CAPÍTULO I: INTRODUCCIÓN

1.1. Conceptos Básicos de Ingeniería de Software

El Software

- No sólo abarca los programas de computación asociados con alguna aplicación o producto.
- Junto con los programas, el software incluye toda la documentación necesaria para instalar, usar, desarrollar y mantener esos programas.
  - ➢ Manuales: Usuario, Administrador, Desarrollador, etc.
  - ➢ Comentarios en el Código: JavaDOC.

Documentación	***//…..*** ***/*…*** ****/***

Software = Programas + Documentación

Características del Software

- Es muy costoso.
- Entregado generalmente con atraso.
- Generalmente, con un costo mayor al presupuestado.
- Generalmente desarrollado de manera artesanal (poco confiable)
- Costo alto de mantención.

Poco confiable, por ejemplo (Fuente: http://bit.ly/cwtX0L)...

- **Sistema de alerta soviético casi causa la III guerra mundial ('83)**: Un error en el software de detección de la red de satélites mal interpreta la reflexión del sol sobre las nubes de gran altitud como un misil.

- **Mars Climate Observer ('98)**: Un error en el software de navegación de la sonda causó que volara a muy baja altura y se estrelló. (sistema métrico vs imperial).

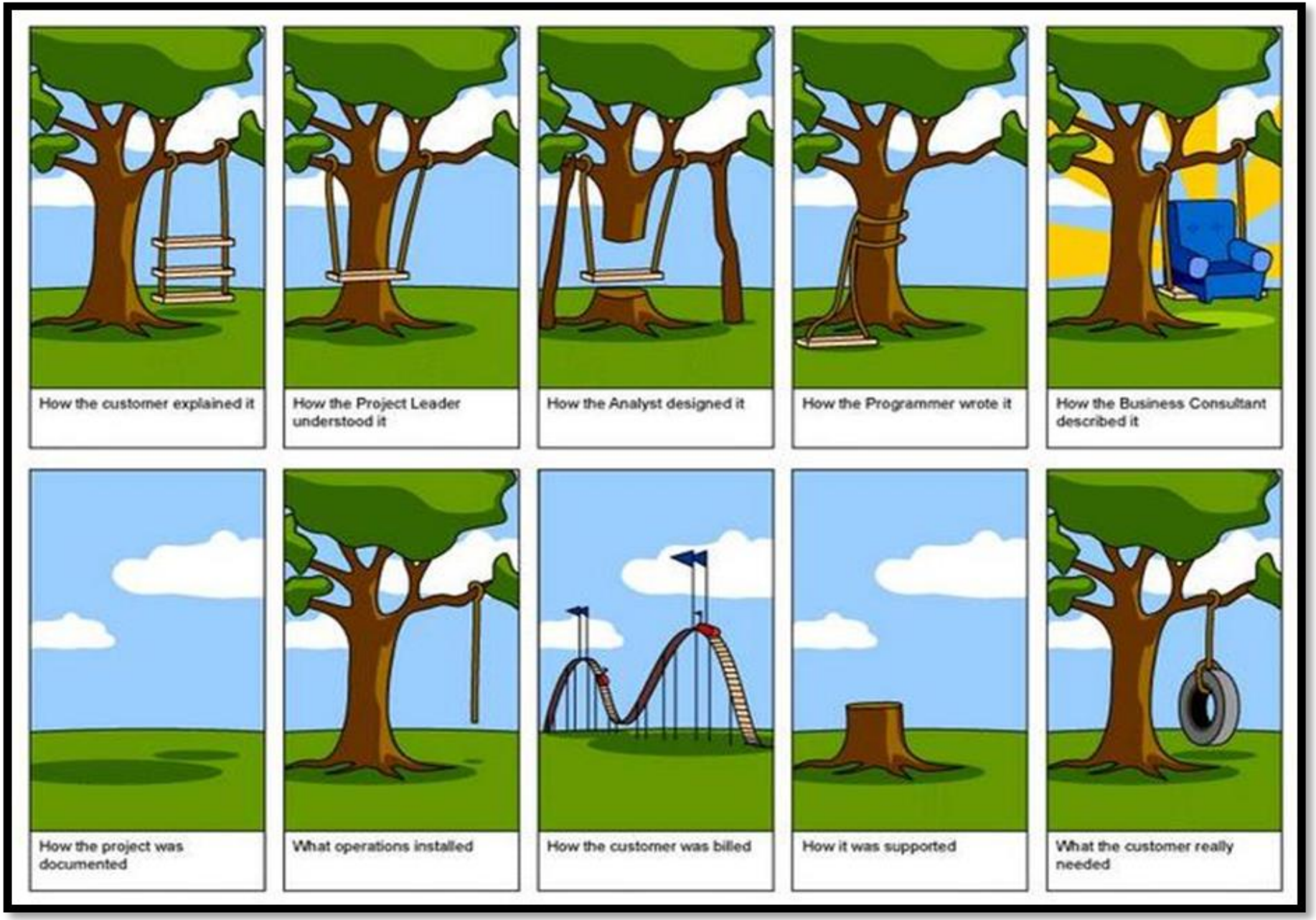
Ingeniería de Software

- Es el estudio de los principios y metodologías para desarrollo de sistemas de software.

