

Programación Visual

III

Daniel Alejandro Nuño
Ramirez

Presentación

- Daniel Alejandro Nuño Ramirez
- Ingeniero en Computación, Universidad de Guadalajara
- Maestro en Informática Aplicada*, ITESO
- Gerente de Desarrollo de Software, Oracle
- TCS, Intel, Bank of America, Flextronics, Estratel, UdeG
- Correr, Cine, Series, Música
- Email: daniel_nuno@hotmail.com

Contenido

Unidad I, Introducción a la programación visual

Unidad II, La plataforma Microsoft .NET

Unidad III, Programación Concurrente

Unidad IV, Programación Distribuida y Paralela

Unidad V, Middleware

UNIDAD I, Introducción a la Programación Visual

UNIDAD I, Introducción a la Programación Visual

- 1.1 Concepto de programación visual
- 1.2 Herramientas para el desarrollo rápido de aplicaciones (RAD)
- 1.3 Relación entre la programación visual y la programación orientada a objetos
- 1.4 La programación basada en eventos
- 1.5 El concepto de componentes visuales
- 1.6 Paradigmas de programación
- 1.7 Entorno de Desarrollo Integrado (IDE)

UNIDAD I, Introducción a la Programación Visual

1.1 Concepto de programación visual

En la actualidad la interacción con la computadora, se ha transformado para poder hacer uso de mas sentidos del usuario, como lo son la voz, el toque de la pantalla y hasta el movimiento.

Los usuarios buscan interactuar con la computadora de una manera más natural para ellos y menos orientada a la computadora.

El usuario que necesita respuestas, cada vez las necesita con menor tiempo y con una mayor facilidad para conseguirlas. Es un usuario impaciente y que rápidamente desecha tecnologías que le parecen poco prácticas.

UNIDAD I, Introducción a la Programación Visual

1.1 Concepto de programación visual

Los desarrolladores de software han ido evolucionando de la misma manera en que las interfaces de usuario así lo han solicitado, como también el uso del software y los productos que se han construido con el.

Se habla del “Front-end” y del “Back-end”, del “Human Computer Interaction” y su evolución, desde CLI, GUI, NUI y hasta OUI.

UNIDAD I, Introducción a la Programación Visual

1.1 Concepto de programación visual

- **CLI**, Command Line Interface, se refiere a la línea de comandos, similar a las terminales UNIX o command de Windows.
- **GUI**, Graphic User Interface, se refiere a una interfaz gráfica, por ejemplo la de Windows 95.
- **NUI**, Natural User Interface, se refiere a las interfaces mas naturales hacia el usuario, por ejemplo las pantallas touch de los celulares y las computadoras.
- **OUI**, Organic User Interface, se refiera a las interfaces con elementos biométricos.

UNIDAD I, Introducción a la Programación Visual

- Human Computer Interaction



CLI

GUI

NUI

OUI

UNIDAD I, Introducción a la Programación Visual

- Human Computer Interaction

- 1940 nacen las computadoras ENIAC
- 1950 UNIVAC
- 1960 System/360
- 1970 PDP/11
- 1980 nace la interacción con el usuario

- Human Computer Interaction

- 1962 Sketchpad (Ivan Sutherlands)
- 1963 Mouse (Douglas Engelbart)
- 1983 The Psychology of Human-Computer Interaction
- 1984 Apple Macintosh

UNIDAD I, Introducción a la Programación Visual

- Front-End

Desarrollo del lado del cliente en el que se convierten los datos a la interfaz grafica del usuario y se programa la interacción con los mismos.

Se le relaciona con HTML, CSS y JavaScript.

- Back-End

Se compone de tres partes, el servidor, la aplicación y la base de datos.

Se le relaciona con Tomcat, Java, Oracle, también con LAMP, Linux, Apache, MySQL y PHP.

UNIDAD I, Introducción a la Programación Visual

1.2 Herramientas para el desarrollo rápido de aplicaciones (RAD)

RAD es una metodología de desarrollo de aplicaciones, que viene de los años 80. Nació principalmente para dar respuesta a las necesidades de los clientes de tener resultados rápidos en sus necesidades de software.

Como hubo en su momento muchas variantes, es difícil tener un proceso unificado que de una definición exacta al término.

A continuación se explican brevemente algunas variantes de RAD.

UNIDAD I, Introducción a la Programación Visual

1.2 Herramientas para el desarrollo rápido de aplicaciones (RAD)

- TimeBox Development
- Domain-Specific Languages
- Reutilización de Software
- Programación orientada a objetos (OOP)
- Arquitectura basada en componentes
- Herramientas de Productividad
- Creación de Prototipos

UNIDAD I, Introducción a la Programación Visual

1.2 Herramientas para el desarrollo rápido de aplicaciones (RAD)

Timebox Development

Técnica de RAD que tiene un tiempo fijo y la funcionalidad de la aplicación, se fija a los límites del tiempo disponible. El objetivo es producir un sistema funcional al final del tiempo dado.

La funcionalidad que no haya sido completada, será removida del producto final.

Esencialmente la práctica de Timebox Development, busca fijar el desarrollo de la aplicación al tiempo dado.

Esta técnica, necesita que el cliente haga una priorización de sus requerimientos.

UNIDAD I, Introducción a la Programación Visual

1.2 Herramientas para el desarrollo rápido de aplicaciones (RAD)

Timebox Development

- Riesgo de ser limitante, el cliente recibe una aplicación funciona en poco tiempo, pero quizá este limitada por el mismo.
- Impide sobre trabajo, a los miembros del equipo no les es permitido divagar en requerimientos futuros, únicamente se permite trabajar bajo lo pactado.
- Mantiene enfocado al equipo, una agresiva planeación y calendario, enfoca la atención del equipo en las necesidades inmediatas de la aplicación.

UNIDAD I, Introducción a la Programación Visual

1.2 Herramientas para el desarrollo rápido de aplicaciones (RAD)

Timebox Development

Si es empleada adecuadamente, esta técnica permite un desarrollo dinámico, donde todo el equipo se ve enfocado en el resultado y en cumplir los tiempos de entrega estipulados.

Esta técnica, examina la importancia en limitar el tiempo en los métodos de desarrollo iterativo, que hoy son parte de las metodologías ágiles.

UNIDAD I, Introducción a la Programación Visual

1.2 Herramientas para el desarrollo rápido de aplicaciones (RAD)

Domain-Specific Languages

En un esfuerzo de reducir la ruta crítica en la construcción de un proyecto, RAD promueve el uso de lenguajes de desarrollo especializados.

Estos lenguajes especializados, son distintos de los de propósito general como Java, sino que resuelven funciones específicas. Algunos ejemplos son:

- Jython, SQL for data access code, Prolog, Tcl/Tk for prototyping user interfaces, JSP for building HTML

UNIDAD I, Introducción a la Programación Visual

1.2 Herramientas para el desarrollo rápido de aplicaciones (RAD)

Reutilización de Software

La reutilización de software cumple con dos importantes necesidades de diseño.

1. **Primero**, permite el desarrollo de módulos comunes que pueden ser compartidos entre las aplicaciones.
2. **Segundo**, impide la duplicidad de código, ya que en la arquitectura de software, un código que tenga duplicidad, hace complicado el mantenimiento y además, produce errores al no poder aplicar los parches adecuadamente.

UNIDAD I, Introducción a la Programación Visual

1.2 Herramientas para el desarrollo rápido de aplicaciones (RAD)

Reutilización de Software

La reutilización de software además tiene los siguientes beneficios:

- Mejora la productividad
- Mayor calidad en el producto
- Mejora la confiabilidad
- Facilita el mantenimiento
- Reduce el tiempo de ingreso del producto de software al mercado (Time to Market)

UNIDAD I, Introducción a la Programación Visual

1.2 Herramientas para el desarrollo rápido de aplicaciones (RAD)

Reutilización de Software

El diseño de software para que sea reusable es difícil de conseguir. A la fecha, el éxito en el área de reutilización de software viene por la práctica del diseño orientado a objetos. Existen dos áreas de interés principalmente:

- Programación orientada a objetos para implementar elementos de software reusable a nivel de código.
- Arquitectura basada en componentes para construir aplicaciones desde componentes prefabricados.

UNIDAD I, Introducción a la Programación Visual

1.2 Herramientas para el desarrollo rápido de aplicaciones (RAD)

Programación orientada a objetos (OOP)

El paradigma de la programación orientada a objetos, nace del deseo de tener lenguajes de programación capaces de representar objetos reales y como una manera de promover la reutilización de software.

Los lenguajes de programación orientados a objetos, utilizan las mejores técnicas para el desarrollo de código reusable y ofrecen las bases de software necesarias, para construir aplicaciones desde módulos de software.

UNIDAD I, Introducción a la Programación Visual

1.2 Herramientas para el desarrollo rápido de aplicaciones (RAD)

Arquitectura basada en componentes

Haciendo una analogía, si C# es un lenguaje orientado a objetos donde se promueve la reutilización del código, la plataforma .NET ha sido creada como una arquitectura basada en componentes.

Los componentes pueden abarcar dominios verticales u horizontales de tal manera que proveen funcionalidad estructural y en su comportamiento.

UNIDAD I, Introducción a la Programación Visual

1.2 Herramientas para el desarrollo rápido de aplicaciones (RAD)

Arquitectura basada en componentes

Las empresas de desarrollo de software tienen ahora la capacidad de desarrollar bibliotecas de componentes que estén orientados a un dominio en particular, tales como servicios financieros, sector gobierno y de telecomunicaciones.

A través de esta estrategia, los fabricantes de software, adquieren una ventaja competitiva. La construcción de estas bibliotecas de componentes, es una importante inversión en la adaptación de un desarrollo.

UNIDAD I, Introducción a la Programación Visual

1.2 Herramientas para el desarrollo rápido de aplicaciones (RAD)

Herramientas de Productividad

En las diferentes vertientes de RAD, se han creado sofisticadas herramientas de desarrollo. Seleccionar la herramienta adecuada, puede incrementar drásticamente la productividad.

Además de incrementar la productividad, las herramientas pueden ayudar a simplificar el proceso de desarrollo de software, un punto clave para los desarrolladores.

UNIDAD I, Introducción a la Programación Visual

1.2 Herramientas para el desarrollo rápido de aplicaciones (RAD)

Herramientas de Productividad

El desarrollo de software empresarial, es una tarea compleja, ya que requiere del uso y dominio de una diversidad de conceptos avanzados de ingeniería, los cuales incluyen programación orientada a objetos, computo distribuido, arquitectura multihilos (multithreading), y middleware orientado a mensajes.

UNIDAD I, Introducción a la Programación Visual

1.2 Herramientas para el desarrollo rápido de aplicaciones (RAD)

Creación de Prototipos

La creación de prototipos es quizá la más conocida de las técnicas de RAD, ya que es aplicable a un gran rango de tareas.

La práctica es idealmente para definir requerimientos de usuario y para explorar la viabilidad de la arquitectura.

La creación de prototipos ofrece un acercamiento de poco riesgo hacia el desarrollo de software.

UNIDAD I, Introducción a la Programación Visual

1.2 Herramientas para el desarrollo rápido de aplicaciones (RAD)

Creación de Prototipos

Algunas de las razones para incluir un prototipo como parte del proyecto incluye:

- Resolver dudas del cliente
- Validar las decisiones de arquitectura
- Resolver problemas de rendimiento
- Capturar requerimientos del cliente

UNIDAD I, Introducción a la Programación Visual

1.2 Herramientas para el desarrollo rápido de aplicaciones (RAD)

Creación de Prototipos

Continuación:

- Comunicar ideas e intenciones hacia el cliente
- Otorgar la elección de tecnología y plataforma
- Demostrar la interfaz de usuario y el flujo de trabajo
- Medir el riesgo

UNIDAD I, Introducción a la Programación Visual

1.2 Herramientas para el desarrollo rápido de aplicaciones (RAD)

Creación de Prototipos

Existen varios acercamientos para realizar prototipos:

- **Throwaway**, donde el prototipo se descarta una vez cumplido su objetivo.
- **Evolutionary**, el prototipo es continuamente usado y evoluciona hasta un sistema de producción.
- **Behavioral**, demuestra el comportamiento y sirve para capturar requerimientos.
- **Structural**, sirve para validar áreas de la arquitectura.

UNIDAD I, Introducción a la Programación Visual

1.3 Relación entre la programación visual y la programación orientada a objetos

- La programación visual otorga los conocimientos necesarios para desarrollar una aplicación con un entorno amigable al usuario, ya sea que se utilice para generar un GUI, un NUI o un OUI.
- La programación orientada a objetos, permite modelar situaciones del mundo real mediante la manipulación abstracta de un objeto, permitiendo que tengan algún estado o característica (datos), comportamiento (métodos) e identidad (propiedad o atributo del objeto) para diferenciarlos de los demás.
- La OOP expresa una aplicación como un conjunto de objetos que colaboran entre ellos para resolver tareas.
- Los lenguajes utilizados para programación visual, suelen estar basados en programación orientada a objetos.

UNIDAD I, Introducción a la Programación Visual

1.3 Relación entre la programación visual y la programación orientada a objetos

- En la programación visual, la estructura como la ejecución de los programas van determinados por los eventos que ocurren en el sistema, definidos por el usuario o que ellos mismos provoquen.
- En la programación estructurada, es el programador el que define el flujo del programa.
- En la programación dirigida por eventos, será el usuario el que dirige el flujo del programa.
- En la programación estructurada, puede haber alguna intervención externa, siempre y cuando el programador así lo haya determinado. En la programación visual un evento puede ocurrir en cualquier momento.
- El desarrollador de un sistema dirigido a eventos, debe definir aquellos que manejarán su programa y las acciones que se realizaran al producirse cada uno de ellos.

UNIDAD I, Introducción a la Programación Visual

1.4 La programación basada en eventos

- Programación síncrona y asíncrona
 - Programación **síncrona** es la que se refiere a que una tarea no puede iniciar hasta que la previa haya terminado.
 - Programación **asíncrona** es en la que múltiples tareas pueden ser realizadas al mismo tiempo y sirve para realizar aplicaciones que responden a eventos.

UNIDAD I, Introducción a la Programación Visual

1.4 La programación basada en eventos

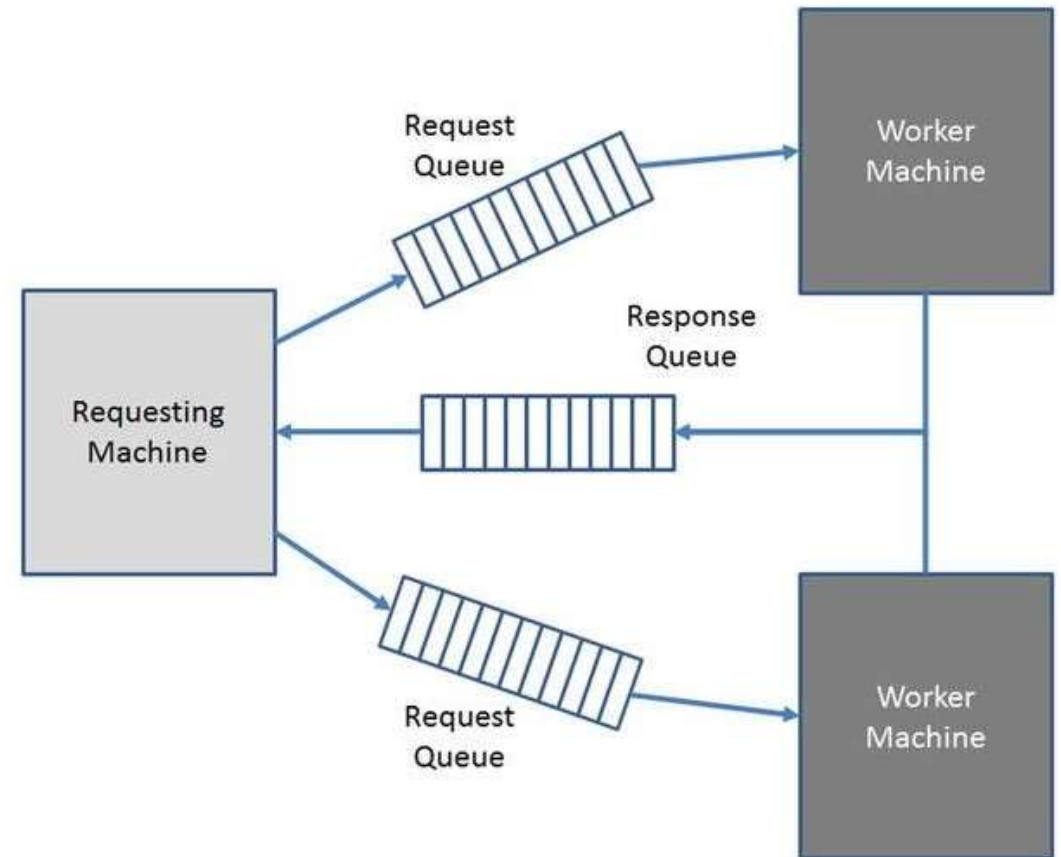
- Programación asíncrona motivación.
 1. Los usuarios están mas al pendiente de la respuesta de las aplicaciones que usan. Con la llegada de los celulares y las tablets, los usuarios se han vuelto mas desesperados por la respuesta de los dispositivos, lo que implica que el desarrollador tenga que captar todos los eventos que suceden en el dispositivo.
 2. La tecnología de procesamiento ha evolucionado de tal manera que ahora es posible poner múltiples núcleos de procesamiento en un paquete, por lo que ahora las maquinas ofrecen enormes cantidades de poder de cómputo.
 3. La tendencia de mover los servicios a la denominada nube significa acceder a funcionalidad que potencialmente esta dispersa geográficamente. La latencia puede causar que las operaciones no tengan el rendimiento necesario, por lo que ejecutar una o mas operaciones de manera concurrente puede beneficiar a que la aplicación tenga un rendimiento aceptable.

UNIDAD I, Introducción a la Programación Visual

1.4 La programación basada en eventos

- Programación asíncrona mecanismos.

1. **Múltiples máquinas,** múltiples máquinas deben ser utilizadas para que cuando hagamos un requerimiento de ejecución remota, no tengamos problemas de bloqueo en la respuesta.
2. Un método común, es utilizar una cola de mensajes, de donde se toma el mensaje y se realiza la acción.

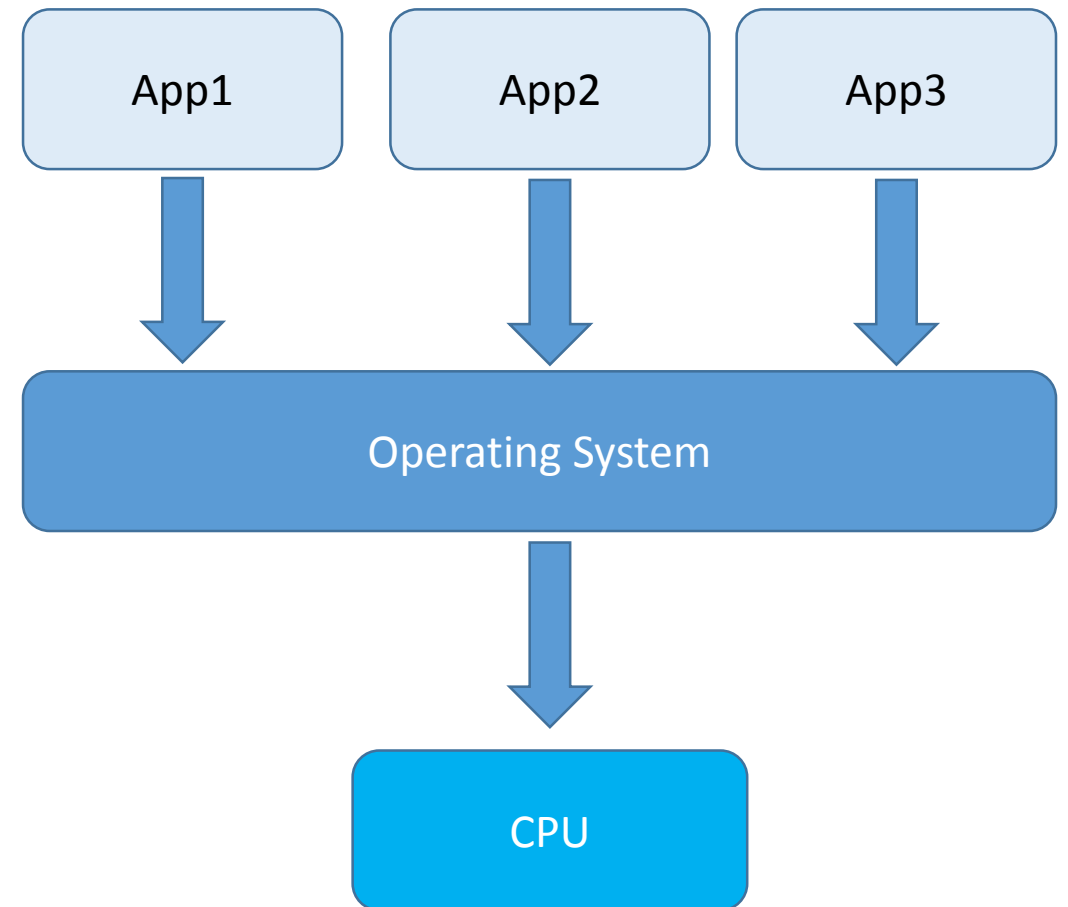


UNIDAD I, Introducción a la Programación Visual

1.4 La programación basada en eventos

- Programación asíncrona mecanismos.

1. Múltiples procesos, un proceso es una unidad aislada en una máquina. Múltiples procesos deben compartir el acceso a las unidades de procesamiento (cores), pero no comparten la memoria virtual y pueden ser ejecutados en diferentes contextos.

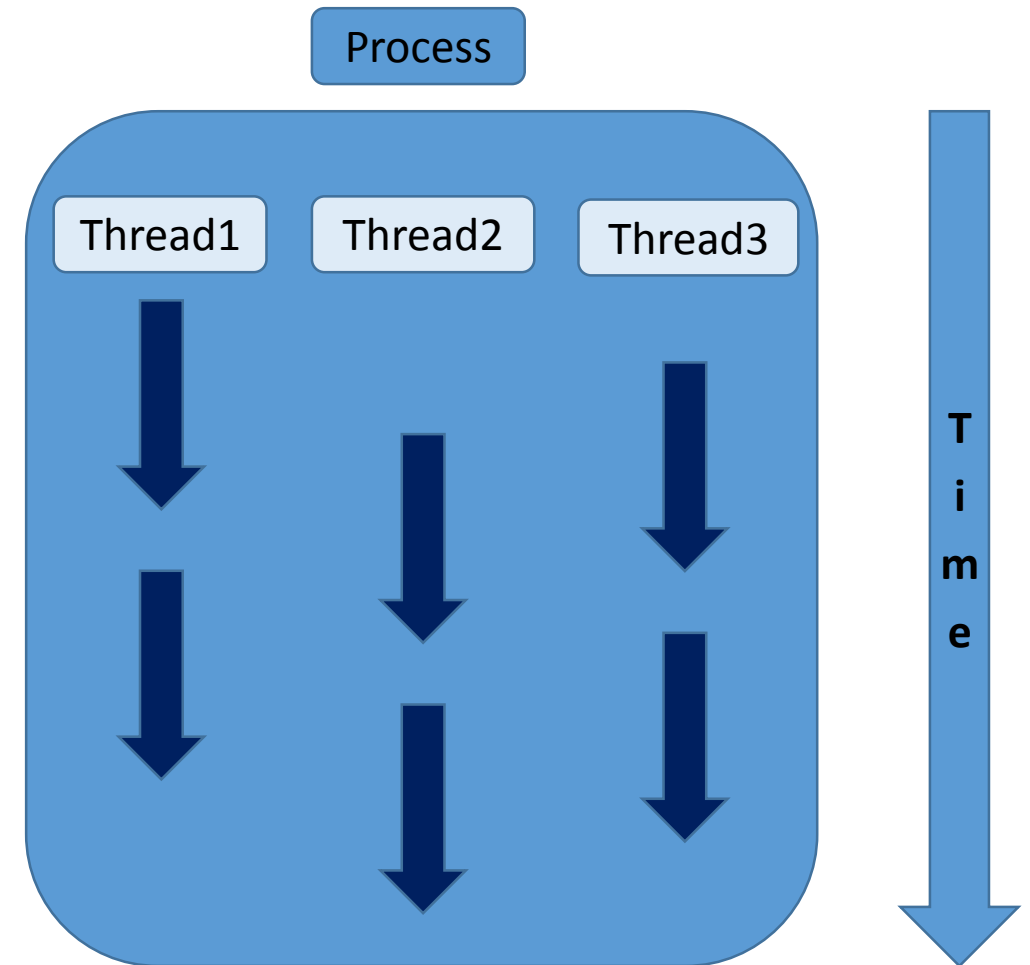


UNIDAD I, Introducción a la Programación Visual

1.4 La programación basada en eventos

- Programación asíncrona mecanismos.

1. **Múltiples hilos (threading)**, un hilo es un conjunto de instrucciones independiente que se puede calendarizar en forma de paquete y que además, no comparte recursos.
2. Un hilo depende de un proceso únicamente, no puede ser movido entre procesos y todos los hilos que forman parte de un proceso, comparten los recursos del proceso tales como memoria, controladores de archivo y sockets.

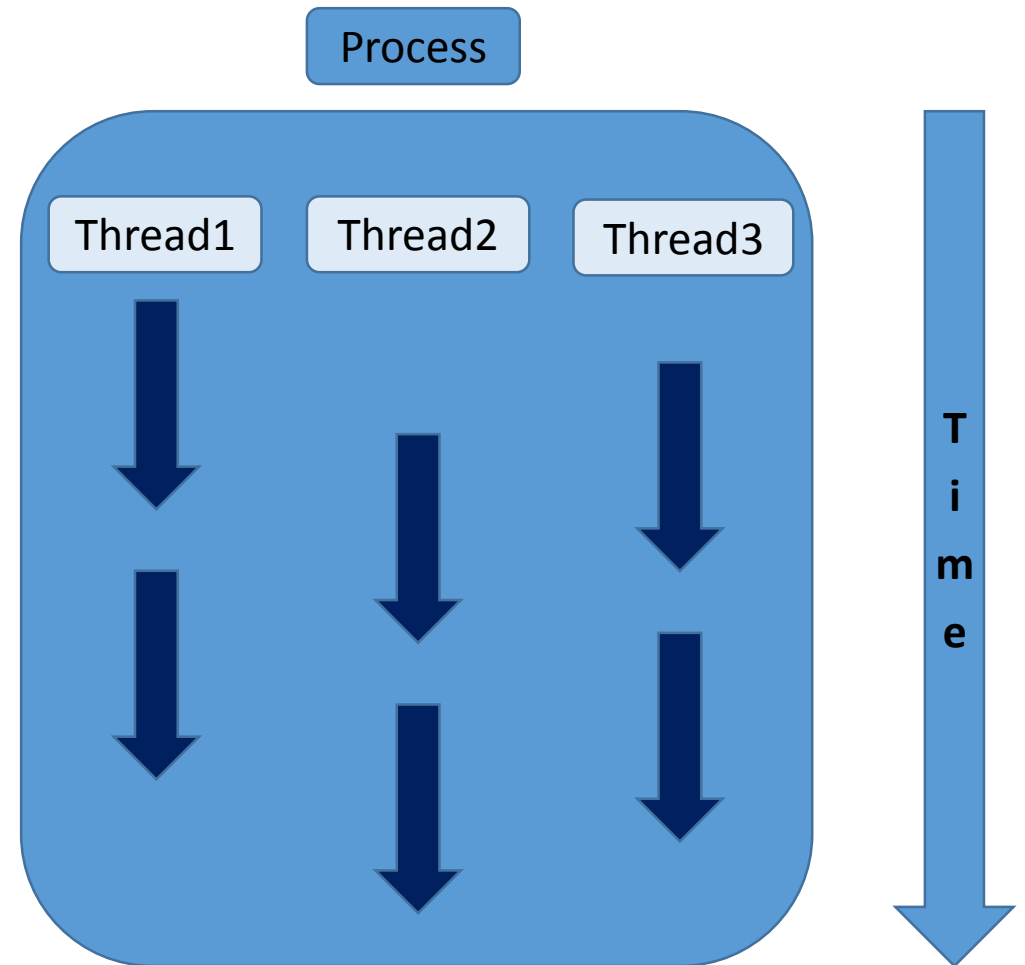


UNIDAD I, Introducción a la Programación Visual

1.4 La programación basada en eventos

- Programación asíncrona mecanismos.

