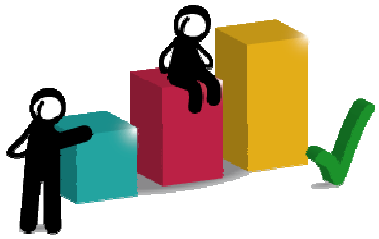
Que los participantes:

- Incorporen los principales conceptos que hacen a la aplicación de los contenidos vistos en el curso en un ambiente laboral actual
- Conozcan los principales lenguajes de programación y plataformas más difundidos en la actualidad: PHP, C#, HTML, Javascript, Ruby y Ruby on Rails, ASP y Java
- Conozcan conceptualmente los patrones de diseño
- Conozcan el concepto de ciclo de vida de un proyecto de desarrollo de software
- Analicen un ejercicio integrador (Material adicional)



Bloques temáticos:

1. Objetivos de la programación
2. Lenguajes modernos
3. Patrones de diseño – conceptos generales
4. Ciclo de vida de un proyecto de desarrollo de software
5. Ejercicio integrador final resuelto



Consignas para el aprendizaje colaborativo

En esta Unidad los participantes se encontrarán con diferentes tipos de actividades que, en el marco de los fundamentos del MEC*, los referenciarán a tres comunidades de aprendizaje, que pondremos en funcionamiento en esta instancia de formación, a los efectos de aprovecharlas pedagógicamente:

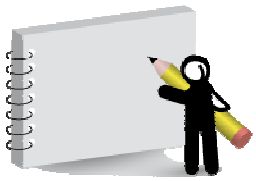
- Los foros proactivos asociados a cada una de las unidades.
- La Web 2.0.
- Los contextos de desempeño de los participantes.

Es importante que todos los participantes realicen algunas de las actividades sugeridas y compartan en los foros los resultados obtenidos.

Además, también se propondrán reflexiones, notas especiales y vinculaciones a bibliografía y sitios web.

El carácter constructivista y colaborativo del MEC nos exige que todas las actividades realizadas por los participantes sean compartidas en los foros.

* El MEC es el modelo de E-learning colaborativo de nuestro Centro.



Tomen nota

Las actividades son opcionales y pueden realizarse en forma individual, pero siempre es deseable que se las realice en equipo, con la finalidad de estimular y favorecer el trabajo colaborativo y el aprendizaje entre pares. Tenga en cuenta que, si bien las actividades son opcionales, su realización es de vital importancia para el logro de los objetivos de aprendizaje de esta instancia de formación. Si su tiempo no le permite realizar todas las actividades, por lo menos realice alguna, es fundamental que lo haga. Si cada uno de los participantes realiza alguna, el foro, que es una instancia clave en este tipo de cursos, tendrá una actividad muy enriquecedora.

Asimismo, también tengan en cuenta cuando trabajen en la Web, que en ella hay de todo, cosas excelentes, muy buenas, buenas, regulares, malas y muy malas. Por eso, es necesario aplicar filtros críticos para que las investigaciones y búsquedas se encaminen a la excelencia. Si tienen dudas con alguno de los datos recolectados, no dejen de consultar al profesor-tutor. También aprovechen en el foro proactivo las opiniones de sus compañeros de curso y colegas.

1. Objetivos de la programación

Si bien la programación es una actividad con una breve pero rica historia, existe poca uniformidad de opinión en muchas de las características que la definen. En lo que sí existe un criterio común es en la necesidad de generar código de calidad en todos los desarrollos, sea cual sea el paradigma o lenguaje de programación que se utilice.

Alguno de los parámetros que definen el concepto de software de calidad, son:

- Claridad
- Corrección
- Eficiencia
- Portabilidad

- **Claridad**. Este atributo de calidad implica que el programa sea lo más simple, claro y legible posible, ya que esto significa que el desarrollo se verá facilitado, así como su posterior mantenimiento. Es importante contemplar que la estructura del programa sea sencilla y coherente, así como cuidar el estilo en la edición; de esta forma se ve facilitado el trabajo del programador, tanto en la fase de desarrollo como en las fases posteriores de corrección de errores y modificaciones. Por esto nos referimos, por ejemplo, al uso de indentación de código, que facilita la visualización y seguimiento de bloques de código. Otro ejemplo serían los comentarios incluidos en los programas, tanto en los procesos/métodos como en las cabeceras de programas/clases. Es importante tener en cuenta que el programador que mantiene un código es probable que no sea el mismo que desarrolló ese código, lo que lleva a la necesidad de mantener una cortesía profesional y solidaridad entre colegas.

  Una práctica que debe ser desterrada es lo que se conoce comúnmente como “hardcode” o (españolizado) “hardcodeo” (que se pronuncia “jarcodeo”), que consiste en asignar valores a atributos o variables en forma programática.

A modo de simplificar los ejemplos, hemos realizado a lo largo del curso varias de estas acciones, que no deben realizarse en un ambiente laboral formal, ya que se muestran en este curso de esta forma para simplificar los ejemplos dados. Si a una variable se le asigna un valor, por ejemplo, nroDNI = 123 se debe pensar si ésta debería ser una variable o una constante. En el caso de creer que su valor podría cambiar (por más remoto que sea) se debe declarar como variable, caso contrario, constante. La forma de evitar esta práctica es enviar y recibir parámetros entre funciones y métodos.

