Manual de prácticas de Técnicas de Programación

Elaborado por:	Revisado por:	Autorizado por:	Vigente desde:
M.F. Gabriel Hurtado Chong **M.A. Luis Yair Bautista Blanco** **M.I. Jorge Armando Rodríguez Vera** **Ing. Miguel Serrano Reyes** **M.I. Gersaín Barrón Velázquez** **Ing. Angelo Sandoval Villegas** **Ing. María Fernanda Merino Morales**	**M.F. Gabriel Hurtado Chong** **M.I. Jorge Armando Rodríguez Vera** **Ing. Miguel Serrano Reyes**	**Dr. Francisco Javier Solorio Ordaz**	**24 de enero de 2020**

Índice de prácticas

Práctica #1

Metodología de la programación orientada a objetos

1. Seguridad en la ejecución

	Peligro o Fuente de energía	Riesgo asociado
1	Tensión alterna	Electrocución

2. Objetivos de aprendizaje

OBJETIVO GENERAL: Entender el paradigma de programación orientada a objetos implementando algunos programas que utilicen los conceptos principales que caracterizan a esta metodología de diseño de software.

OBJETIVOS ESPECÍFICOS:

- Crear un programa que implemente clases.
- Implementar la herencia entre clases.
- Reutilizar código a través de bibliotecas.

3. Introducción

El paradigma de la programación orientada a objetos (POO) tiene como idea fundamental que los datos y las operaciones que los manipulan, el código, estén encapsulados en lo que en este ámbito se conoce como “objetos”; tales objetos representan abstracciones del mundo real llevadas al ámbito de la programación. Los datos almacenados dentro de un objeto representan el estado actual del mismo, en la terminología de POO, esos datos se denominan “atributos”.

Una clase es una construcción que permite crear tipos personalizados de datos mediante la agrupación de diversas variables, métodos y eventos. En el paradigma de la POO, son las clases las que generalizan los objetos y en ellas es donde se determina su comportamiento. Por lo anterior, en el modelado de los datos que antecede a cualquier desarrollo de un programa, el primer paso es identificar todos los objetos que un programador quiere manipular y cómo se relacionan entre sí. Una vez que se conoce un objeto, se generaliza como una clase de objetos que define el tipo de datos que contiene y cualquier secuencia lógica que pueda manipularlo. Cada secuencia lógica distinta se conoce como “método” y por lo general implementan las operaciones que la clase puede llevar a cabo. Esta metodología de programación surgió debido a la necesidad de realizar programas que pudieran ser reutilizados en distintos proyectos.

Entre otros conceptos de la POO se encuentran:

- **Herencia.** Consiste en desarrollar una clase pensando en que se cederán todos sus atributos, propiedades y métodos a una nueva clase, con la finalidad de que la última pueda reutilizar dichas características; en otras palabras, las características deseables en una clase existente se pueden reutilizar en otras clases sin afectar la integridad de la clase original.

- **Polimorfismo.** Consiste en diseñar código que pueda manipular objetos de distintas clases, es decir, las diferencias entre los objetos similares podrán ajustarse con facilidad. El polimorfismo ayuda a construir programas concisos (más cortos de lo que serían si no se utilizara); modulares (las partes no relacionadas se mantienen separadas) y fáciles de modificar y adaptar (por ejemplo, cuando se introducen nuevos objetos).

- **Encapsulamiento.** Implica que los usuarios de un objeto tienen una vista restringida del mismo. Esto quiere decir que la implementación y el estado de cada objeto se mantienen de manera privada dentro de la clase; otros objetos no tienen acceso a esta clase o la autoridad para realizar cambios, por el contrario, regularmente solo pueden llamar a una lista de propiedades y métodos públicos. Esta característica de ocultamiento de datos proporciona una mayor seguridad del programa y evita la manipulación involuntaria de datos.

Los conceptos anteriores forman los conceptos básicos para poder abordar la programación orientada a objetos. En la práctica profesional, también se acostumbra utilizar programas de terceros o que no necesariamente fueron creados desde un inicio en el proyecto original, para ello la incorporación de clases externas o de archivos se hace través de bibliotecas que tienen un código que no necesariamente es accesible para su modificación, pero sí para su uso.

4. Material y equipo

Computadora

5. Desarrollo

Actividad 1. Diseño y creación de una clase.