1. **Múltiples hilos (threading)**, a diferencia de UNIX, en Windows los procesos son construcciones pesadas, esto es debido a la carga de las bibliotecas de ejecución Win32 y sus registros. Por lo que en Windows, se prefiere múltiples hilos para crear programación asíncrona en lugar de utilizar multiprocesos. En particular en Windows, es una buena practica reutilizar los hilos, ya que es costoso crearlos y estarlos destruyendo.

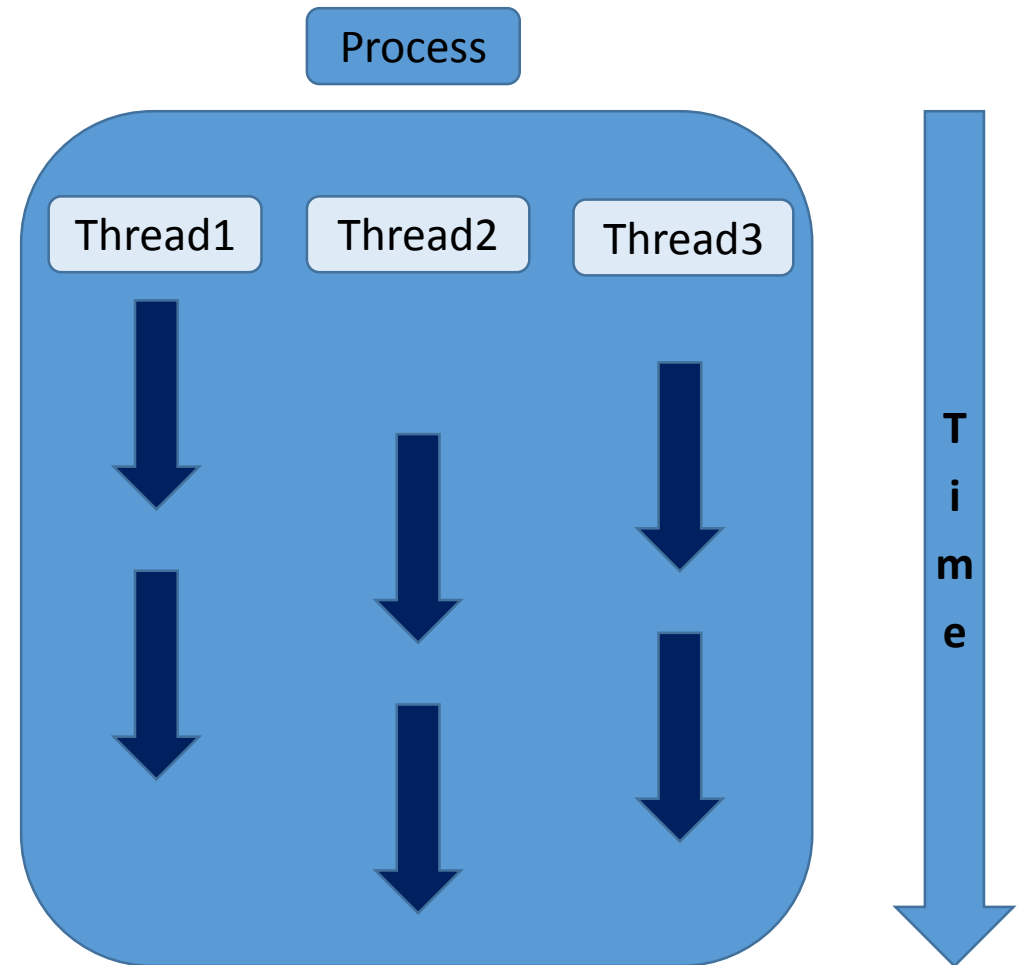


UNIDAD I, Introducción a la Programación Visual

1.4 La programación basada en eventos

- Programación asíncrona mecanismos.

1. **Thread Scheduler (Windows)**, como se puede observar, en algunas ocasiones algún hilo esta pendiente de ejecutarse a la espera de algún evento (SleepWaitJoin).
2. Diferentes procesos pueden ser ejecutados en diferentes prioridades.



UNIDAD I, Introducción a la Programación Visual

1.4 La programación basada en eventos

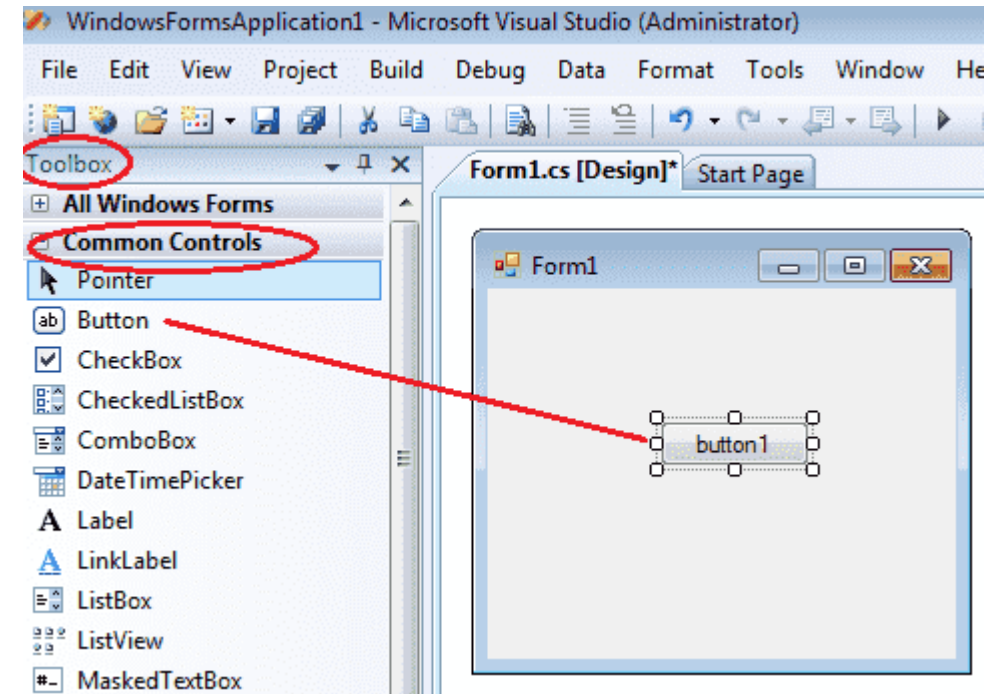
Las interfaces graficas de usuario en C#, están basadas en eventos, tales como:

- Movimientos del mouse
- Teclear por parte del usuario
- Tiempo
- Toques a la pantalla
- Deslizar la pantalla con los dedos

UNIDAD I, Introducción a la Programación Visual

1.5 El concepto de componentes visuales

Un componente es visual cuando tiene una representación gráfica en tiempo de diseño y ejecución por ejemplo: botones, barras de scroll, cuadros de edición, etc., y se dice no visual en caso de que adquieran valores en la fase de ejecución por ejemplo: temporizadores, cuadros de diálogo-no visibles en la fase de diseño, etc.



UNIDAD I, Introducción a la Programación Visual

1.6 Paradigmas de Programación

Un paradigma es ejemplo, modelo o patrón que sigue algún objeto.

En el caso de la programación, es un estilo de desarrollo de programas, o un modelo para resolver problemas computacionales.

Un paradigma te dice que estructuras de programación usar y cuando usarlas.

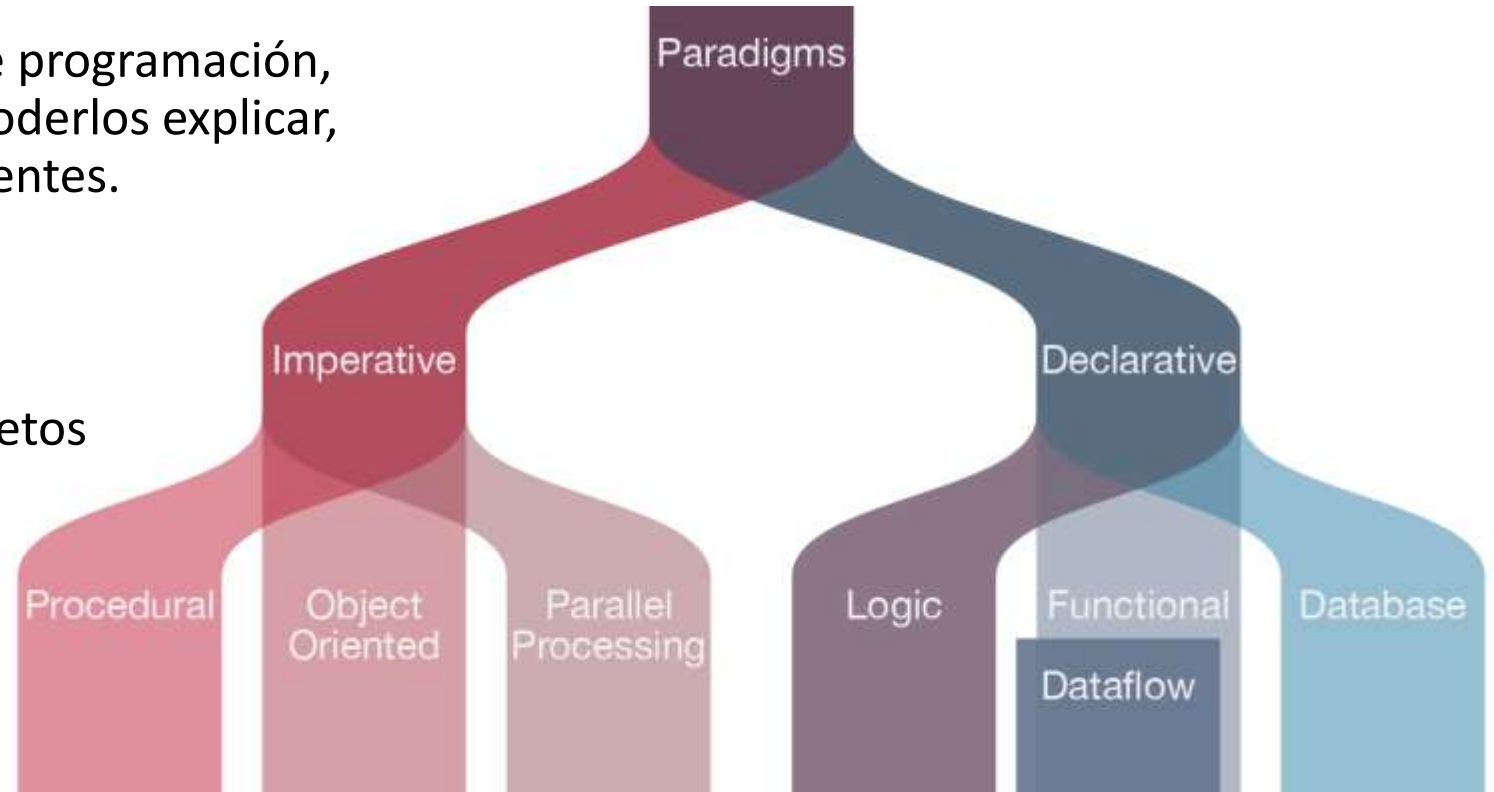
Los lenguajes de programación se encuentran en uno o varios paradigmas (multiparadigmas) a partir del tipo de órdenes que permiten implementar y que tiene una relación directa con su sintaxis.

UNIDAD I, Introducción a la Programación Visual

1.6 Paradigmas de Programación

Existen diferentes paradigmas de programación, sin embargo es necesario para poderlos explicar, entender otros paradigmas existentes.

- Programación Imperativa
- Programación Declarativa
- Programación Orientada a Objetos
- Programación Estructurada
- Programación Funcional



UNIDAD I, Introducción a la Programación Visual

1.6 Paradigmas de Programación

Programación Imperativa

- Históricamente, los primeros programas fueron escritos utilizando este paradigma.
- Un programa estaba construido con una serie de instrucciones específicas, donde cada instrucción hace alguna acción bien definida, como por ejemplo cambiar el valor de una localidad de memoria, imprimir un resultado.
- El lenguaje ensamblador es un ejemplo de este paradigma.

```
Q1:  proc near
      push    sPassword
      call    _strlen
      pop     ecx
      mov     esi, eax
      mov     ebx, offset sMyPassword
      push    ebx
      call    _strlen
      pop     ecx
      cmp     esi, eax
      jz      short loc_4012B2
      xor     eax, eax
      jmp     short end_proc
loc_4012B2:
      push    esi
      push    ebx
      push    sPassword
      call    _strcmp
      add     esp, 8
      test    eax, eax
      jnz     short loc_4012CC
      mov     eax, 1
      jmp     short end_proc
loc_4012CC:
      xor     eax, eax
end_proc:
      pop     esi
      pop     ebx
      pop     ebp
      retn
endp
```

UNIDAD I, Introducción a la Programación Visual

1.6 Paradigmas de Programación

Programación Declarativa

- Tratan de expresar “que hacer” en lugar de “como hacerlo”.
- Un tipo de programación declarativa, es la programación basada en reglas.
- Un tipo importante de programación declarativa es la programación lógica, donde se utilizan axiomas para describir reglas.

Prolog = Programming by Logic

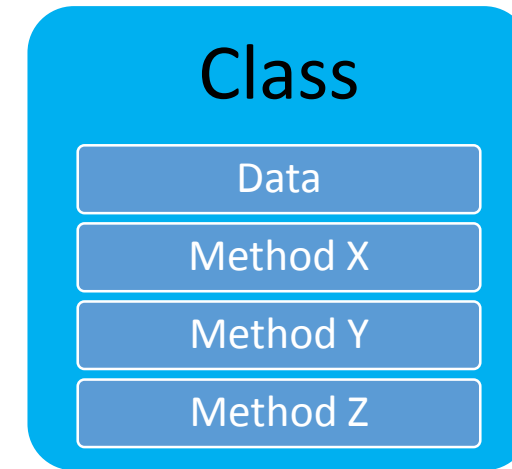
- Prolog is different from all languages you’ve learned
 - Prolog is a Logic Programming (LP) Language
 - you only need to specify the goals (what)
 - but not the strategy to reach this goal (how)
 - Prolog figures it out for you automatically!
 - this is called “declarative programming”
(alone with functional programming, FP)
 - C/C++/Java/Python are Imperative (IP) Languages
 - you specify how to reach some goal (instructions)
 - but leaving the real goal implicit (in comments)
 - LP/FP is cleaner, safer, prettier, while IP is dirtier but faster

UNIDAD I, Introducción a la Programación Visual

1.6 Paradigmas de Programación

Programación Orientada a Objetos

- Descubierta en 1966 por Ole Johan Dahl y Kristen Nygaard
- Conjunto de objetos que son manipulados por acciones
- El estado de cada objeto y las acciones que manipulan ese estado son definidas una vez que el objeto es creado.



UNIDAD I, Introducción a la Programación Visual

1.6 Paradigmas de Programación

Programación Estructurada

- Descubierta por Edsger Wybe Dijkstra in 1968
- Descubrió que el uso de sentencias de “goto” era dañino para la estructura de un programa.
- Reemplazo su uso, con construcciones de “if/then/else” y “do/while/until”
- La programación estructurada impone disciplina en el control del flujo directo del programa.

UNIDAD I, Introducción a la Programación Visual

1.6 Paradigmas de Programación

Programación Funcional

- Alonzo Church inventó en 1936 λ -calculus que da origen al lenguaje de programación llamado LISP.
- Define un programa como una función matemática que convierte unas entradas en unas salidas, sin ningún estado interno y sin ningún efecto colateral.

UNIDAD I, Introducción a la Programación Visual

1.6 Paradigmas de Programación

Imperativo	Declarativo		Orientado a Objetos
	Programación Funcional	Programación Lógica	
Algol Cobol PL/1 Ada C Modula-3	LISP Haskell ML Miranda APL	Prolog	SmallTalk Simula C++ Java C#

UNIDAD I, Introducción a la Programación Visual

1.6 Paradigmas de Programación

¿Cuál paradigma es mejor o cual debo elegir para programar?

Todo depende del propósito del desarrollo que se este tratando de hacer. Si es un desarrollo a bajo nivel, que hará interface con hardware, quizá sea mejor utilizar el imperativo. Si es un desarrollo que deberá simular un comportamiento o una actividad humana, el paradigma orientado a objetos puede dar mayor resultado.

UNIDAD I, Introducción a la Programación Visual

TAREA # 1 INVESTIGAR:

1. En que consiste cada uno de los siguientes paradigmas de programación y ejemplos de lenguajes de programación:
 1. Programación Estructurada
 2. Programación Visual
 3. Programación Orientada a Objetos
 4. Programación Orientada a Eventos
2. ¿A qué se refiere el concepto de RAD (rapid application development)?
3. ¿Qué es un IDE? y mencionar ejemplos
4. Conclusiones

UNIDAD I, Introducción a la Programación Visual

1.7 Entorno de Desarrollo Integrado (IDE)

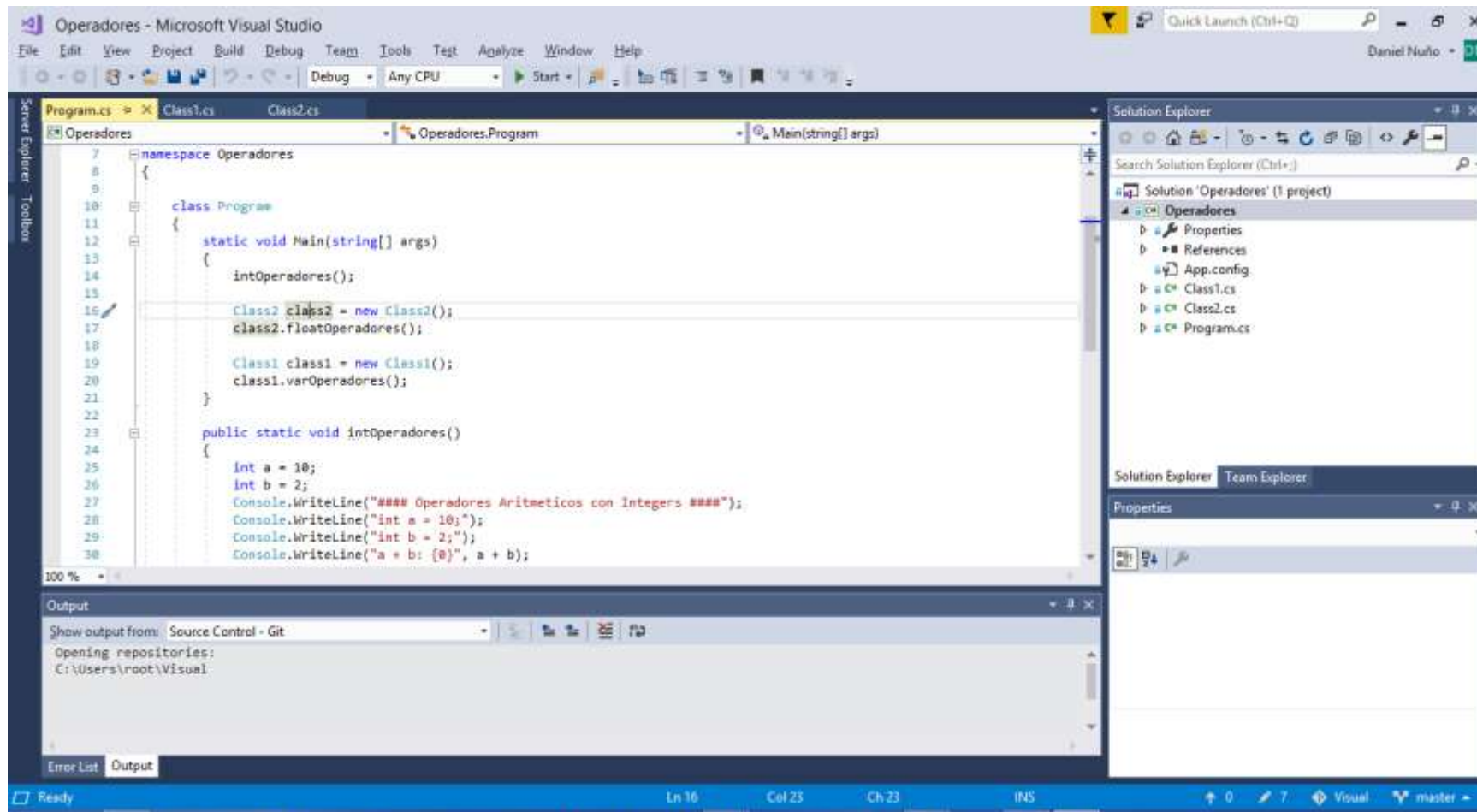
Un entorno de desarrollo integrado, en inglés Integrated Development Environment (IDE), es una aplicación informática que proporciona servicios integrales para facilitarle al desarrollador o programador el desarrollo del software.

Normalmente, un IDE consiste de un editor de código fuente, herramientas de construcción automáticas y un depurador. La mayoría de los IDE tienen auto-completado inteligente de código (IntelliSense).

Algunos IDE contienen un compilador, un intérprete, o ambos, tales como NetBeans y Eclipse; otros no, tales como SharpDevelop y Lazarus

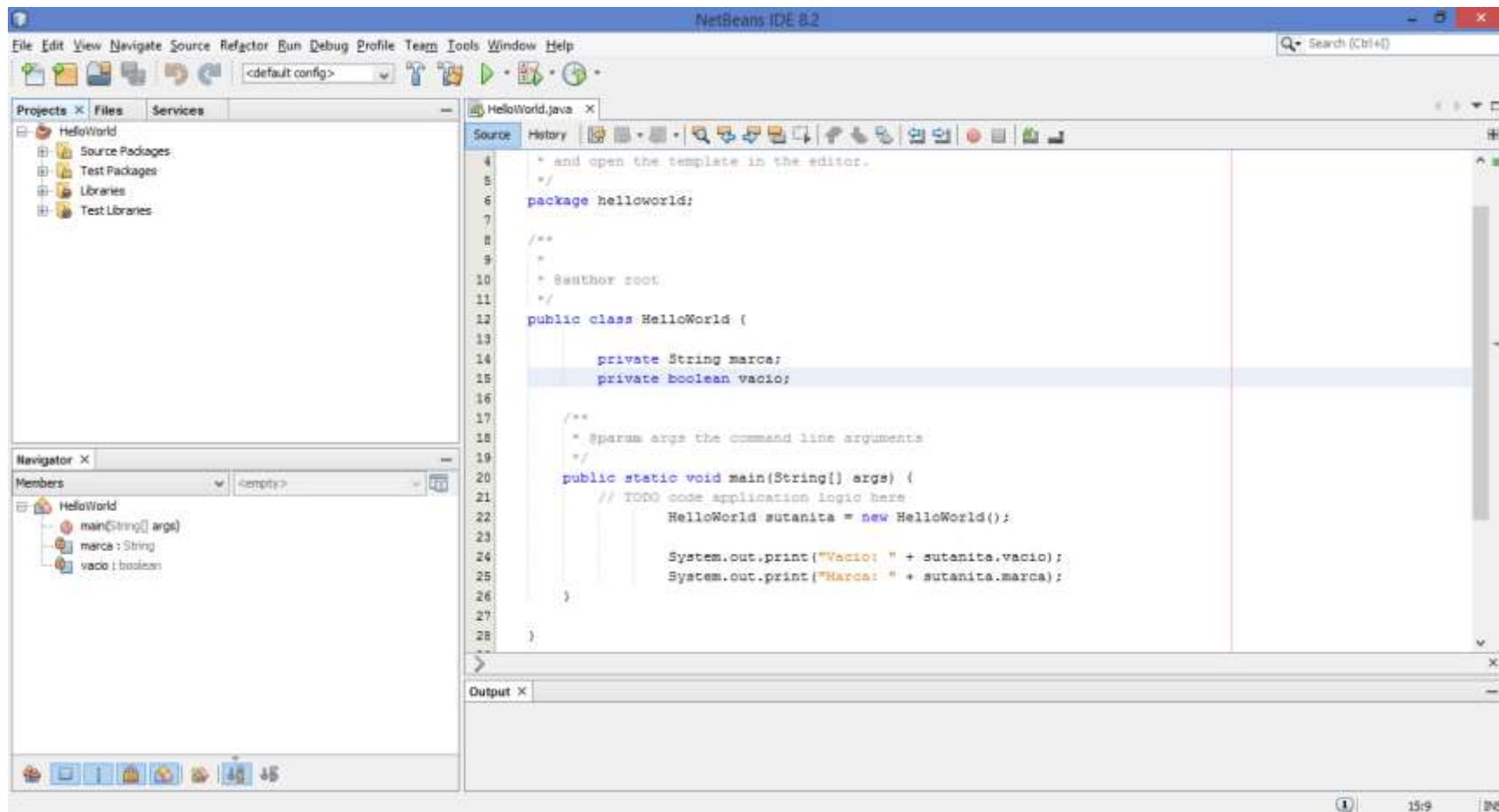
UNIDAD I, Introducción a la Programación Visual