- Aplicación de un enfoque sistemático, disciplinado y cuantificable, al desarrollo, operación y mantenimiento de software.

- Es la aplicación práctica del conocimiento científico en el diseño y construcción de programas de ordenador y la documentación adecuada para desarrollar, operar y mantenerlos.

- Se trata del establecimiento de los principios y métodos de la ingeniería a fin de obtener software de modo rentable.


How the customer explained it
How the Project Leader understood it
How the Analyst designed it
How the Programmer wrote it
How the Business Consultant described it
How the project was documented
What operations installed
How the customer was billed
How it was supported
What the customer really needed

Ciclo de desarrollo de software

Pruebas = Testing

Depuración = Debugging

Mantención

- Correctiva: Corregir Errores
- Adaptiva: Cambios en el Ambiente
- Perfectiva: Cambios en las Solicitudes del Usuario

Problema: Construcción de una casa

Se parte haciendo el análisis, donde se ven los requerimientos de la casa (cantidad de dormitorios, tamaño de la cocina, baños, etc.). Luego se hace el diseño (planos y maquetas) y finalmente se construye la casa. A medida que se construye la casa, se hacen las pruebas respectivas, por ejemplo, se chequea la calidad del hormigón.

1.2. Casos de prueba

Las **pruebas de software** es el proceso por el cual se busca garantizar la calidad de un producto desarrollado. Se utiliza para identificar posibles fallas en los productos software.

Por lo tanto, el proceso de **pruebas** de software tiene dos objetivos:

- Demostrar al desarrollador y cliente que el software satisface sus requerimientos.
- Descubrir defectos en el software (comportamiento incorrecto, no deseado o que no cumple su especificación).

Pruebas de Caja Negra o testing funcional

- Las condiciones de prueba se desarrollan en base a las funcionalidades del programa, es decir, el tester (persona dedicada a ejecutar las pruebas) requiere información sobre los datos de entrada y las salidas observadas.
- Se requiere conocer cuáles son los resultados esperados.
- No se requiere conocer cómo trabaja el software.

Casos de prueba

Casos de Prueba		
Número	**Datos de prueba**	**Resultados Esperados**

Ejemplo de un caso de prueba

Problema: Construir un programa para calcular e imprimir el promedio de edad de los estudiantes. Se leen las edades de los estudiantes. Fin de datos, edad = -1

Casos de Prueba		
Número	**Datos de prueba**	**Resultados Esperados**
1	20 19 18 -1	Edad promedio: 19
2	18 21 -1	Edad promedio: 19.5
3	-1	No se ingresaron edades

Debugging de un programa

Es el proceso de identificar y corregir errores de programación. En su forma más básica se puede depurar (debugging) un programa colocando, en algún lugar de él, instrucciones para desplegar a la pantalla mensajes o resultados intermedios, de manera de poder seguir la ejecución del programa.

Validación de datos de entrada

Verificar correcta lectura de la información ingresada.
Verificar formatos de rut, montos, fechas, etc.
Verificar el ingreso de datos sólo numéricos donde corresponda.
Verificar validación en el ingreso de los días correspondientes al mes.
Verificar validación en el ingreso de los meses correspondientes al año.

Ejemplos de validación de datos de entrada:

- Se están ingresando las edades de los alumnos, se debe validar que no pueden ser menores a 15 y mayores a 70
- Se están ingresando las notas de los alumnos y estas deben ser numéricas y estar entre 1.0 y 7.0
- Se está ingresando un mes, se debe validar que esté entre 1 y 12.

¿Qué hacer si el dato ingresado está incorrecto?