A partir de un objeto real, elija al menos 3 características que lo identifiquen y 2 acciones que definan su comportamiento. Posteriormente, realice una abstracción para definir una clase o plantilla para ese tipo de objetos, que incluya un mínimo de 3 atributos de diferentes tipos (enteros, flotantes, cadenas, booleanos, etc.) y al menos 6 métodos que sean necesarios para escribir/leer los valores de los atributos y/o para interactuar con los objetos. Realice el encapsulamiento apropiado para tales atributos y métodos, además defina las propiedades pertinentes para los atributos incluidos previamente. Incluya al menos 2 constructores para la clase.

Actividad 2. Creación de clases base y derivada.

Realice las modificaciones pertinentes en la clase creada en la actividad anterior para que pueda heredar sus propiedades a otras clases. Una vez hecho eso, cree una clase derivada a partir de la antes modificada, deberá realizar un programa que demuestre que efectivamente fueron heredados todos los atributos y métodos de la clase base, sin necesidad de verificar ni de modificar el código fuente original, además agregue al menos un nuevo atributo y un nuevo método en la clase derivada.

Actividad 3. Reutilización de código.

Cree una biblioteca con las clases creadas hasta el momento y utilícela en un nuevo programa con al menos tres instrucciones diferentes.

6. Bibliografía

- BELL, Douglas y PARR, Mike. **C# para estudiantes**. México, Pearson, 2010
- GARCÍA PÉREZ-SCHOFIELD, Baltasar y ORTÍN SOLER, Francisco. **Programación avanzada orientada a objetos**. España, Andavira Editora, 2013.
- SHALOM, Elad. **SOLID Programming: in C#.** Independently published, 2019.
- LANDA, Nicolás. GUÍA TOTAL DEL PROGRAMADOR C#. Buenos Aires, Redusers,2010.

Práctica #2

Desarrollo de sistemas de cómputo orientados a objetos

1. Seguridad en la ejecución

	Peligro o Fuente de energía	Riesgo asociado
1	Tensión alterna	Electrocución

2. Objetivos de aprendizaje

OBJETIVO GENERAL: El alumno aprenderá a realizar programas orientados a objetos con buenas prácticas de programación.

OBJETIVOS ESPECÍFICOS:

- Fomentar el uso de buenas prácticas de programación.
- Estandarizar nombres de variables utilizadas por un software.
- Promover la correcta documentación de los programas realizados.

3. Introducción

El desarrollo de sistemas orientados a objetos está caracterizado en su mayoría por el trabajo colaborativo entre diferentes programadores y especialistas del área del problema que se requiere resolver, esto hace que la interacción de esa comunidad sea de manera rápida, efectiva y limpia para un buen desarrollo del sistema.

A lo largo de los años, en el área informática se han desarrollado diferentes técnicas para dar orden a todos los problemas de comunicación de las áreas, estandarizando sintaxis y dando recomendaciones de buenas prácticas de programación.

"Me gusta que mi código sea elegante y eficaz. La lógica debe ser directa para evitar errores ocultos, las dependencias deben ser mínimas para facilitar el mantenimiento, el procesamiento de errores completo y sujeto a una estrategia articulada y el rendimiento debe ser óptimo para que los usuarios no tiendan a estropear el código con optimización sin sentido. El código limpio hace bien una cosa" - Bjarne Stroustrup, inventor de C++.

Comenzando con una de las estrategias recurrentes en buenas prácticas de programación: se recomienda seguir una metodología para la resolución de un problema en donde la programación ocupa uno de los últimos pasos, siendo así que lo primero a realizar debe ser un análisis a profundidad de lo que se desea atender a través de software, qué elementos se necesitan para atenderlo, y posteriormente estructurar líneas de código en el lenguaje seleccionado por conveniencia del problema particular. A continuación, se menciona una serie de pasos recomendados dentro de las buenas prácticas de programación para realizar el desarrollo de un programa:

Paso 1 - Definir el problema: describir cualitativamente qué es lo que se desea resolver, qué funciones se espera cumplir y los requerimientos mínimos.

Paso 2 - Definir entradas y salidas: con qué datos, información, se cuenta para resolver el problema y qué se espera obtener de ellos.

Paso 3 - Definir las herramientas: en el caso de programación, específicamente con qué lenguaje, con qué bibliotecas, con qué recursos computacionales se procesarán las entradas para obtener las salidas.

Paso 4 - Proponer una solución verdadera: es decir, qué algoritmo (ya sea basado en un modelo matemático, o en alguna fuente primaria que haya experimentado un problema similar) permite la transformación de entradas en salidas con base en las herramientas seleccionadas.

Paso 5 - Programar: hacer uso de las herramientas, tomando en cuenta todas las entradas consideradas y procesándolas conforme el algoritmo propuesto con la finalidad de generar las salidas deseadas.

