**Ciclo de vida de Software y sus componentes para certificación ISO 29110 – 4-1**



**Ingeniería de Software. Perfiles del Ciclo de Vida para pequeñas entidades (PES). Parte 4-1: Arquitectura y Desarrollo de Software**

**Bogotá 21 de Febrero de 2017.**

Contenido

[Diseño de la Arquitectura 2](#_Toc516718221)

[Definición de la Arquitectura 2](#_Toc516718222)

[Documento de Arquitectura Vistas 4+1 3](#_Toc516718223)

[Documento de decisiones de Arquitectura 4](#_Toc516718224)

[Documento de Integraciones 4](#_Toc516718225)

[Desarrollo de Software 4](#_Toc516718226)

[Plataforma de desarrollo Java 4](#_Toc516718227)

[Plataforma de desarrollo .Net 23](#_Toc516718228)

[Control y seguimiento de Incidentes 33](#_Toc516718229)

[Gestión de la Configuración 35](#_Toc516718230)

[Tipo de proyecto y Definición de los artefactos por tipo de proyecto 35](#_Toc516718231)

[Roles 36](#_Toc516718232)

[Definición del plan de pruebas 37](#_Toc516718233)

[Pruebas funcionales 37](#_Toc516718234)

[Pruebas Integrales 37](#_Toc516718235)

[Pruebas de carga 37](#_Toc516718236)

[Pruebas de estrés 37](#_Toc516718237)

[Pruebas de seguridad 37](#_Toc516718238)

# Diseño de la Arquitectura

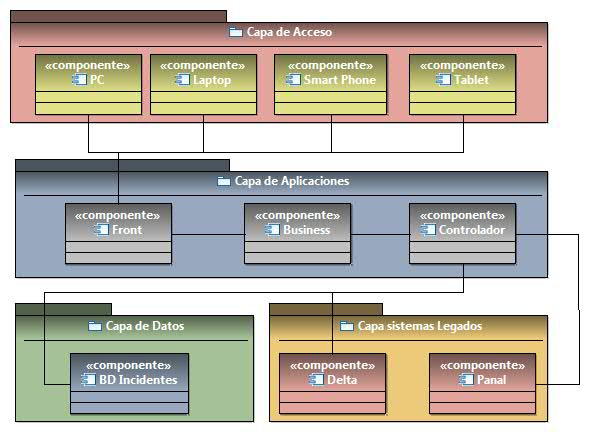
## Definición de la Arquitectura

En Holística Organizacional la Arquitectura de Software está basada para descomponer el alcance de un proyecto en el concepto Funcional y no Funcional.

Dado esta premisa, la Arquitectura de Software está basada en Patrones de:

1. Descomposición por capas y componentes
2. Atributos de Calidad:
   1. Seguridad:
      1. Autenticación
      2. Autorización
      3. No Repudio
      4. Confiabilidad
      5. Integridad
      6. Disponibilidad
   2. Desempeño
   3. Disponibilidad
   4. Usabilidad
   5. Escalabilidad
   6. Modificabilidad

A continuación mostramos gráficamente un ejemplo de la Arquitectura de Software



En la imagen anterior vemos como se descompone el negocio por capas y dentro de cada una de ellas, se tienen los diferentes componentes que a su vez tienen responsabilidades claras dentro de la arquitectura.

## Documento de Arquitectura Vistas 4+1

El documento de Vistas 4+1, está basado en las prácticas de Phillip Krutchen, y plantea la desagregación de los sistemas en las siguientes vistas:

1. Vista Lógica
2. Vista de Desarrollo
3. Vista Física de Despliegue
4. Vista de Datos
5. Escenarios

A continuación adjuntamos dos documentos:

1. Estructura virgen de la plantilla
2. Documento de Arquitectura de Sabaneta



## Documento de decisiones de Arquitectura

El documento de decisiones de Arquitectura, es un Artefacto que define el entorno del porqué se toman las decisiones sobre los artefactos de Software. Ayuda a determinar las alternativas para la toma de decisiones arquitectónicas, así como las restricciones de negocio y de tecnología.

A continuación se adjunta plantilla virgen:

## Documento de Integraciones

El documento de integraciones describe el comportamiento de los sistemas a través de diagramas de secuencia para determinar tanto los parámetros de entrada como de salida de las integraciones entre los diferentes sistemas que componen la solución:



# Desarrollo de Software

Tal como se especificó, en el aparte de tecnologías permitidas para el desarrollo de software, por lo cual se especificará la forma como se debe desarrollar bajo plataforma Java y .Net:

## Plataforma de desarrollo Java

**Introducción**  
  
El objeto del presente documento es el establecimiento de los estándares o convenciones de programación empleados en el desarrollo de software sobre la plataforma Java. Este modelo de programación está basado en los estándares recomendados por Sun Microsystems, que han sido difundidos y aceptados ampliamente por toda la comunidad Java, y que han terminado por consolidarse como un modelo estándar de programación de facto.  
  
Estas normas son muy útiles por muchas razones, entre las que destacan:

* Facilitan el mantenimiento de una aplicación. Dicho mantenimiento constituye el 80% del coste del ciclo de vida de la aplicación.
* Permite que cualquier programador entienda y pueda mantener la aplicación. En muy raras ocasiones una misma aplicación es mantenida por su autor original.
* Los estándares de programación mejoran la legibilidad del código, al mismo tiempo que permiten su compresión rápida.

**Organización de ficheros**

Las clases en Java se agrupan en paquetes. Estos paquetes se deben organizar de manera jerárquica, de forma que todo código desarrollado para el Ayuntamiento de Málaga tendrá que estar incluido dentro del paquete "eu.malaga".

Dentro del paquete principal las clases se organizarán en subpaquetes en función del área, organismo o sección del Ayuntamiento al que pertenezca el código desarrollado. Por ejemplo, si estamos desarrollando un servicio web de inscripción a un curso de programación Java del IMFE las clases de dicho servicio se incluirían en el paquete "eu.malaga.imfe.webservices.cursojava" o similar.  
  
Un fichero consta de secciones que deben estar separadas por líneas en blanco y comentarios opcionales que identifiquen cada sección.  
  
Deben evitarse los ficheros de gran tamaño que contengan más de 1000 líneas. En ocasiones, este tamaño excesivo provoca que la clase no encapsule un comportamiento claramente definido, albergando una gran cantidad de métodos que realizan tareas funcional o conceptualmente heterogéneas.  
  
**Fichero fuente Java (.java)**

Cada fichero fuente Java debe contener una única clase o interfaz pública. El nombre del fichero tiene que coincidir con el nombre de la clase. Cuando existan varias clases privadas asociadas funcionalmente a una clase pública, podrán colocarse en el mismo fichero fuente que la clase pública. La clase pública debe estar situada en primer lugar dentro del fichero fuente.  
En todo fichero fuente Java distinguimos las siguientes secciones:

* Comentarios de inicio.
* Sentencia de paquete.
* Sentencias de importación.
* Declaraciones de clases e interfaces.

**Comentarios de inicio**

Todo fichero fuente debe comenzar con un comentario que incluya el nombre de la clase, información sobre la versión del código, la fecha y el copyright. El copyright indica la propiedad legal del código, el ámbito de distribución, el uso para el que fue desarrollado y su modificación.   
  
Dentro de estos comentarios iniciales podrían incluirse adicionalmente comentarios sobre los cambios efectuados sobre dicho fichero (mejora, incidencia, error, etc.). Estos comentarios son opcionales si los ficheros están bajo un sistema de control de versiones bien documentado, en caso contrario se recomienda su uso. Estos comentarios constituyen el historial de cambios del fichero. Este historial es único para cada fichero y permitirá conocer rápidamente el estado y la evolución que ha tenido el fichero desde su origen.  
  
A continuación se muestra un comentario de inicio para la clase "JceSecurity.java".

/\*  
 \* @(#)JceSecurity.java 1.50 04/04/14  
 \*   
 \* Copyright 2004 Sun Microsystems, Inc. All rights reserved.  
 \* SUN PROPRIETARY/CONFIDENTIAL. Use is subject to license terms.  
 \*/  
  
/\*\*  
 \* This class instantiates implementations of JCE engine classes from  
 \* providers registered with the java.security.Security object.  
 \*  
 \* @author Jan Luehe  
 \* @author Sharon Liu  
 \* @version 1.50, 04/14/04  
 \* @since 1.4  
 \*/

**Sentencias de paquete**

La primera línea no comentada de un fichero fuente debe ser la sentencia de paquete, que indica el paquete al que pertenece(n) la(s) clase(s) incluída(s) en el fichero fuente. Por ejemplo,

package javax.crypto;

**Sentencias de importación**

Tras la declaración del paquete se incluirán las sentencias de importación de los paquetes necesarios. Esta importación de paquetes obligatorios seguirá el siguiente orden:

* Paquetes del JDK de java.
* Paquetes de utilidades no pertenecientes al JDK de Java, de frameworks de desarrollo o de proyectos opensource tales como apache, hibernate, springframework, etc.
* Paquetes desarrollados para el Ayuntamiento de Málaga.
* Paquetes de la aplicación.

Se recomienda minimizar en la medida de lo posible el uso de importaciones del tipo "package.\*", pues dificultan la comprensión de las dependencias existentes entre las clases utilizadas por la aplicación. En caso contrario, se recomienda utilizar comentarios de línea tras la importación.

import java.io.\*; // BufferedReader, PrintWriter, FileInputStream, File  
import java.util.ArrayList;  
  
import org.apache.log4j.Logger;  
import org.apache.lucene.analysis.Analyzer;  
import es.provincia.organismo.corporativas.atlas.vo.AgendaVO;  
import es.provincia.organismo.atlas.vo.AnuncioVO;  
import es.provincia.organismo.atlas.vo.OrganigramaVO;

**Declaraciones de clases e interfaces**

La siguiente tabla describe los elementos que componen la declaración de una clase o interfaz, así como el orden en el que deben estar situados.

|  |  |
| --- | --- |
| Elementos de declaración de una clase / interfaz | Descripción |
| Comentario de documentación de la clase/interfaz /\*\* ... \*/ | Permite describir la clase/interfaz desarrollada. Necesario para generar la documentación de la api mediante javadoc. |
| Sentencia class / interface |  |
| Comentario de implementación de la clase/interfaz, si es necesario /\* ... \*/ | Este comentario incluye cualquier información que no pueda incluirse en el comentario de documentación de la clase/interfaz. |
| Variables de clase (estáticas) | En primer lugar las variables de clase públicas (public), después las protegidas (protected), posteriormente las de nivel de paquete (sin modificador), y por último las privadas (private). |
| Variables de instancia | Primero las públicas (public), después las protegidas (protected), luego las de nivel de paquete (sin modificador), y finalmente las privadas (private). |
| Constructores |  |
| Métodos | Deben agruparse por funcionalidad en lugar de agruparse por ámbito o accesibilidad. Por ejemplo, un método privado puede estar situado entre dos métodos públicos. El objetivo es desarrollar código fácil de leer y comprender. |

**Sangría**

Como norma general se establecen 4 caracteres como unidad de sangría. Los entornos de desarrollo integrado (IDE) más populares, tales como Eclipse o NetBeans, incluyen facilidades para formatear código Java.