- **Corrección**. Se define la correctitud de un programa cuando hace lo que se espera que haga, en relación a lo definido en etapas anteriores al desarrollo. Para determinar si un programa hace lo que debe, es muy importante especificar claramente qué debe hacer el programa antes de desarrollarlo y, una vez acabado, compararlo con lo que realmente hace. Para esto resulta de vital importancia tener dos elementos de para verificar estos puntos:
    - Casos de prueba: este concepto lo definimos en las primeras Unidades del curso y representan una especificación del camino principal y de todos los caminos alternativos posibles que tiene que cumplir un determinado software. Esto incluye el flujo de navegación, los resultados intermedios y el resultado final. Estos caminos deben ser coherentes con las especificaciones hechas con antelación al desarrollo.
    - Datos de prueba: resultan en un complemento para los Casos de prueba. También son llamados “juego de datos” y sirven para que el que esté probando la aplicación sepa qué datos ingresar en qué lugar para lograr cierto efecto. Por ejemplo, si sé que mi programa tiene que validar un DNI, un serie de datos de prueba para ingresar serían: un DNI válido, un DNI inválido, caracteres alfanuméricos, el número con o si punto separador de miles, etc.

- **Eficiencia**. Con esto se busca que un software además de ser “correcto” (según lo antes mencionado) lo haga gestionando de la mejor forma posible los recursos que utiliza, ya sea espacio en disco, memoria, tráfico de red, capacidad de procesador, etc. Normalmente en este aspecto, más allá del uso, también se hace referencia al

tiempo que tarda un programa en realizar una operación para la que ha sido creado y a la cantidad de memoria que necesita.

- **Portabilidad**. Un programa tiene la característica de portable cuando tiene la capacidad de poder ser ejecutado en una plataforma diferente a aquélla en la que se programó. Por plataforma entendemos indistintamente en este caso el sistema operativo y/o el hardware. Este atributo representa una característica muy importante para el software, ya que permite, por ejemplo, a un programa que se ha programado en Unix que pueda ser ejecutado también en una máquina con sistemas operativos Windows.

2. Lenguajes actuales

A continuación se presenta un muy breve resumen de los lenguajes que predominan en la actualidad en el mercado laboral local. Esta no pretende ser una lista excesivamente larga ni minuciosa, sino una breve referencia.

- **PHP**

Es lo que se conoce como lenguaje de programación de scripting del lado del servidor. Scripting (del inglés "script") ya que es un tipo de programación que se realiza por interpretación y no por compilación; y el lado del servidor porque esta interpretación se realiza en el contexto de un servidor de aplicaciones (un tipo especial de software de base). De esta forma, el código fuente se mantiene alojado en el servidor y es el intérprete el que se encarga de compilar instrucción por instrucción antes de ser accedido. Este proceso genera una especie de código objeto llamados opcodes.

PHP, nacido en 1995 y cuyo significado es HypertextPreprocessor ("Preprocesador de hipertexto" en inglés), es un lenguaje que ha logrado una gran popularidad en la programación de páginas y sitios web dinámicos, debido a la rapidez con que se ejecuta, su sintaxis (basada en C) y por el hecho de ser una herramienta de código abierto ("open source" en inglés).

A partir de la versión 5 incorpora elementos de programación orientada a objetos, con lo cual se convierte en un lenguaje híbrido.

- **C#**

C#, que se nombra "ce sharp" literalmente, es un lenguaje orientado a objetos creado por Microsoft como parte de su plataforma .NET, en la que logró destacarse por encima de otros lenguajes que incorpora esta plataforma, en esencia debido a su sintaxis derivada de C y C++. Una de sus ventajas radica en que combina el control a bajo nivel de estos lenguajes con las posibilidades de la programación de lenguajes simples de alto nivel.

El hecho de parecerse a C/C++ se debe principalmente porque surge como la principal competencia de Java, también con similitudes sintácticas con aquellos dos lenguajes de programación, lo que facilitaría la "migración" por parte de los programadores de una tecnología a otra.

Muchas veces se puede ver que se hace referencia a la “programación en .NET”. Conceptualmente esto sería erróneo o, al menos, poco claro, ya que .NET en sí no es un lenguaje de programación sino una completa plataforma (o “framework”) de desarrollo de software, que incluye varios lenguajes de programación además de software de base. Los lenguajes de programación que implementan el framework .NET más difundidos son C# y ASP.NET, aunque no son los únicos: existen adaptaciones de otros lenguajes existentes como A# (basado en Ada), Visual Basic.NET, Visual J# .NET (basado en Java), varias implementación de Cobol (como NetCobol), entre muchos otros. Estos lenguajes tienen en común que, si bien la sintaxis del código fuente es similar a los lenguajes originales, al ser estos compilados generan un mismo código objeto. Dicho de otro forma, no importan en qué lenguaje que implemente .NET se programe, al ser este compilado, el resultado es el mismo.

