Formación de Testing en .NET

Nafarroako Gobernua
Gobierno de Navarra

hiberus©

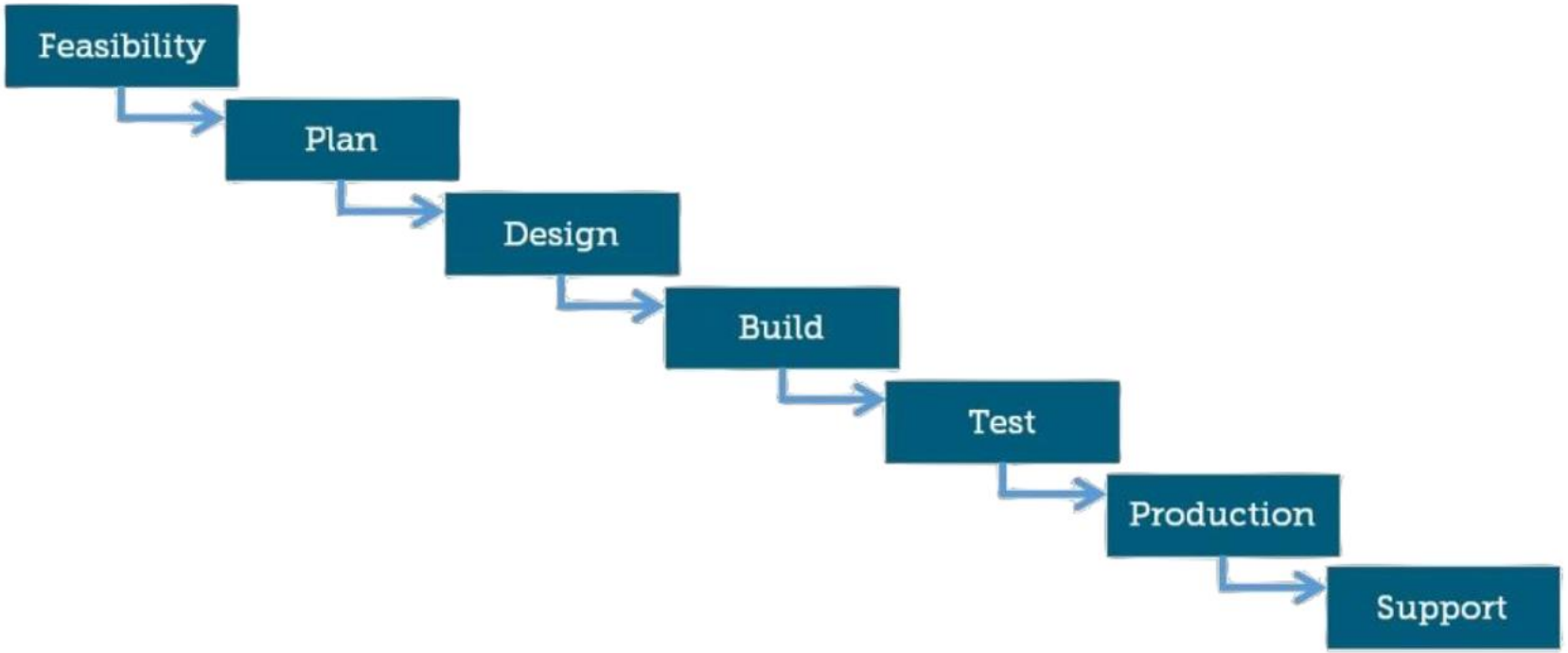
La compañía hiperespecializada en las TIC