**Longitud de línea**

La longitud de línea no debe superar los 80 caracteres por motivos de visualización e impresión.  
  
 **División de líneas**

Cuando una expresión ocupe más de una línea, esta se podrá romper o dividir en función de los siguientes criterios,

* Tras una coma.
* Antes de un operador.
* Se recomienda las rupturas de nivel superior a las de nivel inferior.
* Alinear la nueva línea con el inicio de la expresión al mismo nivel que la línea anterior.
* Si las reglas anteriores generan código poco comprensible, entonces estableceremos tabulaciones de 8 espacios.

Ejemplos:

unMetodo(expresionLarga1, expresionLarga 2, expresionLarga 3,   
 expresionLarga 4, expresionLarga 5);  
  
if ((condicion1 && condicion2)  
 || (condicion3 && condicion4)  
 ||!(condicion5 && condicion6)) {  
 unMetodo();  
}

**Comentarios**  
Distinguimos dos tipos de comentarios: los comentarios de implementación y los de documentación.   
  
**Comentarios de implementación**

Estos comentarios se utilizan para describir el código ("el cómo"), y en ellos se incluye información relacionada con la implementación, tales como descripción de la función de variables locales, fases lógicas de ejecución de un método, captura de excepciones, etc.  
  
Distinguimos tres tipos de comentarios de implementación:

* Comentarios de bloque: Permiten la descripción de ficheros, clases, bloques, estructuras de datos y algoritmos.

/\*  
 \* Esto es un comentario  
 \* de bloque  
 \*/

* Comentarios de línea:

Son comentarios cortos localizados en una sola línea y tabulados al mismo nivel que el código que describen. Si ocupa más de una línea se utilizará un comentario de bloque. Deben estar precedidos por una línea en blanco.

/\* Esto es un comentario de línea \*/  
  
// Esto es otro comentario de línea

* Comentario a final de línea  
    
  Comentario situado al final de una sentencia de código y en la misma línea.

int contador = 4 + 10; // Inicialización del contador  
contador++; /\* Incrementamos el contador \*/

**Comentarios de documentación**

Los comentarios de documentación, también denominados "comentarios javadoc", se utilizan para describir la especificación del código, desde un punto de vista independiente de la implementación, de forma que pueda ser consultada por desarrolladores que probablemente no tengan acceso al código fuente.

El apartado 2 de este documento describe el uso de comentarios de documentación.  
  
**Declaraciones**  
  
**Una declaración por línea**

Se recomienda el uso de una declaración por línea, promoviendo así el uso de comentarios. Ejemplo,

int idUnidad; // Identificador de la unidad organizativa  
String[] funciones; // Funciones de la unidad

**Inicialización**  
  
Toda variable local tendrá que ser inicializada en el momento de su declaración, salvo que su valor inicial dependa de algún valor que tenga que ser calculado previamente.  
  
int idUnidad = 1;  
String[] funciones = { "Administración", "Intervención", "Gestión" };

**Localización**  
  
Las declaraciones deben situarse al principio de cada bloque principal en el que se utilicen, y nunca en el momento de su uso.

public void unMetodo() {  
 int contador = 0; // inicio del método  
  
 ...  
}

La única excepción a esta regla son los índices de los bucles "for", ya que, en Java, pueden incluirse dentro de la propia sentencia "for".

for (int i=0; contador<10; i++) {  
 ...  
}

Se debe evitar el uso de declaraciones que oculten a otras declaraciones de ámbito superior.

int contador = 0; // Inicio del método  
  
public void unMetodo() {  
   
 if (condicion) {  
 int contador = 2; // ¡¡ EVITAR !!  
 ...  
 }  
 ...  
}

**Declaración de clases / interfaces**

Durante el desarrollo de clases / interfaces se deben seguir las siguientes reglas de formateo:

* No incluir ningún espacio entre el nombre del método y el paréntesis inicial del listado de parámetros.
* El carácter inicio de bloque ("{") debe aparecer al final de la línea que contiene la sentencia de declaración.
* El carácter fin de bloque ("}") se sitúa en una nueva línea tabulada al mismo nivel que su correspondiente sentencia de inicio de bloque, excepto cuando la sentencia sea nula, en tal caso se situará detrás de "{".
* Los métodos se separarán entre sí mediante una línea en blanco.

public classe ClaseEjemplo extends Object {  
  
 int variable1;  
 int variable2;  
   
 public ClaseEjemplo() {  
 variable1 = 0;  
 variable2 = 1;  
 }  
 ...  
}

**Sentencias**  
  
Cada línea debe contener como máximo una sentencia. Ejemplo,

int contador++;  
int variable--;

Las sentencias pertenecientes a un bloque de código estarán tabuladas un nivel más a la derecha con respecto a la sentencia que las contiene.

El carácter inicio de bloque "{" debe situarse al final de la línea que inicia el bloque. El carácter final de bloque "}" debe situarse en una nueva línea tras la última línea del bloque y alineada con respecto al primer carácter de dicho bloque.

Todas la sentencias de un bloque deben encerrarse entre llaves "{ ... }", aunque el bloque conste de una única sentencia. Esta práctica permite añadir código sin cometer errores accidentalmente al olvidar añadir las llaves. Ejemplo,

if (condicion) {  
 variable++;  
}

La sentencia "try/catch" siempre debe tener el formato siguiente,

try {  
 sentencias;  
} catch (ClaseException e) {  
 sentencias;  
}

En el bloque "catch" siempre se imprimirá una traza de error indicando el tipo de excepción generada y posteriormente se elevará dicha excepción al código invocante, salvo que la lógica de ejecución de la aplicación no lo requiera.

Siempre se utilizará el bloque "finally" para liberar recursos y para imprimir trazas de monitorización de fin de ejecución.

try {  
 sentencias;  
} catch (ClaseException e) {  
 sentencias;  
} finally {  
 sentencias;  
}

**Espacios en blanco**

Las líneas y espacios en blanco mejoran la legibilidad del código permitiendo identificar las secciones de código relacionadas lógicamente.

Se utilizarán espacios en blanco en los siguientes casos:

Entre una palabra clave y un paréntesis. Esto permite que se distingan las llamadas a métodos de las palabras clave. Por ejemplo:

while (true) {  
 ...  
}

Tras cada coma en un listado de argumentos. Por ejemplo:

objeto.unMetodo(a, b, c);

Para separar un operador binario de sus operandos, excepto en el caso del operador ("."). Nunca se utilizarán espacios entre los operadores unarios (p.e., "++" o "--") y sus operandos. Por ejemplo:

a += b + c;  
  
a = (a + b) / (c + d);  
  
contador++;

Para separar las expresiones incluidas en la sentencia "for". Por ejemplo:

for (expresion1; expresion2; expresion3)

Al realizar el moldeo o "casting" de clases. Ejemplo:

Unidad unidad = (Unidad) objeto;

**Nomenclatura de identificadores**

Las convenciones de nombres de identificadores permiten que los programas sean más fáciles de leer y por tanto más comprensibles. También proporcionan información sobre la función que desempeña el identificador dentro del código, es decir, si es una constante, una variable, una clase o un paquete, entre otros.

**Paquetes**

Se escribirán siempre en letras minúsculas para evitar que entren en conflicto con los nombres de clases o interfaces. El prefijo del paquete siempre corresponderá a un nombre de dominio de primer nivel, tal como: es, eu, org, com, net, etc.

El resto de componentes del paquete se nombrarán de acuerdo a las normas internas de organización de la empresa: departamento, proyecto, máquina, sección, organismo, área, etc.  
  
Generalmente se suele utilizar el nombre de dominio de Internet en orden inverso. Cuando dicho nombre contenga un carácter "-", este se sustituirá por el carácter "\_".  
  
Ejemplos:  
  
es.provincia.organismo1.festivaldecine  
es.provincia.organismo2.vivienda  
es.provincia.organismo3.juventud  
es.provincia.organismo3.formacion  
es.provincia.organismo3.gestionturistica  
  
java.util.ArrayList  
java.util.Date  
java.util.Properties  
  
javax.servlet.http.HttpServletRequest  
javax.servlet.http.HttpServletResponse

**Clases e interfaces**

Los nombres de clases deben ser sustantivos y deben tener la primera letra en mayúsculas. Si el nombre es compuesto, cada palabra componente deberá comenzar con mayúsculas.  
Los nombres serán simples y descriptivos. Debe evitarse el uso de acrónimos o abreviaturas, salvo en aquellos casos en los que dicha abreviatura sea más utilizada que la palabra que representa (URL, HTTP, etc.).

Las interfaces se nombrarán siguiendo los mismos criterios que los indicados para las clases. Como norma general toda interfaz se nombrará con el prefijo "I" para diferenciarla de la clase que la implementa (que tendrá el mismo nombre sin el prefijo "I").

class Ciudadano  
class OrganigramaDAO  
class AgendaService  
class IAgendaService

**Métodos**

Los métodos deben ser verbos escritos en minúsculas. Cuando el método esté compuesto por varias palabras cada una de ellas tendrá la primera letra en mayúsculas.

public void insertaUnidad(Unidad unidad);  
public void eliminaAgenda(Agenda agenda);  
public void actualizaTramite(Tramite tramite)

**Variables**

Las variables se escribirán siempre en minúsculas. Las variables compuestas tendrán la primera letra de cada palabra componente en mayúsculas.

Las variables nunca podrán comenzar con el carácter "\_" o "$". Los nombres de variables deben ser cortos y sus significados tienen que expresar con suficiente claridad la función que desempeñan en el código. Debe evitarse el uso de nombres de variables con un sólo carácter, excepto para variables temporales.

Unidad unidad;  
Agenda agenda;  
Tramite tramite;

**Constantes**

Todos los nombres de constantes tendrán que escribirse en mayúsculas. Cuando los nombres de constantes sean compuestos las palabras se separarán entre sí mediante el carácter de subrayado "\_".

int LONGITUD\_MAXIMA;  
int LONGITUD\_MINIMA;

**Prácticas de programación**

**Visibilidad de atributos de instancia y de clase**

Los atributos de instancia y de clase serán siempre privados, excepto cuando tengan que ser visibles en subclases herederas, en tales casos serán declarados como protegidos.