1.7 Entorno de Desarrollo Integrado (IDE)



UNIDAD I, Introducción a la Programación Visual

1.7 Entorno de Desarrollo Integrado (IDE)



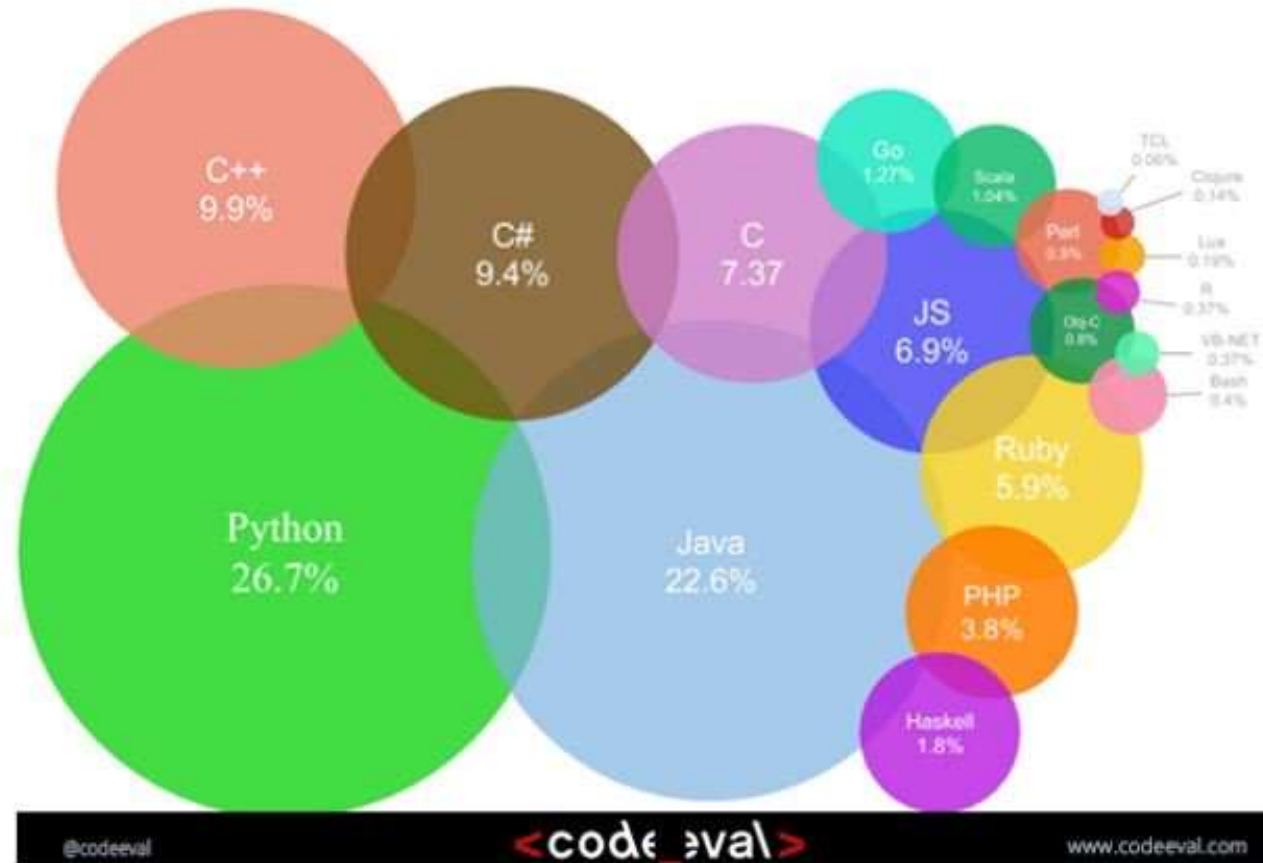
UNIDAD I, Introducción a la Programación Visual

1.7 Entorno de Desarrollo Integrado (IDE)

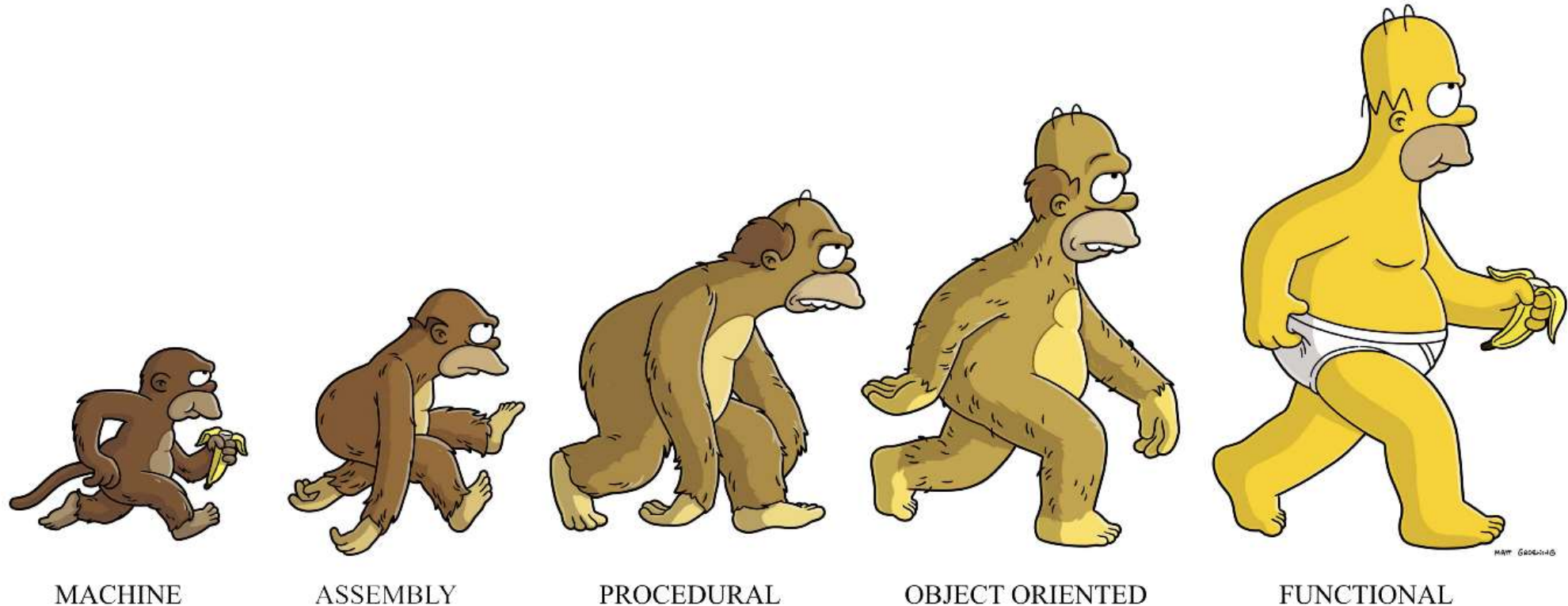


UNIDAD I, Introducción a la Programación Visual

Most Popular Coding Languages of 2016



UNIDAD I, Introducción a la Programación Visual



UNIDAD I, Introducción a la Programación Visual

TAREA # 2 INVESTIGAR:

IDE	Características	Lenguajes de Programación	Sistema Operativo	Paradigma

Unidad II, la plataforma Microsoft .NET

Unidad II, la plataforma Microsoft .NET

2.1 Conceptos básicos de la plataforma .NET

2.2 Introducción al Framework de .NET

2.3 Introducción al Managed Execution Environment

2.4 Desarrollo y versiones

2.5 El Common Type System

2.6 La Biblioteca de Clases

2.7 Delegación y eventos en .NET

2.8 Gestión de memoria y recursos

2.9 Serialización

2.10 Remoting y servicios Web XML

Unidad II, la plataforma Microsoft .NET

2.1 Conceptos básicos de la plataforma .NET

¿Qué NO es .NET?

- .NET no es un sistema operativo
- .NET no es un Lenguaje de Programación
- .NET no es un Entorno de Desarrollo (IDE)
- .NET no es un servidor de aplicaciones (Application Server)

Unidad II, la plataforma Microsoft .NET

2.1 Conceptos básicos de la plataforma .NET

¿Qué Sí es .NET?

- Plataforma que engloba distintas aplicaciones, servicios y conceptos, y que en conjunto permiten el desarrollo y la ejecución de aplicaciones.

Unidad II, la plataforma Microsoft .NET

2.1 Conceptos básicos de la plataforma .NET

Componentes

- **Un entorno de ejecución de aplicaciones**, también llamado “Runtime”, que es un componente de software cuya función es la de ejecutar las aplicaciones .NET e interactuar con el sistema operativo ofreciendo sus servicios y recursos.
- **Un conjunto de bibliotecas de funcionalidades y controles reutilizables**, con una enorme cantidad de componentes ya programados listos para ser consumidos por otras aplicaciones.
- **Un conjunto de lenguajes de programación de alto nivel**, junto con sus compiladores y linkers, que permitirán el desarrollo de aplicaciones sobre la plataforma .NET.
- **Un conjunto de utilitarios y herramientas de desarrollo** para simplificar las tareas más comunes del proceso de desarrollo de aplicaciones
- **Documentación y guías de arquitectura**, que describen las mejores prácticas de diseño, organización, desarrollo, prueba e instalación de aplicaciones .NET

Unidad II, la plataforma Microsoft .NET

2.1 Conceptos básicos de la plataforma .NET

Características

- Es una ***plataforma de ejecución intermedia***, ya que las aplicaciones .NET no son ejecutadas directamente por el sistema operativo, como ocurre en el modelo tradicional de desarrollo. Las aplicaciones .NET son ejecutadas contra un componente de software llamado **Entorno de Ejecución (“Runtime”, o “Máquina Virtual”)**.
- Este componente es el encargado de manejar el ciclo de vida de cualquier aplicación .NET, iniciándola, deteniéndola, interactuando con el Sistema Operativo y proveyéndole servicios y recursos en tiempo de ejecución.
- La plataforma Microsoft .NET está **completamente basada en el paradigma de Orientación a Objetos**
- .NET es **multi-lenguaje**: esto quiere decir que para poder codificar aplicaciones sobre esta plataforma no necesitamos aprender un único lenguaje específico de programación de alto nivel, sino que se puede elegir de una amplia lista de opciones.

Unidad II, la plataforma Microsoft .NET

2.1 Conceptos básicos de la plataforma .NET

Características

- .NET permite el desarrollo de aplicaciones empresariales de misión crítica para la operación de tipos variados de organizaciones.
- Si bien también es muy atrayente para desarrolladores no profesionales, estudiantes y entusiastas, su verdadero poder radica en su capacidad para soportar las aplicaciones más grandes y complejas.
- .NET fue diseñado para proveer un único modelo de programación, uniforme y consistente, para todo tipo de aplicaciones (formularios Windows, de consola, aplicaciones Web, aplicaciones móviles, etc.) y para cualquier dispositivo de hardware (PC's, Pocket PC's, Teléfonos Celulares Inteligentes, también llamados "SmartPhones", Tablet PC's, etc.).
- Esto representa un gran cambio con respecto a las plataformas anteriores a .NET, las cuales tenían modelos de programación, bibliotecas, lenguajes y herramientas distintas según el tipo de aplicación y el dispositivo de hardware.

Unidad II, la plataforma Microsoft .NET



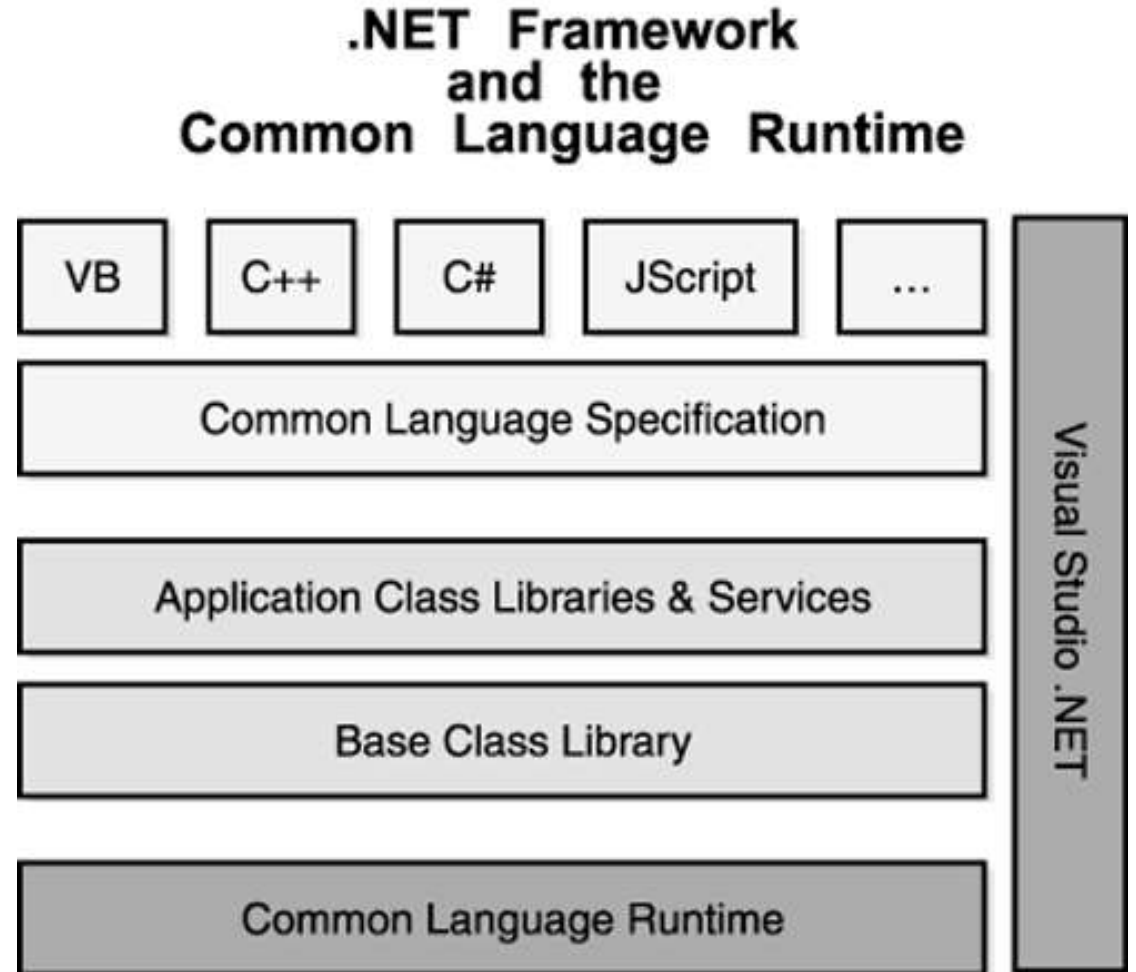
2.1 Conceptos básicos de la plataforma .NET

Características

- Uno de los objetivos de diseño de .NET fue que tenga la posibilidad de interactuar e integrarse fácilmente con aplicaciones desarrolladas en plataformas anteriores
- Particularmente en **Component Object Model (COM)**, ya que aún hoy existen una gran cantidad de aplicaciones desarrolladas sobre esa base.
- .NET no sólo se integra fácilmente con aplicaciones desarrolladas en otras plataformas Microsoft, sino también con aquellas desarrolladas en otras plataformas de software, sistemas operativos o lenguajes de programación.
- Para esto hace un uso extensivo de numerosos estándares globales que son de uso extensivo en la industria. Algunos ejemplos de estos estándares son XML, HTTP, SOAP, WSDL y UDDI.

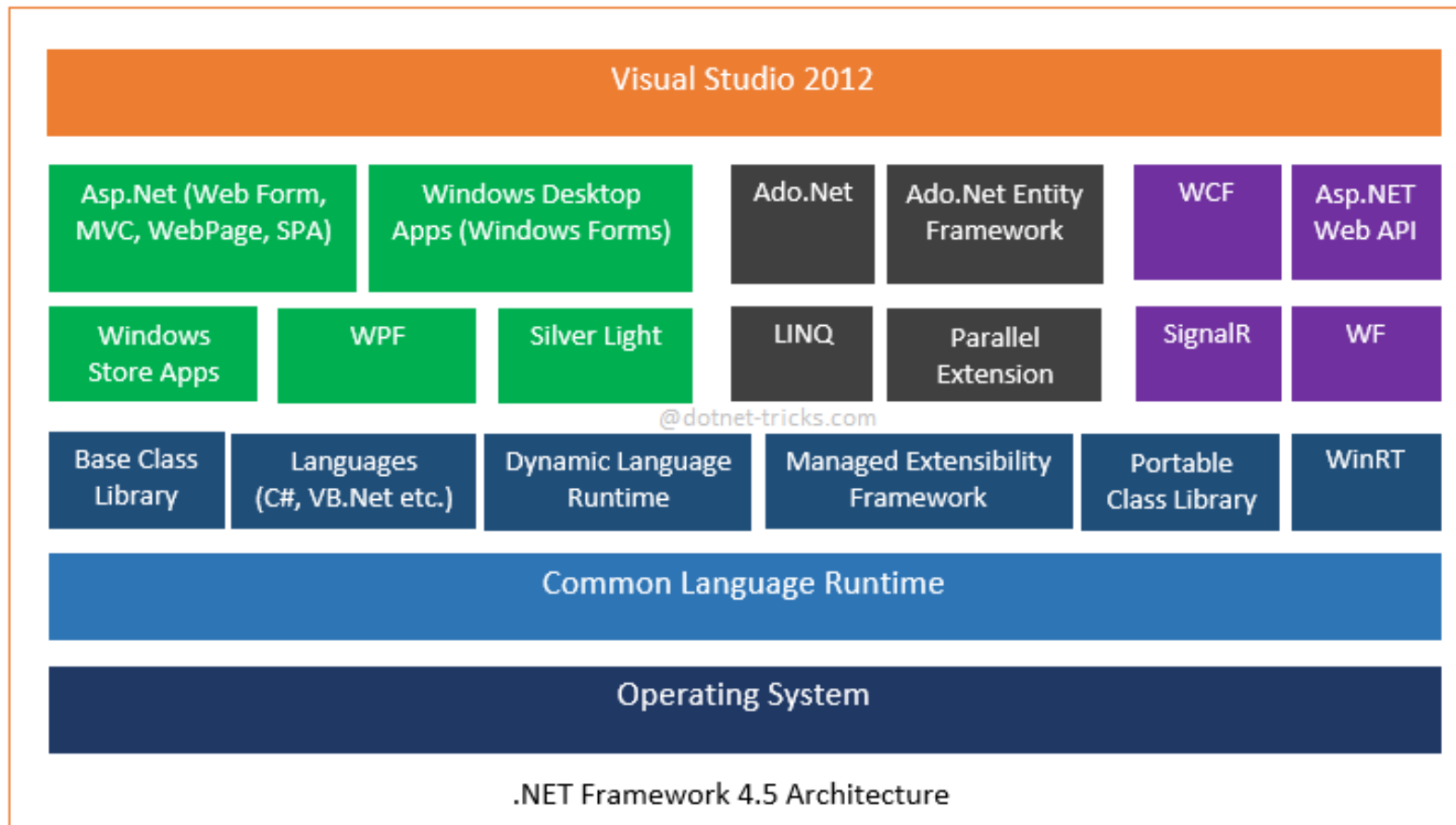
Unidad II, la plataforma Microsoft .NET

2.2 Introducción al Framework de .NET



Unidad II, la plataforma Microsoft .NET

2.2 Introducción al Framework de .NET



Unidad II, la plataforma Microsoft .NET

2.2 Introducción al Framework de .NET

Common Language Runtime (CLR)

El CLR es el verdadero núcleo del Framework de .NET, ya que es el entorno de ejecución en el que se cargan las aplicaciones desarrolladas en los distintos lenguajes.

La herramienta de desarrollo compila el código fuente de cualquiera de los lenguajes soportados por .NET en un mismo código intermedio **Microsoft Intermediate Language (MSIL)**. Para generar dicho código el compilador se basa en el **Common Language Specification (CLS)** que determina las reglas necesarias para crear código MSIL compatible con el CLR.

De esta forma, indistintamente de la herramienta de desarrollo utilizada y del lenguaje elegido, el código generado es siempre el mismo, ya que el MSIL es el único lenguaje que entiende directamente el CLR.

Este código es transparente al desarrollo de la aplicación ya que lo genera automáticamente el compilador.

Unidad II, la plataforma Microsoft .NET

2.2 Introducción al Framework de .NET

Common Language Runtime (CLR) execution model

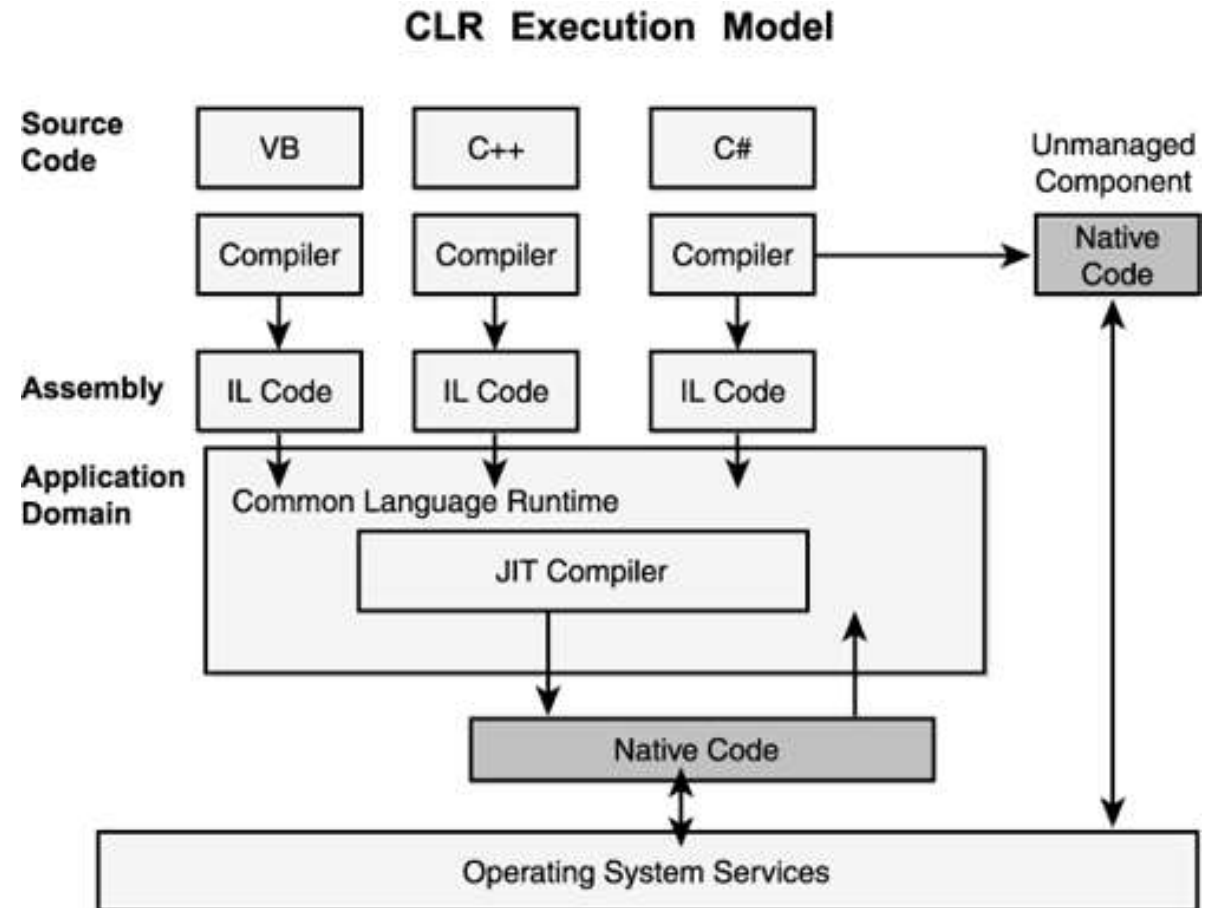
Assembly, unidad principal en .NET.

PE, Portable Executable.

Módulo, archivo que contiene código ejecutable, un Assembly puede contener uno o varios módulos.

AppDomain, el dominio de una aplicación, se refiere a un proceso ligero. Se utiliza para separar el dominio de la aplicación del resto de los procesos en ejecución.

Actividad: (PEDump)

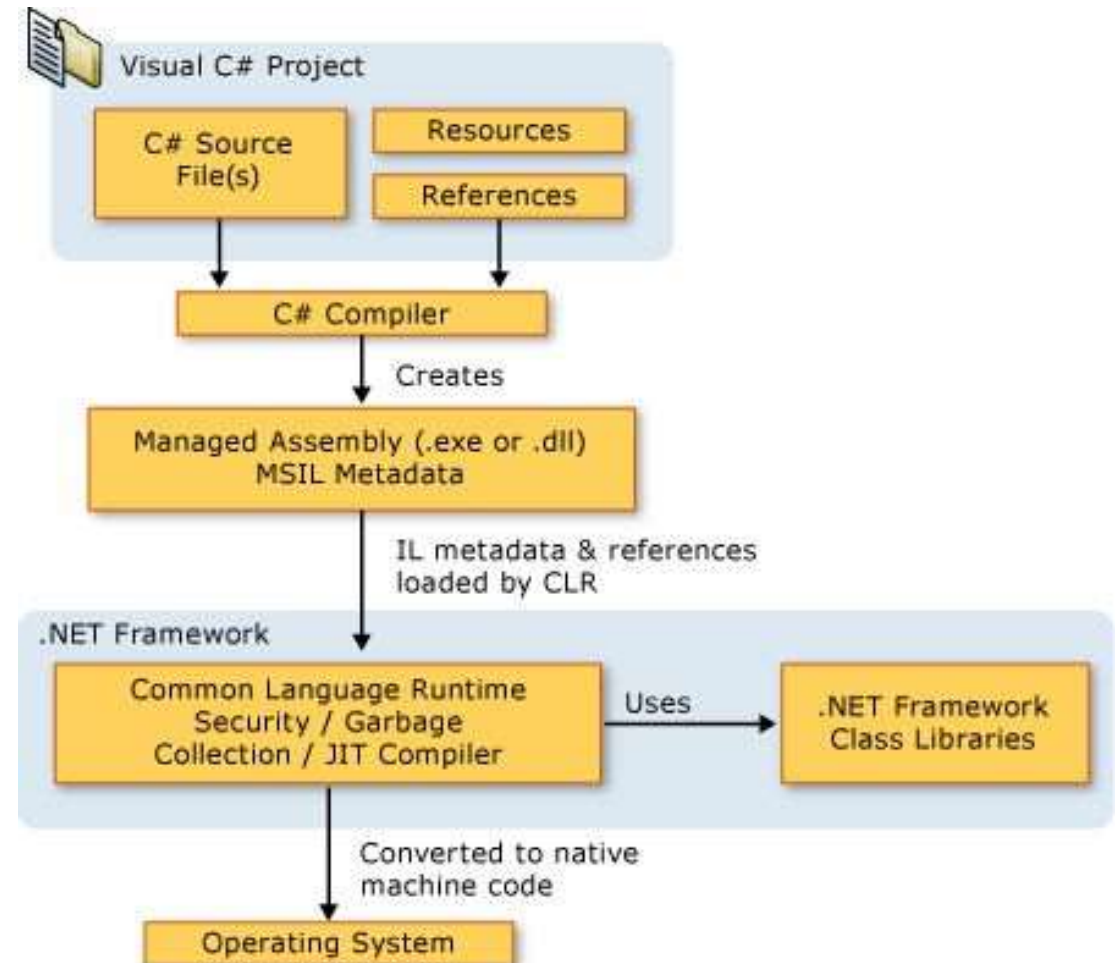


Unidad II, la plataforma Microsoft .NET

2.2 Introducción al Framework de .NET

Common Language Runtime (CLR) funciones

- Convierte código en IL
- Manejo de excepciones
- Seguridad en los tipos de dato
- Manejo de Memoria (garbage collector)
- Seguridad
- Performance
- Independencia del lenguaje
- Independencia de la plataforma
- Independencia en la arquitectura



Unidad II, la plataforma Microsoft .NET

2.2 Introducción al Framework de .NET

Common Language Runtime (CLR) **componentes:**

Cargador de clase, Se usa para cargar todas las clases en tiempo de ejecución.

MSIL a código nativo, El compilador Just In Time (JIT) convertirá el código MSIL en código nativo.

Administrador de código, Administra el código en tiempo de ejecución.

Recolector de basura, Gestiona la memoria. Recoge todos los objetos no utilizados y destrúyelos para reducir la memoria.

Subproceso de soporte, Es compatible con multihilo de nuestra aplicación.

Controlador de excepciones, Maneja excepciones en tiempo de ejecución.