Paso 6 – Validar: una vez que el programa realizado esté ejecutándose sin errores, hacer cambios a las variables de entrada para identificar que las salidas siguen generándose conforme el algoritmo establecido y no únicamente para casos particulares. Este paso también se conoce coloquialmente como *testing*.

Específicamente, cuando se está generando código de programación, las buenas prácticas consideran que la definición del nombre de alguna variable es un aspecto fundamental para el correcto desarrollo del sistema no sólo en ese momento sino para su mutabilidad a largo plazo. Ejemplo de lo anterior es la siguiente línea de código:

```
int d; //Tiempo transcurrido en días
```

El nombre asignado carece de especificidad sobre lo que guarda en su interior, se debe aclarar la información que manipulará, como en la siguiente línea:

```
int TimeInDays;
```

Como se observa en la línea anterior, otra buena práctica consiste en que, para definir frases o palabras compuestas, se use una letra mayúscula al inicio de cada palabra y sin espacio entre estas. En las siguientes líneas de código, se hace referencia a una fecha en meses, días, horas, minutos y segundos primero de forma incorrecta y posteriormente con base en la implementación de buenas prácticas.

```
Private Date genymdhms

Private Date  GenerationTime;
```

Cuando se define una clase, se recomienda evitar el uso de nombres que describan su existencia, así también el nombre deberá ser un sustantivo y evitar verbos en medida de lo posible ya que éstos se recomiendan para los métodos (recordando que los métodos declarados en una clase representan las acciones que realizarán los objetos creados a partir de dicha clase). Ejemplos de buenas prácticas conforme lo dicho anteriormente serían:

```
class Empleado

class FiguraGeometrica
```

private void VaciarLista()

public double CalcularArea()

Se recomienda que los métodos o funciones sean de tamaño reducido y, dicha función, solo realice una acción; a tal práctica se le conoce como **principio de responsabilidad única** y para favorecerlo se sugiere que las instrucciones de una función se encuentren en el mismo nivel de abstracción.

Referente a los comentarios que se agregan como parte de la documentación, las buenas prácticas mencionan que no hay nada más útil que un cometario bien colocado, así como nada más inútil y molesto que comentarios innecesarios. Documentar el código ayuda a que otros programadores puedan seguir el flujo de acciones del programa y puedan ocupar de manera eficiente las clases y los métodos creados por el equipo. La recomendación se basa en ordenar la información en bloques para no buscar la ubicación de una función en toda la estructura del programa, de tal forma se mantiene un modo ágil de realizar cambios o mejoras, así como advertir de código peligroso o sensible con el cual poner especial atención a modificaciones o cambios que podrían afectar el desempeño del programa.

Un ejemplo de cómo las buenas prácticas de documentación ayudan no sólo a un equipo de programación para actualizar su software, sino también a los usuarios, se encuentra en las bibliotecas o APIs de terceros, ya que con base en su correcta documentación más usuarios podrían acceder a sus funciones y servicios, incidiendo positivamente en la reputación de los programadores responsables (en el caso de servicios gratuitos) o significando un incremento de ventas (en el caso de las aplicaciones de pago).

4. Material y equipo

Computadora

5. Desarrollo

Actividad 1. Metodología del desarrollo de sistemas.

Para un problema específico a ser resuelto por un programa, seguir los pasos recomendados para la metodología de desarrollo de sistemas orientados a objetos hasta el paso 4. Documentar el proceso paso a paso, de forma ordenada.

Actividad 2. Uso de buenas prácticas.

Con base en el resultado de la actividad 1, programar la solución al problema (paso 5 de la metodología) utilizando las buenas prácticas sugeridas para nombrar cada elemento y comentando las líneas o módulos.

Actividad 3. Verificación del código.

Comprobar que el programa funcione correctamente (paso 6 de la metodología). Realizar un breve texto, no mayor a una cuartilla, respecto a las diferencias percibidas entre un código sin buenas prácticas de programación y uno tomándolas en cuenta, que cumplen una misma tarea.

6. Bibliografía

- MARTIN, Robert C. **Código limpio: Manual de estilo para el desarrollo ágil de software.** España, Anaya Multimedia, 2012
- BELL, Douglas y PARR, Mike. **C# para estudiantes**. México, Pearson, 2010.
- GARCÍA PÉREZ-SCHOFIELD, Baltasar y ORTÍN SOLER, Francisco. **Programación avanzada orientada a objetos**. España, Andavira Editora, 2013.
- SHALOM, Elad. **SOLID Programming: in C#.** Independently published, 2019.
- LANDA, Nicolás. GUÍA TOTAL DEL PROGRAMADOR C#. Buenos Aires, Redusers,2010.