El acceso a los atributos de una clase se realizará por medio de los métodos "get" y "set" correspondientes, incluso cuando el acceso a dichos atributos se realice en los métodos miembros de la clase.

public class Unidad {  
  
 private int id;  
 private String nombre;  
 ...  
  
 public void actualizaUnidad(Unidad unidad) {  
 this.setId(unidad.getId());  
 this.setNombre(unidad.getNombre());  
 }  
  
 ...  
}

**Referencias a miembros de una clase**

Evitar el uso de objetos para acceder a los miembros de una clase (atributos y métodos estáticos). Utilizaremos en su lugar el nombre de la clase. Por ejemplo:

metodoUtilidad(); // Acceso desde la propia clase estática  
ClaseUtilidad.metodoUtilidad(); // Acceso común desde cualquier clase

**Constantes**

Los valores constantes (literales) nunca aparecerán directamente en el código. Para designar dichos valores se utilizarán constantes escritas en mayúsculas y se declararán, según su ámbito de uso, o bien en una Clase de constantes creada para tal efecto, o bien en la clase donde sean utilizadas.  
  
// Uso incorrecto  
codigoErrorUsuarioNoEncontrado = 1;  
...  
switch (error) {  
 case codigoErrorUsuarioNoEncontrado:  
 ...  
}  
  
// Uso correcto  
public final int CODIGOERROR\_USUARIONOENCONTRADO = 1;  
...  
switch (error) {  
 case CODIDOGERROR\_USUARIONOENCONTRADO:  
 ...  
}

**Asignación sobre variables**

Se deben evitar las asignaciones de un mismo valor sobre múltiples variables en una misma sentencia, ya que dichas sentencias suelen ser difíciles de leer.

int a = b = c = 2; // Evitar

No utilizar el operador de asignación en aquellos lugares donde sea susceptible de confusión con el operador de igualdad. Por ejemplo:

// INCORRECTO  
if ((c = d++) == 0) { }  
  
// CORRECTO  
c = d++;  
if (c == 0) { }

No utilizar asignaciones embebidas o anidadas. Ejemplo:

c = (c = 3) + 4 + d; // Evitar

Debería escribirse

c = 3;  
c = c + 4 + d;

**Otras prácticas**

* Paréntesis  
    
  Es una buena práctica el uso de paréntesis en expresiones que incluyan distintos tipos de operadores para evitar problemas de precedencia de operadores. Aunque la precedencia de operadores nos pueda parecer clara, debemos asumir que otros programadores no tengan un conocimiento exhaustivo sobre las reglas de precedencia.

if (w == x && y == z) // INCORRECTO  
if ((w == x) && (y == z)) // CORRECTO

* Valores de retorno

Los valores de retorno tendrán que ser simples y comprensibles, de acuerdo al propósito y comportamiento del objeto en el que se utilicen.

// INCORRECTO  
public boolean esProgramador(Empleado emp) {  
  
 if (emp.getRol().equals(ROL\_PROGRAMADOR)) {  
 return true;  
 } else {  
 return false;  
 }  
  
}  
  
// CORRECTO  
public boolean esProgramador(Empleado emp) {  
  
 boolean esUnProgramador = false;  
  
 if (emp.getRol().equals(ROL\_PROGRAMADOR)) {  
 esUnProgramador = true;  
 }  
  
 return esUnProgramador;  
}

* Expresiones en el operador condicional ternario

Toda expresión compuesta, por uno o más operadores binarios, situada en la parte condicional del operador ternario deberán ir entre paréntesis. Ejemplo:

(x >= y) ? x : y;

* Comentarios especiales (TODO, FIXME, XXX)

Utilizaremos XXX para comentar aquella porción de código que, aunque no tenga mal funcionamiento, requiera modificaciones. Usaremos FIXME para señalar un bloque de código erróneo que no funciona. Emplearemos TODO para comentar posibles mejoras de código, como puedan ser las debidas a optimizaciones, actualizaciones o refactorizaciones.

**Documentación: javadoc**

Se aconseja, como buena práctica de programación, incluir en la entrega de la aplicación la documentación de los ficheros fuente de todas las clases. Dicha documentación será generada por la herramienta "javadoc".

La herramienta "javadoc" construirá la documentación a partir de los comentarios (incluidos en las clases) encerrados entre los caracteres "/\*\*" y "\*/". Distinguimos tres tipos de comentarios javadoc, en función del elemento al que preceden: de clase, de variable y de método.

Dentro de los comentarios "javadoc" podremos incluir código html y etiquetas especiales de documentación. Estas etiquetas de documentación comienzan con el símbolo "@", se sitúan al inicio de línea del comentario y nos permiten incluir información específica de nuestra aplicación de una forma estándar.

Como norma general utilizaremos las siguientes etiquetas:

* @author Nombre  
    
  Añade información sobre el autor o autores del código.
* @version InformacionVersion  
    
  Permite incluir información sobre la versión y fecha del código.
* @param NombreParametro Descripción

Inserta el parámetro especificado y su descripción en la sección "Parameters:" de la documentación del método en el que se incluya. Estas etiquetas deben aparecer en el mismo orden en el que aparezcan los parámetros especificados del método. Este tag no puede utilizarse en comentarios de clase, interfaz o campo. Las descripciones deben ser breves.

* @return Descripción

Inserta la descripción indicada en la sección "Returns:" de la documentación del método. Este tag debe aparecer en los comentarios de documentación de todos los métodos, salvo en los constructores y en aquellos que no devuelvan ningún valor (void).

* @throws NombreClase Descripción

Añade el bloque de comentario "Throws:" incluyendo el nombre y la descripción de la excepción especificada. Todo comentario de documentación de un método debe contener un tag "@throws" por cada una de las excepciones que pueda elevar. La descripción de la excepción puede ser tan corta o larga como sea necesario y debe explicar el motivo o motivos que la originan.

* @see Referencia

Permite incluir en la documentación la sección de comentario "See also:", conteniendo la referencia indicada. Puede aparecer en cualquier tipo de comentario "javadoc". Nos permite hacer referencias a la documentación de otras clases o métodos.

* @deprecated Explicación

Esta etiqueta indica que la clase, interfaz, método o campo está obsoleto y que no debe utilizarse, y que dicho elemento posiblemente desaparecerá en futuras versiones. "javadoc" añade el comentario "Deprecated" en la documentación e incluye el texto explicativo indicado tras la etiqueta. Dicho texto debería incluir una sugerencia o referencia sobre la clase o método sustituto del elemento "deprecado".

* @since Version

Se utiliza para especificar cuando se ha añadido a la API la clase, interfaz, método o campo. Debería incluirse el número de versión u otro tipo de información.

El siguiente ejemplo muestra los tres tipos de comentarios "javadoc",

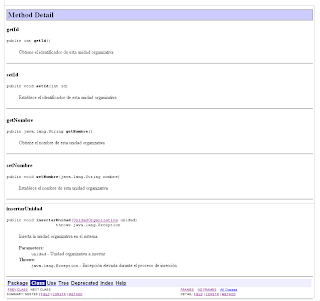
/\*\*  
 \* UnidadOrganizativa.java:  
 \*   
 \* Clase que muestra ejemplos de comentarios de documentación de código.   
 \*   
 \* @author jlflorido  
 \* @version 1.0, 05/08/2008  
 \* @see documento "Normas de programación v1.0"  
 \* @since jdk 5.0   
 \*/  
public class UnidadOrganizativa extends PoolDAO {  
  
 /\*\* Trazas de la aplicación \*/  
 private Logger log = Logger.getLogger(UnidadOrganizativa.class);  
  
 /\*\* Identificador de la unidad organizativa \*/  
 private int id;  
   
 /\*\* Nombre de la unidad organizativa \*/  
 private String nombre;  
  
 /\*\* Obtiene el identificador de esta unidad organizativa \*/  
 public int getId() {  
 return id;  
 }  
  
 /\*\* Establece el identificador de esta unidad organizativa \*/  
 public void setId(int id) {  
 this.id = id;  
 }  
   
 /\*\* Obtiene el nombre de esta unidad organizativa \*/  
 public String getNombre() {  
 return nombre;  
 }  
  
 /\*\* Establece el nombre de esta unidad organizativa \*/  
 public void setNombre(String nombre) {  
 this.nombre = nombre;  
 }  
   
 /\*\*  
 \* Inserta la unidad organizativa en el sistema.  
 \*   
 \* @param unidad Unidad organizativa a insertar  
 \* @throws Exception Excepción elevada durante el proceso de inserción  
 \*/  
 public void insertarUnidad(UnidadOrganizativa unidad) throws Exception{  
   
 log.debug("-> insertarUnidad(UnidadOrganizativa unidad)");  
   
 Connection conn = null;  
 PreparedStatement pstmt = null;  
 StringBuffer sqlSb = null;  
   
 try {  
 conn = this.dameConexion();  
   
 sqlSb = new StringBuffer("")  
 .append("INSERT INTO ORG.UNIDAD\_ORGANIZATIVA ")  
 .append("(ID, NOMBRE) VALUES (?, ?)");  
   
 pstmt = conn.prepareStatement(sqlSb.toString());  
 pstmt.setInt(1, unidad.getId());  
 pstmt.setString(2, unidad.getNombre());  
 pstmt.executeUpdate();  
   
 } catch (Exception e) {  
  
 log.error("Error: error al insertar la unidad. " +  
 "Descripción:" + e.getMessage(), e);  
  
 throw e;  
  
 } finally {  
  
 log.debug("<- insertarUnidad(UnidadOrganizativa unidad)");  
  
 }  
 }  
}

La documentación generada por "javadoc" será la siguiente:

1. Página índice de toda la documentación generada:

[](http://3.bp.blogspot.com/_YhLVAlexHp4/TDLM_u-4I_I/AAAAAAAAAB8/8knGNHhDMls/s1600/01.jpg)

b) Documentación de la clase "UnidadOrganizativa.java":

[](http://1.bp.blogspot.com/_YhLVAlexHp4/TDLVZDSI2-I/AAAAAAAAACE/_kBdmBmAX2A/s1600/01.jpg)  
  
[](http://1.bp.blogspot.com/_YhLVAlexHp4/TDLVgokSr0I/AAAAAAAAACM/_wGs5O7D6Bg/s1600/02.jpg)  
  
[](http://2.bp.blogspot.com/_YhLVAlexHp4/TDLVmkbrGtI/AAAAAAAAACU/c4AvM6XfsHs/s1600/03.jpg)  
  
[](http://1.bp.blogspot.com/_YhLVAlexHp4/TDLVr7kpNpI/AAAAAAAAACc/S8sf8txMduI/s1600/05.png)

## Plataforma de desarrollo .Net

Convenciones de nomenclatura

* En ejemplos breves que no incluyen [directivas using](https://docs.microsoft.com/es-es/dotnet/csharp/language-reference/keywords/using-directive), use calificaciones de espacio de nombres.Si sabe que un espacio de nombres se importa en un proyecto de forma predeterminada, no es necesario completar los nombres de ese espacio de nombres. Los nombres completos pueden partirse después de un punto (.) si son demasiado largos para una sola línea, como se muestra en el ejemplo siguiente.