- **HTML**

En principio, no deberíamos estar haciendo referencia a HTML en este curso, ya que en sentido estricto no es un lenguaje de programación. HTML significa “HiperTextMarkupLanguage” o Lenguaje de marcado de hipertexto. Decimos que no es un lenguaje de programación ya que no incorpora capacidades como estructuras condicionales o repetitivas. Tampoco tiene tipos de datos. Más bien es lo que se denomina un “lenguaje de tags”, siendo un tag (o etiqueta) un elemento determinado entre signos “<>”. El HTML está regido y estandarizado por un consorcio internacional que define que elementos o etiquetas son válidos. Cabe aclarar que TODO lo que vemos en la web, en un navegador, en una página es HTML. Se puede generar de diversas maneras, pero el resultado es el mismo: una serie de instrucciones predefinidas entre “<>” que son interpretadas por un navegador. Por ejemplo, para mostrar un botón en una página web deberíamos recurrir a la instrucción:

<input type=”button”/>

Como podemos ver, poco tiene que ver con las estructuras que vimos anteriormente.

El HTML se vuelve realmente poderoso cuando se lo combina con JavaScript (lenguaje que veremos a continuación), CSS (que facilita los aspectos estéticos y cosméticos de las páginas web) y lenguajes de programación orientados a web (como también veremos a continuación).

- **JavaScript**

Es un lenguaje de programación interpretado, que se ejecuta del lado del cliente. Esto último significa que el código fuente no se encuentra ubicado en un servidor, sino que es compilado instrucción por instrucción (interpretado) dentro de otro programa. En el caso de JavaScript, el cliente es un navegador (o "browser", en inglés), siendo los más difundidos en la actualidad Chrome, Firefox, Internet Explorer y Safari, entre otros. La forma de verificar esto podría ser recurrir a cualquiera de estos programas y buscar la opción de "Ver fuente" ("View source"). De esta forma, se puede ver el código HTML y JavaScript que incluye una determinada página. Este mismo código es que el viaja desde un servidor, llega al navegador y este lo "interpreta" mostrando una página web.

Este lenguaje se transformó en el lenguaje de programación de lado del cliente para web más difundido, habiendo prevalecido sobre el VBScript de Microsoft, convirtiéndose en un estándar de facto. Su sintaxis es muy similar a Java (de allí parte de su nombre, que se complementa con el "script" haciendo referencia a que es un lenguaje interpretado).

Su uso principal se da en dos aspectos: le da dinamismo o "movimiento" a las páginas web mejorando el diseño, como complemento al HTML (por ejemplo, para crear menús desplegables) y permite agregar un nivel básico de funcionalidades "de negocio" (por ejemplo, al permitir validar el formato de una fecha). A esto se suman alguna tendencias actuales, relativamente reciente, como AJAX, que permite el envío asincrónico de mensajes entre el cliente y el servidor, con lo que se logra, por ejemplo, que se valide un campo o rellene una tabla sin que se tenga que recargar toda la página. Correos web como GMail o Yahoo! aprovechan en gran medida estas capacidades.

- **Ruby y Ruby on Rails**

Ruby on Rails (conocido como RoR o simplemente Rails) es un framework de desarrollo de aplicaciones web de código abierto programado en el lenguaje de programación Ruby. Surgió en el 1995 y vio demorada su difusión y masificación dado que inicialmente toda la documentación de encontraba escrita en japonés, lo que dificultaba su aprendizaje, problema que con el tiempo se fue mitigando. Su principal impulso llegó de la mano de RoR, que toma ideas de otros lenguajes (como Pearl y el mencionado PHP) para acelerar la programación al concentrar todo le desarrollo en un único lenguaje y simplificar algunos temas complejos, como el uso de bases de datos.

- **Java**

Si bien se hizo referencia a este lenguaje en numerosos apartados del curso, podremos hacer algunos aportes más. Podemos decir que hoy el término “Java” tiene dos interpretaciones: referencia directa al lenguaje de programación original o una mención a una plataforma, o familia de productos, en forma similar a lo mencionado antes para .NET. Esto se debe a la gran difusión que tuvo el lenguaje y por el hecho de ser open source, lo que facilitó la proliferación de comunidades que ampliaron e hicieron crecer el lenguaje en prestaciones y posibilidades de uso. Por eso, al hablar de programación web en Java, en realidad se puede estar haciendo referencia a JSP (“Java Server Pages”) o JSF (“Java Server Faces”). El primero fue uno los primeros medios que permitieron embeber código Java dentro de código HTML para darle dinamismo y mayor prestaciones a las páginas web. El segundo es el actual estándar de programación web de Java. Ninguno de los dos son lenguajes o tienen instrucciones: son especificaciones o, dicho de otra forma, estándares, documentos que dicen cómo se debe programar.

3. Patrones de diseño – conceptos generales

Un patrón de diseño es una propuesta de solución a un problema determinado, y que tiene que cumplir con ciertas pautas: debe estar comprobado que funciona y que es reutilizable. Esto significa que existe una experiencia previa que afirma que la solución ya ha resuelto problemas similares anteriormente en forma efectiva y que es aplicable a diferentes situaciones (similares entre sí).