Práctica #3

Estructuras de datos compuestas

1. Seguridad en la ejecución

	Peligro o Fuente de energía	Riesgo asociado
1	Tensión alterna	Electrocución

2. Objetivos de aprendizaje

OBJETIVO GENERAL: Entender el uso y funcionamiento de las principales estructuras de datos lineales y no lineales, así como conocer algunos de los algoritmos empleados en las acciones de ordenación y de búsqueda.

OBJETIVOS ESPECÍFICOS:

- Entender el funcionamiento de dos estructuras de datos y ser capaz de utilizarlas junto con los métodos pertinentes para trabajar con ellas.
- Implementar un algoritmo de ordenación y uno de búsqueda en alguna de las estructuras de datos creadas.

3. Introducción

Una estructura de datos es un grupo de elementos que se pueden procesar de manera uniforme. Las estructuras básicas suelen tener la limitación de tener tamaño fijo; por otra parte, las estructuras dinámicas tienen la capacidad de variar su tamaño en tiempo de ejecución, siempre y cuando se tenga suficiente espacio en memoria.

Las listas enlazadas, las pilas y las colas son colecciones de elementos de datos "alineados en una fila"; mientras que los árboles son estructuras de datos que almacenan sus nodos de forma jerárquica. En la Figura 1 se pueden apreciar las estructuras lineales, como las listas, y un árbol, que es una estructura no lineal.

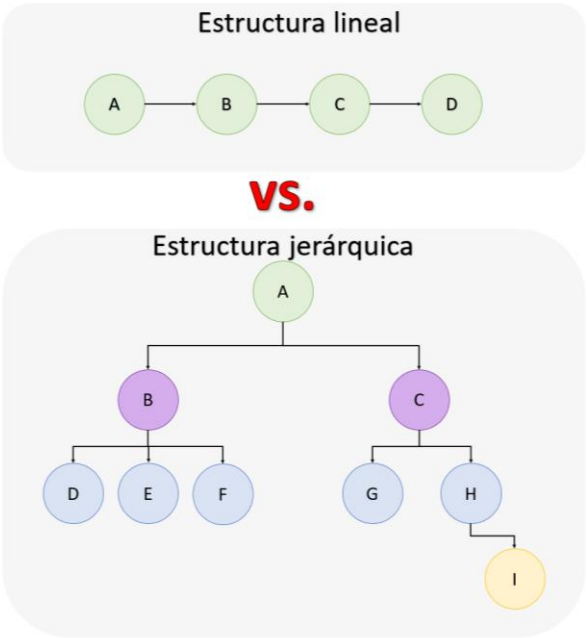

Figura 1. Representación gráfica de la diferencia entre las estructuras de datos lineales y las no lineales.

Estructuras de datos lineales

Las listas, las pilas y las colas son colecciones lineales (es decir, secuencias) de elementos de datos del mismo tipo.

En las **listas** los usuarios pueden insertar y eliminar elementos en cualquier parte de ellas; mientras que, en las **pilas** y **colas** se tienen restricciones en la inserción y eliminación de elementos.

Las **pilas** son conocidas como estructuras de datos LIFO (Last In First Out; último en entrar, primero en salir) esto es, el elemento más reciente en ser añadido será el primero en ser removido.

En las **colas**, los nodos se remueven de la estructura en el mismo orden en el que se agregan; por esta razón se les conoce como estructuras de datos tipo FIFO (First In First Out; primero en entrar, primero en salir).

Estructuras de datos no lineales

También conocidas como árboles, son estructuras de datos bidimensionales no lineales, con propiedades especiales; los nodos de un árbol contienen dos o más enlaces.

El nodo raíz es el primer nodo en un árbol. Cada enlace en el nodo raíz hace referencia a un hijo. En la Figura 1, el nodo raíz es A. Un nodo es padre de otro si existe un enlace desde el primer nodo al segundo (en ese caso, también decimos que el segundo nodo es hijo del primero). Por ejemplo, en la Figura 1 el nodo B es padre de E.

A un nodo que no tiene hijos se le llama hoja, como los nodos D, E, F, G e I de la Figura 1.