C#Copiar

var currentPerformanceCounterCategory = new System.Diagnostics.

PerformanceCounterCategory();

* No es necesario cambiar los nombres de objetos que se crearon con las herramientas del diseñador de Visual Studio para que se ajusten a otras directrices.

Convenciones de diseño

Un buen diseño utiliza un formato que destaque la estructura del código y haga que el código sea más fácil de leer. Las muestras y ejemplos de Microsoft cumplen las convenciones siguientes:

* Utilice la configuración del Editor de código predeterminada (sangría automática, sangrías de 4 caracteres, tabulaciones guardadas como espacios). Para obtener más información, vea [Opciones, editor de texto, C#, formato](https://docs.microsoft.com/es-es/visualstudio/ide/reference/options-text-editor-csharp-formatting).
* Escriba solo una instrucción por línea.
* Escriba solo una declaración por línea.
* Si a las líneas de continuación no se les aplica sangría automáticamente, hágalo con una tabulación (cuatro espacios).
* Agregue al menos una línea en blanco entre las definiciones de método y las de propiedad.
* Utilice paréntesis para que las cláusulas de una expresión sean evidentes, como se muestra en el código siguiente.

C#Copiar

if ((val1 > val2) && (val1 > val3))

{

// Take appropriate action.

}

Convenciones de los comentarios

* Coloque el comentario en una línea independiente, no al final de una línea de código.
* Comience el texto del comentario con una letra mayúscula.
* Finalice el texto del comentario con un punto.
* Inserte un espacio entre el delimitador de comentario (//) y el texto del comentario, como se muestra en el ejemplo siguiente.

C#Copiar

// The following declaration creates a query. It does not run

// the query.

* No cree bloques con formato de asteriscos alrededor de comentarios.

Convenciones de lenguaje

En las secciones siguientes se describen las prácticas que sigue el equipo C# para preparar las muestras y ejemplos de código.

**String (Tipo de datos)**

* Use [interpolación de cadenas](https://docs.microsoft.com/es-es/dotnet/csharp/language-reference/tokens/interpolated) para concatenar cadenas cortas, como se muestra en el código siguiente.

C#Copiar

string displayName = $"{nameList[n].LastName}, {nameList[n].FirstName}";

* Para anexar cadenas en bucles, especialmente cuando se trabaja con grandes cantidades de texto, utilice un objeto [StringBuilder](https://docs.microsoft.com/es-es/dotnet/api/system.text.stringbuilder).

C#Copiar

var phrase = "lalalalalalalalalalalalalalalalalalalalalalalalalalalalalala";

var manyPhrases = new StringBuilder();

for (var i = 0; i < 10000; i++)

{

manyPhrases.Append(phrase);

}

//Console.WriteLine("tra" + manyPhrases);

Variables locales con asignación implícita de tipos

* Use [tipos implícitos](https://docs.microsoft.com/es-es/dotnet/csharp/programming-guide/classes-and-structs/implicitly-typed-local-variables) para las variables locales cuando el tipo de la variable sea obvio desde el lado derecho de la asignación, o cuando el tipo exacto no sea importante.

C#Copiar

// When the type of a variable is clear from the context, use var

// in the declaration.

var var1 = "This is clearly a string.";

var var2 = 27;

var var3 = Convert.ToInt32(Console.ReadLine());

* No use [var](https://docs.microsoft.com/es-es/dotnet/csharp/language-reference/keywords/var) cuando el tipo no sea evidente desde el lado derecho de la asignación.

C#Copiar

// When the type of a variable is not clear from the context, use an

// explicit type.

int var4 = ExampleClass.ResultSoFar();

* No confíe en el nombre de variable para especificar el tipo de la variable. Puede no ser correcto.

C#Copiar

// Naming the following variable inputInt is misleading.

// It is a string.

var inputInt = Console.ReadLine();

Console.WriteLine(inputInt);

* Evite el uso de var en lugar de [dynamic](https://docs.microsoft.com/es-es/dotnet/csharp/language-reference/keywords/dynamic).
* Use tipos implícitos para determinar el tipo de la variable de bucle en bucles [for](https://docs.microsoft.com/es-es/dotnet/csharp/language-reference/keywords/for) y [foreach](https://docs.microsoft.com/es-es/dotnet/csharp/language-reference/keywords/foreach-in).

En el ejemplo siguiente se usan tipos implícitos en una instrucción for.

C#Copiar

var syllable = "ha";

var laugh = "";

for (var i = 0; i < 10; i++)

{

laugh += syllable;

Console.WriteLine(laugh);

}

En el ejemplo siguiente se usan tipos implícitos en una instrucción foreach.

C#Copiar

foreach (var ch in laugh)

{

if (ch == 'h')

Console.Write("H");

else

Console.Write(ch);

}

Console.WriteLine();

Tipo de datos sin signo

* En general, utilice int en lugar de tipos sin signo. El uso de int es común en todo C#, y es más fácil interactuar con otras bibliotecas cuando se usa int.

Matrices

* Utilice sintaxis concisa para inicializar las matrices en la línea de declaración.

C#Copiar

// Preferred syntax. Note that you cannot use var here instead of string[].

string[] vowels1 = { "a", "e", "i", "o", "u" };

// If you use explicit instantiation, you can use var.

var vowels2 = new string[] { "a", "e", "i", "o", "u" };

// If you specify an array size, you must initialize the elements one at a time.

var vowels3 = new string[5];

vowels3[0] = "a";

vowels3[1] = "e";

// And so on.

Delegados

* Utilice sintaxis concisa para crear instancias de un tipo de delegado.

C#Copiar

// First, in class Program, define the delegate type and a method that

// has a matching signature.

// Define the type.

public delegate void Del(string message);

// Define a method that has a matching signature.

public static void DelMethod(string str)

{

Console.WriteLine("DelMethod argument: {0}", str);

}

C#Copiar

// In the Main method, create an instance of Del.

// Preferred: Create an instance of Del by using condensed syntax.

Del exampleDel2 = DelMethod;

// The following declaration uses the full syntax.

Del exampleDel1 = new Del(DelMethod);

Instrucciones try-catch y using en el control de excepciones

* Use una instrucción [try-catch](https://docs.microsoft.com/es-es/dotnet/csharp/language-reference/keywords/try-catch) en la mayoría de casos de control de excepciones.

C#Copiar

static string GetValueFromArray(string[] array, int index)

{

try

{

return array[index];

}

catch (System.IndexOutOfRangeException ex)

{

Console.WriteLine("Index is out of range: {0}", index);

throw;

}

}

* Simplifique el código mediante la [instrucción using](https://docs.microsoft.com/es-es/dotnet/csharp/language-reference/keywords/using-statement) de C#. Si tiene una instrucción [try-finally](https://docs.microsoft.com/es-es/dotnet/csharp/language-reference/keywords/try-finally) en la que el único código del bloque finally es una llamada al método [Dispose](https://docs.microsoft.com/es-es/dotnet/api/system.idisposable.dispose), use en su lugar una instrucción using.

C#Copiar

// This try-finally statement only calls Dispose in the finally block.

Font font1 = new Font("Arial", 10.0f);

try

{

byte charset = font1.GdiCharSet;

}

finally

{

if (font1 != null)

{

((IDisposable)font1).Dispose();

}

}

// You can do the same thing with a using statement.

using (Font font2 = new Font("Arial", 10.0f))

{

byte charset = font2.GdiCharSet;

}

Operadores && y ||

* Para evitar excepciones y aumentar el rendimiento omitiendo las comparaciones innecesarias, use [&&](https://docs.microsoft.com/es-es/dotnet/csharp/language-reference/operators/conditional-and-operator) en lugar de [&](https://docs.microsoft.com/es-es/dotnet/csharp/language-reference/operators/and-operator) y [||](https://docs.microsoft.com/es-es/dotnet/csharp/language-reference/operators/conditional-or-operator) en lugar de [|](https://docs.microsoft.com/es-es/dotnet/csharp/language-reference/operators/or-operator) cuando realice comparaciones, como se muestra en el ejemplo siguiente.

C#Copiar

Console.Write("Enter a dividend: ");

var dividend = Convert.ToInt32(Console.ReadLine());

Console.Write("Enter a divisor: ");

var divisor = Convert.ToInt32(Console.ReadLine());

// If the divisor is 0, the second clause in the following condition

// causes a run-time error. The && operator short circuits when the

// first expression is false. That is, it does not evaluate the

// second expression. The & operator evaluates both, and causes

// a run-time error when divisor is 0.

if ((divisor != 0) && (dividend / divisor > 0))

{

Console.WriteLine("Quotient: {0}", dividend / divisor);

}

else

{

Console.WriteLine("Attempted division by 0 ends up here.");

}

New (Operador)

* Utilice la forma concisa de la creación de instancias de objeto con tipos implícitos, como se muestra en la siguiente declaración.

C#Copiar

var instance1 = new ExampleClass();

La línea anterior es equivalente a la siguiente declaración.

C#Copiar

ExampleClass instance2 = new ExampleClass();

* Utilice inicializadores de objeto para simplificar la creación de objetos.

C#Copiar

// Object initializer.

var instance3 = new ExampleClass { Name = "Desktop", ID = 37414,

Location = "Redmond", Age = 2.3 };

// Default constructor and assignment statements.

var instance4 = new ExampleClass();

instance4.Name = "Desktop";

instance4.ID = 37414;

instance4.Location = "Redmond";

instance4.Age = 2.3;

Control de eventos

* Si va a definir un controlador de eventos que no es necesario quitar más tarde, utilice una expresión lambda.