Unidad II, la plataforma Microsoft .NET

2.2 Introducción al Framework de .NET

Compilador Just In Time (JIT)

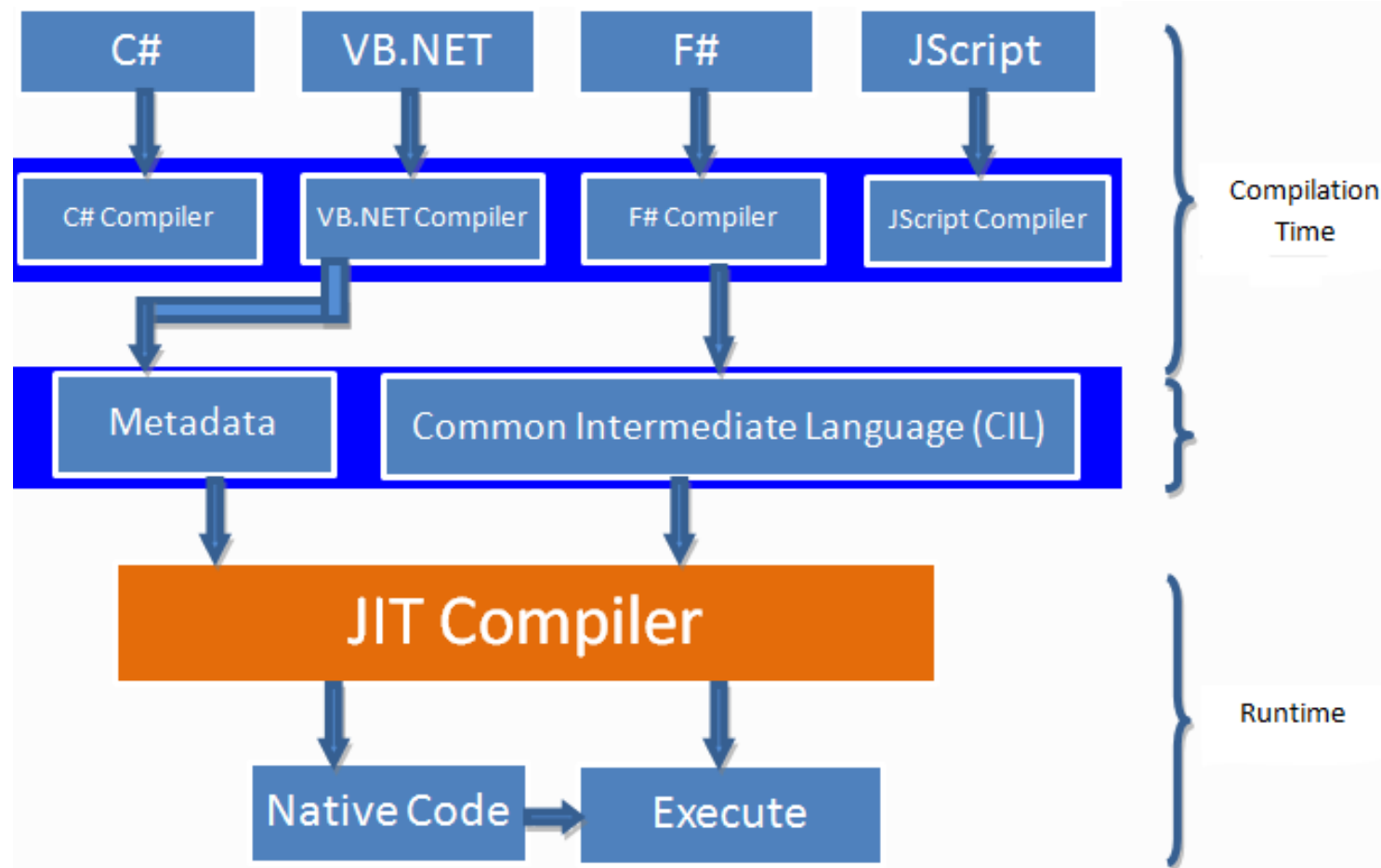
Sin embargo, el código generado en MSIL no es código máquina y por tanto no puede ejecutarse directamente.

Se necesita un segundo paso en el que una herramienta denominada compilador JIT (Just-In-Time) genera el código máquina real que se ejecuta en la plataforma que tenga la computadora.

De esta forma se consigue con .NET cierta independencia de la plataforma, ya que cada plataforma puede tener su compilador JIT y crear su propio código máquina a partir del código MSIL.

La compilación JIT la realiza el CLR a medida que se invocan los métodos en el programa y el código ejecutable obtenido, se almacena en la memoria caché de la computadora, siendo recompilado sólo cuando se produce algún cambio en el código fuente.

Unidad II, la plataforma Microsoft .NET



Unidad II, la plataforma Microsoft .NET

2.3 Introducción al *Managed Execution Environment*

Common Language Runtime (CLR) es el ambiente de la maquina virtual en el que todos los lenguajes de .NET se ejecutan.

Es un ambiente de gestión de la ejecución (**Managed Execution Environment**), que provee servicios para la ejecución de programas.

Los servicios incluyen: seguridad, manejo de memoria, etc.

El soporte de CLR para la integración a través de distintos lenguajes, permite que componentes desarrollados en un lenguaje puedan ser usados por componentes desarrollados en otros.

Todos los lenguajes compilan hacia un lenguaje intermedio común. (IL)

Todas las aplicaciones creadas comparten un sistema de tipos común **Common Type System (CTS)**.

Unidad II, la plataforma Microsoft .NET

2.4 Desarrollo y versiones

- Cada versión de .NET Framework contiene **Common Language Runtime (CLR)**, las bibliotecas de clases base y otras bibliotecas
- Los usuarios pueden instalar y ejecutar varias versiones de .NET Framework en sus equipos. Al desarrollar o implementar una aplicación, puede que necesite conocer las versiones de .NET Framework que están instaladas en el equipo del usuario
- Las actualizaciones instaladas para cada versión de .NET Framework instalada en un equipo se enumeran en el Registro de Windows. Se puede utilizar el Editor del Registro (regedit.exe) para ver esta información.

Unidad II, la plataforma Microsoft .NET

Version Number	CLR Version	Release Date	Support Ended	Development Tool	Windows	Windows Server	Replaces
1	1	2/13/2002	2009-07-14[5]	Visual Studio .NET[6]	XP SP1[a]	N/A	N/A
1.1	1.1	4/24/2003	2015-06-14[5]	Visual Studio .NET 2003[6]	XP SP2, SP3[b]	2003	1.0[7]
2	2	11/7/2005	2011-07-12[5]	Visual Studio 2005[8]	N/A	2003, 2003 R2,[9] 2008 SP2, 2008 R2 SP1	N/A
3	2	11/6/2006	2011-07-12[5]	Expression Blend[10][c]	Vista	2008 SP2, 2008 R2 SP1	2
3.5	2	11/19/2007	N/A[5]	Visual Studio 2008[11]	7, 8, 8.1, 10[d]	2008 R2 SP1	2.0, 3.0
4	4	4/12/2010	2016-01-12[5]	Visual Studio 2010[12]	N/A	N/A	N/A
4.5	4	8/15/2012	2016-01-12[5]	Visual Studio 2012[13]	8	2012	4
4.5.1	4	10/17/2013	2016-01-12[5]	Visual Studio 2013[14]	8.1	2012 R2	4.0, 4.5
4.5.2	4	5/5/2014	N/A[5]	N/A	N/A	N/A	4.0–4.5.1
4.6	4	7/20/2015	N/A[5]	Visual Studio 2015[15]	10 v1507	N/A	4.0–4.5.2
4.6.1	4	2015-11-30[16]	N/A[5]	Visual Studio 2015 Update 1	10 v1511	N/A	4.0–4.6
4.6.2	4	2016-08-02[17]	N/A[5]		10 v1607	2016	4.0–4.6.1
4.7	4	2017-04-05[18]	N/A[5]	Visual Studio 2017	10 v1703	N/A	4.0–4.6.2
4.7.1	4	2017-10-17[19]	N/A[5]	Visual Studio 2017	10 v1709	2016 v1709	4.0–4.7
4.7.2	4	2018-04-30[20]	N/A[5]	Visual Studio 2017	10 v1803	N/A	4.0–4.7.1
4.8	4	Developing[21]	N/A	Visual Studio 2019 (Planning)[22]	10 v1903 (Planning)	N/A	4.0–4.7.2

Unidad II, la plataforma Microsoft .NET

TAREA # 3 Glosario: 1er Parcial

- Descripciones en no menos de tres líneas
- Ordenado alfabéticamente
- Con excelente ortografía
- Incrementando a lo largo del curso
- En cada parcial se preguntaran términos del Glosario que cuentan para la calificación

Unidad II, la plataforma Microsoft .NET

2.5 El Common Type System

- El **Common Type System (CTS)** define como los tipos de datos son declarados, usados y gestionados en el **Common Language Runtime (CLR)**.
- El CTS define un vasto grupo de tipos y operaciones, suficiente para soportar la completa implementación de varios lenguajes de programación.
- El CTS provee las bases para la integración a través de diferentes lenguajes de programación.
- Cada lenguaje de programación ofrece diferentes capacidades, pero no todos los lenguajes soportan cada una de las características del CTS.
- Un sub grupo de los tipos necesarios para soportar la integración a través de diferentes lenguajes se ha definido como el **Common Language Specification (CLS)**.

Unidad II, la plataforma Microsoft .NET

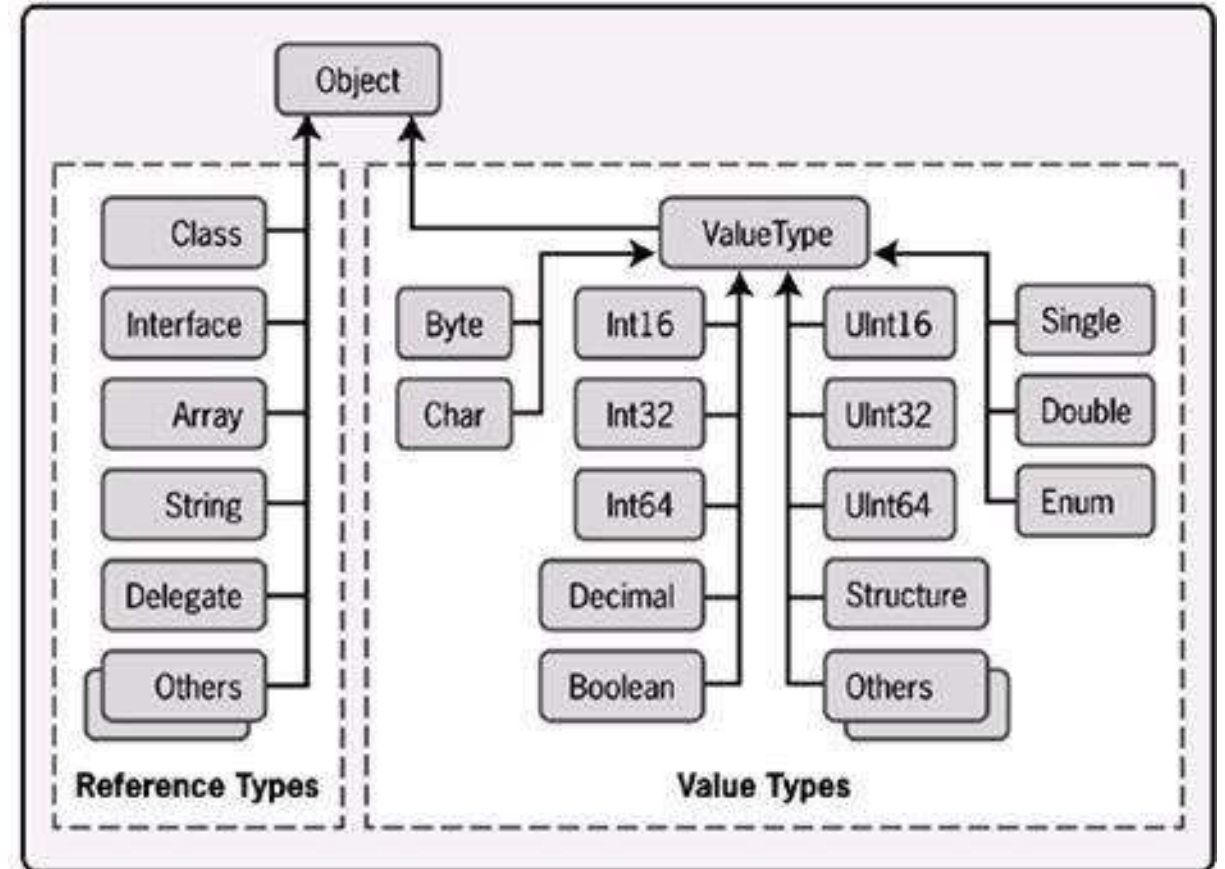
2.5 El Common Type System

- El Sistema de Tipos Común realiza las funciones siguientes:
- Establece un marco de trabajo que ayuda a permitir la integración entre lenguajes, la seguridad de tipos y la ejecución de código de alto rendimiento.
- Proporciona un modelo orientado a objetos que admite la implementación completa de muchos lenguajes de programación.
- Define reglas que deben seguir los lenguajes, lo que ayuda a garantizar que los objetos escritos en distintos lenguajes puedan interactuar unos con otros.
- Proporciona una biblioteca que contiene los tipos de datos primitivos (Boolean, Byte, Char, Int32 y UInt64) que se emplean en el desarrollo de aplicaciones.

Unidad II, la plataforma Microsoft .NET

2.5 El Common Type System

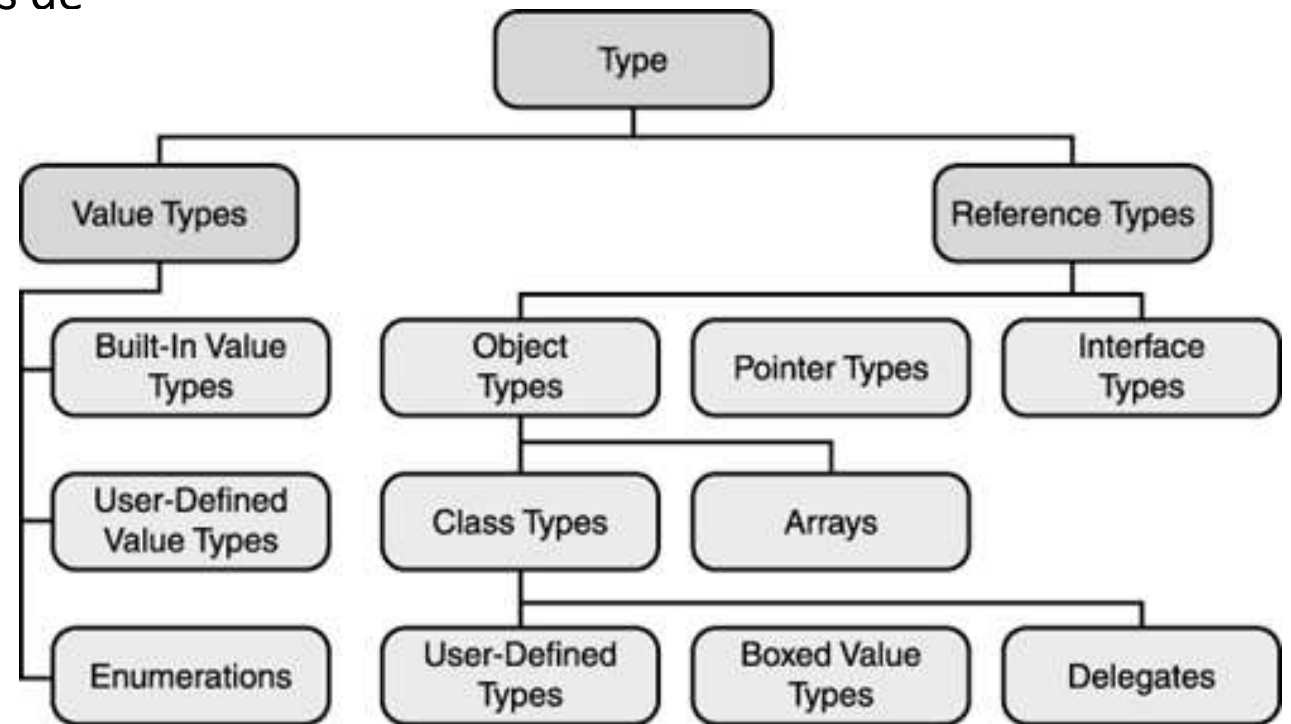
- Define un conjunto común de tipos de datos orientados a objeto
- Todos los lenguajes de programación que forman parte de .NET deben implementar los tipos definidos por el CTS
- Todo tipo hereda directa o indirectamente del tipo **System.Object**



Unidad II, la plataforma Microsoft .NET

2.5 El Common Type System

- El “Common Type System” tiene dos tipos de objetos:
 - Referencia
 - Valor



Unidad II, la plataforma Microsoft .NET

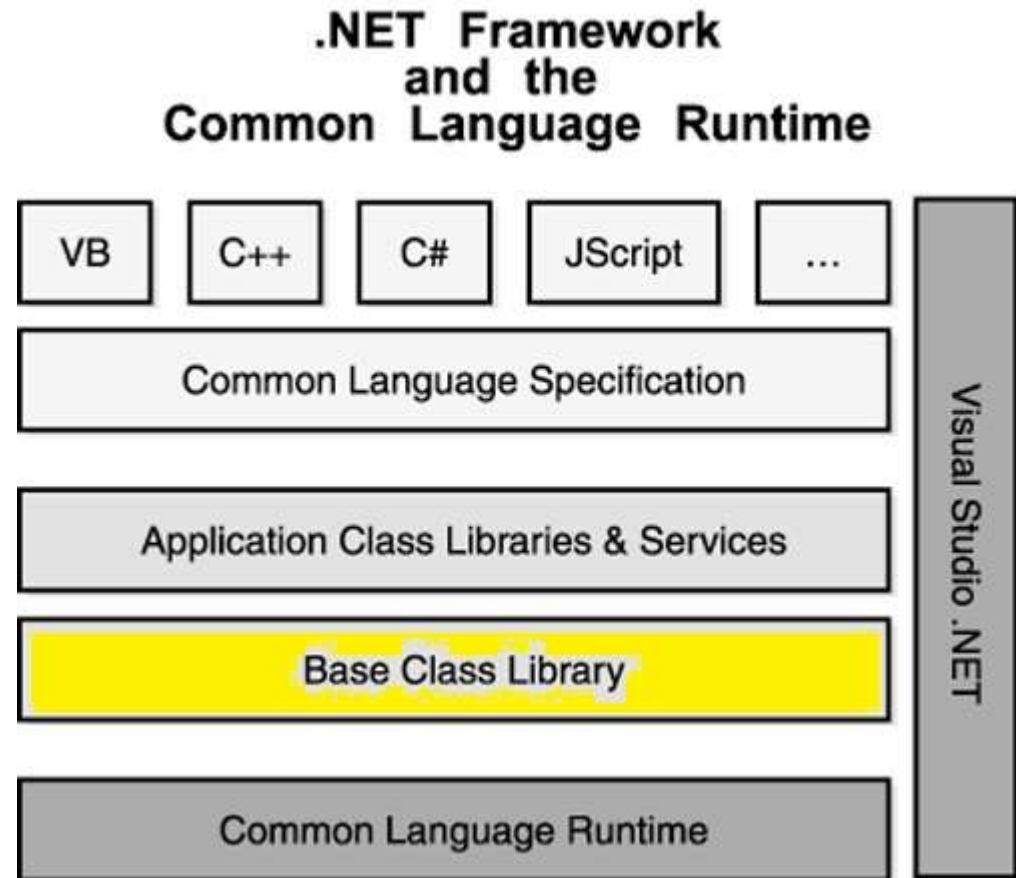
Common Language Specification (CLS)

- Conjunto de reglas que han de seguir las definiciones de tipos que se hagan usando un determinado lenguaje gestionado, si se desea que sean accesibles desde cualquier otro lenguaje gestionado por .NET
- Ejemplo de reglas significativas del CLS:
- Los tipos de datos básicos admitidos son: bool, char, byte, short, int, long, float, double, string y object
- Las tablas han de tener una o más dimensiones, y el número de dimensiones de cada tabla ha de ser fijo. Además, han de indexarse empezando a contar desde 0.
- En las definiciones de atributos sólo pueden usarse enumeraciones o datos de los siguientes tipos: System.Type, string, char, bool, byte, short, int, long, float, double y object

Unidad II, la plataforma Microsoft .NET

2.6 La biblioteca de clases

Base Class Library (BCL)



Unidad II, la plataforma Microsoft .NET

2.6 La biblioteca de clases

La BCL está constituida por espacios de nombres (**namespaces**). Cada espacio de nombres contiene tipos que se pueden utilizar en el programa: clases, estructuras, enumeraciones, delegados e interfaces.

Cuando se crea un proyecto de Visual C# en Visual Studio, se sigue haciendo referencia a las DLL más comunes de la clase base, pero, si necesita usar un tipo incluido en una DLL a la que aún no se hace referencia, deberá agregar la referencia de esa DLL.

La plataforma .NET incluye una colección de clases bien organizada cuya parte independiente del sistema operativo ha sido propuesta para su estandarización.

La BCL integra todas las tecnologías Windows en un marco único para todos los lenguajes de programación (Windows Forms, GDI+, Web Forms, Web Services, impresión, redes...).

La BCL proporciona un modelo orientado a objetos que sustituye a los componentes COM.

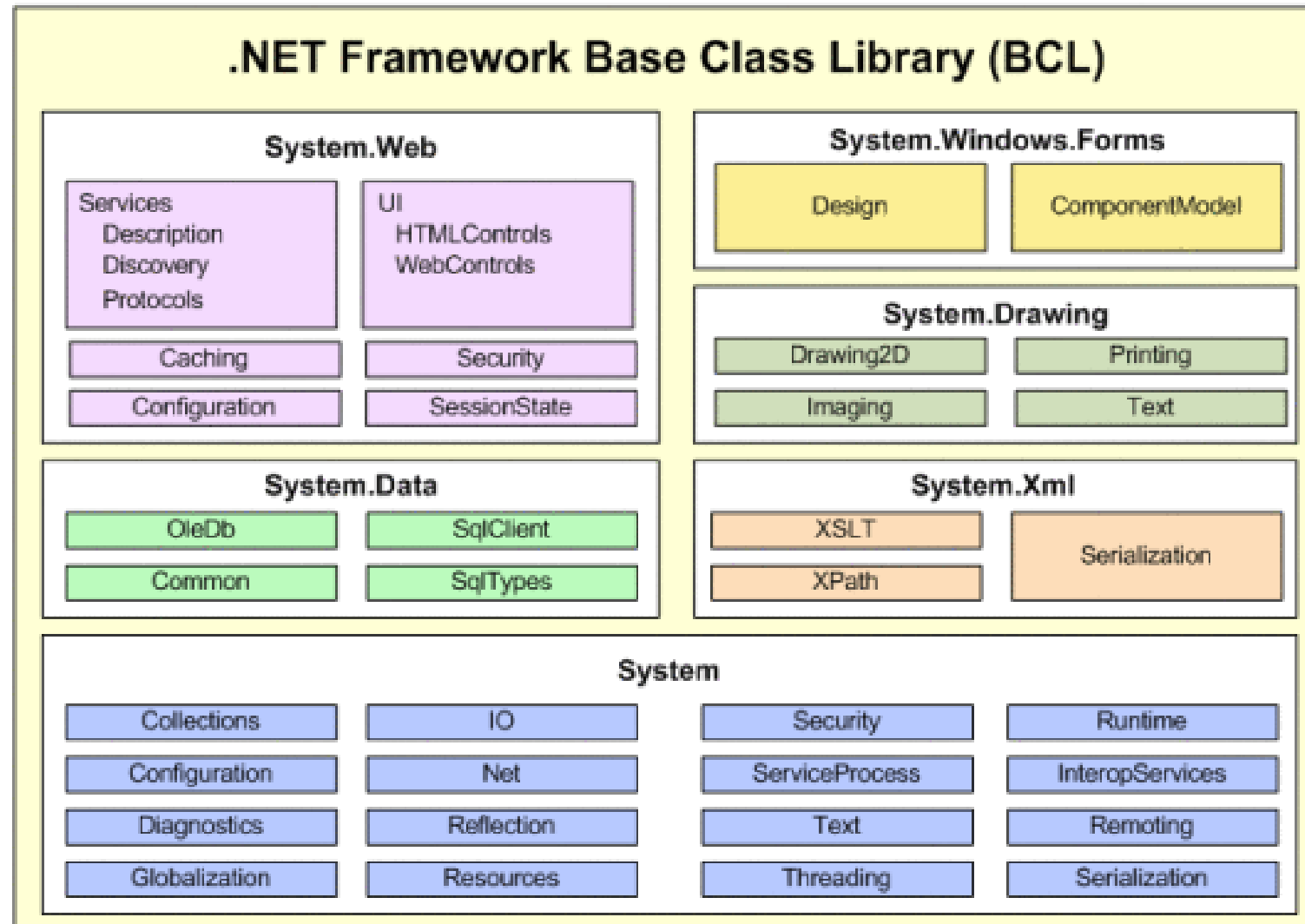
Unidad II, la plataforma Microsoft .NET

2.6 La biblioteca de clases

- System namespace

- Namespace raíz para tipos de datos de .NET
- Contiene clases que representan los tipos de datos base: Object, Byte, Char, Array, Int32, String
- Contiene mas de 100 clases que van de un rango para manejo de excepciones y hasta ejecución.
 - <https://docs.microsoft.com/en-us/dotnet/api/system>
- Además, contiene namespaces de segundo nivel.
- <https://docs.microsoft.com/dotnet/api>

Unidad II, la plataforma Microsoft .NET



Unidad II, la plataforma Microsoft .NET

~~2.1 Conceptos básicos de la plataforma .NET~~

~~2.2 Introducción al Framework de .NET~~

~~2.3 Introducción al Managed Execution Environment~~

~~2.4 Desarrollo y versiones~~

~~2.5 El Common Type System~~

~~2.6 La Biblioteca de Clases~~

2.7 Delegación y eventos en .NET

2.8 Gestión de memoria y recursos

2.9 Serialización

2.10 Remoting y servicios Web XML

Unidad II, la plataforma Microsoft .NET

2.7 Delegación y eventos en .NET

- Un **delegado** es una estructura de programación que nos permite *invocar a uno o varios métodos a la vez*.
- Estos métodos pueden encontrarse en la misma clase desde la que se invocan o en clases distintas asociadas a ésta.
- **Delegate** Permite extender y reutilizar la funcionalidad de una clase sin utilizar el mecanismo de herencia, donde la **Clase C** puede acceder a los métodos de la **Clase B** por medio de una instancia de esta última.

Unidad II, la plataforma Microsoft .NET

2.7 Delegación y eventos en .NET

Delegación

- Técnica en la que un objeto de cara al exterior expresa cierto comportamiento pero en realidad delega la responsabilidad de implementar dicho comportamiento a un objeto asociado en una relación inversa de responsabilidad.

Uso de los Delegados

- *En general, son útiles en todos aquellos casos en que interese pasar métodos como parámetros de otros métodos.*
- Se utilizan para diseñar marcos de referencia que pueden ser extendidos y flexibles.

Unidad II, la plataforma Microsoft .NET

- Un delegado es como un apuntador a una función
- Es un tipo de referencia y mantiene la referencia a un método.
- Derivado de System.Delegate
- Se declara usando la palabra “delegate”, seguido de la firma de una función.

```
public delegate void Print(int value);  
  
Print printDel = PrintNumber;  
  
public static void PrintNumber(int num)  
{  
    Console.WriteLine("Number: {0,-12:N0}", num);  
}
```

The diagram includes the following annotations:

- Access modifier**: Points to `public` in the delegate declaration.
- Delegate type**: Points to `void` in the delegate declaration.
- Delegate function signature**: Points to `Print(int value);` in the delegate declaration.
- Function signature must match with delegate signature**: Points to the `Print` type in the assignment and the `PrintNumber` method signature.
- © TutorialsTeacher.com**: A watermark is present in the center.