Fundamentos de Testing

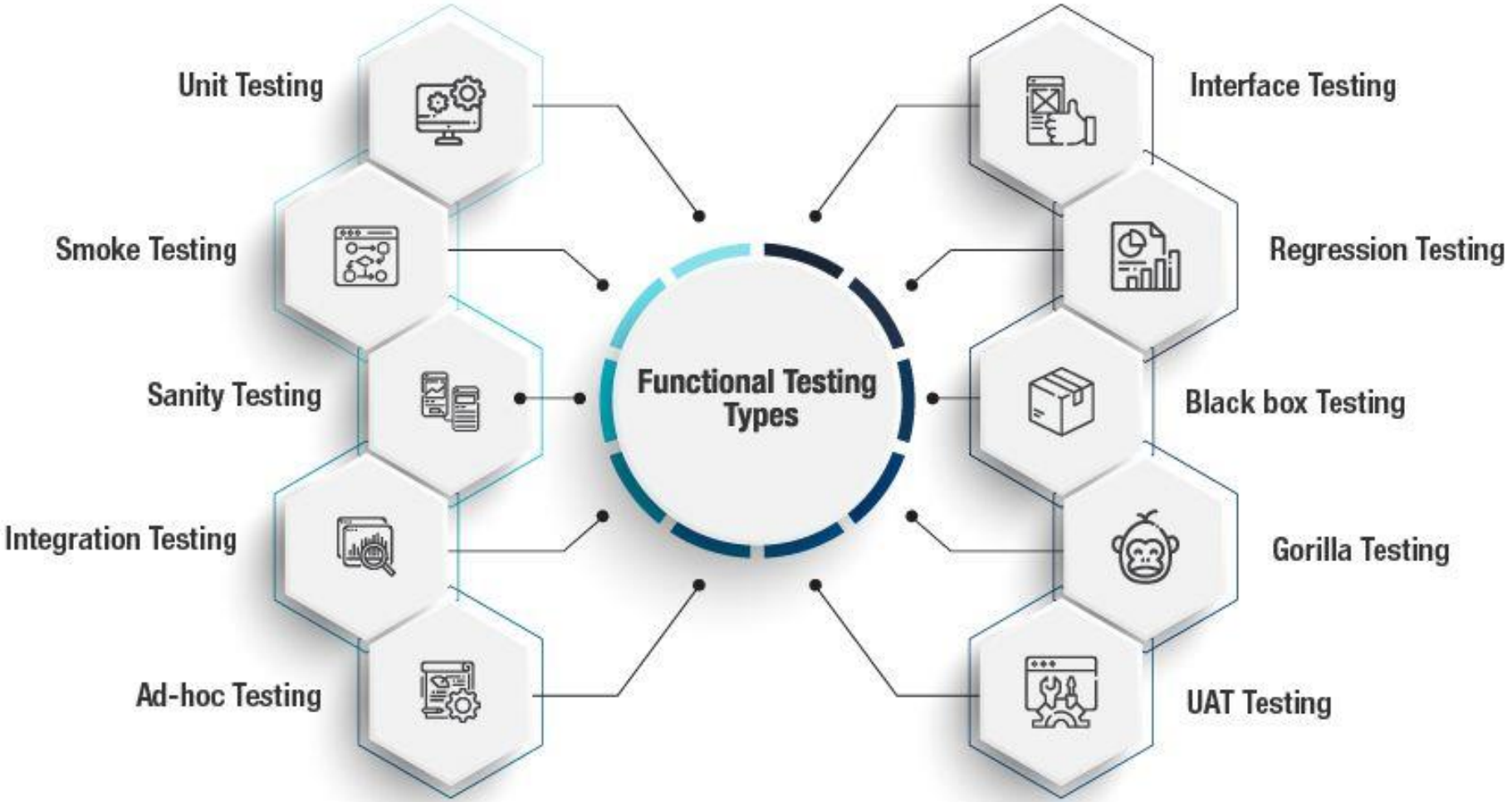
Nafarroako Gobernua
Gobierno de Navarra

www.hiberus.com

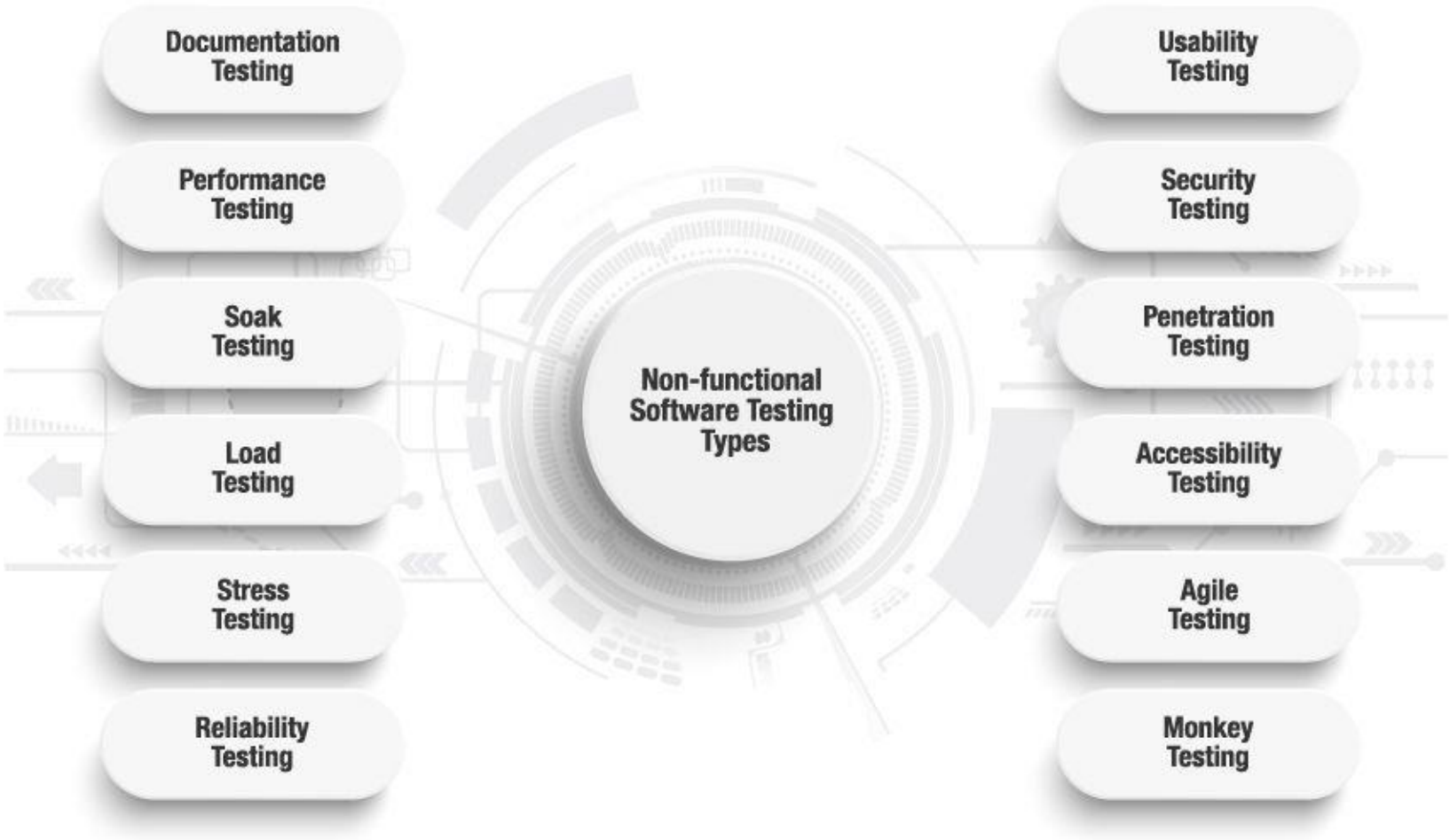
Ciclo de vida del Software



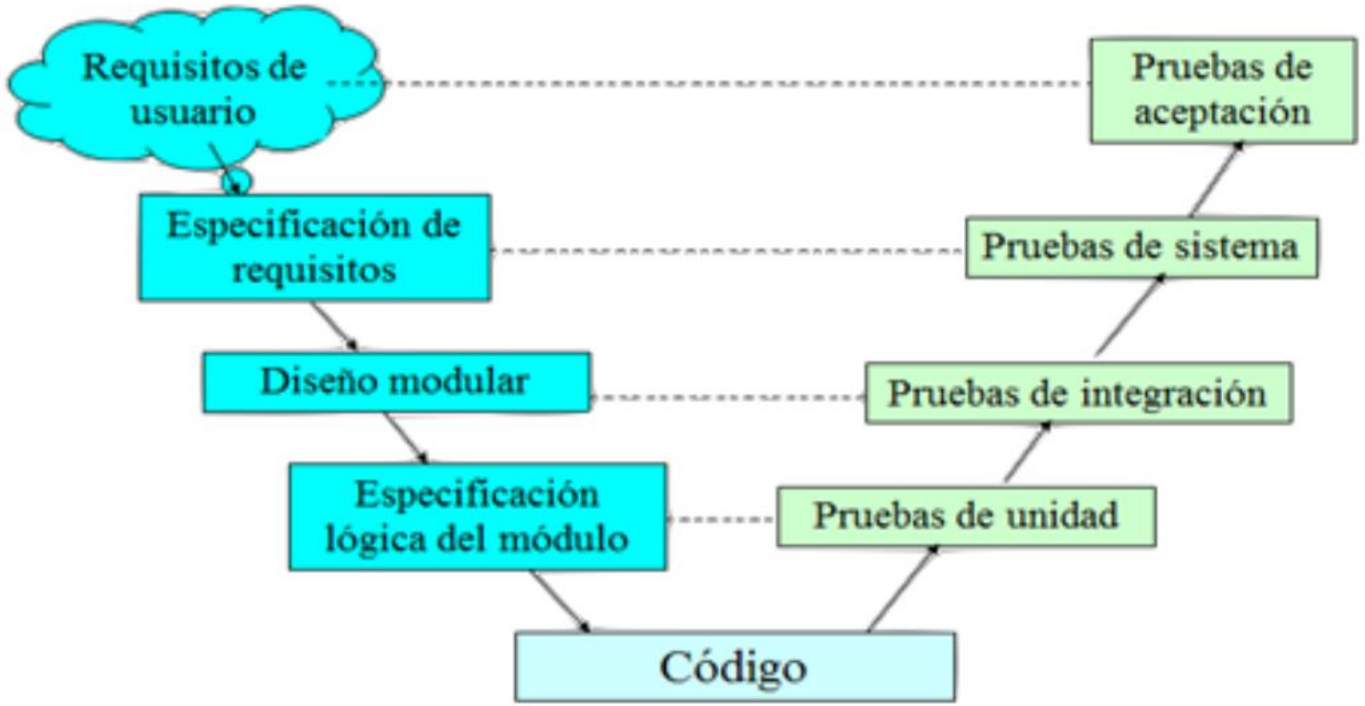
Tipos de pruebas