C#Copiar

public Form2()

{

// You can use a lambda expression to define an event handler.

this.Click += (s, e) =>

{

MessageBox.Show(

((MouseEventArgs)e).Location.ToString());

};

}

C#Copiar

// Using a lambda expression shortens the following traditional definition.

public Form1()

{

this.Click += new EventHandler(Form1\_Click);

}

void Form1\_Click(object sender, EventArgs e)

{

MessageBox.Show(((MouseEventArgs)e).Location.ToString());

}

Miembros estáticos

* Llame a miembros [estáticos](https://docs.microsoft.com/es-es/dotnet/csharp/language-reference/keywords/static) con el nombre de clase ClassName.StaticMember. Esta práctica hace que el código sea más legible al clarificar el acceso estático. No califique un miembro estático definido en una clase base con el nombre de una clase derivada. Mientras el código se compila, su legibilidad se presta a confusión, y puede interrumpirse en el futuro si se agrega a un miembro estático con el mismo nombre a la clase derivada.

Consultas LINQ

* Utilice nombres descriptivos para las variables de consulta. En el ejemplo siguiente, se utiliza seattleCustomers para los clientes que se encuentran en Seattle.

C#Copiar

var seattleCustomers = from cust in customers

where cust.City == "Seattle"

select cust.Name;

* Utilice alias para asegurarse de que los nombres de propiedad de tipos anónimos se escriben correctamente con mayúscula o minúscula, usando para ello la grafía Pascal.

C#Copiar

var localDistributors =

from customer in customers

join distributor in distributors on customer.City equals distributor.City

select new { Customer = customer, Distributor = distributor };

* Cambie el nombre de las propiedades cuando puedan ser ambiguos en el resultado. Por ejemplo, si la consulta devuelve un nombre de cliente y un identificador de distribuidor, en lugar de dejarlos como Name e ID en el resultado, cambie su nombre para aclarar que Name es el nombre de un cliente e ID es el identificador de un distribuidor.

C#Copiar

var localDistributors2 =

from cust in customers

join dist in distributors on cust.City equals dist.City

select new { CustomerName = cust.Name, DistributorID = dist.ID };

* Utilice tipos implícitos en la declaración de variables de consulta y variables de intervalo.

C#Copiar

var seattleCustomers = from cust in customers

where cust.City == "Seattle"

select cust.Name;

* Alinee las cláusulas de consulta bajo la cláusula [from](https://docs.microsoft.com/es-es/dotnet/csharp/language-reference/keywords/from-clause), como se muestra en los ejemplos anteriores.
* Use cláusulas [where](https://docs.microsoft.com/es-es/dotnet/csharp/language-reference/keywords/where-clause) antes de otras cláusulas de consulta para asegurarse de que las cláusulas de consulta posteriores operan en un conjunto de datos reducido y filtrado.

C#Copiar

var seattleCustomers2 = from cust in customers

where cust.City == "Seattle"

orderby cust.Name

select cust;

* Use varias cláusulas from en lugar de una cláusula [join](https://docs.microsoft.com/es-es/dotnet/csharp/language-reference/keywords/join-clause) para obtener acceso a colecciones internas. Por ejemplo, una colección de objetos Student podría contener cada uno un conjunto de resultados de exámenes. Cuando se ejecuta la siguiente consulta, devuelve cada resultado superior a 90, además del apellido del alumno que recibió la puntuación.

C#Copiar

// Use a compound from to access the inner sequence within each element.

var scoreQuery = from student in students

from score in student.Scores

where score > 90

select new { Last = student.LastName, score };

# Control y seguimiento de Incidentes

Para el control y seguimiento de Incidentes se han diseñado dos plantillas:

1. Plantilla de incidentes
2. Plantilla de trazabilidad End to End

El objetivo de la plantilla de incidentes es llevar la trazabilidad de los errores que se presentan en los ambientes de QA y Producción, definir el responsable y llevar el tiempo de resolución del mismo.

Por otro lado el objetivo de la plantilla de trazabilidad end to end tiene por objeto trazar los diferentes frentes involucrados en el desarrollo de Software:

Issues

Desarrollo

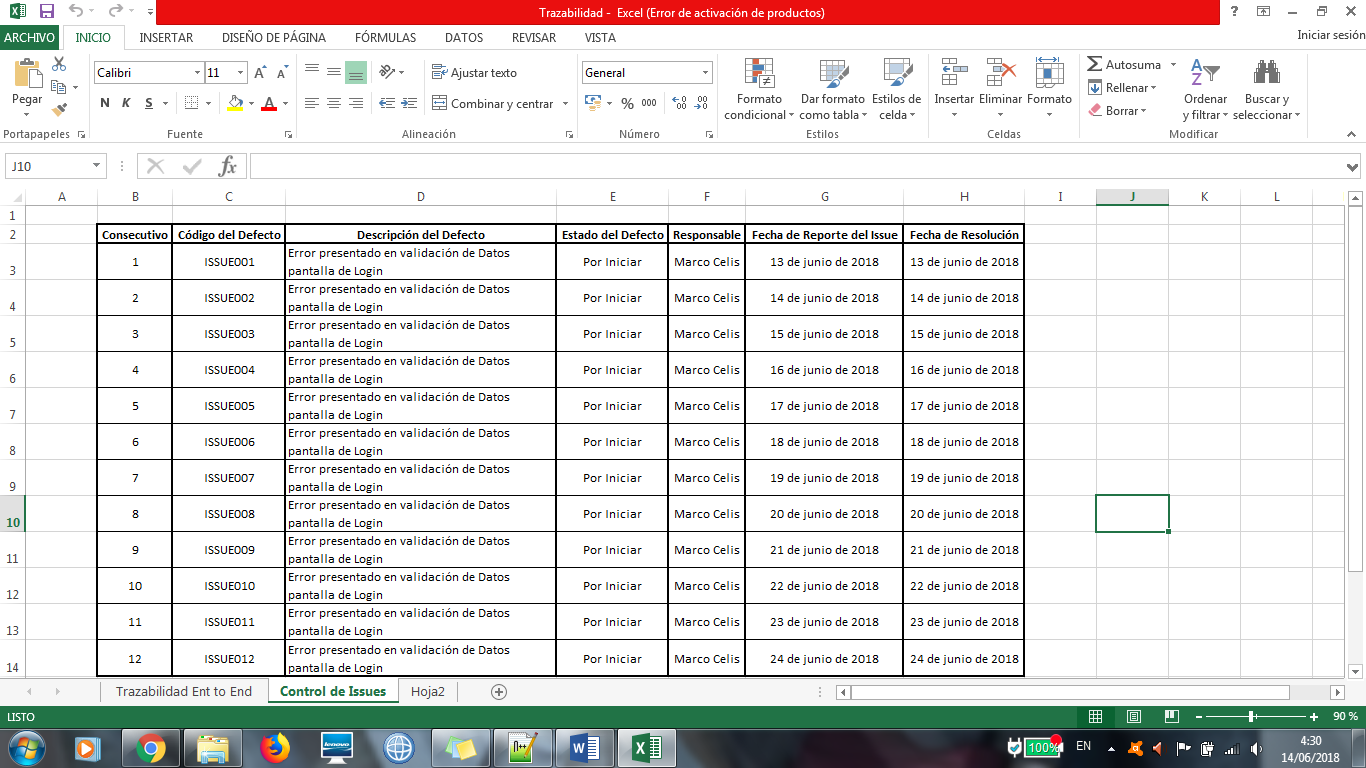
Arquitectura

Requerimientos

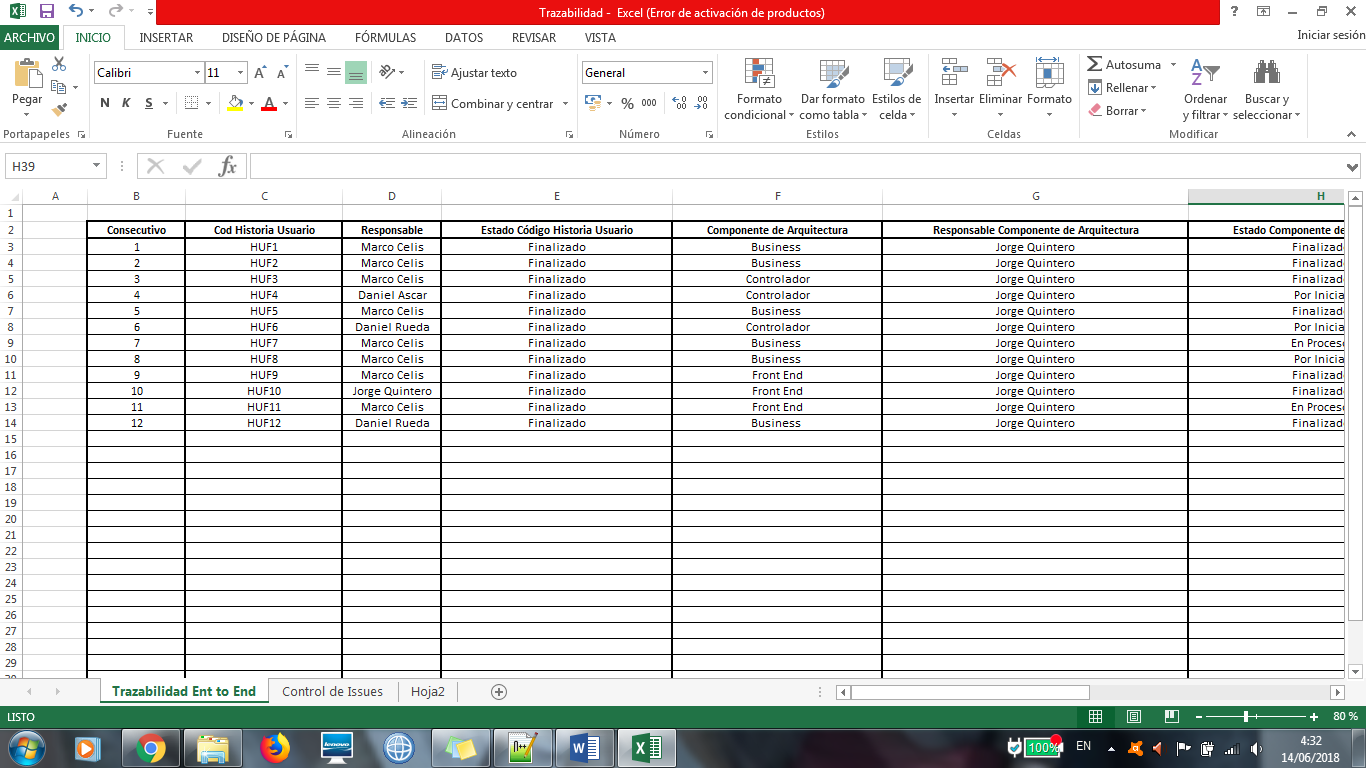
Roles y Responsables

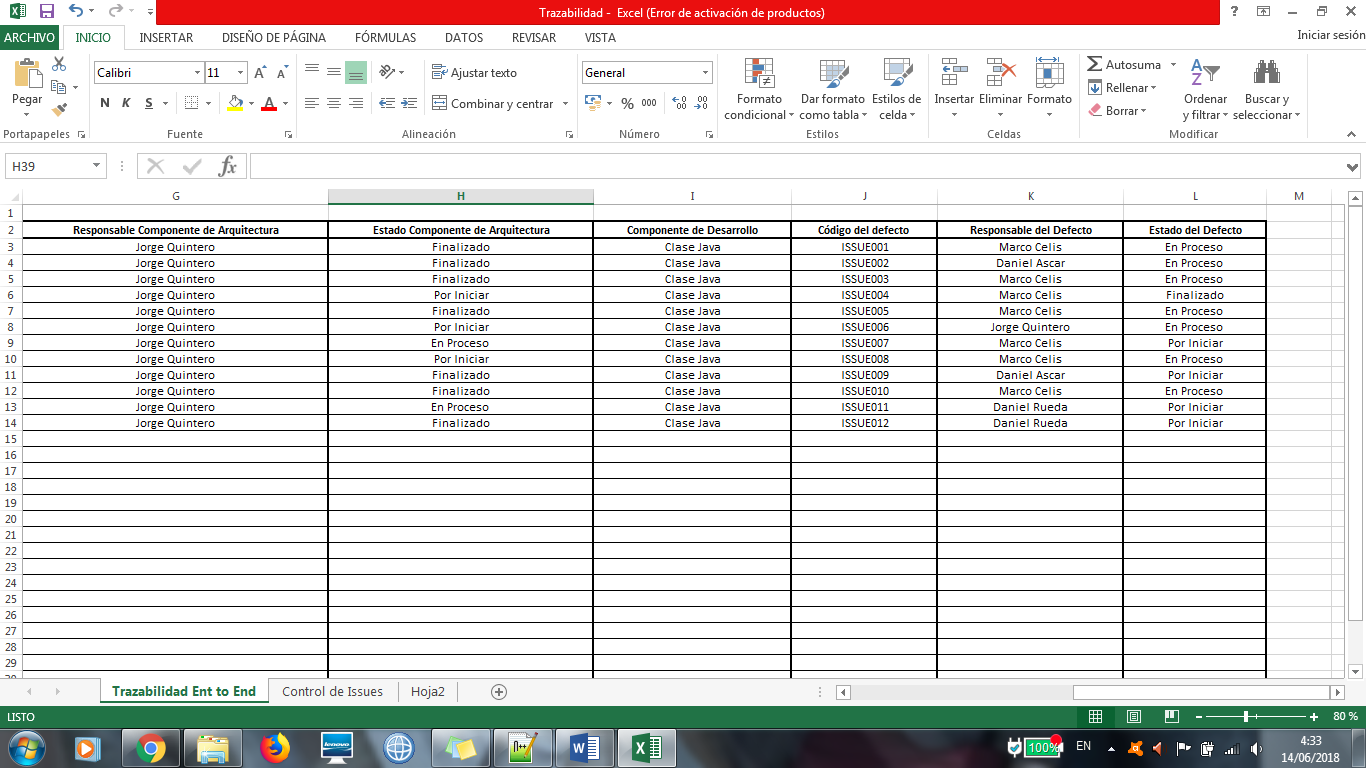
A continuación mostramos ejemplos de cada una de las planillas:

1. Plantilla de incidentes



1. Plantilla end to end:





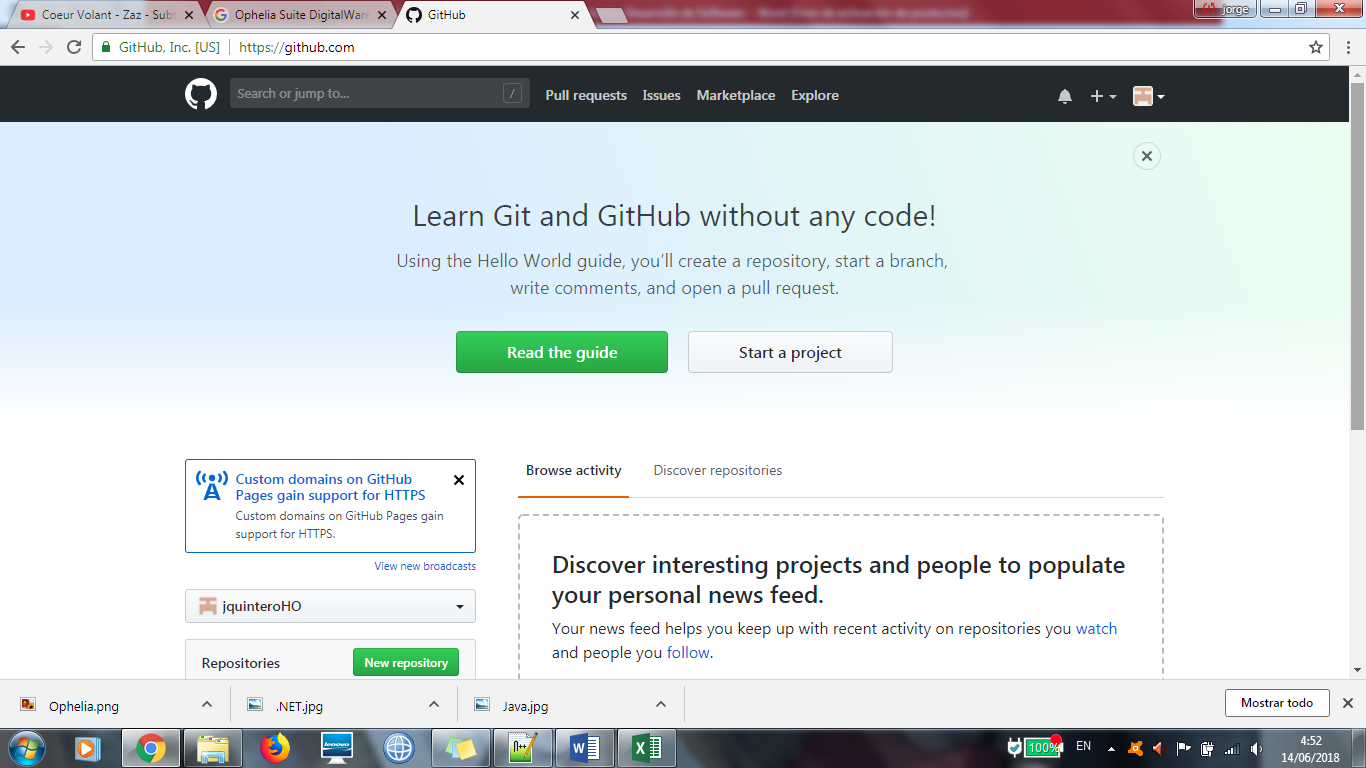
# Gestión de la Configuración

## Tipo de proyecto y Definición de los artefactos por tipo de proyecto

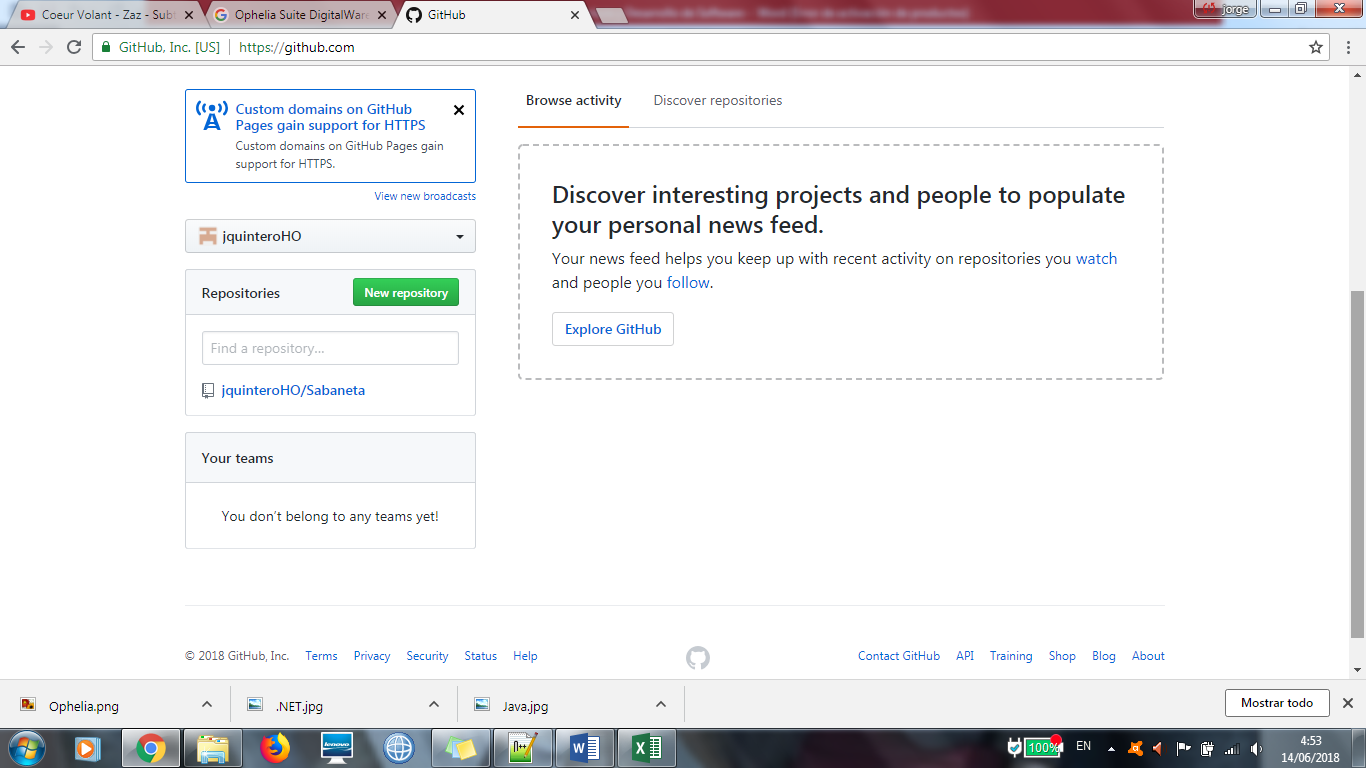
Holística Organizacional dentro de su portafolio de implementación de proyectos de software, maneja 3 tipos de Tecnología:

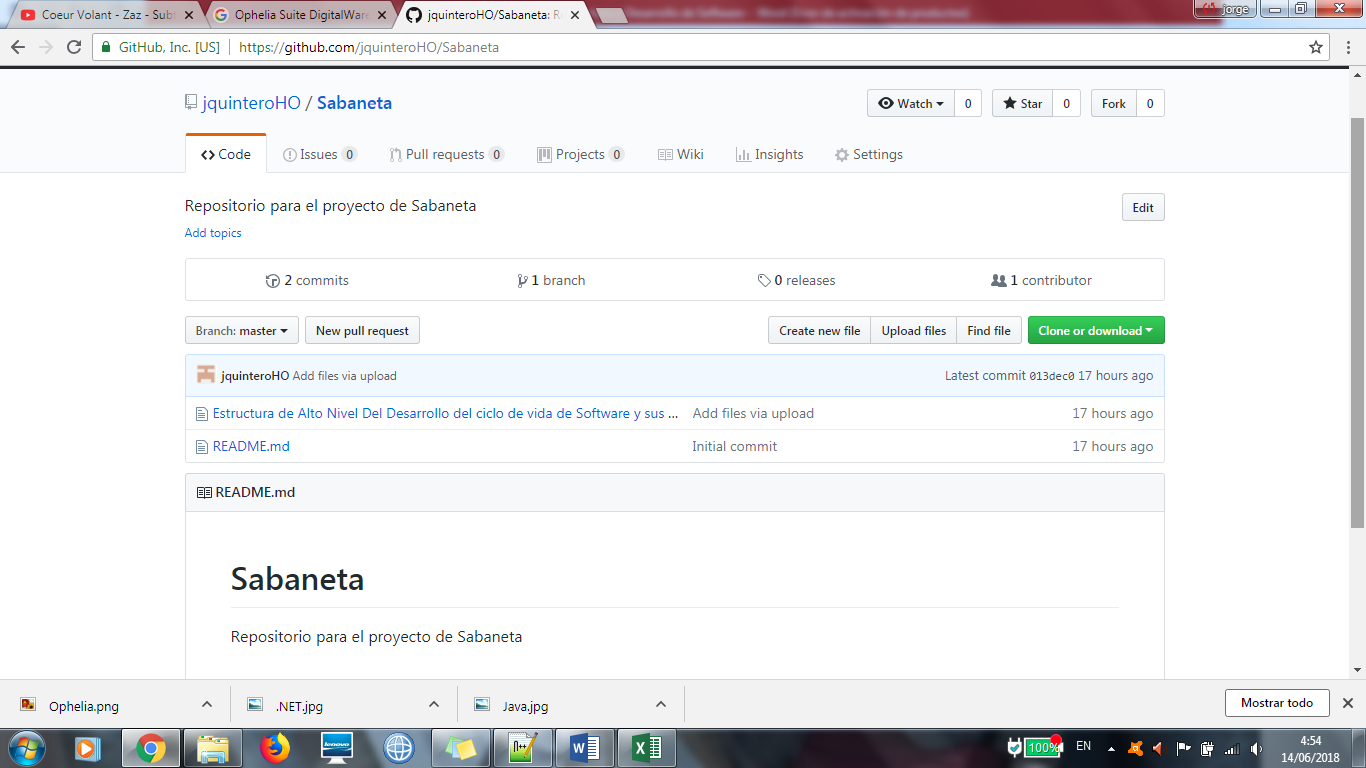


Sin importar el tipo de Proyecto, nuestro repositorio será GitHUB. GitHUB es una plataforma ***Cloud*** la cual nos permite tener repositorios individuales para cada proyecto y tener control de cada una de las versiones que vamos Generando:



En nuestro caso crearemos un proyecto para el proyecto de Sabaneta como ejemplo para este manual:





GitHUB nos pemite tener permisos por usuarios y crear Branches por proyecto. En este caso también aplica el concepto de sin importar el tipo de proyecto de software que se maneja dentro del portafolio de Holística Organizacional, siempre vamos a tener 4 Branches:

1. Desarrollo
2. Quality Assurance
3. Producción
4. Documentación

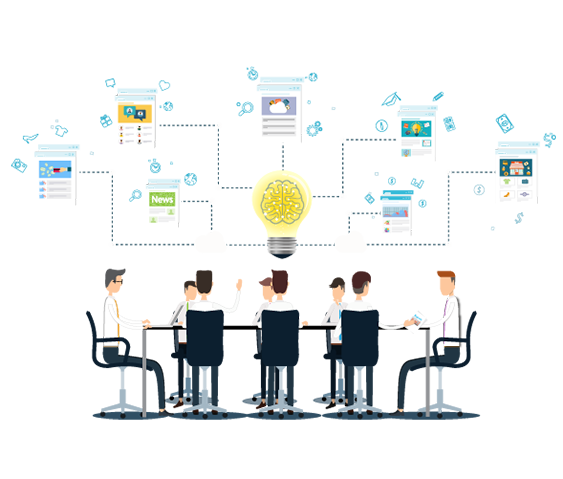
Dentro de cada proyecto, cobijado dentro de cada una de las tecnologías de Holística, tendremos los siguientes artefactos:

1. Java: Workspaces y paquetes.
2. .NET: DLL.
3. Ophelia: Snapshot.

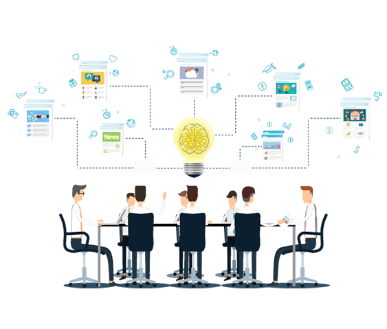
## Roles

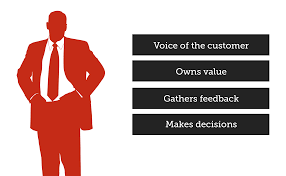
Para la gestión de la configuración, se requiere que intervengan varios roles, por lo cual tendremos:

1. **Equipo de desarrollo:** 
   1. Tendrá autonomía para el desarrollo colaborativo sobre el Branch de desarrollo.
   2. Solamente cuando cumpla las especificaciones de las pruebas podrá promover su desarrollo a ambiente de QA.
   3. El líder técnico deberá revisar el cumplimiento mínimo de la calidad del desarrollo y aprobar el paso a través de Email.



1. **Equipo de QA:**
   1. Tendrá autonomía sobre ambiente de QA para realizar las pruebas establecidas.
   2. Tendrá autonomía para devolver versiones a ambiente de desarrollo, si el desarrollo no cumple con las especificaciones mínimas requeridas para estar en ambiente de QA o si se encuentran fallas en la funcionalidad.
   3. El líder del equipo de QA debe generar una lista de artefactos que pasarán a producción y discutirlo en comité de aprobación con el Product Owner y el Arquitecto de Software.





**Software Architect**

**QA Team**

**Product Owner**

**Development Team**

# Definición del plan de pruebas

## Pruebas funcionales Caja negra y Caja Blanca

Unas de las vías más importantes para determinar el estado de la calidad de un producto de software es el proceso de pruebas. Estas están dirigidas a componentes del sistema en su totalidad, con el objetivo de medir el grado en que cumple con los requerimientos. En ellas se usan casos de prueba, especificados de forma estructurada mediante técnicas. Sus objetivos, métodos y técnicas usadas se describen en el plan de prueba.

La prueba es una actividad fundamental en muchos procesos de desarrollo, incluyendo el del software. Estas permiten detectar la presencia de errores que pudieran generar las entradas o salidas de datos y comportamientos inapropiados durante su ejecución. Un concepto más específico dado por algunos desarrolladores de software es:

***“Cualquier intento de demostrar que el software tiene propiedades por debajo de la calidad requerida”.***

De acuerdo a la IEEE [8] el concepto de prueba se define como:

***“Una actividad en la cual un sistema o componente es ejecutado bajo condiciones específicas, se observan o almacenan los resultados y se realiza una evaluación de algún aspecto del sistema o componente”.***

Teniendo en cuenta las definiciones anteriores se puede concluir que la prueba de software es una actividad en la cual el sistema es ejecutado bajo condiciones específicas para demostrar que no tiene la madurez necesaria para ser implantado. Dentro de las actividades que se practican para obtener un software con la madurez necesaria están:

* **Revisiones:** Consiste en que cada integrante del equipo de desarrollo revisa el producto que va generando.
* **Inspecciones:** Revisión de cada producto por parte de colegas.
* **Validaciones:** Es el cliente quien revisa el producto para decir si cumple con sus necesidades.

Esta definición implica que se considera una prueba exitosa si se demuestran deficiencias en el software. Las fallas pueden ser en el código o en el modelado, en dependencia del tipo de pruebas que se le apliquen al software.

Se distinguen pruebas técnicas y pruebas funcionales. Las pruebas técnicas son la responsabilidad de los ingenieros de software que han desarrollado el producto, pero estos ingenieros en ocasiones deben hacerse cargo de las pruebas funcionales.

En proyectos a gran escala las pruebas funcionales son la responsabilidad de un equipo de pruebas, formado por uno o varios técnicos, un coordinador de pruebas y un gestor de pruebas o de calidad.

### Características generales de la Estrategia de Prueba.

Al aplicarles las pruebas al software se deben seguir un conjunto de estrategias para lograr que estas se hagan en el menor tiempo posible y con la calidad requerida, además de garantizar que arrojen los resultados esperados.

La prueba comienza en el nivel de módulo y trabaja “hacia fuera”, hacia la integración completa del sistema completo.

En diferentes puntos es adecuada la utilización de técnicas de prueba distintas.

La prueba la lleva a cabo el que desarrolla el software y para grandes proyectos, un grupo de prueba independiente.

La prueba y la depuración son actividades diferentes, pero la depuración puede entrar en cualquier estrategia de prueba.

Hay dos estrategias generales para la prueba de software: las estrategias de pruebas de especificación (Caja Negra) y pruebas de código (Caja Blanca).

### Métodos de Pruebas.

Existen diversos métodos para realizar las pruebas de software, entre las más importantes se encuentran la prueba de Caja Blanca, prueba de Caja Negra y prueba de la Estructura de Control.

El uso de la prueba de Caja Blanca es mejor para verificar que se recorran todos los caminos y detectar un mayor número de errores. La Caja Negra brinda la posibilidad de cubrir la mayor parte de las combinaciones de entradas y lograr así un juego de pruebas más eficaz.