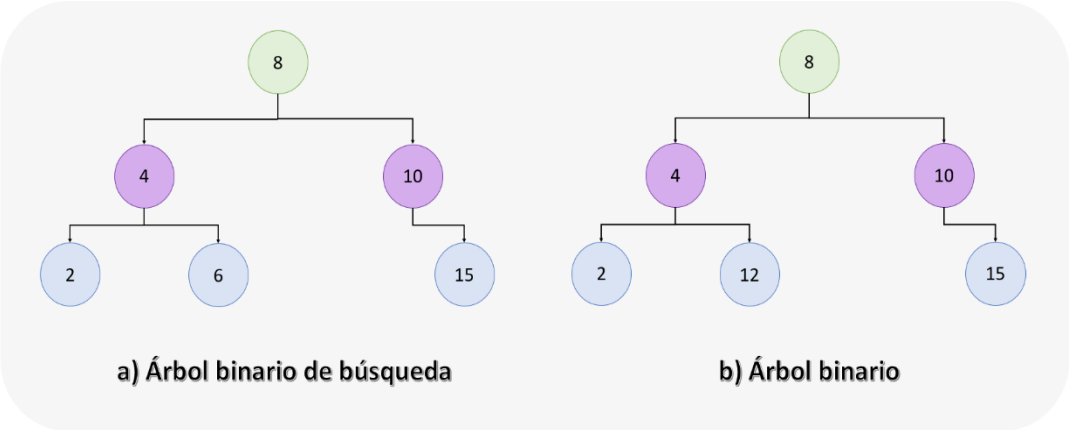

Figura 2. Tipos de árboles.

Un árbol binario es aquel en el que cada nodo tiene máximo dos hijos; mientras que un árbol binario de búsqueda es un árbol binario en el que para cada nodo “n”, el nodo a la izquierda (“i”) de “n” debe satisfacer: i≤n; mientras que el nodo a la derecha de (“d”) “n” se debe cumplir que n<d. Esto debe satisfacerse para todo nodo “n”. En la Figura 2 se muestra un árbol binario y un árbol binario de búsqueda; el árbol mostrado en la Figura 2 b), no es de búsqueda debido a que “12” no está a la derecha de “8”.

Las operaciones básicas en árboles son: insertar y localizar elementos, así como recorrer y moverse a través del árbol.

Con ayuda de los árboles se pueden representar datos jerárquicos y facilitar las operaciones de búsqueda en estos conjuntos de datos. Un ejemplo del uso de árboles es el manejo de carpetas de archivos en un sistema operativo.

Algoritmos de ordenación y búsqueda

Una vez que se tiene un grupo de datos almacenados en una estructura de datos, o en un archivo, se pueden ordenar siguiendo algún criterio. Ordenar es el proceso de reorganizar un conjunto de objetos en una secuencia determinada; este proceso generalmente tiene como objetivo facilitar la búsqueda de elementos pertenecientes a un conjunto.

Los métodos de ordenación, para estructuras lineales, se suelen dividir en dos grandes grupos:

- Directos: Intercambio, burbuja, selección, inserción.
- Indirectos o avanzados: Shell, quicksort, ordenación por mezcla, radixsort.

A la localización de un elemento dentro de la estructura de datos, o determinar si este elemento se encuentra dentro del conjunto de datos, se le conoce como búsqueda. Para encontrar elementos dentro de una estructura de datos lineal se pueden usar los métodos de búsqueda secuencial o búsqueda binaria.

Para los árboles, los métodos de búsqueda son informados y no informados (a ciegas, ya que recorre todo el árbol sin tener una pista de donde pueda estar el dato deseado).

4. Material y equipo

Computadora

5. Desarrollo

Actividad 1. Implementación de estructuras de datos

Realizar un programa en el que se empleen al menos dos estructuras de datos de diferente tipo para generar registros de datos de algún tipo de objeto, utilizando al menos 3 operaciones fundamentales de cada estructura usada.

Actividad 2. Algoritmo de Ordenación

Ordenar los datos guardados en una estructura generada en la Actividad 1 mediante alguno de los métodos de ordenación, conforme a dos criterios distintos.

Actividad 3. Algoritmo de Búsqueda

Partiendo de una estructura de datos generada en la Actividad 1, realizar diferentes búsquedas, mediante al menos dos métodos, y comparar el número de iteraciones realizadas por cada método para llegar al resultado.

6. Bibliografía

- LAAKMANN MCDOWELL, Gayle. **Cracking the Coding Interview: 189 Programming Questions and Solutions**. Palo Alto, CA., CareerCup, 2016.
- BELL, Douglas y PARR, Mike. **C# para estudiantes**. México, Pearson, 2010
- CEBALLOS SIERRA, Francisco Javier. **Microsoft C#. Curso de programación**. México, Alfaomega, 2007
- DEITEL, Harvey y Deitel, PAUL. **C# Cómo programar**. España, Pearson, 2007
- LÓPEZ ROMÁN, Leobardo. **Metodología de la programación orientada a objetos**. México, Alfaomega, 2007

Práctica #4

Interfaces gráficas de usuario

1. Seguridad en la ejecución

	Peligro o Fuente de energía	**Riesgo asociado**
1	Tensión alterna	Electrocución

2. Objetivos de aprendizaje

OBJETIVO GENERAL: Entender e implementar interfaces gráficas, para aplicaciones de computadora, que se enfoquen en brindar una grata experiencia al usuario con una notable facilidad de uso (usabilidad) de la misma.