Unidad II, la plataforma Microsoft .NET

Se usa un delegado cuando:

- El que llama no necesita acceder otras propiedades o métodos en el objeto que implementa el método.
- Se usa un diseño orientado a eventos.

Unidad II, la plataforma Microsoft .NET

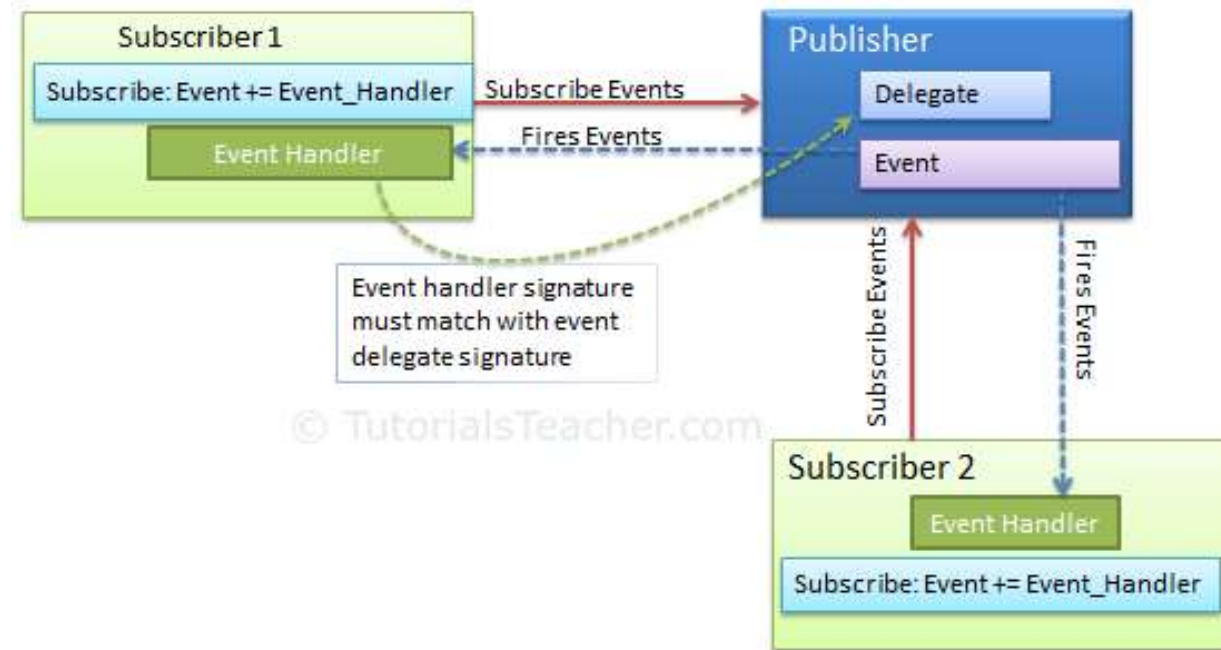
TAREA # 4 Delegados en C#:

1. Realizar ejercicios basados en:
https://www.tutorialspoint.com/csharp/csharp_delegates.htm
2. Pedir al usuario 5 números enteros.
3. Usar las siguientes operaciones (métodos):
 1. Sumar 5 números enteros
 2. Sacar el promedio de esos 5 números enteros
 3. Sacar el mayor de esos 5 números enteros
4. Subir el programa a github o bitbucket y enviar por email (daniel_nuno@hotmail.com) la liga al mismo.

Unidad II, la plataforma Microsoft .NET

2.7 Delegación y eventos en .NET

- Un **evento** es un mensaje que envía un objeto cuando ocurre una acción.
- La acción podría ser causada por la interacción del usuario (clic del botón), o podría ser iniciado por lógica de programa (cambiar un valor de propiedad).
- Es un mecanismo de comunicación entre objetos.
- Sirven para extender aplicaciones.



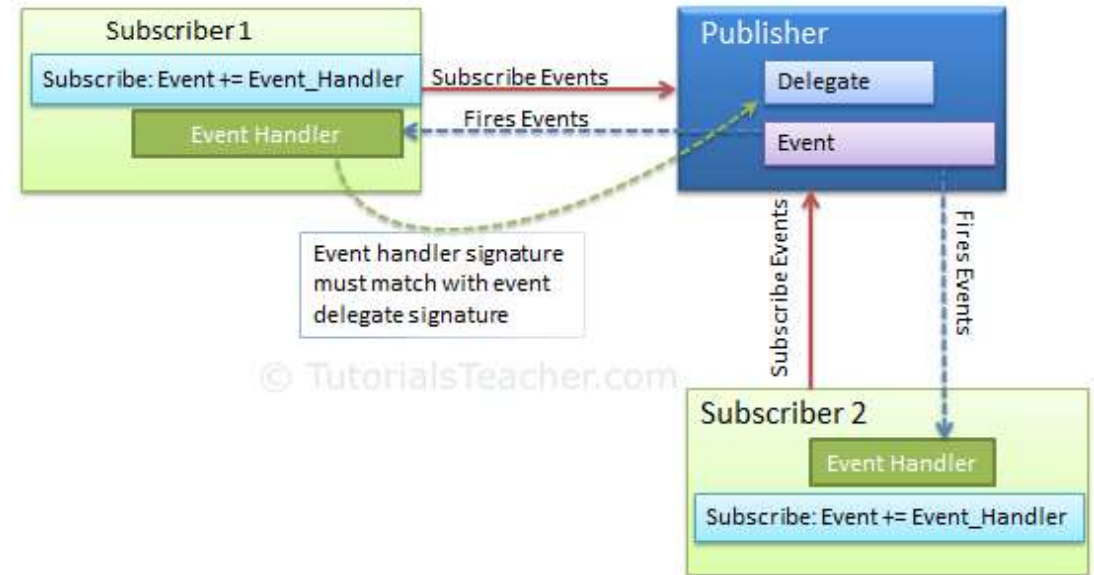
Unidad II, la plataforma Microsoft .NET

2.7 Delegación y eventos en .NET

Ejemplo:

Una marca reconocida publica un evento y además, usted recibe una notificación de ese evento por correo.

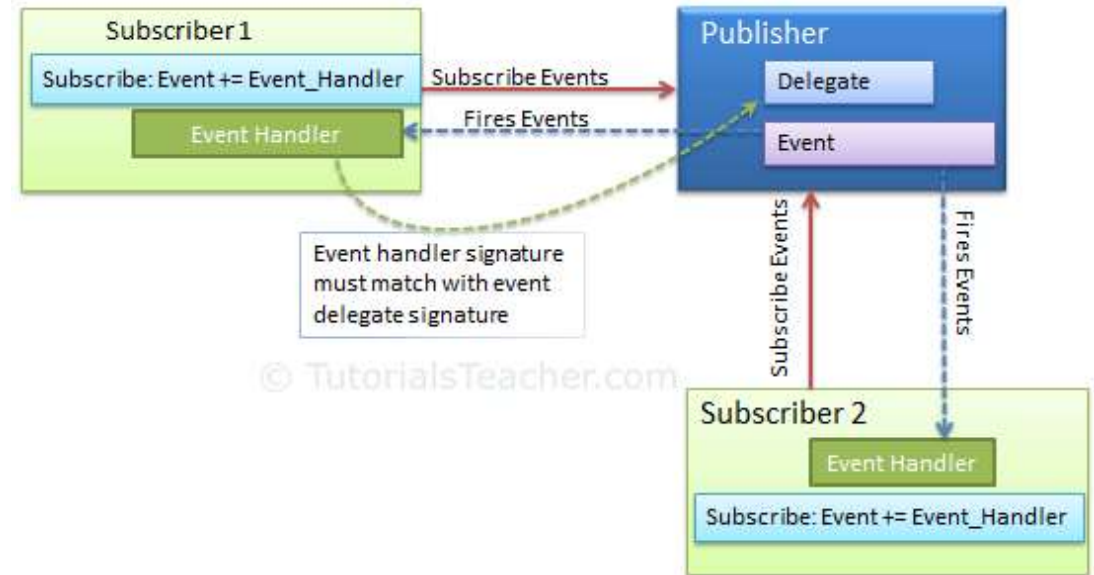
La marca publica un **evento** y **notifica** a las personas que son **subscriptores** acerca de ese evento. Los subscriptores **realizan una acción** con esa notificación (atienden/ignoran).



Unidad II, la plataforma Microsoft .NET

2.7 Delegación y eventos en .NET

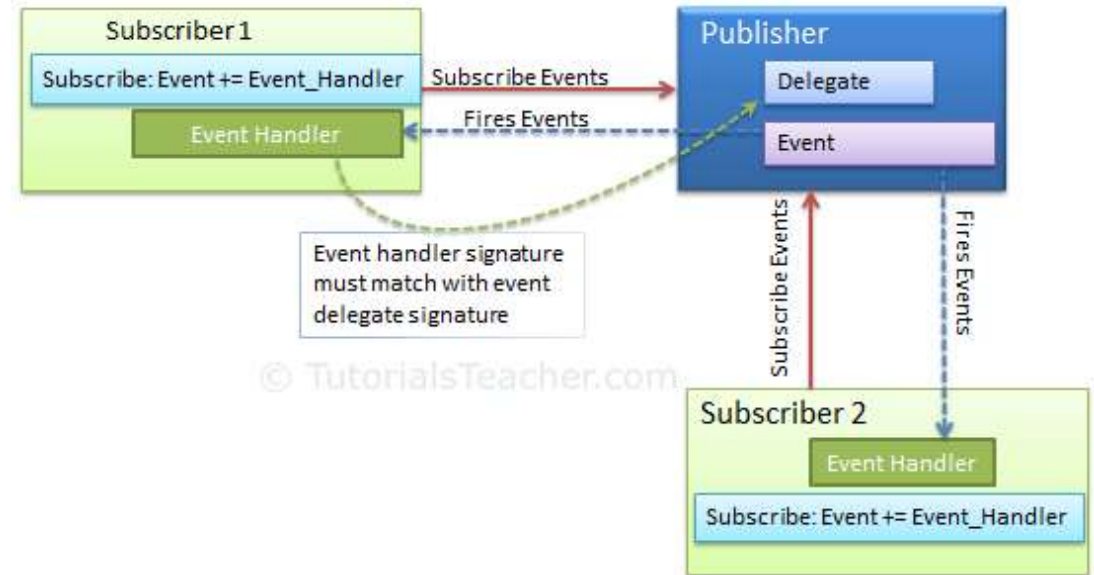
- Un evento tiene un ‘**Publisher**’, un “**Subscriber**”, una “**Notification**” y un “**Handler**”.
- Un evento no es otra cosa mas que un delegado encapsulado.



Unidad II, la plataforma Microsoft .NET

2.7 Delegación y eventos en .NET

- Un “**Publisher**” es un objeto que contiene la definición del evento y el delegado.
- Un “**subscriber**” es un objeto que acepta el evento y provee un “**handler**” para el mismo.



Unidad II, la plataforma Microsoft .NET

El modelo de delegación presupone la existencia de tres tipos de objetos:

1. Un conjunto de eventos(**events**) que pueden suceder y para los cuales es de interés registrar información de estado que detalle para cada tipo de evento los aspectos particulares del caso.
 1. *Ejemplos de eventos de interés podrían ser:* un evento de mouse o uno de teclado. En el primer caso es importante conocer la ubicación del puntero del mouse en la pantalla y/o cual botón se oprimió. En el segundo lo importante saber cual tecla se oprimió o liberó.
2. Un conjunto de fuentes(**sources**) de eventos que los disparan, en general frente a una acción del usuario. Fuentes de eventos puede ser: la ventana de diálogo donde se visualiza la imagen, o un actor en esta.
3. Un conjunto de observadores(**listeners**) que se subscriben a las fuentes de eventos de su interés y son comunicados oportunamente por estas con el evento adecuado al caso. Es responsabilidad de cada observador implementar las acciones a tomar frente al evento informado.

Unidad II, la plataforma Microsoft .NET

Ejercicio # 1 en clase:

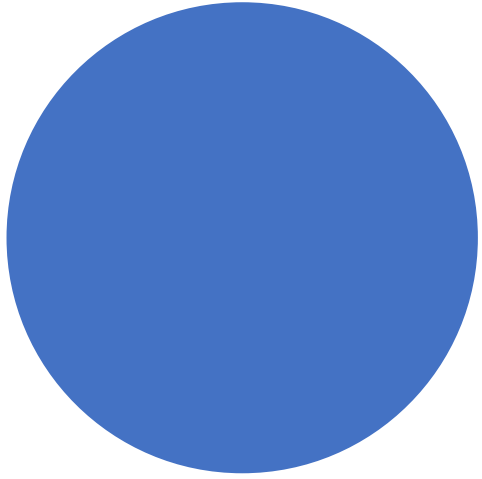
Eventos en C#:

1. Realizar ejercicios basados en:
https://www.tutorialspoint.com/csharp/csharp_events.htm
2. Seleccionar de entre:
 1. Registro para eventos
 2. Simulador visual de compuertas lógicas
 3. Calcular promedio de calificaciones de alumnos
3. Subir el programa a github o bitbucket y enviar por email (daniel_nuno@hotmail.com) la liga al mismo.

Unidad II, la plataforma Microsoft .NET

TAREA # 5 Eventos en C#:

1. Realizar ejercicios basados en:
https://www.tutorialspoint.com/csharp/csharp_events.htm
2. Calculadora para generar ofertas laborales, calcular en base al salario mensual:
 1. Ley: Aguinaldo, vacaciones, Infonavit, IMSS, RCV
 2. Empresa: SGMM, Vales Despensa, Comedor*, Seguro de Vida*
 3. *fija, resto es %.
3. Subir el programa a github o bitbucket y enviar por email (daniel_nuno@hotmail.com) la liga al mismo.



Unidad III, Programación Concurrente

Unidad III, Programación concurrente

- 3.1 Programación concurrente y paralelismos.
- 3.2 El concepto de exclusión mutua.
- 3.3 Bloqueo mediante variables compartidas.
- 3.4 Algoritmo de Peterson.
- 3.5 Algoritmo de Dekker.
- 3.6 Conceptos básicos de Sincronización.
- 3.7 Bloqueos.
 - 3.7.1 Locking
 - 3.7.2 Mutex
 - 3.7.3 Semaphores
- 3.9 Monitores.
- 3.10 Ínter-bloqueó.

Unidad III, Programación concurrente

- 3.1 Programación concurrente y paralelismos.
- Un programa ordinario consiste en declaraciones, asignaciones y control de flujo en algún lenguaje programación.
- Los lenguajes modernos, incluyen procedimientos y módulos para organizar grandes sistemas, mediante abstracciones y encapsulamiento.
- Luego de la compilación, estas instrucciones de lenguaje maquina, son ejecutadas de manera secuencial en una computadora.

Unidad III, Programación concurrente

- 3.1 Programación concurrente y paralelismos.
- Un programa concurrente es un conjunto de programas secuenciales que pueden ser ejecutados en paralelo.
- **Ejecución en paralelo** es usado en sistemas donde la ejecución de varios programas se superpone en el tiempo mediante el uso de varios procesadores.
- **Ejecución concurrente** se reserva para un potencial paralelismo, en el cual la ejecución podría, pero no necesariamente, superponerse en el tiempo.

Unidad III, Programación concurrente

- 3.1 Programación concurrente y paralelismos.
- **Concurrencia es hacer mas de una cosa a la vez.**
- Una aplicación utiliza concurrencia para responderle al usuario mientras se escribe en la base de datos.
- Muchas aplicaciones usan concurrencia para responder a un segundo requerimiento mientras se finaliza el primer requerimiento.

Unidad III, Programación concurrente

- 3.1 Programación concurrente y paralelismos.
- **Multithreading** es una de las formas de concurrencia que utiliza múltiples hilos de ejecución, irónicamente los *threads* fueron implementados para aislar a los programas unos de otros.
- **Multithreading** literalmente se refiere a usar múltiples hilos de ejecución (*threads*).
- **Multithreading** es una forma de concurrencia, pero no es la única.

Unidad III, Programación concurrente

- 3.1 Programación concurrente y paralelismos.

Procesamiento en paralelo se refiere a hacer varias tareas mediante su división en varios hilos (*threads*) de ejecución, que se ejecutan de manera concurrente.

- Procesamiento en paralelo o programación en paralelo utiliza *multithreading* para maximizar el uso de múltiples procesadores. Los procesadores modernos tienen múltiples CPUs.
- Procesamiento en paralelo es un tipo de *multithreading* y este a su vez, es un tipo de concurrencia.

Unidad III, Programación concurrente

- 3.1 Programación concurrente y paralelismos.

Programación asíncrona es una forma de concurrencia que utiliza “futures” o “callbacks” para evitar hilos innecesarios.

- Un “future” (o promesa) es un tipo que representa alguna operación que será completada en el futuro.
- La programación asíncrona se centra en la idea que una operación que se inicia será completada en algún punto en el futuro y mientras este en progreso, no bloquea el hilo original. Cuando se complete la operación, se notifica que la operación termino.

Unidad III, Programación concurrente

- 3.1 Programación concurrente y paralelismos.

Programación asíncrona los tipos modernos de promesas en .NET son Task y Task<Tresult>.

- Las APIs asíncronas de antes utilizan *callbacks* o eventos en lugar de promesas.

Unidad III, Programación concurrente

- 3.1 Programación concurrente y paralelismos.

Programación reactiva es otra forma de concurrencia, donde la aplicación reacciona a eventos.

- Programación reactiva esta relacionada con la programación asíncrona, pero se construye sobre eventos asíncronos en lugar de operaciones asíncronas.
- Un evento asíncrono podría no tener un inicio, podría pasar en cualquier momento y podría presentarse muchas veces.

Unidad III, Programación concurrente

- 3.1 Programación concurrente y paralelismos.

Programación reactiva

- Si se considerara una aplicación como una gran máquina de estados, entonces el comportamiento de la aplicación se podría describir como que reacciona a una series de eventos y que su estado se actualiza con cada evento.
- La programación reactiva no es necesariamente concurrente, pero están muy relacionada.

Unidad III, Programación concurrente

- 3.1 Programación concurrente y paralelismos.

Problemas de la programación concurrente

En la programación concurrente, se puede tener múltiple acceso a recursos compartidos sin conflicto, siempre y cuando sean accesos de no modificación.

Los problemas se originan cuando se intenta acceder un recurso compartido para modificar su estado.

Ejemplo de algunos problemas:

- Tratar de actualizar un archivo de manera concurrente
- Tratar de actualizar un segmento de memoria compartida

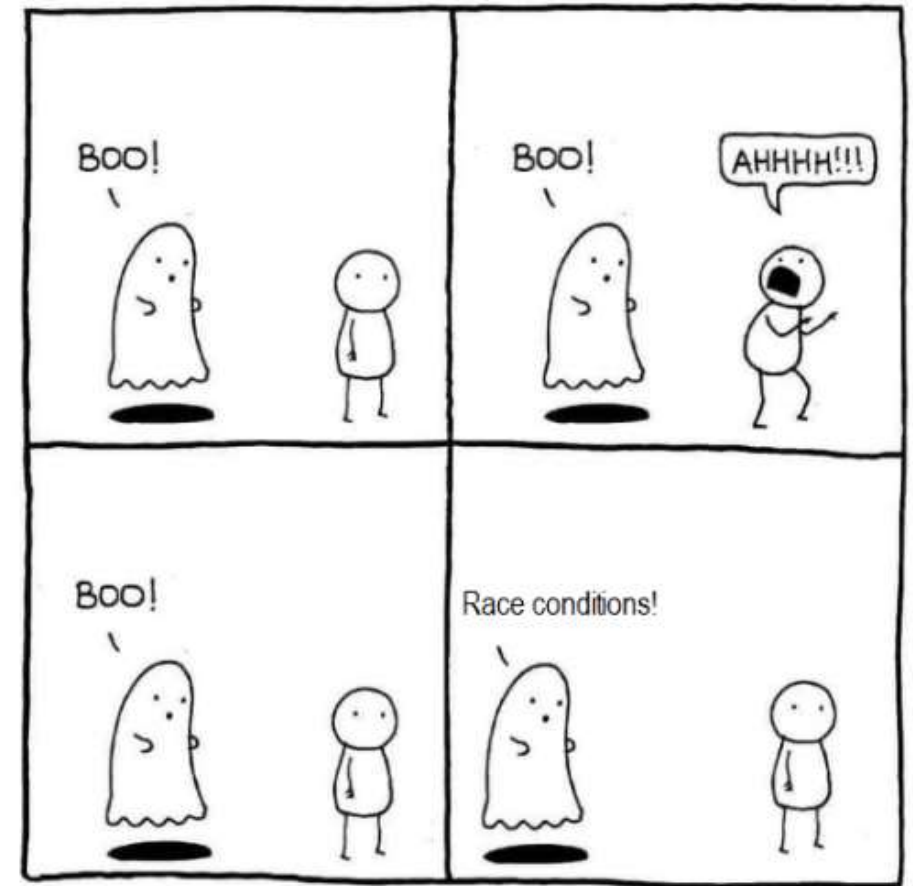
Unidad III, Programación concurrente

- 3.1 Programación concurrente y paralelismos.

Problemas de la programación concurrente

Al intentar acceder de manera concurrente para actualizar un recurso compartido, se pueden tener los siguientes problemas:

Condición de Carrera (*race condition*), se refiere a cuando un dispositivo o un sistema intentan realizar una o mas operaciones a la vez, pero debido a la naturaleza del dispositivo o del sistema, se deben realizar en la secuencia correcta, en otro caso se presenta una condición de carrera.



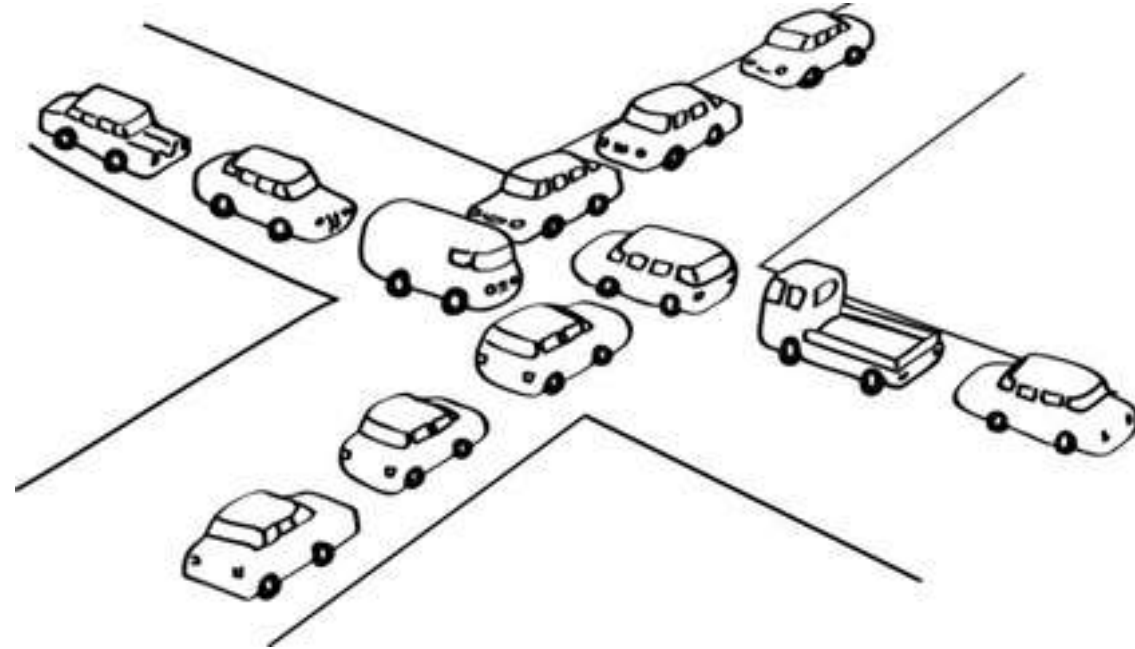
Multithreading can be difficult

Unidad III, Programación concurrente

- 3.1 Programación concurrente y paralelismos.

Problemas de la programación concurrente

Deadlocks, se refiere a cuando dos objetos agarran a un mismo elemento compartido y ninguno de los dos cede para soltarlo, entonces los objetos se quedan esperando al otro.



Unidad III, Programación Concurrente

TAREA # 6 Spinning, Deadlock:

1. Basado en el ejercicio de *Deadlock*, modificarlo para en lugar de utilizar un 'sleep', se haga un ciclo y se cuente cuanto tiempo se debería usar en el 'sleep', para que el programa funcione.
2. <https://github.com/danunora/unedl/tree/master/Command/DeadLock>
3. Subir el programa a github o bitbucket y enviar por email (daniel_nuno@hotmail.com) la liga al mismo.

Unidad III, Programación concurrente

- 3.2 El concepto de exclusión mutua.

Exclusión Mutua se refiere al control de cuantos hilos de ejecución pueden trabajar sobre una región del código.

El objetivo de la exclusión mutua es garantizar que únicamente un hilo de ejecución accede a los datos a la vez mediante el uso de un bloqueo (**lock**).

Unidad III, Programación concurrente

- 3.2 El concepto de exclusión mutua.

Atomicidad se dice que una operación es atómica si no puede ser interrumpida. Las operaciones que no son atómicas, son vulnerables a las condiciones de carrera (*race condition*).

Ejemplos:

- 1) $x = 1$, es una operación atómica, la asignación se hace en una sola operación.
- 2) $x++$, no es una operación atómica, el incremento esta compuesto de dos operaciones:
 - 1) $tmp = x+1;$
 - 2) $x = tmp;$

Unidad III, Programación concurrente

- 3.2 El concepto de exclusión mutua.

Sección crítica, es aquella parte de los procesos concurrentes que no pueden ejecutarse de manera concurrente o que su composición es atómica y no pueden dividirse.

Si un proceso entra a ejecutar una sección crítica en la que se accede a variables compartidas, entonces otro proceso no puede entrar a ejecutar una sección crítica en la que esas variables se modifiquen en el interior.

Las secciones críticas se agrupan mutuamente exclusivas de las regiones críticas de cada clase.

Ejemplo:

[https://github.com/danunora/unedl/tree/master/Command/Atomic CriticalSection](https://github.com/danunora/unedl/tree/master/Command/Atomic%20CriticalSection)

Unidad III, Programación concurrente

- 3.2 El concepto de exclusión mutua.

La exclusión se consigue con protocolos que bloqueen el acceso a la sección crítica mientras es utilizada por un proceso.



Unidad III, Programación concurrente

- 3.2 El concepto de exclusión mutua.

Se realiza el bloqueo de una sección crítica mediante el uso de una variable compartida de tipo booleano que sirve de indicador.

La acción del bloqueo se realiza con la actividad del indicador y la del desbloqueo con su desactivación.

Este método, no resuelve el problema de la exclusión mutua ya que la comprobación y la inicialización del indicador son operaciones separadas y en ese caso se puede entrelazar el uso del recurso por ambos procesos.

Unidad III, Programación concurrente

- 3.2 El concepto de exclusión mutua.

```
class ExclusionMutua
```

```
bool bandera;
```

```
process P1 {  
    while (bandera) {  
        // Esperando a que se libere  
    }  
    bandera = true; // bloqueo  
    // sección crítica  
    bandera = false;  
    // resto del proceso  
}
```

```
process P2 {  
    while (bandera) {  
        // Esperando a que se libere  
    }  
    bandera = true; // bloqueo  
    // sección crítica  
    bandera = false;  
    // resto del proceso  
}
```

```
main {  
    bandera = false;  
    P1();  
    P2();  
}
```

Ejemplo:

<https://github.com/danunora/unedl/tree/master/Command/ThreadTest>

<https://github.com/danunora/unedl/tree/master/Command/ThreadSafe>

Unidad III, Programación concurrente

- 3.3 Bloqueo mediante variables compartidas

Se realiza el bloqueo de una sección crítica mediante el uso de dos **banderas**.