Las pruebas mencionadas permiten probar cada una de las condiciones existentes en el programa, identificar claramente las entradas, salidas y estudiar las relaciones que existen entre ellas, permitiendo así maximizar la calidad de las pruebas y en dependencia del resultado se constará con un sistema más estable y confiable.

### Prueba de Especificación (Caja Negra).

**Pruebas de Caja Negra:** También suelen ser llamadas funcionales y basadas en especificaciones. En ellas se pretende examinar el programa en busca de que cuente con las funcionalidades que debe tener y como lleva a cabo las mismas, analizando siempre los resultados que devuelve y probando todas las entradas en sus valores válidos e inválidos.

Al ejecutar las pruebas de Caja Negra se desarrollan casos de prueba reales para cada condición o combinación de condiciones y se analizan los resultados que arroja el sistema para cada uno de los casos. En esta estrategia se verifica el programa considerándolo una caja negra. Las pruebas no se hacen en base al código, sino a la interfaz. No importa que se cubran todas las rutas dentro del programa, lo importante es probar todas las entradas en sus valores válidos e inválidos y lograr que el sistema tenga una interfaz amigable.

### Limitaciones

Lograr una buena cobertura con pruebas de caja negra es un objetivo deseable; pero no suficiente a todos los efectos. Un programa puede pasar con holgura millones de pruebas de especificación y sin embargo tener defectos internos que surgen en el momento más inoportuno.

Por ejemplo, una computadora que contenga el virus Viernes-13 puede estar pasando pruebas de caja negra durante años y años. Sólo falla si es viernes y es día 13; pero ¿a quién se le iba a ocurrir hacer esa prueba?

Las pruebas de caja negra nos convencen de que un programa realizar bien sus funcionalidades programadas, pero no de que haga (además) otras cosas menos aceptables.

### Prueba de Código (Caja Blanca).

**Pruebas de Caja Blanca:** También suelen ser llamadas estructurales o de cobertura lógica. En ellas se pretende investigar sobre la estructura interna del código, exceptuando detalles referidos a datos de entrada o salida, para probar la lógica del programa desde el punto de vista algorítmico. Realizan un seguimiento del código fuente según se va ejecutando los casos de prueba, determinándose de manera concreta las instrucciones, bloques, etc. que han sido ejecutados por los casos de prueba.

En las pruebas de Caja Blanca se desarrollan casos de prueba que produzcan la ejecución de cada posible ruta del programa o módulo, considerándose una ruta como una combinación específica de condiciones manejadas por un programa.

Hay que señalar que no todos los errores de software se pueden descubrir verificando todas las rutas de un programa, hay errores que se descubren al integrar unidades del sistema y pueden existir errores que no tengan relación con el código específicamente.

### Características de las pruebas de Caja Blanca.

En las pruebas de Caja Blanca, se pretende indagar sobre la estructura interna del código, omitiendo detalles referidos a datos de entrada o salida. Su objetivo principal es probar la lógica del programa desde el punto de vista algorítmico.

Estas se basan en el diseño de Casos de Prueba que usa la estructura de control del diseño procedimental para derivarlos. Mediante las pruebas de Caja Blanca el ingeniero de software puede obtener Casos de Prueba que:

1. Garanticen que se ejerciten por lo menos una vez todos los caminos independientes de cada módulo, programa o método.
2. Ejerciten todas las decisiones lógicas en las vertientes verdadera y falsa.
3. Ejecuten todos los bucles en sus límites operacionales.
4. Ejerciten las estructuras internas de datos para asegurar su validez.

Las pruebas de Caja Blanca son consideradas entre las más importantes que se aplican a los sistemas, con la que se obtienen como resultados la disminución en un gran porciento el número de errores existentes en el software y por ende una mayor calidad y confiabilidad en la codificación.

### Tipos de pruebas de Caja Blanca.

* **De estructura de datos locales:**

Se centran en el estudio de las variables del programa. Busca que toda variable esté declarada y que no existan con el mismo nombre, ni declaradas local y globalmente, que haya referencias a todas las variables y para cada variable, analiza su comportamiento en comparaciones.

* **De cobertura lógica**
* **De Cobertura de Sentencias:** Comprueba que todas las sentencias se ejecuten al menos una vez.
* **De Cobertura de Decisión:** Ejecuta casos de prueba de modo que cada decisión se pruebe al menos una vez a Verdadero (True) y otra a Falso (False).
* **De Cobertura de Condición:** Ejecuta un caso de prueba a True y otro a False por cada condición, teniendo en cuenta que una decisión puede estar formada por varias condiciones.
* **De Cobertura de Condición/Decisión:** Se realizan las pruebas de cobertura de condición y las de decisión a la vez.
* **De Condición Múltiple:** Cada decisión multicondición se traduce a una condición simple, aplicando posteriormente la cobertura de decisión.
* **De Cobertura de Caminos:** Se escriben casos de prueba suficientes para que se ejecuten todos los caminos de un programa. Entendiéndose camino como una secuencia de sentencias encadenadas desde la entrada del programa hasta su salida. [11]

### Prueba del Camino Básico.

Buscando una mejor comprensión de los contenidos, se hace importante definir primeramente algunos conceptos fundamentales:

* **Camino:** “Secuencia de todas las instrucciones de un programa de principio a fin”. Un camino se puede definir como la ruta de secuencias que se siguen dentro del código de fuente de un programa, un ejemplo es, desde la entrada de valores al sistema hasta la devolución de resultados que arroja, respetando la estructura de código.
* **Camino Básico:** Es una técnica de prueba de Caja Blanca que permite obtener una medida de complejidad lógica para generar un conjunto básico de caminos que se ejecutan por lo menos una vez durante la ejecución del programa.
* **Camino independiente:** “Es cualquier camino del programa que incluye nuevas instrucciones de un proceso o una nueva condición”.
* **Complejidad:** Es proporcional al número de errores en un segmento de código. “Entre más complejo, más susceptible a errores”. Se relaciona con el esfuerzo requerido para probar. “Entre más complejo, mayor atención para probar”.
* **Complejidad ciclomática:** Es la “medida de la complejidad lógica de un módulo “G” y el esfuerzo mínimo necesario para calificarlo. Es el número de rutas lineales independientes de un módulo “G”, por lo tanto es el número mínimo de rutas que deben probarse”.

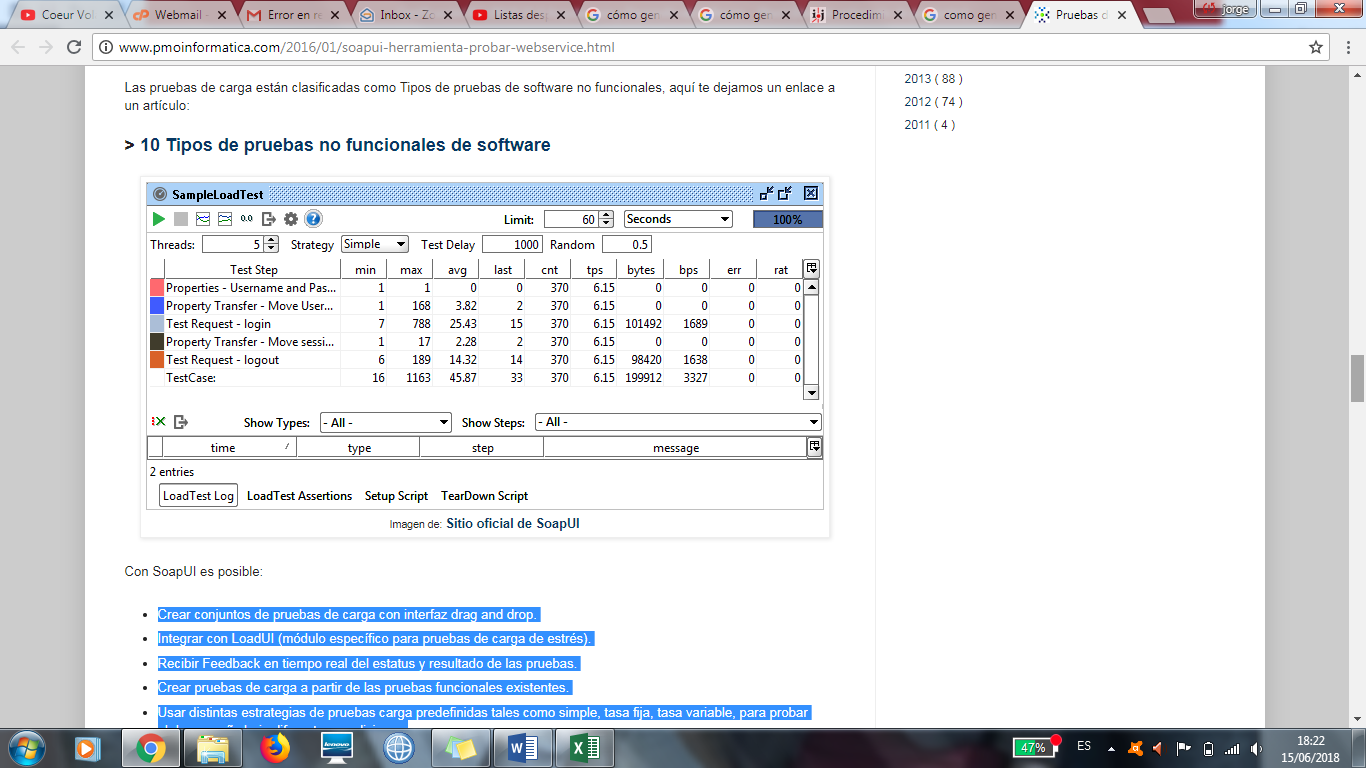
## 

## Pruebas de carga y estrés

Para las pruebas de carga y estrés se utilizará SOAPUI.

SOAPUI es una herramienta que permite el consumo de Web Services de forma independiente y podemos verificar su funcionamiento, pero además nos permite:

* Crear conjuntos de pruebas de carga con interfaz drag and drop.
* Integrar con LoadUI (módulo específico para pruebas de carga de estrés).
* Recibir Feedback en tiempo real del estatus y resultado de las pruebas.
* Crear pruebas de carga a partir de las pruebas funcionales existentes.
* Usar distintas estrategias de pruebas carga predefinidas tales como simple, tasa fija, tasa variable, para probar el desempeño bajo diferentes condiciones.



En la imagen anterior podemos ver cómo podemos lanzar hilos de forma simultánea (Threads) con intervalos de tiempo y analizar los pasos de los componentes que componen la lógica de la respuesta y los tiempos involucrados.

## Pruebas de seguridad

Las pruebas de seguridad son sub contratadas por Holística a una compañía llamada SECPRO.