OBJETIVOS ESPECÍFICOS:

- Comprender qué es una interfaz gráfica, sus partes y eventos.
- Entender la aplicación de los principios de usabilidad y la experiencia del usuario desde el punto de vista de programación.

3. Introducción

Interfaz gráfica de usuario

Una interfaz gráfica de usuario (GUI) es la parte de un programa que actúa de intermediario entre el usuario y la computadora utilizando un conjunto de imágenes y objetos gráficos para representar la información y acciones que se pueden realizar. La interfaz gráfica se compone de dos elementos principales: ventana y controles.

Ventanas y controles

Las ventanas son los objetos sobre los que se dibujan los controles como cajas de texto (*textbox*), botones (*buttons*) o etiquetas (*labels*) que dan lugar a la interfaz gráfica. Tanto las ventanas como los controles, al ser objetos, tienen un conjunto de propiedades para modificar sus características, por ejemplo: la propiedad *name* es el nombre del control con el que se le hará referencia en el código del programa.

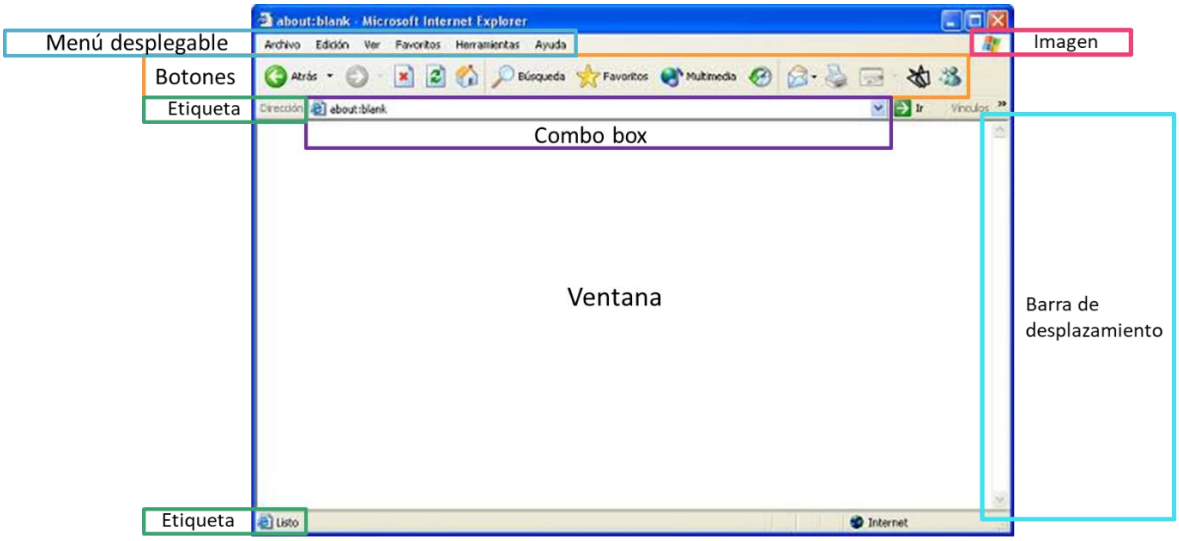

Figura 1. Ejemplo de interfaz, ventana y sus controles.

Eventos

Cuando ocurre un suceso sobre un control, como cuando un usuario pulsa un botón, se produce un evento; para que la aplicación responda a ese evento, se tiene que escribir un método que contendrá el código a ejecutar ante tal suceso.

Experiencia del usuario (UX)

El diseño basado en la experiencia del usuario tiene por objetivo la creación de productos que resuelvan necesidades concretas, consiguiendo la mayor satisfacción y mejor experiencia de uso con el mínimo esfuerzo. Este concepto está relacionado con la usabilidad.

Entre los principios de usabilidad desarrollados por Jakob Nielsen se encuentran:

- **Relación entre el sistema y el mundo real:** Uso de palabras y frases que sean familiares al usuario.

- **Visibilidad del estado del sistema:** Siempre mantener informado a los usuarios de lo que está ocurriendo a través de retroalimentación.