Haciendo un poco de historia, nos podemos remitir al año 1979 cuando el arquitecto austríaco Chistopher Alexander publicó su obra “The Timeless Way of Building” (“El modo intemporal de construir”) donde propone una serie de lineamientos genéricos destinados a la construcción de edificios con el fin de mejorar la calidad de las obras.

Sostenía que ante la repetición de un mismo problema una y otra vez, se podría proponer una solución al mismo por única vez y que ésta, al ser replicada, aceleraría y abarataría los costos de producción. Con este fin, junto a sus colegas, publicó una serie de obras con el objetivo de documentar en forma unificada la experiencia de varias generaciones de conocimiento y experiencia arquitectónica.

Tomando el trabajo de Alexander como base, a fines de los ’80 los norteamericanos Cunningham y Beck (padre también de las metodologías ágiles) publicaron la primera divulgación articulando los patrones de diseño arquitectónicos al desarrollo de software orientado a objetos. En el artículo presentaron los cinco primeros patrones de interacción hombre-ordenador.

Posteriormente, ya a principios de la década de los ’90 es cuando los patrones de diseño de software obtienen una relevancia significativa a partir de la publicación del libro “DesingPatterns” (Patrones de diseño), del grupo conocido por “Gang of four” (algo así como “La banda de los cuatro”) formada por Gamma, Helm, Johnson y Vlisides, donde compilaban 23 patrones de diseño comunes.

Los componentes principales que forman un patrón se pueden resumir en: nombre, problema o situación en la que se aplica, solución genérica y las consecuencias de su implementación en términos de pros, contras, costos, beneficios y riesgos. Estos pueden ir acompañados de un ejemplo implementado en algún lenguaje de programación OO (poco recomendable, ya que pierde la generalidad y acota la solución a un simple ejemplo programático) y/o (idealmente) como documentación gráfica basada en UML.

Los tipos de patrones se corresponden las siguientes taxonomías:

- **Patrones creacionales o de creación**: se ocupan de la creación, inicialización y configuración de los objetos
- **Patrones estructurales o de estructura**: se enfocan en la separación de responsabilidades en capas, principalmente dividiendo la programación de la interfaz de usuario con la lógica de negocio, por ejemplo y cómo estos se agrupan.
- **Patrones de comportamiento**: se ocupan de la forma en que se comunican (envían mensajes) los objetos entre sí.

Los patrones de diseño nos deben servir como referencia. Para saber que probablemente "alguien ya estuvo allí". Deben ser nuestra fuente de consulta permanente, ya que nos van a facilitar las respuestas a estas grandes e importantes preguntas que todo desarrollador debe hacerse antes de empezar a programar:

- ¿Alguien ya resolvió este problema antes?
    - Si: busquemos el patrón que lo resuelve
        - Si no existe el patrón, cabe preguntarnos: ¿Es un problema con el que puede encontrar otra persona? ¿Tengo forma de obtener una solución que me sirva a mí y a otra persona?
            - Resolvemos el problema y (si aplica) lo documentamos y difundimos, así ayudaremos a otro colega que potencialmente se pueda encontrar con este mismo problema en el futuro.

4. Ciclo de vida de un proyecto de desarrollo de software

Los desarrollos de software tienden a enmarcarse en proyectos, entiendo por esto como una serie de actividades únicas e irrepetibles que tienen como finalidad obtener como resultado un producto o servicio. Se dice que son actividades únicas e irrepetibles ya que de otra forma se podría interpretar como que esas actividades ya fueron realizadas en el pasado por otras personas, en el marco de otro proyecto, por el que se obtuvo el mismo producto y servicio que estamos buscando crear. Otra forma de decir "reinventar la rueda".

Estos proyectos suelen dividirse en fases o etapas, en los que la mayoría no tienen que ver con la programación, sino con darle el contexto o generar los artefactos necesarios para poder programar posteriormente. De la misma forma, no todo termina en la programación, ya que hay actividades bien diferenciadas que se realizan posteriormente.

Normalmente, un proyecto de software tiene las siguientes fases o etapas:

- **Análisis Funcional (o simplemente Análisis)**: en esta fase se evalúa la viabilidad del proyecto y lo que se busca lograr, además de definir los objetivos del proyecto, se recopila y catalogan los requerimientos del cliente y se ponderan las restricciones y condiciones. Normalmente de estas actividades surgen varios artefactos, siendo el más importante para el programador los relacionados a los requisitos funcionales del proyecto, que suelen estar documentados como requerimientos o casos de uso.
- **Diseño Técnico**: es el punto intermedio entre los requerimientos y la programación. Básicamente consiste en pasar a un documento técnico de alto nivel los documentos obtenidos en la fase anterior, a través de gráficos (como diagramas de flujo o UML, siendo este un estándar para documentar la POO en forma gráfica). Cabe destacar la diferencia entre "Diseño Técnico" y "Diseño Gráfico", siendo esta una actividad que puede estar tanto en esta fase como en la anterior de "Análisis". Esta es la documentación principal que recibirá el programador y es la que lo orientará para realizar un desarrollo correcto.