Se asocia una bandera a cada uno de los procesos. Antes de acceder al recurso un proceso debe activar su bandera y comprobar que el otro no tiene su bandera activada.

Interbloqueo, se presenta cuando dos procesos llaman al “bloqueo” simultáneamente. Las dos banderas quedan activas y los dos procesos a la espera que se libere el recurso.

Unidad III, Programación concurrente

- 3.3 Bloqueo mediante variables compartidas

```
class ExclusionMutua {  
    bool bandera1, bandera2;  
  
    // Realiza bloqueo  
    void bloqueo(bool flag1, bool flag2) {  
        flag1 = true;  
        while (flag2) {  
            // Espera  
        }  
    }  
  
    // Realiza desbloqueo  
    void desbloqueo(bool flag3) {  
        flag3 = true;  
    }  
}
```

```
void P1  
{  
    bloqueo(bandera1, bandera2);  
    // sección crítica  
    // uso de recurso  
    desbloqueo(bandera1);  
}  
  
void P2  
{  
    bloqueo(bandera2, bandera1);  
    // sección crítica  
    // uso de recurso  
    desbloqueo(bandera2);  
}
```

```
main {  
    bandera1 = false;  
    bandera2 = false;  
    P1();  
    P2();  
}
```

Unidad III, Programación concurrente

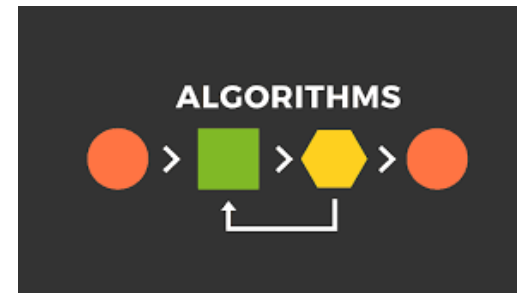
TAREA # 7 Algoritmo de Peterson/Dekker:

1. Realizar una exposición acerca de:
 1. Algoritmo de Peterson (Sergio, Brian, Kevin, Luis)
 2. Algoritmo de Dekker (Miguel, Ivan, Octavio, Santos)
2. Subir el programa a github o bitbucket y enviar por email (daniel_nuno@hotmail.com) la liga al mismo.

Unidad III, Programación concurrente

3.4 Algoritmo de Peterson

- Kevin Ricardo Ceja Ramos
- Sergio Herrera Rivera
- Luis Manuel de Alba Villaseñor
- Brian Eduardo Preciado Limón



Descripción

- El **algoritmo de Peterson**, también conocido como **solución de Peterson**, es un algoritmo de programación concurrente para exclusión mutua, que permite a dos o más procesos o hilos de ejecución compartir un recurso sin conflictos, utilizando sólo memoria compartida para la comunicación.
- Peterson desarrolló el primer algoritmo (1981) para dos procesos que fue una simplificación del algoritmo de Dekker para dos procesos. Posteriormente este algoritmo fue generalizado para N procesos.

Sección crítica

En programación concurrente, se define como a la porción de código de un programa de computador el cual accede a un recurso compartido (estructura de datos ó dispositivo) que no debe de ser accedido por mas de un hilo en ejecución (thread). La sección crítica por lo general termina en un tiempo determinado y el hilo, proceso o tarea solo tendrá que esperar un período determinado de tiempo para entrar. Se necesita de un mecanismo de sincronización en la entrada y salida de la sección crítica para asegurar la utilización exclusiva del recurso, por ejemplo un semáforo.

Exclusión mutua

- Los algoritmos de exclusión mutua (comúnmente abreviada como mutex por mutual exclusión) se usan en programación concurrente para evitar que fragmentos de código conocidos como secciones críticas accedan al mismo tiempo a recursos que no deben ser compartidos. La mayor parte de estos recursos son las señales, contadores, colas y otros datos que se emplean en la comunicación entre el código que se ejecuta cuando se da servicio a una interrupción y el código que se ejecuta el resto del tiempo. Se trata de un problema de vital importancia porque, si no se toman las precauciones debidas, una interrupción puede ocurrir entre dos instrucciones cualesquiera del código normal y esto puede provocar graves fallos.

Algoritmo para 2 procesos

```
Bandera [0] = 0
```

```
Bandera [1] = 0
```

```
Turno          = 4
```

```
P0: bandera[0] = 1
```

```
    turno = 1
```

```
while(bandera[1] && turno == 1);
```

```
//no hace nada, espera
```

```
//sección critica
```

```
//fin de la sección critica
```

```
Bandera[0] = 0
```

```
P0: bandera[1] = 1
```

```
    turno = 1
```

```
while(bandera[1] && turno == 1);
```

```
//no hace nada, espera
```

```
//sección critica
```

```
//fin de la sección critica
```

```
Bandera[1] = 0
```

Los procesos p0 y p1 no pueden estar en la sección crítica al mismo tiempo: si p0 está en la sección crítica, entonces bandera[0] = 1, y ocurre que bandera[1] = 0, con lo que p1 ha terminado la sección crítica, o que la variable compartida turno = 0, con lo que p1 está esperando para entrar a la sección crítica. En ambos casos, p1 no puede estar en la sección crítica...

Algoritmo para n procesos

// variables compartidas

Bandera: array[0..N-1] of -1...n-2; //inicializada a -1

Turno: array[0..N-2] of 0..n-1; // inicializada a 0

// Protocolo para P_i ($i = 0, \dots, N-1$)

j: 0.. N-2; // variable local indicando la etapa

For(j = 0 to N - 2){

 bandera[i] = j;

 turno[j] = i;

 while $[(\exists k \neq i : \text{bandera}[k] \geq j) \wedge (\text{turno}[k] == i)]$

}

<sección crítica>

bandera[i] = - 1;

Referencias

- <http://diccionario.sensagent.com/Algoritmo%20de%20Peterson/es-es/>
- <https://sistemasoper2.wordpress.com/2014/10/21/seccion-critica/>
- <https://www.aiu.edu/publications/student/spanish/180-207/PDF/sistemas-operativos-procesos-concurrentes.pdf>

ALGORITMO DE DEKKER

LUIS ENRIQUE SANTOS LANDEROS

MIGUEL ALEJANDRO GUEVARA MENDOZA

IVAN MICHEL MADRIZ DELGADO

OCTAVIO CAMACHO PONCE

¿Qué es el algoritmo de Dekker?

El algoritmo de Dekker es un algoritmo de programación concurrente para exclusión mutua, que permite a dos procesos o hilos de ejecución compartir un recurso sin conflictos. Fue uno de los primeros algoritmos de exclusión mutua inventados, implementado por Edsger Dijkstra.

Si ambos procesos intentan acceder a la sección crítica simultáneamente, el algoritmo elige un proceso según una variable turno. Si el otro proceso está ejecutando en su sección crítica, deberá esperar su finalización.

Existen cinco versiones del algoritmo Dekker, teniendo ciertos fallos los primeros cuatro. La versión 5 es la que trabaja más eficientemente, siendo una combinación de la 1 y la 4.



Versión 1 - Alternancia Estricta

La primer versión del algoritmo de Dekker es llamado Alternancia Estricta, es llamado de esta manera ya que obliga a que cada proceso tenga un turno, o sea que hay un cambio de turno cada vez que un proceso sale de la sección crítica, por lo tanto si un proceso es lento atrasara a otros procesos que son rápidos.

Características:

- Garantiza la exclusión mutua
- Su sincronización es forzada
- Acopla fuertemente a los procesos (procesos lentos atrasan a procesos rápidos)
- No garantiza la progresión, ya que si un proceso por alguna razón es bloqueado dentro o fuera de la sección puede bloquear a los otros procesos.

Algoritmo

```
1  int turno_proceso; //1 proceso 1, 2 proceso 2
2  Proceso1()
3  {
4      while( true )
5      {
6          [REALIZA_TAREAS_INICIALES]
7          while( turno_proceso == 2 ){
8              [SECCION_CRITICA]
9              turno_proceso = 2;
10             [REALIZA_TAREAS_FINALES]
11         }
12     }
13
14     Proceso2()
15     {
16         while( true )
17         {
18             [REALIZA_TAREAS_INICIALES]
19             while( turno_proceso == 1 ){
20                 [SECCION_CRITICA]
21                 turno_proceso = 1;
22                 [REALIZA_TAREAS_FINALES]
23             }
24         }
25
26     iniciar(){
27         turno_proceso = 1;
28         Proceso1();
29         Proceso2();
30     }
```

Descripción del algoritmo

- Cuando un proceso es ejecutado verifica si es su turno, si no es su turno se queda en espera por medio de un ciclo while.(línea 7 y 19)
- De lo contrario si es su turno avanza a la sección crítica.
- Cuando el proceso sale de la sección crítica cambia de turno.(línea 9 y 21)

Versión 2 - Problema de Interbloqueo

Segunda versión del algoritmo de Dekker es llamado Problema de Interbloqueo, su nombre se debe a que si en cada ráfaga de CPU, cada proceso queda en el mismo estado, en el estado donde se le asigna que puede entrar a la sección crítica (línea 8 para el proceso 1 y línea 21 para el proceso 2). Entonces estando los dos procesos con opción a entrar, a la siguiente ráfaga de CPU ambos procesos verificarán si el proceso alterno puede entrar (línea 9 y 22), viendo que el proceso alterno tiene la opción de entrar, los procesos quedan bloqueados ya que se quedarán encicladados bloqueándose mutuamente ya que no podrán entrar nunca a la sección crítica.

Características:

- Garantiza la exclusión mutua
- No garantiza espera limitada

Algoritmo

```
1  boolean p1_puede_entrar, p2_puede_entrar;
2
3  Proceso1()
4  {
5      while( true )
6      {
7          [REALIZA_TAREAS_INICIALES]
8          p1_puede_entrar = true;
9          while( p2_puede_entrar ){
10             [SECCION_CRITICA]
11             p1_puede_entrar = false;
12             [REALIZA_TAREAS_FINALES]
13         }
14     }
15
16  Proceso2()
17  {
18      while( true )
19      {
20          [REALIZA_TAREAS_INICIALES]
21          p2_puede_entrar = true;
22          while( p1_puede_entrar ){
23             [SECCION_CRITICA]
24             p2_puede_entrar = false;
25             [REALIZA_TAREAS_FINALES]
26         }
27     }
28
29  iniciar()
30  {
31      p1_puede_entrar = false;
32      p2_puede_entrar = false;
33      Proceso1();
34      Proceso2();
35  }
```

Descripción del algoritmo

- El proceso que es ejecutado después de realizar sus tareas iniciales, a este proceso se le permite entrar (línea 8 y 21).
- Cuando ya puede entrar verifica si otro proceso tiene la opción de poder entrar, si otro proceso también tiene la opción de poder entrar se da un interbloqueo. De lo contrario el proceso avanza a la sección crítica (línea 9 y 22).
- Al salir de la sección crítica el proceso cambia su opción (línea 11 y 24). Y permite al otro proceso avanzar a la sección crítica.

Versión 3 - Colisión región crítica no garantiza la exclusión mutua

La Tercera versión del algoritmo de Dekker es llamado Colisión región crítica no garantiza la exclusión mutua, como su nombre lo indica se da una colisión en la región crítica por la forma en que son colocados por así decirlo los permisos, ya que primero se comprueba si otro proceso esta dentro y luego se indica que el proceso en el que se esta actualmente cambia diciendo que esta dentro. Y el problema se da cuando los procesos después de haber tenido sus ráfagas de CPU pasan de la fase de comprobación (línea 8 y 21) y se tiene libre el camino para entrar a la región crítica, generando esto una colisión.

Características

- No garantiza la exclusión mutua
- Colisión en la región crítica

Algoritmo

```
1  boolean p1_esta_dentro, p2_esta_dentro;
2
3  Proceso1()
4  {
5      while( true )
6      {
7          [REALIZA_TAREAS_INICIALES]
8          while( p2_esta_dentro ){
9              p1_esta_dentro = true;
10             [SECCIÓN_CRITICA]
11             p1_esta_dentro = false;
12             [REALIZA_TAREAS_FINALES]
13         }
14     }
15
16  Proceso2()
17  {
18      while( true )
19      {
20          [REALIZA_TAREAS_INICIALES]
21          while( p1_esta_dentro ){
22              p2_esta_dentro = true;
23              [SECCIÓN_CRITICA]
24              p2_esta_dentro = false;
25              [REALIZA_TAREAS_FINALES]
26          }
27      }
28
29  iniciar()
30  {
31      p1_esta_dentro = false;
32      p2_esta_dentro = false;
33      Proceso1();
34      Proceso2();
35  }
```

Descripción del Algoritmo

- Al ejecutarse el proceso y después de realizar sus tareas iniciales, verifica si otro proceso esta dentro de la sección critica (linea 8 y 21).
- Si el otro proceso esta dentro entonces espera a que salga de la sección critica. De lo contrario pasa la fase de comprobación y cambia su estado a que esta dentro (linea 9 y 22).
- Luego de pasar la sección critica cambia su estado (linea 11 y 24), termina sus tareas finales.

Versión 4 - Postergación Indefinida

Cuarta versión del algoritmo de Dekker es llamado Postergación Indefinida, su nombre se debe a que en una parte del código (línea 12 y 30) es colocado un retardo con un tiempo aleatorio, y el retardo puede ser muy grande que no se sabe hasta cuando entrara a la sección crítica.

Características

- Garantiza la exclusión mutua.
- Un proceso o varios se quedan esperando a que suceda un evento que tal vez nunca suceda.

Algoritmo

```
1  boolean p1_puede_entrar, p2_puede_entrar;
2
3  Proceso1()
4  {
5      while( true )
6      {
7          [REALIZA_TAREAS_INICIALES]
8          p1_puede_entrar = true;
9          while( p2_puede_entrar )
10         {
11             p1_puede_entrar = false;
12             retardo( tiempo_x ); //tiempo_x es un tiempo aleatorio
13             p1_puede_entrar = true;
14         }
15         [SECCION_CRITICA]
16         p1_puede_entrar = false;
17         [REALIZA_TAREAS_FINALES]
18     }
19 }
20
21 Proceso2()
22 {
23     while( true )
24     {
25         [REALIZA_TAREAS_INICIALES]
26         p2_puede_entrar = true;
27         while( p1_puede_entrar )
28         {
29             p2_puede_entrar = false;
30             retardo( tiempo_x ); //tiempo_x es un tiempo aleatorio
31             p2_puede_entrar = true;
32         }
33         [SECCION_CRITICA]
34         p2_puede_entrar = false;
35         [REALIZA_TAREAS_FINALES]
36     }
37 }
38
39 iniciar()
40 {
41     p1_puede_entrar = false;
42     p2_puede_entrar = false;
43     Proceso1();
44     Proceso2();
45 }
```

Descripción del algoritmo

- Luego de realizar sus tareas iniciales el procesos solicita poder entrar en la sección crítica, si el otro proceso no puede entrar (línea 9 y 27) ya que su estado es falso entonces el proceso entra si problema a la sección crítica.
- De lo contrario si el otro proceso también puede entrar entonces se entra al ciclo donde el proceso actual se niega el paso así mismo y con un retardo de x tiempo siendo este aleatorio (línea 12 y 30) se pausa el proceso, para darle vía libre a los otros procesos.
- Luego de terminar su pausa entonces el proceso actual nuevamente puede entrar y nuevamente si el otro proceso puede entrar se repite el ciclo y si no hay otro proceso, entonces el proceso puede entrar en la sección crítica.
- Cambia su estado (línea 16 y 34) y luego realiza sus tareas finales.

Versión 5 - Algoritmo Optimo

Quinta versión del algoritmo de Dekker Algoritmo Optimo, este algoritmo es una combinación del algoritmo 1 y 4.

Características

- Garantiza la exclusión mutua.
- Progreso
- Espera limitada

Algoritmo

```
1  boolean p1_puede_entrar, p2_puede_entrar;
2  int turno;
3
4  Proceso1()
5  {
6      while( true )
7      {
8          [REALIZA_TAREAS_INICIALES]
9          p1_puede_entrar = true;
10         while( p2_puede_entrar )
11         {
12             if( turno == 2 )
13             {
14                 p1_puede_entrar = false;
15                 while( turno == 2 ){ }
16                 p1_puede_entrar = true;
17             }
18         }
19         [REGION_CRITICA]
20         turno = 2;
21         p1_puede_entrar = false;
22         [REALIZA_TAREAS_FINALES]
23     }
24 }
25
26 Proceso2()
27 {
28     while( true )
29     {
30         [REALIZA_TAREAS_INICIALES]
31         p2_puede_entrar = true;
32         while( p1_puede_entrar )
33         {
34             if( turno == 1 )
35             {
36                 p2_puede_entrar = false;
37                 while( turno == 1 ){ }
38                 p2_puede_entrar = true;
39             }
40         }
41         [REGION_CRITICA]
42         turno = 1;
43         p2_puede_entrar = false;
44         [REALIZA_TAREAS_FINALES]
45     }
46 }
47
48 iniciar()
49 {
50     p1_puede_entrar = false;
51     p2_puede_entrar = false;
52     turno = 1;
53     Proceso1();
54     Proceso2();
55 }
```

Descripción del algoritmo

- Se realiza las tareas iniciales, luego se verifica si hay otro procesos que puede entrar, si lo hay se entra al ciclo y si es el turno de algún otro proceso (línea 12 y 34) cambia su estado a ya no poder entrar a la sección crítica y nuevamente verifica si es el turno de algún otro proceso (línea 15 y 37) si lo es se queda enciclado hasta que se da un cambio de turno, luego nuevamente retoma su estado de poder entrar a la sección crítica, regresa al ciclo y verifica si hay otro proceso que puede entrar entonces nuevamente se encicla, de lo contrario entra a la sección crítica.
- Al salir de la sección crítica el proceso cambia su turno, cambia su estado y realiza sus tareas finales.(línea 20 y 42)

Referencias

<https://bourneshell.wordpress.com/2012/10/26/algoritmos-de-dekker/>

<http://aprendiendo-software.blogspot.com/2011/12/algoritmo-de-dekker-version-1.html>

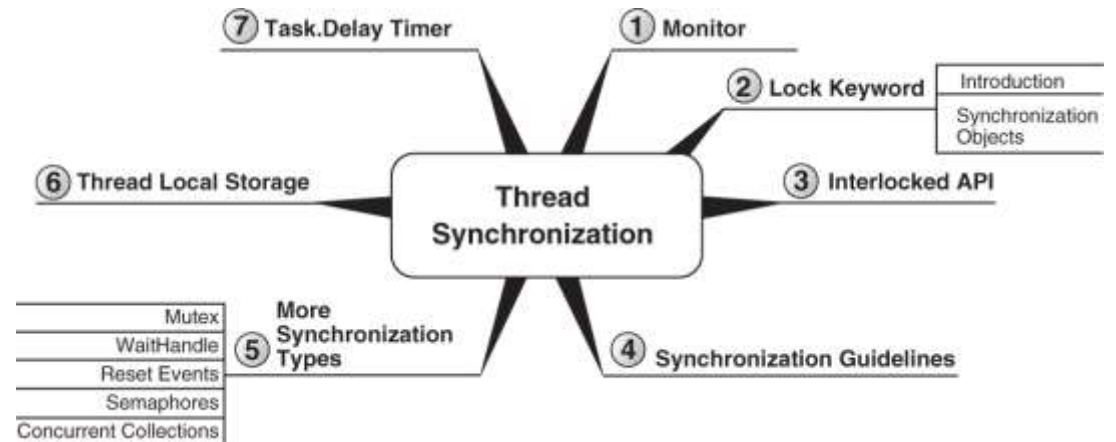
<http://www.esacademic.com/dic.nsf/eswiki/65551>

Unidad III, Programación concurrente

- 3.6 Conceptos básicos de Sincronización.

La **sincronización** nos permite coordinar las acciones de los hilos para obtener un resultado predecible.

La sincronización es particularmente importante cuando se requiere tener acceso a la misma información.



Unidad III, Programación concurrente

- 3.6 Conceptos básicos de Sincronización.

En C#, existen varios constructores que permiten la sincronización en los siguientes tipos:

1. Métodos de bloqueo simple
2. Métodos de sincronización con bloqueo
3. Métodos de señalización
4. Métodos de sincronización sin bloqueo

Unidad III, Programación concurrente

- 3.6 Conceptos básicos de Sincronización.

1. Métodos de bloqueo simple

1. Se utilizan para esperar a que otro hilo termine o a que un tiempo determinado se consuma. (**Sleep**, **Wait**, **Task.Wait**)

2. Métodos de sincronización con bloqueo

1. Limitan la cantidad de hilos que pueden realizar alguna actividad o ejecutar una sección de código a la vez.
2. Los constructores de acceso exclusivo son los mas comunes, únicamente permiten un hilo a la vez y permiten el acceso a información en común sin interferir entre uno y otro. **Lock**, **Mutex** y **SpinLock** son los mas utilizados.
3. Los constructores de acceso no exclusivo son **Semaphore**, **SemaphoreSlim** y los **read/write locks**.

Unidad III, Programación concurrente

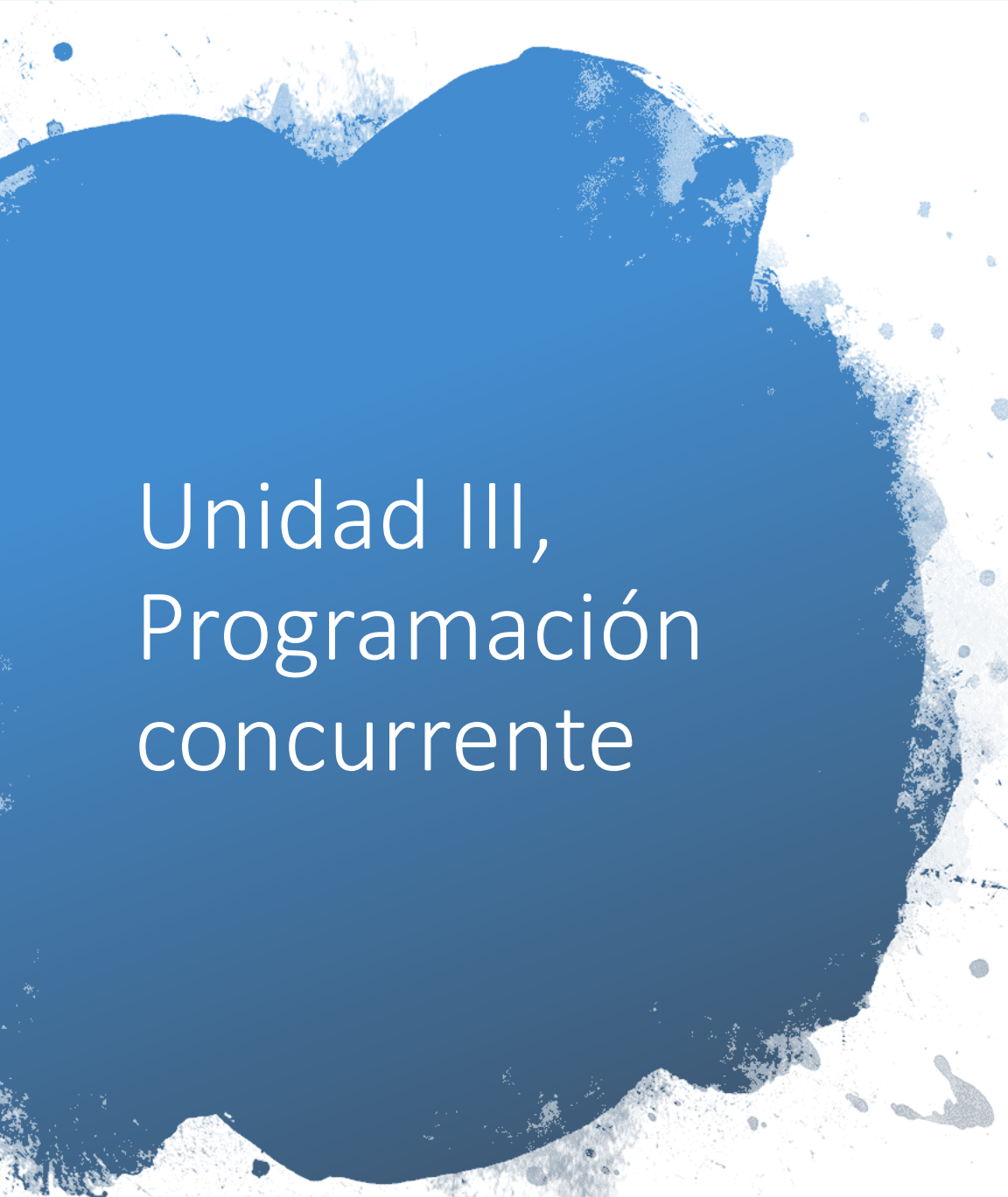
- 3.6 Conceptos básicos de Sincronización.

3. Métodos de señalización

3. Estos permiten pausar a un hilo hasta recibir una notificación de otro hilo, impidiendo la necesidad de estar verificando (**polling**)
4. Existen **event wait handles**, métodos **Monitor Wait/Pulse** y en .NET 4.0 se introdujeron las clases: **CountdownEvent** y **Barrier**

4. Métodos de sincronización sin bloqueo

3. Estos protegen el acceso mediante el uso de primitivas del procesador. Existen en CLR y en C# los siguientes métodos: **Thread.MemoryBarrier**, **Thread.VolatileRead**, **Thread.VolatileWrite**, the **volatile keyword**, and the **Interlocked class**.



Unidad III, Programación concurrente

- 3.7 Métodos de bloqueo simple
 - 3.7.1 Sleep
 - 3.7.2 Wait
 - 3.7.3 Task.Wait

Unidad III, Programación concurrente

- 3.6 Conceptos básicos de Sincronización.

Bloqueo simple (Blocking), se dice que un hilo esta bloqueado cuando se pausa su ejecución cuando esta durmiendo (**Sleeping**) o cuando esta esperando por algún otro hilo que finalice (**Join, EndIvoke**).

Un hilo bloqueado no consume CPU hasta que es desbloqueado.

El desbloqueo sucede cuando:

- 1) Se satisface la condición del bloqueo
- 2) Se acaba el tiempo (time out)
- 3) Recibe una interrupción (Thread.Interrupt)
- 4) Se aborta (Thread.Abort)

Unidad III, Programación concurrente

- 3.6 Conceptos básicos de Sincronización.

Spinning, un hilo puede esperar por una condición de desbloqueo al girar (**spin**) entorno a un ciclo de revisión (**polling loop**).