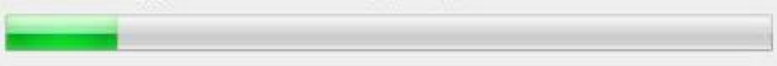

Figura 2. Ejemplo de visibilidad del estado: barra de carga de un proceso.

- **Control y libertad del usuario:** Darle al usuario la posibilidad de subsanar el error (botón de deshacer, posibilidad de editar).

Figura 3. Ejemplo de control: botón para deshacer modificaciones recientes.

- **Consistencia y estándares:** Seguir las convenciones establecidas para ciertos iconos.

Figura 4. Ejemplo de estándar: ícono ligado al guardado de un programa.

- **Prevención de errores:** poder reaccionar ante cualquier error que pueda cometer el usuario y sugerirle correcciones.

- **Reconocimiento antes que recordar:** deben ser visibles objetos, acciones y opciones, evitar que el usuario recuerde información que se dé en una parte del proceso para seguir adelante.

- **Flexibilidad y eficiencia de uso:** consideración de que el usuario puede ser una persona con experiencia (power users) o completamente nueva (early bird).

- **Estética y diseño minimalista:** no contener información innecesaria.

- **Ayuda y documentación:** Agregar una sección de preguntas frecuentes (FAQ, por sus siglas en inglés), menús de ayuda, tutoriales, etc.

Leyes y efectos en usabilidad

Existen leyes que fueron establecidas en la interacción humano – máquina o computadora y que son fundamentales para el diseño de usuario, como son:

Ley de Fitts: Al esperar que el usuario pulse un control, el tamaño del elemento y su posición respecto al punto de partida del cursor son importantes. Entre más lejos y más pequeño sea un control, más difícil será para el usuario llegar a él.

Ley de Hick: El tiempo que el usuario tarda en tomar una decisión aumenta a medida que se incrementa el número de opciones. Es importante priorizar menús de navegación y reducir opciones.

Ley de Jakob: Mantener informado al usuario todo el tiempo de lo que está ocurriendo, es posible utilizando barras de carga, indicadores de procesos de compra, etc.

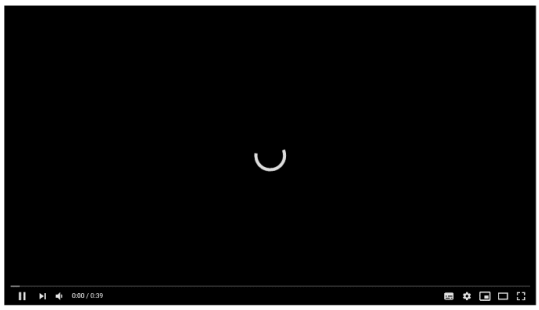

Figura 5. Ejemplo de Jakob: indicador de carga en un video on-line.

Ley de Miller: Las personas pueden recordar hasta siete elementos distintos (± 2 elementos) en su memoria de trabajo. Es necesario reducir la cantidad de elementos y mantenerlos organizados por grupos que no excedan los nueve ítems.

Principio de Pareto: El 80% del uso de un sistema está relacionado con el 20% de las características o variables de éste. Los menús en la interfaz gráfica son un ejemplo de conciliar cuál es el 20% de la funcionalidad que debe ser visible, por lo que es necesario identificar las funciones más utilizadas.

Efecto de Von Restorff: También llamado “efecto de aislamiento”, dice que cuando hay varios objetos similares, se recordará el que difiere del resto.

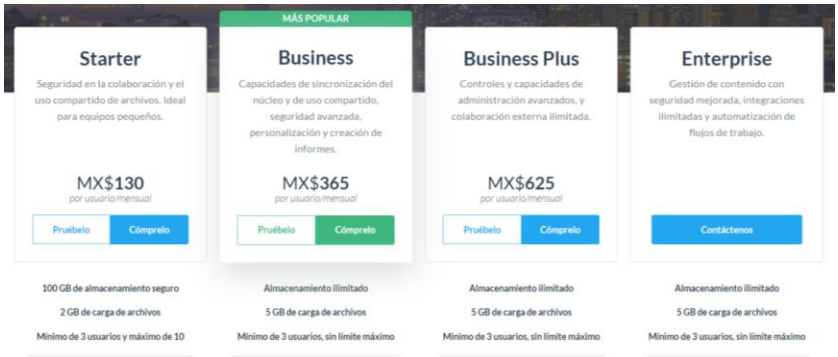

Figura 6. Ejemplo de Von Restorff: resalta un plan de compra sobre los demás favoreciendo su elección.