- **Desarrollo**: en esta fase se realizan todas las actividades de programación propiamente dichas. Se generan los programas, se compilan y se prueban. Estas pruebas a nivel de programador suelen llamarse “pruebas unitarias”, no siendo estas las únicas pruebas que se deban realizar en esta fase.
- **Pruebas**: normalmente estas actividades son realizadas por profesionales llamados “testers”, que se encuentran adiestrados en diferentes tipos de pruebas (funcionales, de integración, de aceptación, etc.) y técnicas de test (de “caja negra” o “caja blanca”, etc.). El objetivo es obtener un producto o servicio con un número de fallas reducido, tendiendo a cero, al menos sobre los errores conocidos (ya que es importante ser consciente que con el uso se van a encontrar nuevas fallas). Las fallas detectadas deben ser reportadas a quien corresponda (un responsable del proyecto, un programador, un equipo especial dedicado a resolver problemas u otro).
- **Despliegue**: es el proceso en el cual se toma el software desarrollado y probado y se lo pone a disposición de los usuarios, también conocido como “puesta en vivo” o “pasaje a producción” y otras variantes.
- **Mantenimiento**: esta fase se orienta a realizar distintos tipos de mantenimientos (correctivo, que soluciona problemas detectados; evolutivos, que agrega funcionalidades; preventivos, que busca entrar problemas antes de que surjan; etc.).

Estas fases pueden ser agrupadas y ejecutadas de diferentes formas, ya sea una a continuación de otras en forma única, o en forma solapada, o una a continuación de otra pero varias veces, etc. Más allá de cómo se agrupen, prácticamente siempre se realizan estas actividades de análisis-diseño-desarrollo-prueba-despliegue-mantenimiento. Tal vez juntando alguna de estas fases o dividiéndolas, pero siempre son necesarias.

Estas diversas formas de agrupar y ejecutar estas fases, normalmente se las conoce como:

- **Cascada**: ejecutado en forma estricta, una fase sólo puede ser iniciada cuando finaliza la anterior. Los criterios de finalización (aceptación, por parte del cliente) suelen llamarse puerta o “gate”, en inglés, siendo este el término mayormente difundido y utilizado. La principal crítica que recibe este modelo es su falta de flexibilidad, ya que si se detecta algún cambio o inconveniente, suele ser necesario llevar complejas gestiones de cambio con el cliente, que suelen entorpecer, posponer o incluso cancelar proyectos.

- **Incremental**: este tipo de modelo sugiere realizar un conjunto de actividades en forma repetitiva, desarrollando porciones pequeñas de código, con el objetivo de ir aumentando la funcionalidad incorporando el desarrollo de nuevos requerimientos. Al finalizar cada ciclo se obtiene un conjunto de funcionalidades que pueden servir para detectar nuevos requerimientos o problemas de planteo en forma temprana.

- **Ágil**: los modelos ágiles de desarrollo de software se basan en procesos iterativos como base para lograr un ciclo más ligero, centrado en las personas por sobre las herramientas y procedimientos formales, típicos de los modelos tradicionales. En este caso resulta fundamental la retroalimentación en lugar de planificación, como principal mecanismo de control. Esta retroalimentación se logra al liberar en períodos cortos de tiempo nuevas versiones del software que pueda ser visto y probado por el cliente.

- **Espiral**: combina características de los modelos anteriores, entre otros, aunque poniendo énfasis en el análisis de riesgos, haciendo hincapié en las condiciones de las opciones y limitaciones para facilitar la reutilización de software. Una de las críticas que recibe este modelo son las actividades adicionales que deben realizar los programadores, más allá de las de programación exclusivamente.

5. Ejercicio integrador final resuelto

A continuación vamos a desarrollar un ejemplo integrador, similar al que nos podríamos encontrar en un ambiente laboral.

En la mayoría de los casos, hablando siempre de compañías en las que desarrolla software comercial (en contraposición con sistemas de alta complejidad, de medicina o de investigación) se suelen programar “aplicaciones de gestión”, nombre general que se le da al software de propósito general, que tiene como finalidad administrar o gestionar distintos tipos de recursos, sean pólizas de seguros, cuentas bancarias, alumnos o despacho de automóviles en una automotriz.

Estas aplicaciones suelen estar plagadas de lo que se llama ABM o ABMC, cuyo significado es Alta-Baja-Modificación y Consulta, haciendo referencia a las principales acciones que se realizan con los datos y a los procesos que las realizan.

Podemos citar el ejemplo de una póliza de seguro, que podríamos contratar para nuestro vehículo:

- Se dan de alta los datos de una persona
- Luego se dan de alta los datos del vehículo
- Se realiza una consulta para verificar la cotización
- Se modifican los datos para consultar un plan más conveniente
- Se borran (o dan “de baja”) las cotizaciones anteriores
- Y (para resumir) posteriormente se disparan una serie de procesos que realizan validaciones de negocio, entre otras acciones, y terminan emitiendo la póliza.
- O, simplemente, el cliente decide no contratar la póliza, quedando parte de sus datos dados de alta como histórico.