Por ejemplo si el tipo debe tomar el valor 1 o 2

- Desplegar un mensaje a la pantalla y no se hace nada.

```
Leer tipo
if (tipo != 1 && tipo != 2){
        StdOut.println("tipo incorrecto");
}
else {
        ….
}
```

- Leer hasta que esté correcto lo que se está leyendo.

```
do
        leer tipo
while(tipo != 1 &&  tipo != 2);
```

En los talleres es obligatorio validar los datos de entrada

Validación del rut

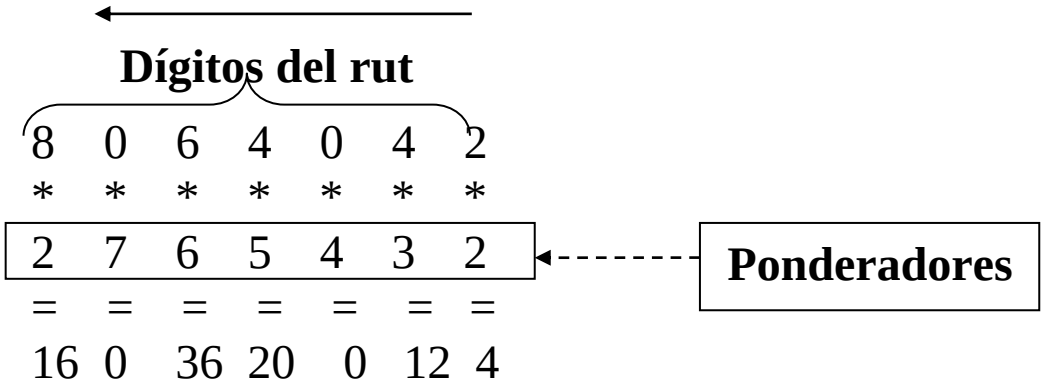
Para validar el rut se utiliza su dígito verificador, es decir, se debe comparar el dígito verificador del rut con un dígito verificador que se calcula con los dígitos que tiene el rut hasta antes del guión. Si son iguales está correcto, sino significa que hay un error en el rut ingresado.

¿Cómo se calcula el dígito verificador (DV)?

rut: 8.064.042 -0

1. Se multiplica cada dígito por un ponderador que parte en 2, se va incrementando en 1 por cada dígito y su valor máximo es 7.

Si hay más dígitos se vuelve a partir desde el 2.
El rut se considera de atrás para adelante.



2. Se hace la suma de los valores obtenidos anteriormente.

16 + 0 + 36 + 20 + 0 + 12 + 4 = 88

3. El resultado de la suma anterior se divide por 11, obteniéndose el resto de la división

88: 11 = 8

0 ◄---------- **Resto de la división**

4. Se resta 11 menos el resto de la división anterior

11 – 0 = 11 ◄---------- **Resultado de la resta**

5. Si resultado

- es 11 ➔ DV = 0
- es 10 ➔ DV = K
- está entre 1 y 9 ➔ DV = resultado

1.3. Excepciones

Uno de los aspectos más importantes a la hora de programar es el manejo de errores. En Java una **Excepción** es un evento que ocurre durante la ejecución del programa interrumpiendo el flujo normal de las instrucciones. A diferencia de un error en tiempo de compilación (error de sintáxis), estos errores deben ser controlados durante tiempos de ejecución mediante el control de excepciones.

¿Qué pasaría si tratáramos de realizar una división entre cero?

```
public class App {
    public static void main(String[] args) {
        double division = 2/0;
    }
}
```

La ejecución de este programa arrojaría el siguiente error:

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
```

¿Cómo se podría **controlar**?
En Java este tipo de errores o **excepciones** se controlan mediante la sentencia **try-catch**.

```
try{
    // sentencias de código que podrían originar un error

}catch(TipoExcepcion  objetoExcepcion) {
    // sentencias de código que “capturan” el error

}finally{
    // sentencias de código que se ejecutarán siempre después del
    // try-catch
}
```

```
public class App {
    public static void main(String[] args) {

        try{
            double division = 2/0;

        }catch(ArithmeticException  ex){
            StdOut.println("¡Error! División entre cero");

        }finally{
            StdOut.println("El programa ha terminado");
        }
    }
}
```

Se puede observar que en el bloque **catch** se ha definido una excepción de tipo “ArithmeticException” o excepción aritmética, del mismo modo, es posible definir otros tipos de excepciones.

```
        try{
            String numero = “HolaMundo”;
            int b = Integer.parseInt(numero);

        }catch(NumberFormatException ex){
            StdOut.println("¡Error! No es posible realizar la conversión");
        }
```

En este caso se trató de convertir una cadena de texto a número, el bloque **catch** capturó el error de tipo “formato numérico”, es decir, el formato “HolaMundo” no tiene las características de un valor de tipo numérico.

Los errores más comunes en tiempos de ejecución son:

- **ArithmeticException**: producto de una condición aritmética (ejemplo: división entre cero).
- **NumberFormatException**: al tratar de convertir una cadena de texto con formato incorrecto a número.
- **ClassCastException**: cuando se trata de convertir una instancia (objeto) a otra clase de la cual no es objeto.
- **NullPointerException**: al trabajar con objetos nulos o no instanciados (ejemplo: modificar los atributos de un objeto nulo).
- **IndexOutOfBoundsException**: cuando se trata de acceder a una posición inexistente dentro de un arreglo (ejemplo: acceder a la posición -1 de un vector).
- **IllegalArgumentException**: se utiliza para indicar que un método ha recibido parámetros incorrectos o "argumentos" inapropiados (ejemplo: nombre nulo)

También es posible definir una o más excepciones dentro de un mismo bloque **try-catch**:

```
        try{
            //….
        }catch(NumberFormatException excepcion){
            StdOut.println("¡Error! No es posible realizar la conversión");

        } catch(NullPointerException excepcion){
            StdOut.println("¡Error! Instancia de clase nula");

        } catch(ArithmeticException excepcion){
            StdOut.println("¡Error! Error aritmético");
        }
```

De este modo cada vez que se produzca un error específico, se ejecutará el **catch** correspondiente a la naturaleza de dicho error.

¿Qué pasaría si no estuviera definida la excepción correcta en el cuerpo del **try-catch**?

```
public class App {
    public static void main(String[] args) {
        try{
            double division = 2/0;
        }catch(NumberFormatException excepcion){
            StdOut.println("¡Error! No es posible realizar la conversión");
        }
    }
}
```

En este caso el error producido es de tipo aritmético y el bloque **catch** hace referencia a una excepción de tipo formato numérico, por lo cual, este error **no ha sido controlado correctamente y el programa se cae**

En Java todas las excepciones heredan de la clase **Exception,** la cual puede ser usada dentro del bloque **catch** el cual se ejecutará automáticamente si es que en el bloque del **try-catch** no ha sido definida la excepción correspondiente al error. La desventaja de usar la clase **Exception** en este caso, radica en que no es posible identificar cuál es el tipo de error que gatilla la excepción.

```
try{
        double division = 2/0;
}catch(NumberFormatException excepcion){
        StdOut.println("¡Error! No es posible realizar la conversión");
}catch(Exception excepcion){
        StdOut.println("¡Error! Ha ocurrido un error");
}
```

Al no encontrar la excepción correspondiente a **ArithmeticException**, se ejecuta el **catch** de la clase **Exception**.

Es importante señalar que la verificación de las excepciones se realizará en el orden que han sido definidas dentro del bloque **try-catch.** Por lo tanto, si inicialmente se ha definido el **catch(Exception e)** este será el primer **catch** que se ejecutará (ya que es la super clase de las excepciones en Java).

```
try{
        double division = 2/0;
}catch(Exception excepcion){
        StdOut.println("¡Error! No es posible realizar la conversión");
}catch(ArithmeticException excepcion){
        StdOut.println("¡Error! Ha ocurrido un error");
}
```

No compila. El siguiente catch tiene que ser el último

```
catch(Exception excepcion)
```

El ejemplo anterior arrojará un error en tiempos de compilación ya que no es posible definir otras excepciones después de la excepción genérica **Exception** (se entiende que la excepción ya ha sido capturada).

Uso de la Excepción IllegalArgumentException

Supongamos que se tiene la clase **Alumno** la cual posee los atributos **nombre** y **edad** ¿Qué pasaría si tratáramos de instanciar un objeto de la clase con un nombre nulo y una edad negativa?

En este caso es el **Constructor** de la clase quien debe verificar la correctitud de los parámetros recibidos y realizar las validaciones necesarias para cada tipo de dato, por ejemplo, que el nombre no sea nulo o que la edad se encuentre en un rango válido (entre 0 y 100).

```
public class Alumno {
    private String nombre;
    private int edad;

    public Alumno(String nombre, int edad) {

        if( nombre == null || nombre.length() <=0){
            throws new IllegalArgumentException(“El nombre del
                                                alumno no puede ser nulo”);
        }

        if( edad < 0 | | edad > 100){
            throws new IllegalArgumentException(“La edad del alumno
                                                se encuentra fuera de rango”);
        }
        this.nombre = nombre;
        this.edad = edad;
    }
}
```

La sentencia **throws new IllegalArgumentException** lanzará una excepción que será capturada en el bloque **catch** definido para ese error.

```
public class App {
	public static void main(String[] args) {
		try{
			Alumno alumno = new Alumno(null, 17);
		}catch(IllegaArgumentException ex){
			StdOut.println(ex.getMessage());
		}
	}
}
```

En el ejemplo anterior se intentó instanciar un objeto **alumno** con un nombre nulo, en el constructor de la clase se gatillará la excepción correspondiente al atributo “nombre” (previa verificación). El mensaje “lanzado” por la excepción es capturado por el objeto **ex** que se ha definido dentro del bloque **catch**. De este modo, mediante el método “getMessage()” es posible obtener el mensaje gatillado por el error; El nombre del alumno no puede ser nulo.

En los talleres se tiene que manejar excepciones