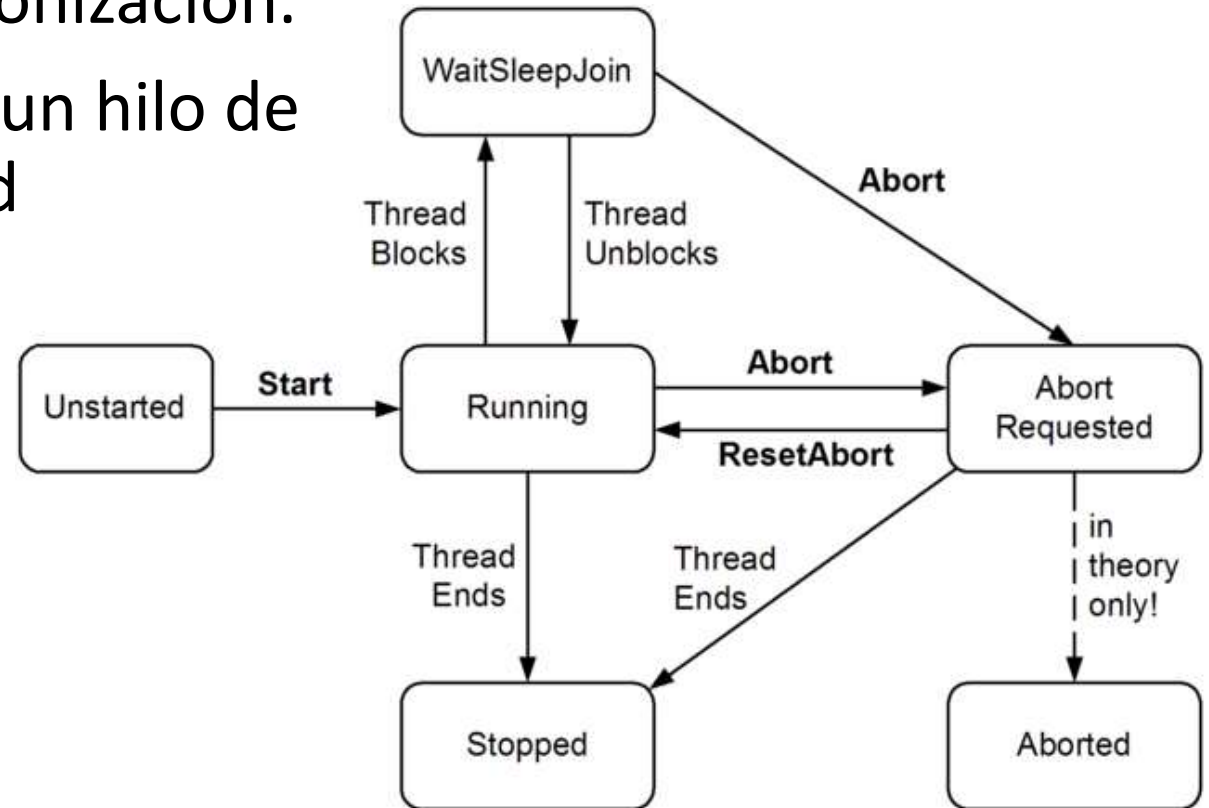
El hacer spinning es un gasto de procesamiento por parte del CPU por lo que se debe utilizar cuando se espere que la condición de desbloqueo se reciba pronto, sin embargo impide la latencia (**latency**) y sobretrabajo (**overhead**) de un cambio de contexto (**context switch**).

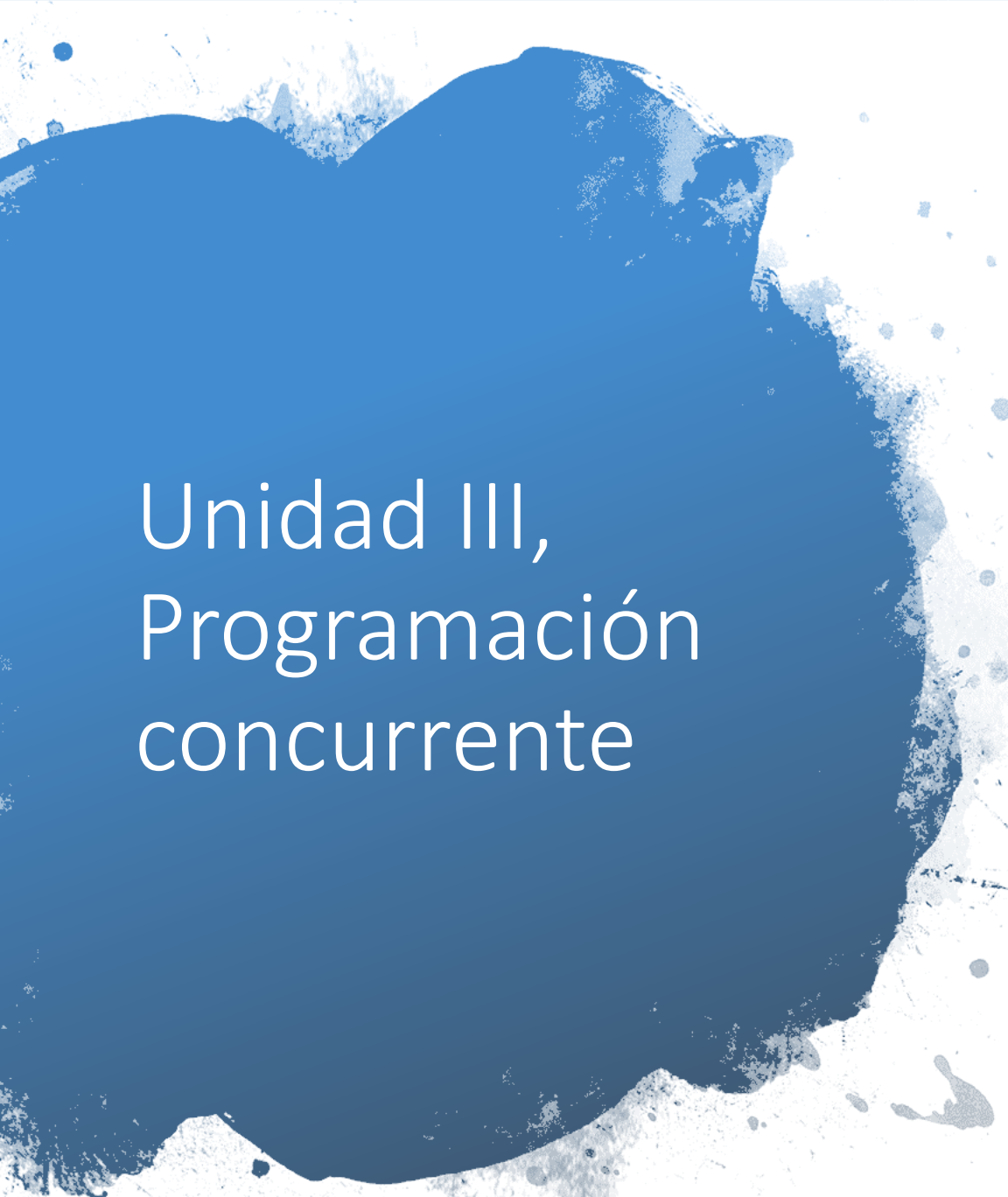
Ejemplo:

<https://github.com/danunora/unedl/tree/master/Command/SpinLock>

Unidad III, Programación concurrente

- 3.6 Conceptos básicos de Sincronización.
Se puede consultar el estado de un hilo de ejecución utilizando la propiedad **“ThreadState”**.





Unidad III, Programación concurrente

- 3.7 Sincronización con Bloqueos.
 - 3.7.1 Locking
 - 3.7.2 Mutex
 - 3.7.3 Semaphores

Unidad III, Programación concurrente

- 3.7.1 Bloqueos (Locking)

Los dos métodos principales para bloqueo exclusivo son **lock** y **Mutex**.

Lock es mas rápido y conveniente.

Mutex tiene un nicho en el que el bloqueo puede extenderse a varias aplicaciones y diferentes procesos en la computadora.

Unidad III, Programación concurrente

- 3.7.1 Bloqueos

Lock

Bloqueo exclusivo (**locking**) es usado para que un solo hilo pueda tener acceso a una sección particular del código a la vez.

Ejemplo:

<https://docs.microsoft.com/en-us/dotnet/api/system.threading.readerwriterlock>

Unidad II, la plataforma Microsoft .NET

Ejercicio # 2 en clase:

Reloj Checador Multithread:

1. En base al ejercicio del Reloj Checador (FileStream), realizar una simulación de que varias personas checan su entrada/salida.
 1. Cada persona será un *thread*,
 2. La persona puede checar entrada o salida
 3. <http://www.johandorper.com/log/thread-safe-file-writing-csharp>
2. Subir el programa a github o bitbucket y enviar por email (daniel_nuno@hotmail.com) la liga al mismo.

Unidad III, Programación concurrente

- 3.7.2 Mutex

Mutex es similar a un lock en C#, pero puede extenderse a múltiples procesos. Es decir, **Mutex** puede ser a nivel computadora o a nivel aplicación.

Adquirir y liberar un **Mutex** puede tardar varios microsegundos, alrededor de 50 veces mas lento que un **lock**.

WaitOne, método para bloquear.

ReleaseMutex, método para desbloquear.

Referencia:

<https://docs.microsoft.com/en-us/dotnet/api/system.threading.mutex?view=netframework-4.7.2>

Ejemplo:

<https://github.com/danunora/unedi/tree/master/Command/Mutex>

Unidad III, Programación concurrente

- 3.7.3 Semáforos
- Como ya se ha mencionado, el problema con los hilos, es relacionado a los recursos compartidos, como por ejemplo las variables.
- Un hilo puede estar modificándola, mientras otro hilo también puede estar haciendo lo mismo.
- Para resolver este conflicto, se pueden utilizar semáforos.
- Su funcionamiento se base en la cooperación entre procesos.
- Se utilizan señales obligando a un proceso a detenerse hasta que reciba una señal.



Unidad III, Programación concurrente

- 3.7.3 Semáforos
- En C#, existe la clase '[System.Threading.Semaphore](#)' que nos permite crear semáforos que a su vez, nos sirven para limitar el número de hilos que tienen acceso a una sección crítica de manera concurrente.
- Se utilizar un semáforo para controlar el acceso a los recursos.
- Un semáforo indica la capacidad de recursos que pueden ser utilizados, y el resto de requerimientos son rechazados, hasta que los usados se vayan liberando.

Unidad III, Programación concurrente

- 3.7.3 Semáforos
- En C#, existe la clase '[System.Threading.SemaphoreSlim](#)' que nos permite crear semáforos ligeros, que nos permiten limitar el número de *threads* que pueden acceder a un recurso o a un conjunto de recursos.
- Se le llama semáforos ligeros a aquellos implementados en C# y que **no** son delegados al sistema operativo para su manejo (Windows Kernel).
- No pueden ser nombrados y se recomiendan para una sola aplicación.
- Deben ser usados cuando el tiempo de espera es mínimo.

Ejemplo:

<https://github.com/danunora/unedi/tree/master/Command/SemaforoSlimTest>

Unidad III, Programación concurrente

- 3.7.3 Semáforos (Semaphore and SemaphoreSlim)

Ejemplos:

<http://www.albahari.com/threading/part2.aspx>

<http://www.tutorialspoint.com/Semaphore-in-Chash>

<https://github.com/danunora/unedl/tree/master/Command/Semaphores>

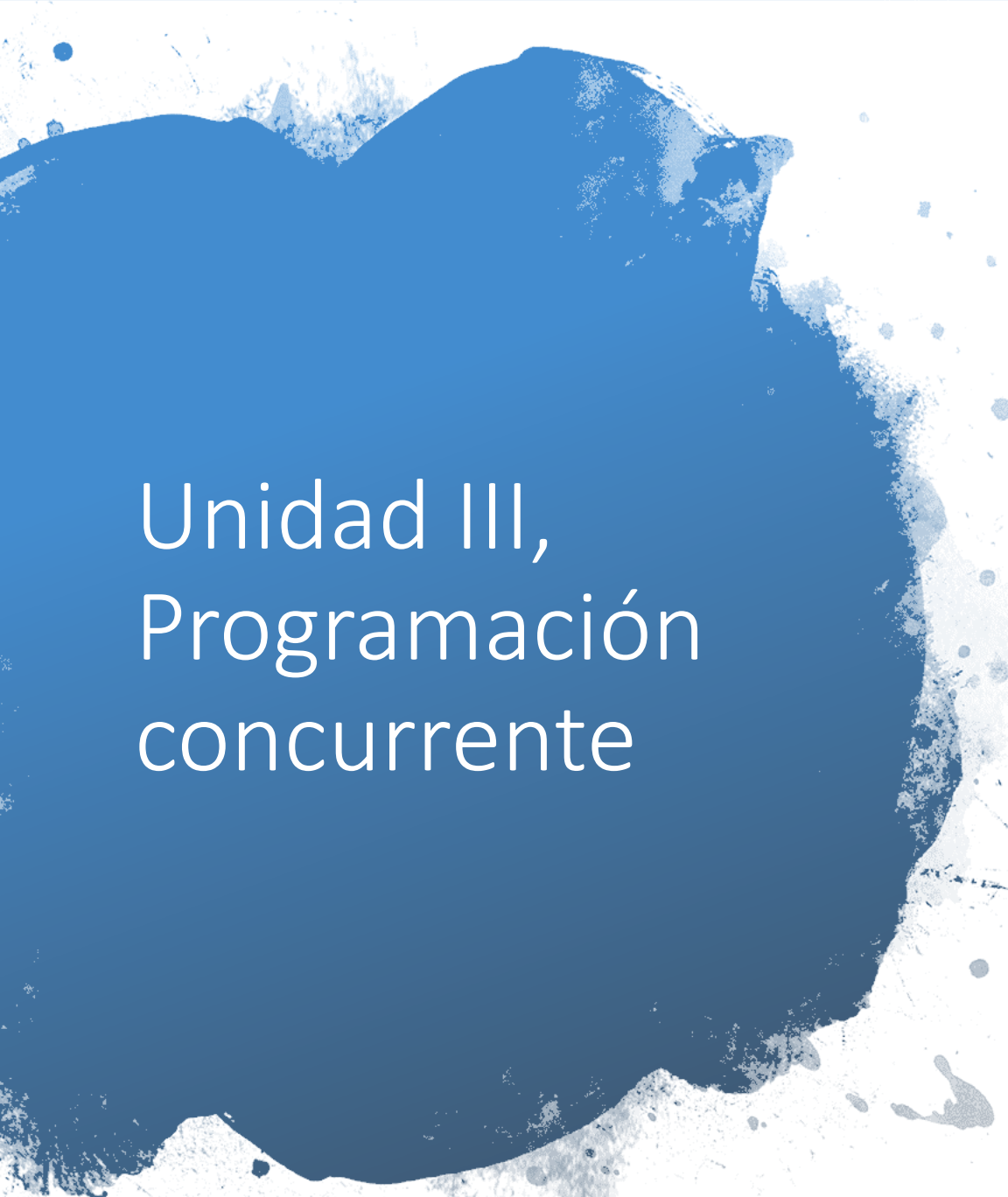
<https://github.com/danunora/unedl/tree/master/Command/SemaphoresTheClub>

<https://github.com/danunora/unedl/tree/master/Command/SemaforoSlimTest>

UNIDAD I, Introducción a la Programación Visual

TAREA # 8 Glosario: 2do Parcial

- Descripciones en no menos de tres líneas
- Ordenado alfabéticamente
- Con excelente ortografía
- Incrementando a lo largo del curso
- En cada parcial se preguntaran términos del Glosario que cuentan para la calificación



Unidad III, Programación concurrente

- 3.7 Sincronización sin Bloqueos.
 - 3.7.1 Interlock class

Unidad III, Programación concurrente

- 3.7 Sincronización sin bloqueos

Interlock Class

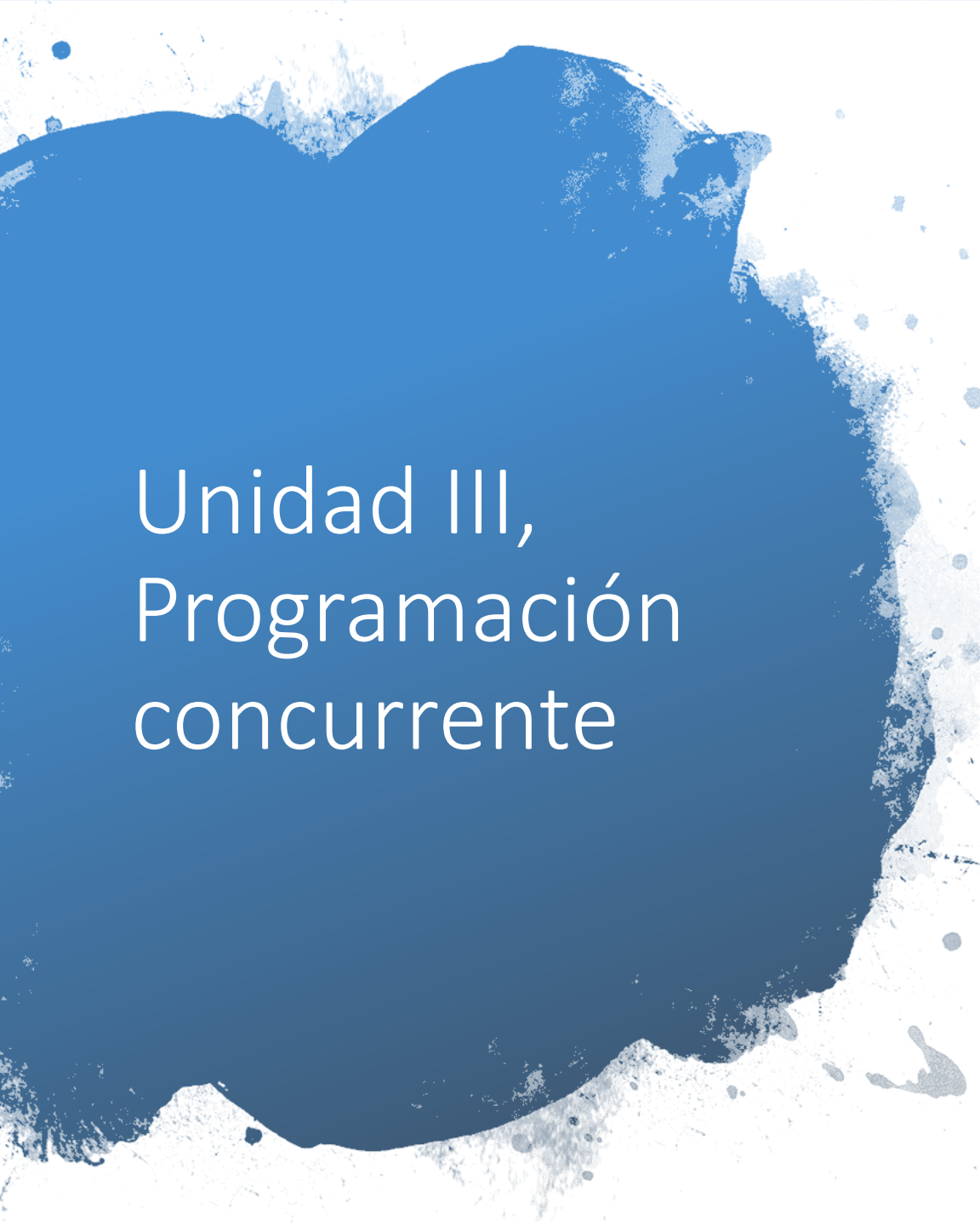
Provee operaciones atómicas para las variables que están compartidas entre múltiples *threads*.

Referencia:

<https://docs.microsoft.com/en-us/dotnet/api/system.threading.interlocked>

Ejemplo:

<https://github.com/danunora/unedi/tree/master/Command/Interlocked>

A large, dark blue, irregular splash-like graphic on the left side of the slide, with a textured, watercolor-like appearance. It has several smaller, lighter blue splatters extending from its right edge into the white background.

Unidad III, Programación concurrente

- 3.9 Microsoft C# parallelism API
- Task Parallel Library (TPL)

Unidad III, Programación concurrente

- 3.9 Microsoft C# parallelism API
- Task Parallel Library (TPL)

Conjunto de tipos públicos y APIs que forman parte de los espacios de nombres (namespaces) usados para trabajar con *threads*.

Task is a TPL's unit of work

Su propósito es simplificar el trabajo del desarrollador para agregar paralelismo y concurrencia a sus aplicaciones.

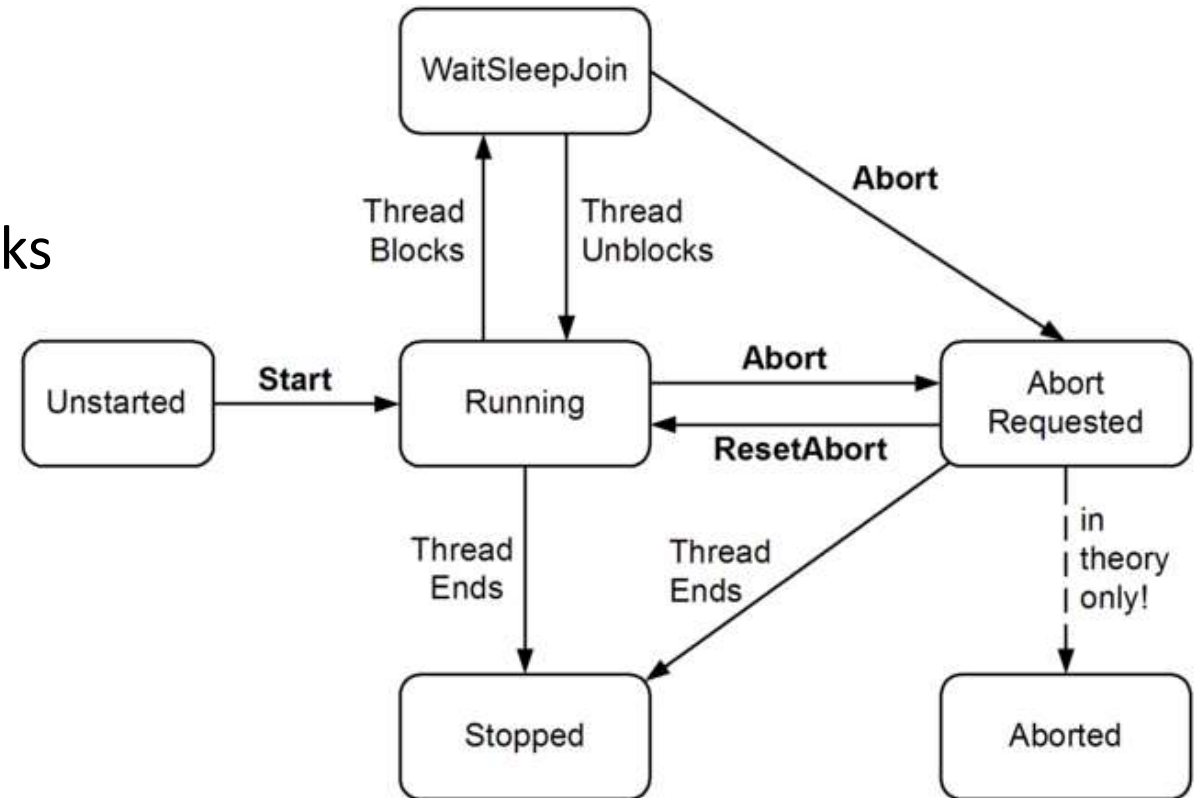
- [System.Threading](#)
- [System.Threading.Tasks](#)

Unidad III, Programación concurrente

- 3.9 Microsoft C# parallelism API
- Parallel LINQ (PLINQ) es una implementación de los objetos LINQ(Language-Integrated Query, parallelized)
- Referencia:
- <https://docs.microsoft.com/en-us/dotnet/standard/parallel-programming/parallel-linq-plinq>

Unidad III, Programación concurrente

- 3.9 Microsoft C# parallelism API
- Task Parallel Programming
 - Creating and Starting Tasks
 - Cancelling Tasks
 - Waiting for Time to Pass or for Tasks
 - Exception Handling



Unidad III, Programación concurrente

- 3.9 Microsoft C# parallelism API
- Creating and Starting Tasks
 - Method1, Task is getting created and started
 - `Task.Factory.StartNew();`
 - Method2, Task is getting created and needs to be started
 - `var t = new Task();`
 - `t.Start();`

Example:

<https://github.com/danunora/unedl/tree/master/Command/TaskExampleStarting>

Unidad III, Programación concurrente

- 3.9 Microsoft C# parallelism API
- Cancelling Tasks
 - Method1
 - IsCancellationRequested?
 - break
 - Method2
 - IsCancellationRequested?
 - throw new OperationCanceledException()
 - Method3 (Recommended)
 - ThrowIfCancellationRequested()

Example:

<https://github.com/danunora/unedi/tree/master/Command/TaskExampleCancelling>

Unidad III, Programación concurrente

- 3.9 Microsoft C# parallelism API
- Waiting for Time to Pass
 - Thread.Sleep()
 - Pauses the thread of execution
 - Scheduler can get another task executed while sleep
 - [Thread.SpinWait\(\)](#)
 - SpinWait.SpinUnit(),
 - Pause the thread, but don't give up your place in the execution list
 - Wasting resources, but it's avoiding expensive context switches

Example:

<https://github.com/danunora/unedl/tree/master/Command/TaskExampleSpinWait>

<https://github.com/danunora/unedl/tree/master/Command/TaskExample2SpinWait>

<https://github.com/danunora/unedl/tree/master/Command/TaskExampleWaiting>

Unidad III, Programación concurrente

- 3.9 Microsoft C# parallelism API
- Task Status:

Task Value	Description
Created	Created but not started
WaitingForActivation	Waiting to be scheduled until some dependant operation has completed
WaitingToRun	Waiting for the scheduler
Running	Currently running.
WaitingForChildrenToComplete	Waiting for all attached children to complete
RanToCompletion	Final state. Success
Canceled	Final state. Was cancelled
Faulted	Final state. Unhandled exception.

- Source:
 - <https://blogs.msdn.microsoft.com/pfxteam/2009/08/30/the-meaning-of-taskstatus/>

Unidad III, Programación concurrente

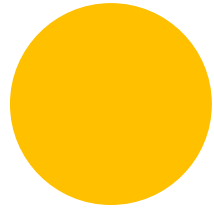
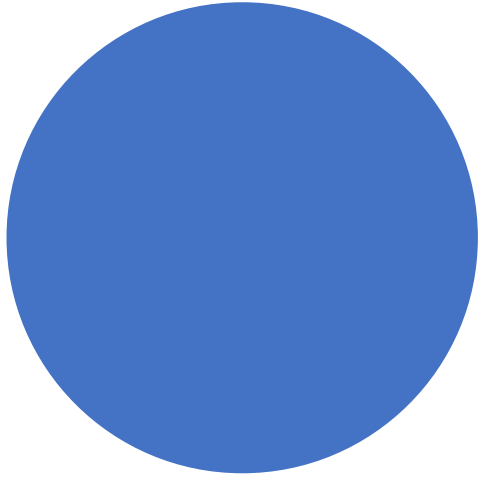
- 3.9 Microsoft C# parallelism API
- Exception Handling
 - Las tareas generan diferentes excepciones
 - Si no se manejan las excepciones en una tarea diferente podrían ser ignoradas
 - Se debe utilizar “**AggregateException**” para manejarlas y hacer un ciclo.