Por procesos (o “procesos batch”), entendemos a un algoritmo que realiza una cantidad variada de acciones (pueden ser de AMBC) pero que no suelen requerir de interacción con el usuario de la aplicación. Estos procesos suelen lanzarse o dispararse por pedido (como en el caso de la emisión de la póliza) o se establece para que se ejecuten en forma periódica (por ejemplo, se podrían borrar los datos históricos de los cliente sin pólizas una vez por mes o por semana). En ambos casos, estos procesos son independientes y no requieren de la interacción o el ingreso de datos por parte de un usuario u operador del sistema.

Por otro lado, en el ejemplo que veremos a continuación, omitiremos algunos detalles de implementación, relacionadas a esas actividades ya resueltas que habíamos nombrado en Unidades anteriores. Concretamente, evitaremos detalles de la implementación relacionada a la lógica de uso de una base de datos en la cual estaríamos guardando, modificando, eliminando y consultando datos (la que veremos en el ejemplo con el nombre de “BaseDeDatos”). Si bien no es tema de este Curso abundar en detalles sobre las bases de datos, diremos simplemente que son la forma estándar de guardar datos de una forma conocida, aceptada y difundida.

En el siguiente ejemplo veremos cómo crear una Póliza de distintos tipos.

```
Clase publica Persona
    privado String nombre nuevaInstancia String()
    privado String nroDNI nuevaInstancia String()

    metodo publico String obtenerNombre()
        retornar: nombre
    fin metodo

    metodo publico escribirNombre(String nombreNuevo)
        nombre = nombreNuevo
    fin metodo

    metodo publico String obtenerNroDNI ()
        retornar: nroDNI
    fin metodo

    metodo publico escribirNroDNI (String nroDNINuevo)
        nroDNI= nroDNINuevo
    fin metodo
fin clase


Interface publica Cotizador
    metodo publico Poliza calculaPrecio(Poliza)
fin interface
```

```
Clase publica CotizadorAutos implementa Cotizador
    metodo publico Poliza calculaPrecio(Poliz apoliza)
        Float precio nuevaInstancia Float()
        si ((PolizaAutos)poliza).obtenerAnio() < 2010 entonces
            precio = 100
        sino
            precio = 150
        fin si
        poliza.escribirPrecio(precio)
        retornar: poliza
    fin metodo
fin clase


Clase publica CotizadorHogar implementa Cotizador
    metodo publico Poliza calculaPrecio(Polizapoliza)
        Float precio nuevaInstancia Float()
        precio = 0
        si ((PolizaHogar)poliza).obtenerMetros() >= 150 entonces
            precio = 55
        sino
            precio = 45
        fin si
        poliza.escribirPrecio(precio)
        retornar: poliza
    fin metodo
fin clase
```

```
Clase publica PolizaABM
    metodo publico creaPoliza(Poliza poliza)
        //creo un supuesto objeto encargado de manejar la base de datos
        BaseDeDatos db nuevaInstancia BaseDeDatos()
        db.guardarEnLaBaseDeDatos(poliza)
    fin metodo

    metodo publico modificaPoliza(Poliza polizaAModificar)
        //creo un supuesto objeto encargado de manejar la base de datos
        BaseDeDatos db nuevaInstancia BaseDeDatos()
        polizaAModificar = db.obtenerDeLaBaseDeDatos(poliza.obtenerNroPoliza())
        polizaAModificar.escribirPersona(poliza.obtenerPersona())
        polizaAModificar.escribirPrecio(poliza.obtenerPrecio())
        db.guardarEnLaBaseDeDatos(polizaAModificar)
    fin metodo

    metodo publico eliminaPoliza(Poliza poliza)
        //creo un supuesto objeto encargado de manejar la base de datos
        BaseDeDatos db nuevaInstancia BaseDeDatos()
        db.eliminarDeLaBaseDeDatos(poliza)
    fin metodo
fin clase


Interface publica Poliza
    metodo publico Integer obtenerNroPoliza()
    metodo publico escribirNroPoliza(Integer)
    metodo publico Persona obtenerPersona()
    metodo publico escribirPersona(Persona)
    metodo publico Float obtenerPrecio()
    metodo publico escribirPrecio(Float)
fin interface
```

```
Interface publica Poliza
    metodo publico Integer obtenerNroPoliza()
    metodo publico escribirNroPoliza(Integer)
    metodo publico Persona obtenerPersona()
    metodo publico escribirPersona(Persona)
    metodo publico Float obtenerPrecio()
    metodo publico escribirPrecio(Float)
fin interface


clase publica PolizaHogar heredaDe PolizaRamosGenerales
    privado Float metros nuevaInstancia Float()

    metodo publico PolizaHogar ()
        tipo = "HOGAR"
    fin metodo

    metodo publico Float obtenerMetros ()
        retornar: metros
    fin metodo

    metodo publico escribirMetros(Float metrosNuevo)
        metros = metrosNuevo
    fin metodo

    metodo publico costo()
        costo = 500
    fin metodo
fin clase
```

```
clase publica PolizaAutos heredaDe PolizaRamosGenerales
    privado Integer anio nuevaInstancia Integer()

    metodo publico PolizaAutos()
        tipo = "AUTO"
    finmetodo

    metodo publico Integer obtenerAnio()
        retornar: anio
    finmetodo

    metodo publico escribirAnio(Integer anioNuevo)
        anio = anioNuevo
    finmetodo

    metodo publico costo()
        costo = 1000
    fin metodo
fin clase
```

```
Clase publica abstracta PolizaRamosGenerales implementa Poliza
    privado Integer nroPoliza nuevaInstancia Integer()
    privado Persona persona nuevaInstancia Persona()
    privado Float precio nuevaInstancia Float()
    privado String tipo nuevaInstancia String()
    privado Float costo nuevaInstancia Float()

    metodo publico Integer obtenerNroPoliza()
        retornar: nroPoliza
    fin metodo

    metodo publico escribirNroPoliza(Integer nroPolizaNueva)
        nroPoliza = nroPolizaNueva
    fin metodo

    metodo publico Persona obtenerPersona()
        retornar: persona
    fin metodo

    metodo publico escribirPersona(Persona personaNueva)
        persona = personaNueva
    fin metodo

    metodo publico Float obtenerPrecio()
        retornar: precio
    fin metodo

    metodo publico escribirPrecio(Float precioNuevo)
        precio = precioNuevo
    fin método

    metodo publico abstracto costo()
fin clase
```

```
Clase publica ComprarPoliza
    metodo publico ComprarPoliza (Integer anio, String nombre, String DNI)
        Persona persona = crearPersona(nombre, DNI)                 //1
        Poliza polizaAuto nuevaInstanciaPolizaAuto()                //2
        //creo un supuesto objeto encargado de manejar la base de datos
        BaseDeDatos dbnueva InstanciaBaseDeDatos()                  //3
        //busco en la supuesta base de datos el próximo número disponible
        polizaAuto.escribirNroPoliza(db.obtenerNuevoNroPoliza())    //4
        //asigno la Persona a la póliza
        polizaAuto.escribirPersona(persona)                         //5
        polizaAuto.escribirAnio (anio)
        //creo instancia del cotizador para autos
        Cotizador cotizadorAutos nuevaInstancia CotizadorAutos()    //6
        polizaAutos = cotizarPoliza(polizaAutos, cotizadorAutos)    //7
        guardarPoliza(polizaAutos)                                  //8
    fin metodo

    metodo privado Poliza cotizarPoliza(Poliza poliza, Cotizador cotizador)
        // mando la póliza para que calcule el precio. Me devuelve la misma póliza
        // pero con el agregado del precio, por lo que "piso la póliza anterior" (sin precio)
        retornar: cotizador.calculaPrecio(poliza)
    fin metodo
```