Efecto de Zeigarnik: Define la tendencia de las personas a recordar tareas inacabadas o interrumpidas con mayor facilidad. En la interfaz se indican ciertas tareas “incompletas” para que el usuario sienta que no ha terminado.

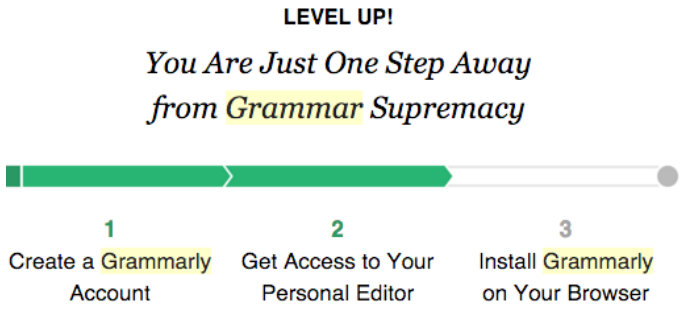

Figura 7. Ejemplo de Zeigarnik: indicador de progreso en aplicación de gramática.

Efecto de posición en serie: Entre los elementos de una serie, los usuarios siempre recordarán el primero y el último de los elementos. En barras de navegación las acciones más importantes se colocan en los extremos izquierdo y derecho.

Evaluación de interfaces gráficas de usuario

La evaluación de interfaces de usuario se realiza mediante métodos subjetivos y objetivos. Los métodos subjetivos se basan en la opinión del usuario (tanto principiantes como experimentados), puede ser de manera oral, donde el usuario utiliza la interfaz para comentar los problemas encontrados, opiniones, sentimientos, etc. También puede ser de manera escrita con cuestionarios o *checklist* como QUIS (*Questionnare for User Interaction Satisfaction*), entre otros.

4. Material y equipo

Computadora

5. Desarrollo

Actividad 1. Implementación de una GUI

De un problema propuesto y tomando en cuenta los principios de usabilidad, leyes y efectos antes mencionados, realizar una interfaz gráfica de usuario agregando los controles necesarios para resolver el problema, agregar un control para la representación gráfica de al menos una de las variables procesadas. Una vez terminada la interfaz, realizar un documento en donde se exprese qué leyes y efectos se utilizaron en el diseño justificando su uso, y cuáles no justificando su ausencia.

Actividad 2. Evaluación de una GUI

Verificar la funcionalidad de la interfaz gráfica de manera subjetiva mínimo con un compañero de clase. Sin explicación previa, registrar las acciones que realiza, en qué centra su atención, si resuelve el problema, los errores que comete, así como comentarios y opiniones posteriores al uso de la interfaz.

6. Bibliografía

- **Las leyes de UX con casos prácticos.** (s.f.). Recuperado el Mayo de 2019, de https://medium.com/startups-es/las-leyes-de-ux-con-casos-pr%C3%A1cticos-b838ddf7ff9b
- **10 reglas heurísticas de usabilidad.** (s.f.). Recuperado el Mayo de 2019, de http://www.braintive.com/10-reglas-heuristicas-de-usabilidad-de-jakob-nielsen/
- **Applying Fitts Law to mobile interface design.** (s.f.). Recuperado el Mayo de 2019, de https://webdesign.tutsplus.com/es/articles/applying-fitts-law-to-mobile-interface-design--webdesign-6919
- **Ley de Tesler.** (s.f.). Recuperado el Mayo de 2019, de https://jummp.wordpress.com/2010/05/01/ley-de-tesler-de-conservacion-de-la-complejidad/
- CANZIBA, Elvis. Hands-on UX Design for Developers. UK, Packt, 2018.
- REITERER, Harald. **Encyclopedia of library and information science.** http://nbn-resolving.de/urn:nbn:de:bsz:352-251693
- NIELSEN, Jakob y LORANGER, Hoa. **Usabilidad, Prioridad en el diseño web. (Diseño y Creatividad).** Anaya multimedia, 2006.
- MCKAY, Everett. **UI is communication**. UK, Elsevier, 2013.
- C. BENEYTO. (2018) "***Las leyes de UX con casos prácticos*.** Accesado en mayo de 2019. Disponible: https://medium.com/startups-es/las-leyes-de-ux-con-casos-pr%C3%A1cticos-b838ddf7ff9b
- J. SMITH (2012) "**Applying Fitts Law to mobile interface design."** Accesado en mayo de 2019. Disponible: https://webdesign.tutsplus.com/es/articles/applying-fitts-law-to-mobile-interface-design--webdesign-6919