Referencia:

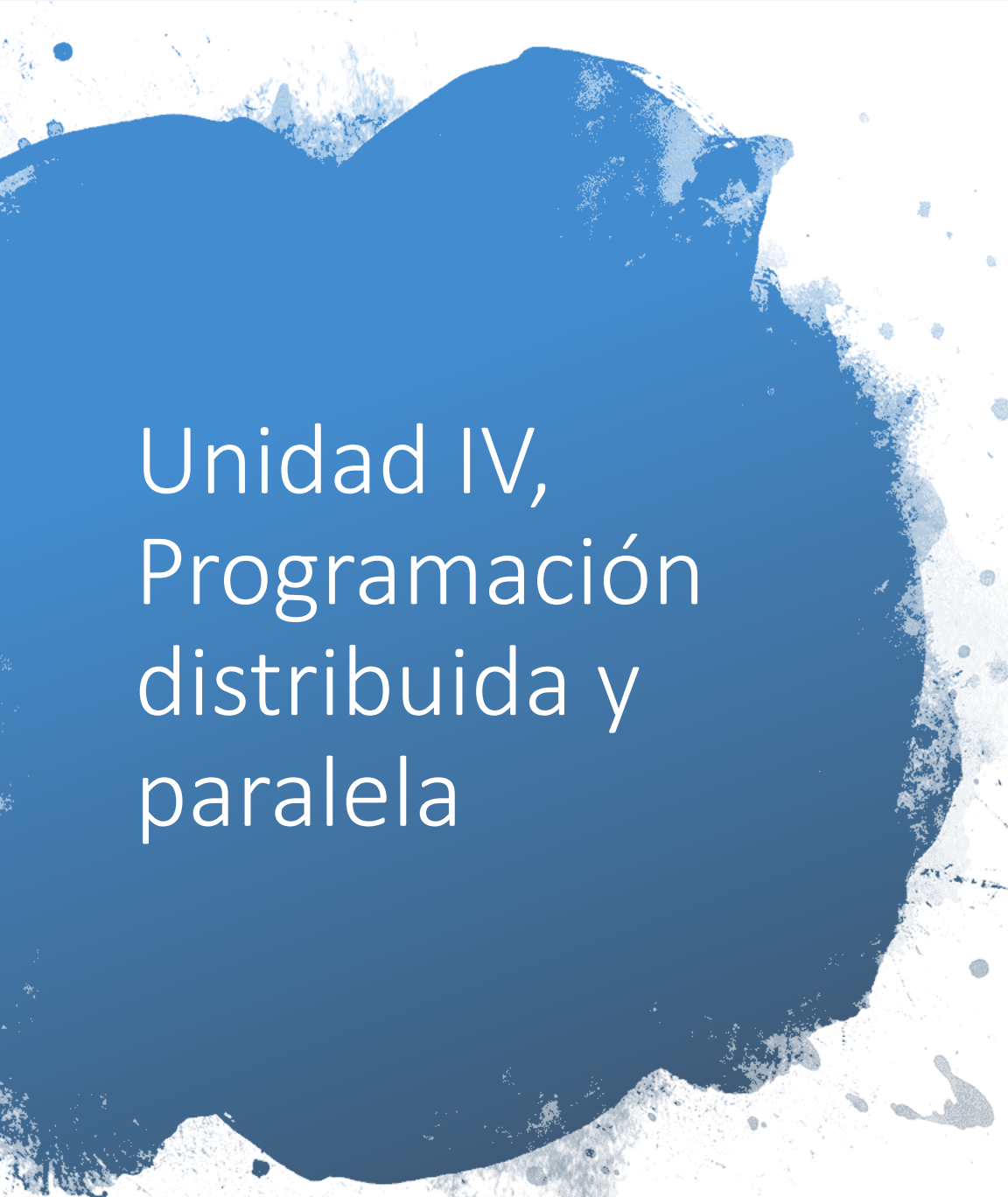
[https://msdn.microsoft.com/en-us/library/dd537614\(v=vs.110\).aspx](https://msdn.microsoft.com/en-us/library/dd537614(v=vs.110).aspx)

Example:

<https://github.com/danunora/unedi/tree/master/Command/TaskExampleExceptions>



Unidad IV, Programación distribuida y paralela



Unidad IV, Programación distribuida y paralela

- 4.1 Introducción
- 4.2 Esquemas algorítmicos básicos
- 4.3 Diseño de algoritmos paralelos y distribuidos
- 4.4 Notación y lenguajes
- 4.5 Aplicaciones distribuidas

Unidad IV, Programación distribuida y paralela

- 4.1 Introducción

La computación distribuida se refiere a la capacidad de llevar a cabo una tarea mediante la división de la misma en subtareas, las cuales pueden ser resueltas por una red de computadoras denominado sistema distribuido.

Estos sistemas distribuidos tienen la habilidad de comunicarse y coordinar sus actividades mediante el intercambio de información y el estado de sus tareas individuales.

Los sistemas distribuidos permiten tener un sistema compuesto por computadoras de menor costo, en comparación a un solo sistema con mucha capacidad.

Unidad IV, Programación distribuida y paralela

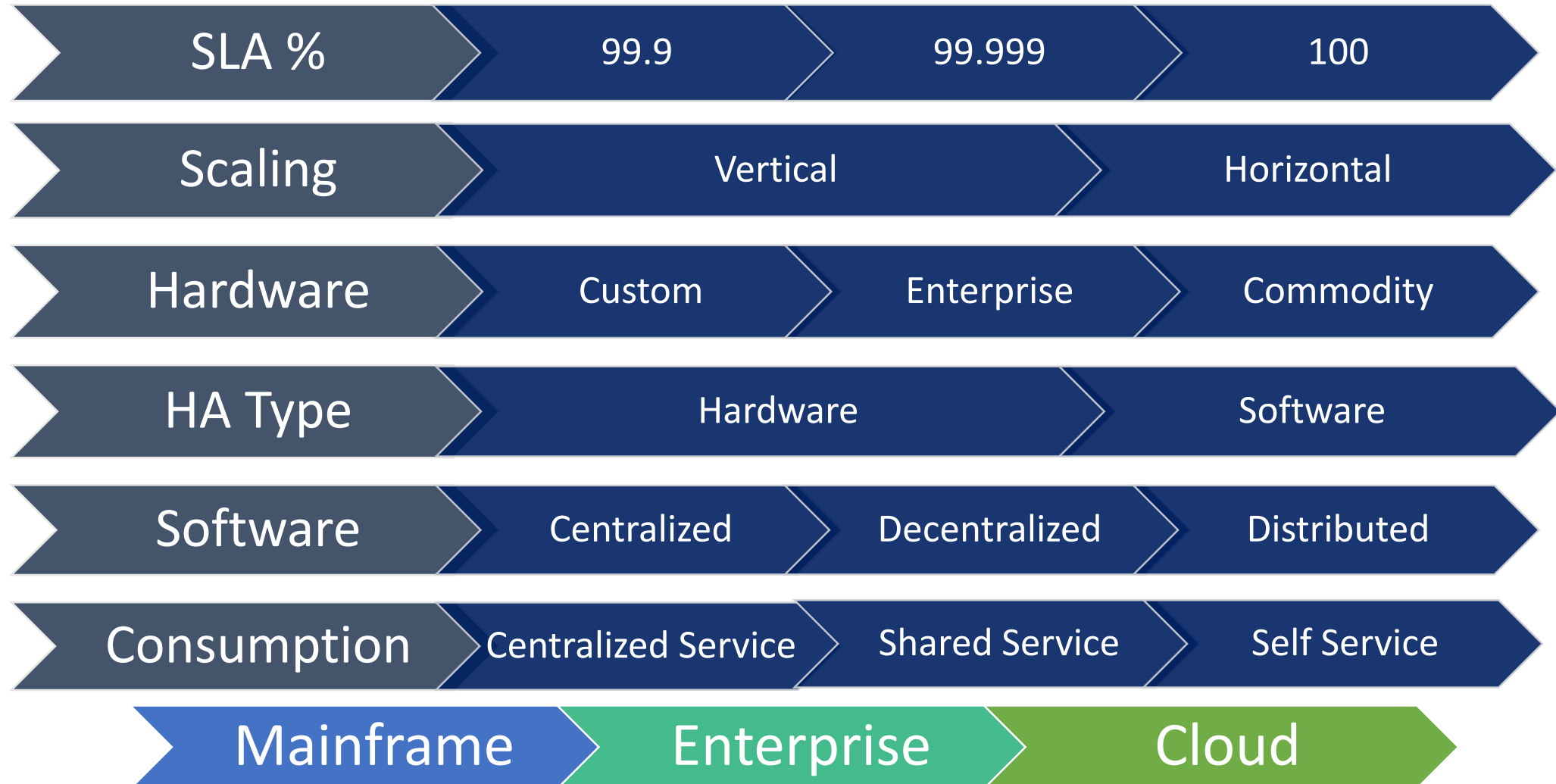
- 4.1 Introducción

El modelo de sistemas distribuidos distingue entre nodos y procesos.

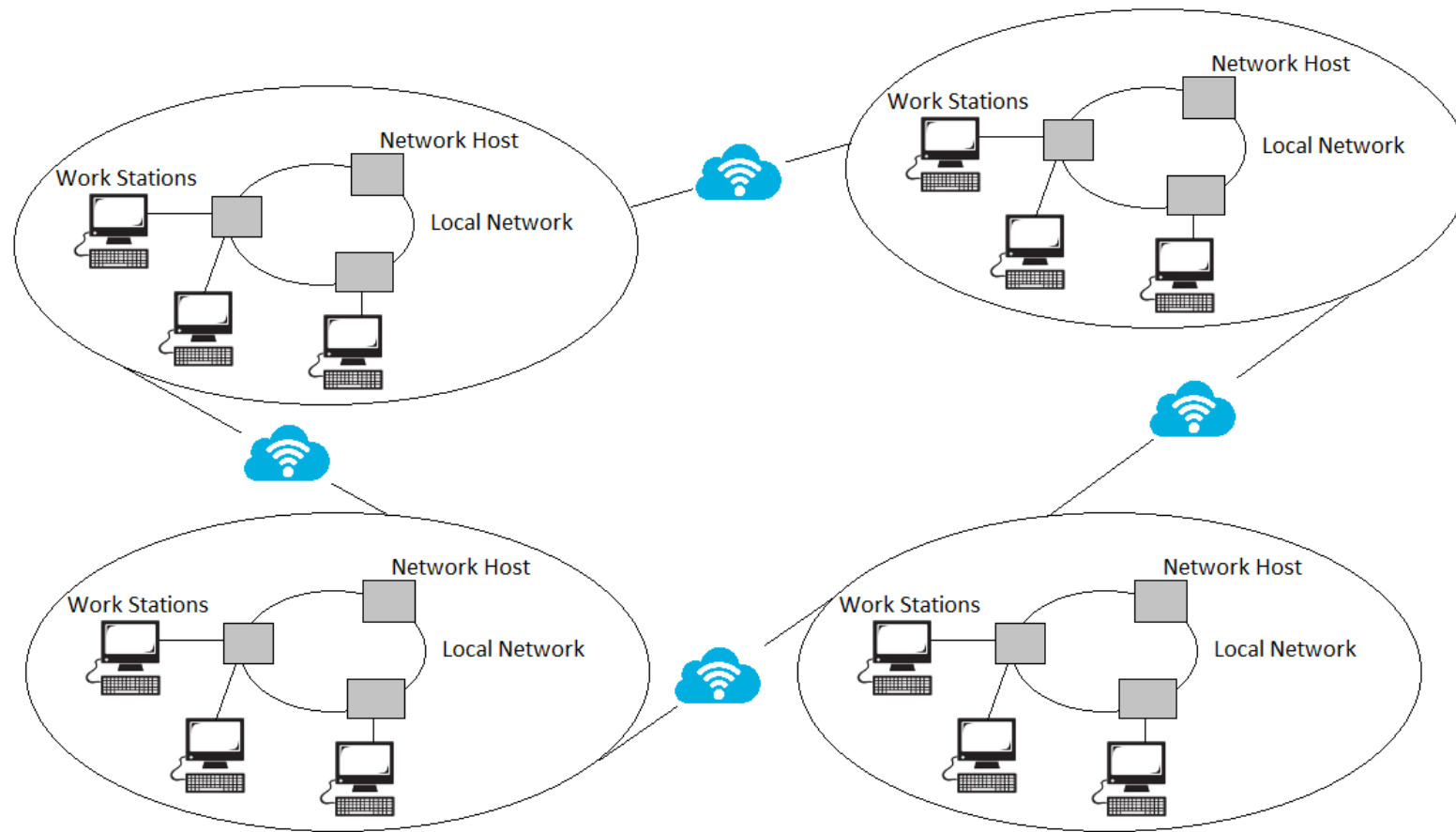
Un **nodo** intenta representar a un objeto físicamente identificable como una computadora, mientras que una computadora puede estar ejecutando múltiples **procesos**, ya sea por compartir un procesador o múltiples procesadores.

- La comunicación y sincronización entre procesos ejecutándose en un nodo, se realiza utilizando primitivas de **memoria compartida**.
- La comunicación y sincronización ente procesos ejecutándose en diferentes nodos, se realiza **intercambiando mensajes**.

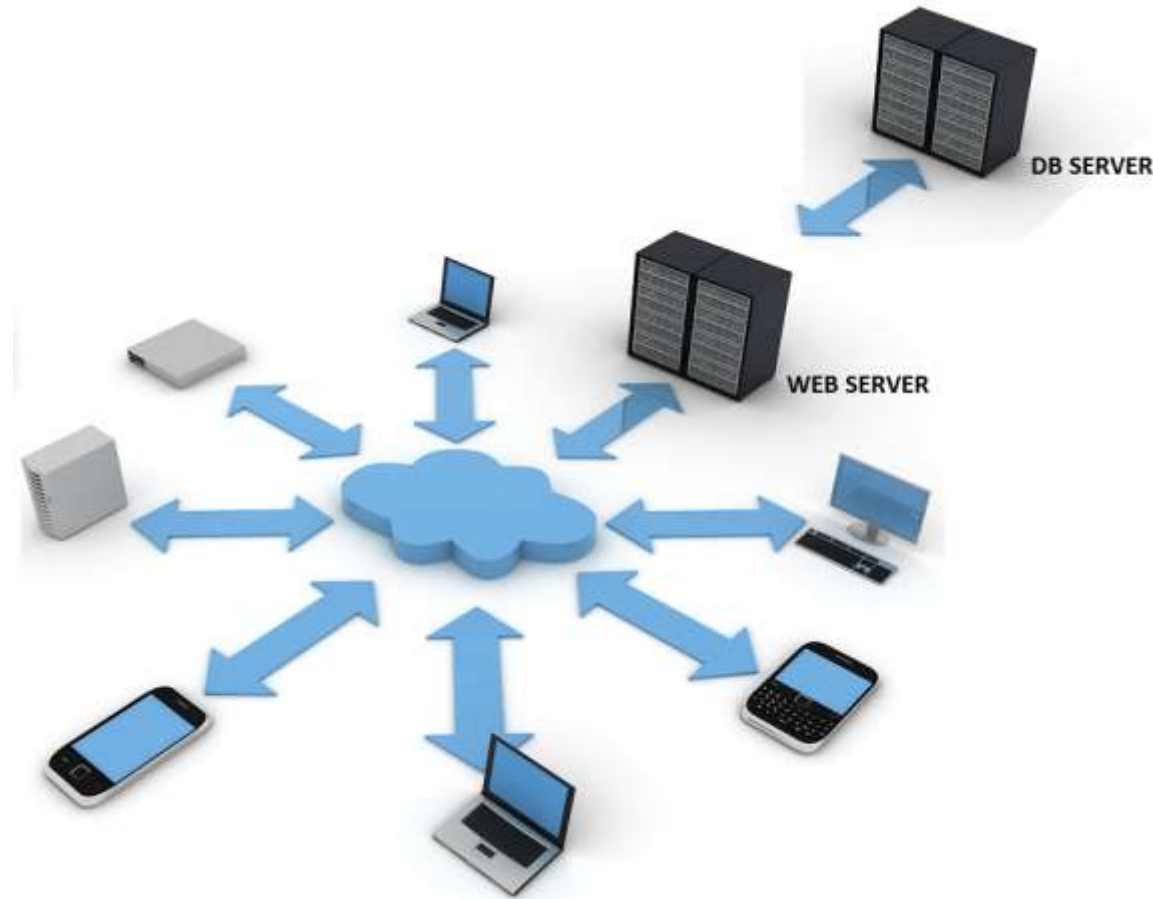
Unidad IV, Programación distribuida y paralela



Unidad IV, Programación distribuida y paralela



Unidad IV, Programación distribuida y paralela

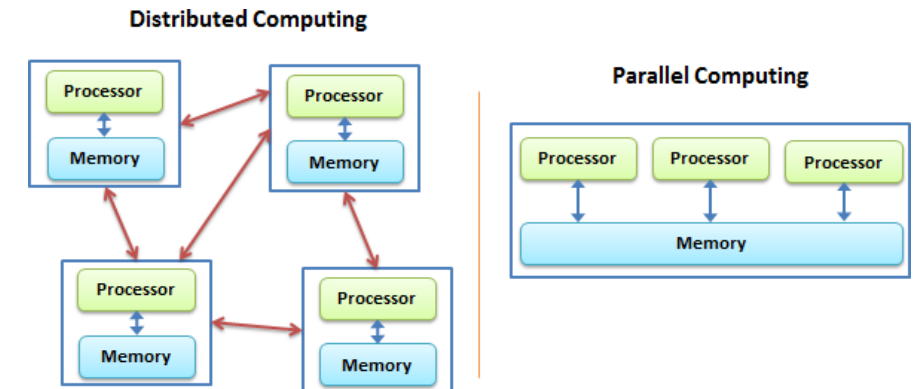


Unidad IV, Programación distribuida y paralela

- 4.1 Introducción

La computación distribuida VS la computación en paralelo

- Computación en paralelo, consiste en múltiples procesadores que se comunican entre si utilizando memoria compartida (***shared memory***).
- Computación distribuida, consiste en múltiples procesadores que se comunican entre si utilizando una red de comunicación.



Unidad IV, Programación distribuida y paralela

• 4.1 Introducción

Razones para construir un sistema distribuido y además en paralelo.

- **Escalabilidad**, un sistema distribuido no tiene los problemas asociados con memoria compartida y debido a que se pueden agregar nuevos procesadores, es mas escalable que sistemas en paralelo.
- **Confiabilidad**, en el sistema distribuido, el impacto en uno de sus componentes no impacta en la totalidad del sistema.
- Compartir datos,
- **Compartir recursos**, en un sistema distribuido se comparten todos los recursos, aun los que son muy exclusivos.
- **Modular**, un sistema distribuido puede ser flexible como para aceptar nuevos procesadores y también, para tolerar las fallas mientras se reemplaza o se remueve un procesador.
- **Geografía**, los sistemas distribuidos permiten la distribución geográfica de los recursos.
- **Economía**, con la evolución de las nuevas computadoras y las redes de banda ancha, el costo se reduce.

Unidad IV, Programación distribuida y paralela

- 4.1 Introducción

Uniform Memory Access (UMA)

Sistema que contiene mas de un procesador y que tiene acceso directo a la memoria compartida, la cual forma un espacio de direcciones común.

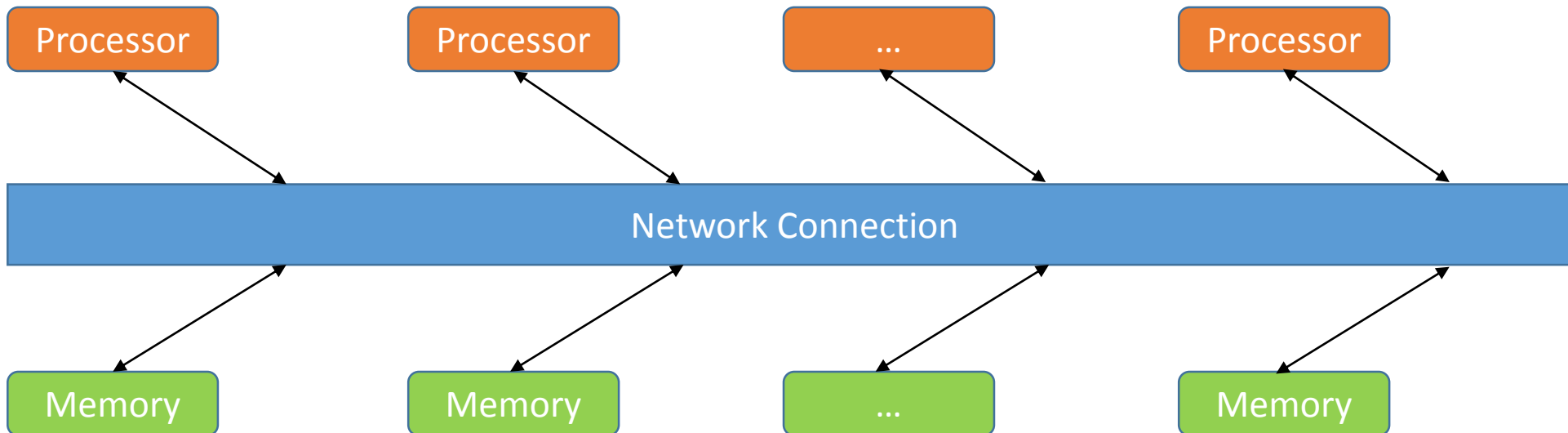
La latencia de acceso es la misma para cualquier dirección de memoria.

La comunicación interprocesos, se realiza mediante operaciones de lectura y escritura a través de la memoria compartida.

Unidad IV, Programación distribuida y paralela

- 4.1 Introducción

Uniform Memory Access (UMA)



Unidad IV, Programación distribuida y paralela

- 4.1 Introducción

Non-Uniform Memory Access (UMA)

Un sistema paralelo compuesto de múltiples computadoras es otro tipo de sistema distribuido y contiene múltiples procesadores, configurados sin tener acceso directo a memoria compartida.

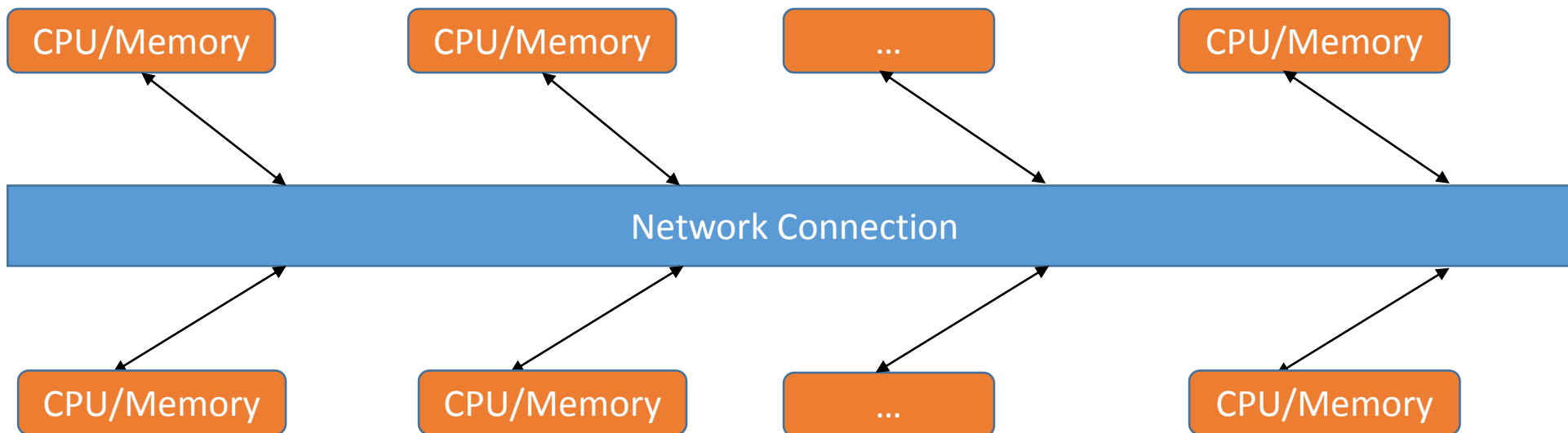
Los procesadores pueden establecer comunicación ya sea con un espacio de direcciones en común o opciones de intercambio de mensajes.

La latencia de acceso a diferentes localidades de memoria es variable.

Unidad IV, Programación distribuida y paralela

- 4.1 Introducción

Non-Uniform Memory Access (NUMA)



Unidad IV, Programación distribuida y paralela

- 4.1 Introducción

La ley Amdahl

Es una ley que frecuentemente es considerada en computación en paralelo para pronosticar la mejora en los tiempos de procesamiento, cuando se incrementa la cantidad de múltiples procesadores de sistema.

$$S_{\text{latency}}(s) = \frac{1}{(1 - p) + \frac{p}{s}}$$

- S_{latency} is the calculated improvement of the latency (execution) of the complete task.
- s is the improvement in execution of the part of the task that benefits from the improved system resources.
- p is the proportion of the execution time that the part benefiting from improved resources actually occupies.

UNIDAD I, Introducción a la Programación Visual

TAREA # 9 Java RMI, C# WCF: 3er Parcial

- Realizar una investigación acerca de:
 - Java RMI (Remote Method Invocation)
 - WCF (Windows Communication Foundation)
- Subir el documento a github o bitbucket y enviar por email (daniel_nuno@hotmail.com) la liga al mismo.

Unidad IV, Programación distribuida y paralela

- 4.2 Esquemas algorítmicos básicos

Unidad IV, Programación distribuida y paralela

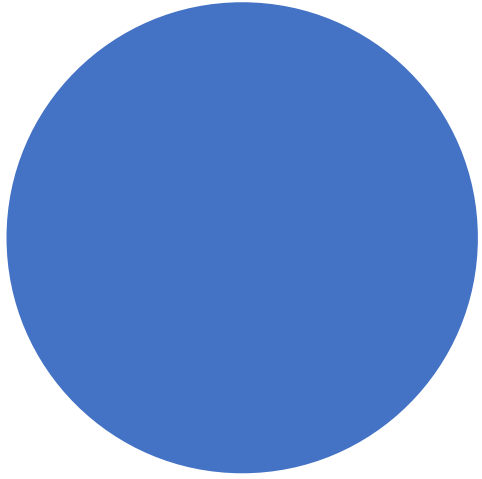
- 4.3 Diseño de algoritmos paralelos y distribuidos

Unidad IV, Programación distribuida y paralela

- 4.4 Notaciones y lenguajes

Unidad IV, Programación distribuida y paralela

- 4.5 Aplicaciones distribuidas



Unidad V,
Middleware



Unidad V Middleware

- 5.1 Conceptos básicos
- 5.2 Categorías
 - 5.2.1 Tuplas distribuidas
 - 5.2.2 Procedimientos remotos
 - 5.2.3 Objetos Distribuidos
- 5.3 Sistemas de alto desempeño
- 5.4 Principales aplicaciones



Anexos

Bibliografía

- Rapid J2EE™ Development: An Adaptive Foundation for Enterprise Applications
 - by Alan Monnox
 - Publisher: Prentice Hall
 - Release Date: March 2005
 - ISBN: 9780131472204
- Pro Asynchronous Programming with .NET
 - by Andrew Clymer, Richard Blewett
 - Publisher: Apress
 - Release Date: December 2013
 - ISBN: 9781430259206

Bibliografía

- C# for Java Developers
 - by Adam Freeman, Allen Jones
 - Publisher: Microsoft Press
 - Release Date: August 2002
 - ISBN: 9780735617797
- .NET Common Language Runtime Unleashed
 - by Kevin Burton
 - Publisher: Sams
 - Release Date: April 2002
 - ISBN: 0672321246

Bibliografía

- C# 7.0 in a Nutshell
 - by Ben Albahari, Joseph Albahari
 - Publisher: O'Reilly Media, Inc.
 - Release Date: October 2017
 - ISBN: 9781491987650
- .NET Common Language Runtime Unleashed
 - by Kevin Burton
 - Publisher: Sams
 - Release Date: April 2002
 - ISBN: 0672321246

Referencias

- <https://docs.microsoft.com/en-us/dotnet/index>
- <https://visualstudio.microsoft.com/>
- <https://www.tutorialspoint.com/csharp/>
- <http://www.tutorialsteacher.com/csharp/csharp-tutorials>
- <http://www.albahari.com/threading/>
- Online Test
- https://www.tutorialspoint.com/csharp/csharp_online_test.htm