(sigue)

```
    metodo publico ComprarPoliza (Float metros, String nombre, String DNI)
        Persona persona nuevaInstancia Persona()
        persona = crearPersona(nombre, DNI)
        Poliza polizaHogar nuevaInstancia PolizaHogar()
        //creo un supuesto objeto encargado de manejar la base de datos
        BaseDeDatos db nuevaInstancia BaseDeDatos()
        //busco en la supuesta base de datos el próximo número disponible
        polizaHogar.escribirNroPoliza(db.obtenerNuevoNroPoliza())
        //asigno la Persona a la póliza
        polizaHogar.escribirPersona(persona)
        polizaHogar.escribirMetros(metros)
        //creo instancia del cotizador para hogar
        Cotizador cotizadorHogar nuevaInstanciaCotizadorHogar()
        polizaHogar = cotizarPoliza(polizaHogar, cotizadorHogar)
        guardarPoliza(polizaHogar)
    fin metodo

    metodo privado Persona crearPersona(String nombre, String DNI)
        Persona persona nuevaInstancia Persona()
        persona.escribirNombre(nombre)
        persona.escribirDNI(DNI)
        retornar: persona
    fin metodo

    metodo privado guardarPoliza(Poliza poliza)
        PolizaABM abm nuevaInstancia PolizaABM()
        abm.creaPoliza(poliza)
    fin metodo
fin clase
```

El ejemplo comienza en “ComprarPoliza”. A continuación, paso a paso la explicación de cada instrucción. Para facilitar la comprensión se numeraron las líneas de uno de los métodos de esta clase.

0. Para crear una instancia de “ComprarPoliza” voy a tener que usar alguno de sus dos constructores: uno recibe como parámetro un Integer y dos Strings. El otro recibe un Float y dos Strings. Dependiendo del constructor que elija, es lo que voy a estar creando: si le envió como parámetros un Integer y dos Strings va a ser un auto, si envío un Float y dos String va a ser un hogar. Este juego con los constructores es un ejemplo de sobrecarga de métodos (métodos constructores en este caso).

1. Se crea una instancia de Persona. Se invoca a un método interno privado, donde se crea la instancia y se cargan los valores de nombre y DNI según los métodos públicos de “Persona” (“escribir...”). Los métodos “obtener...” en este ejemplo no se utilizan, pero siempre deben estar presentes. Esta clase tiene como única responsabilidad contener los datos de una persona, que en este caso sería un cliente.

2. Se crea un objeto de tipo “Poliza” que es instancia de “PolizaAuto”. “Poliza” es una interface que es implementada por una clase abstracta llamada “PolizaRamosGenerales”, que contiene todos métodos que escriben y leen sus atributos (“nropoliza”, “persona” y “precio”). Al ser creado como una “PolizaAuto” (ver esa clase) su constructor por defecto asigna al atributo “tipo” el valor "AUTOS". Para el atributo “tipo” no haremos los métodos de "escribir…" y "obtener…" ya que en este constructor es el único lugar en cual se modifica este atributo.

3. En este punto se crea una instancia de una clase ficticia “BaseDeDatos” (como se explico anteriormente) sobre la cual omitiremos la implementación ya que la mayoría de los lenguajes ya provee librerías (clases auxiliares) que realizan estas actividades.

4. Con este objeto ficticio obtengo cual es el próximo número de póliza disponible (esto se conoce normalmente como "ID" y se pronuncia "aidí", por "identificador". Se trata en este caso de un número único e irrepetible que identifica una póliza en particular. En este punto, tengo cargada en la póliza nueva su identificador único.

5. A la misma póliza le asigno la persona que creé y cargue anteriormente en el punto 1.

6. Se crea un objeto de tipo “Cotizador” que es instancia de “CotizadorAutos”.

7. En este punto se sobrescribe ("pisa") la “poliza” actual por la misma “poliza” pero con el precio cargado. Para esto, se hacen las siguientes acciones:
    a. Llamar al método interno privado “CotizarPoliza”. A este método le envío la póliza actual y el tipo de cotizado que quiero que use.
    b. Dentro de este método voy a retornar la póliza cotizada (con precio). Para esto voy a usar el “Cotizador” que en tiempo de ejecución puede ser una instancia de “CotizadorAutos” o “CotizadorHogar”. En tiempo de compilación no sé de qué tipo concreto va a ser el “Cotizador" que voy a usar. Si el parámetro de tipo “Poliza” llamado “poliza” que recibe este método es una instancia de “PolizaAutos”, se va a invocar al “CotizadorAutos”, si es del tipo “PolizaHogar” va a invocar al “CotizadorHogar”. Este es un ejemplo de polimorfismo.
    c. Es importante prestar atención a una expresión compleja en el método “calculaPrecio”, tanto en “CotizadorHogar” como “CotizadorAutos”:
        i. “((PolizaAutos)poliza)” esto se usa para invocar los métodos “obtenerMetros” y “obtenerAnio”. Es un ejemplo de casting o casteo, ya que se está forzando al objeto “Poliza” que se comporte como una “PolizaAutos” o “PolizaHogar” dependiendo del caso. Esto se debe a que la interfaz “Poliza” no sabe si va ser del tipo “auto” u “hogar”, por lo que se debe forzar la conversión.

8. Finalmente, se invoca a un método privado interno que se va a encargar de guardar en la base de datos la póliza. Para esto se crea una instancia de "PolizaABM" y se invoca al método "crearPoliza". Esta clase tiene otros métodos que no se utilizaron en este ejemplo, pero que se explican a continuación:
    a. "modificaPoliza": lo primero que hace es obtener desde la base de datos la póliza que se quiere modificar, a través de un supuesto método "obtenerDeLaBaseDeDatos" que buscaría en la base de datos una póliza según el identificador al enviar como parámetro "poliza.obtenerNroPoliza()". Luego se modifican los datos de la "persona" y el "precio" (el identificador nunca debe modificarse y por lógica, en este caso el "tipo" tampoco, ya que sería otra póliza en ese caso). Finalmente se guarda la póliza en la base de datos.
    b. "eliminaPoliza": en este caso habría un método que borra una póliza en base al objeto del tipo "Poliza" que llega como parámetro.

Finalmente, en la misma clase, vamos a encontrar el constructor "ComprarPoliza (Float metros, String nombre, String DNI)". Como dijimos en el punto 0, cuando creamos una instancia de la clase ComprarPoliza y se utiliza este constructor sobrecargado, lo que vamos a estar haciendo es comprar una "Póliza de Hogar", ya que los parámetros que recibe este constructor son los necesarios para crear un Póliza de este tipo. El resto de los pasos que se realizan son similares a los de la "Póliza de Autos", siendo la lógica similar y habiendo puesto en clases internas las funcionalidades que se comparte entre ambos constructores ("cotizarPoliza", "crearPersona", "guardarPoliza") logrando, de esta forma, escribir estas instrucciones una única vez y siendo utilizadas en más de una ocasión.

Lo que vimos

- Los principales conceptos que hacen a la aplicación de los contenidos vistos en el curso en un ambiente laboral actual
- Los principales lenguajes de programación y plataformas más difundidos en la actualidad: PHP, C#, HTML, Javascript, Ruby y Ruby on Rails y Java
- Conceptos los patrones de diseño
- Concepto de ciclo de vida de un proyecto de desarrollo de software
- Ejercicio integrador