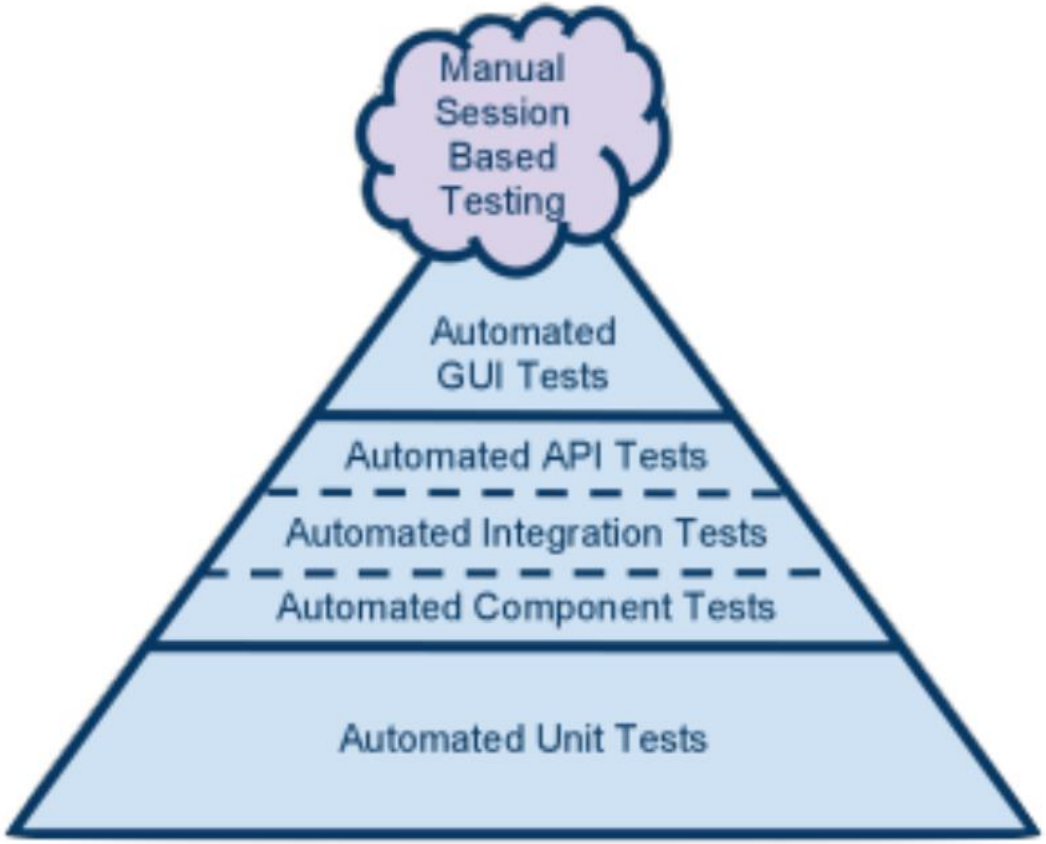
Tipos de pruebas



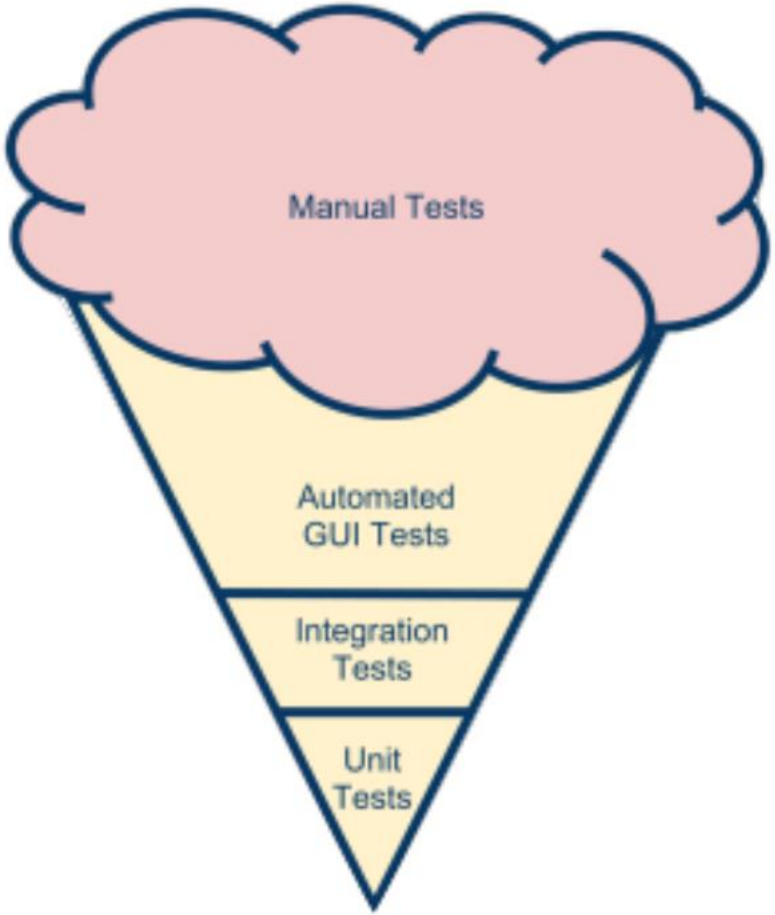
Estrategia de pruebas



Pirámide de Cohn’s

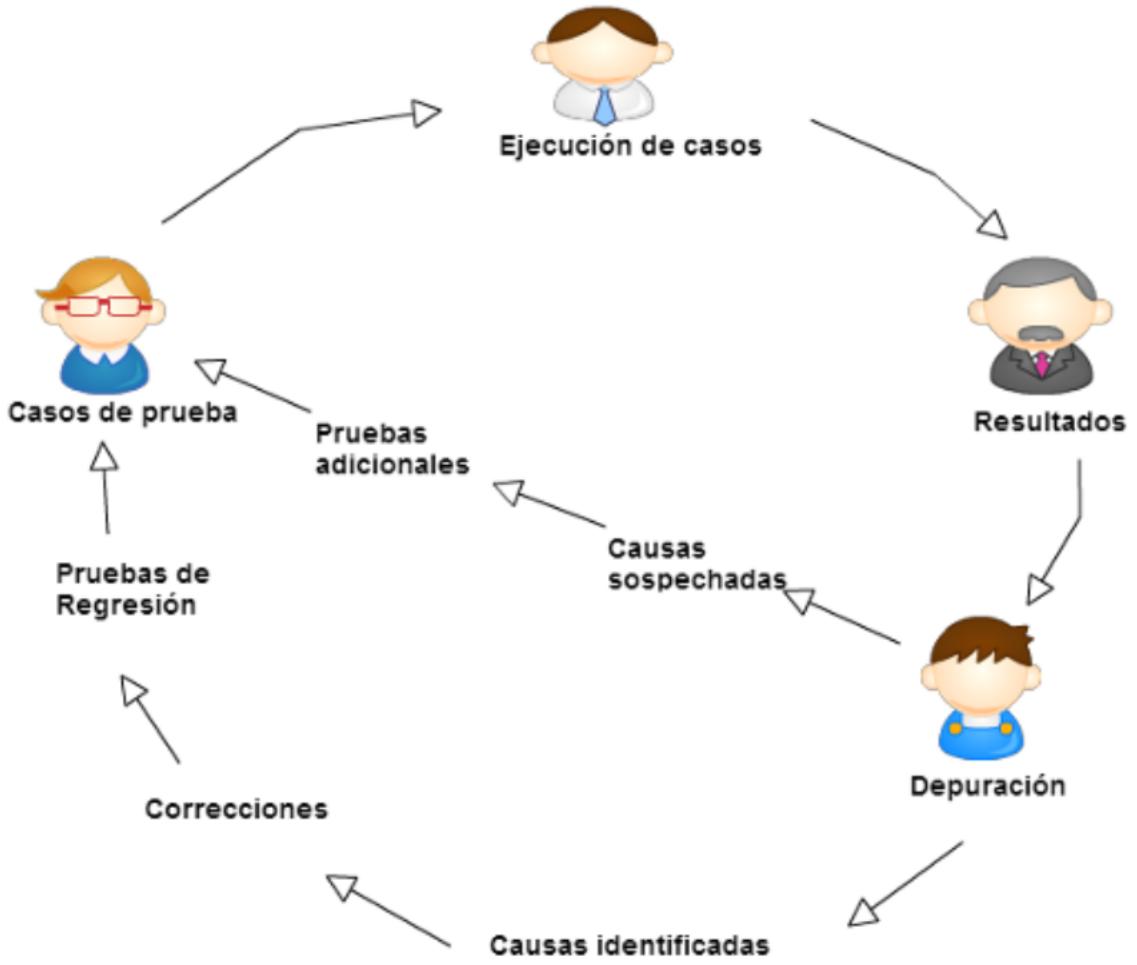


Cono de helado

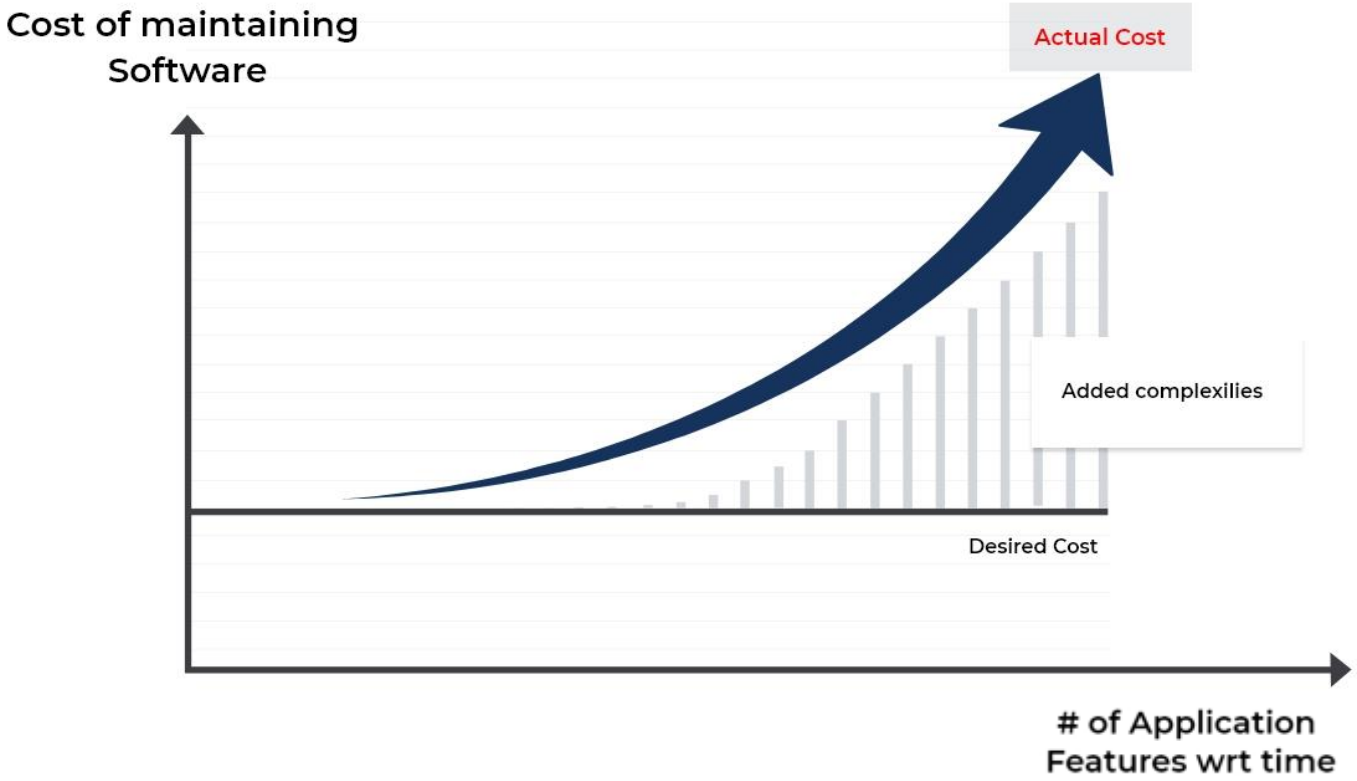


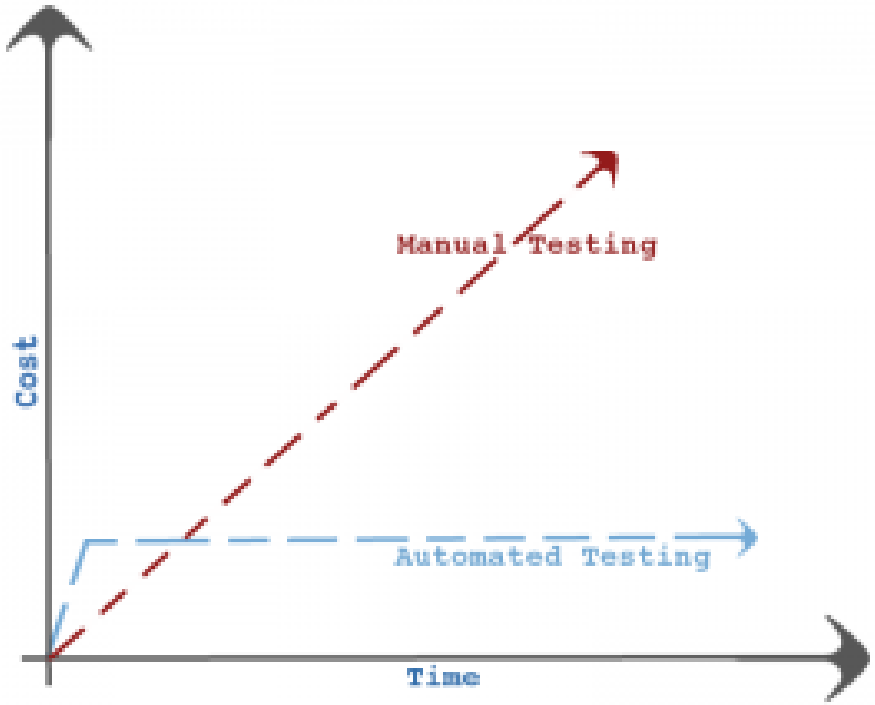
Complejidad en los proyectos

Consumo de tiempo



Beneficios de automatizar





Técnicas de diseño

Nafarroako Gobernua
Gobierno de Navarra

Técnicas de diseño

Técnicas de diseño en casos de prueba

Una buena prueba debe centrarse en dos objetivos:

- Probar si el software no hace lo que debe hacer.
- Probar si el software hace lo que no debe hacer.

Un caso de prueba (test case) se define como “un conjunto de entradas, condiciones de ejecución, y resultados esperados desarrollados para un objetivo particular como, por ejemplo, ejercitar un camino concreto de un programa o verificar el cumplimiento de un determinado requisito, incluyendo toda la documentación asociada”.

El diseño de casos de prueba está totalmente condicionado por la imposibilidad de probar exhaustivamente el software, ya que no se puede probar todos los casos posibles en el desarrollo de un software.

Las técnicas de diseño de casos de prueba tienen como objetivo conseguir una confianza aceptable en que se detectarán los defectos existentes sin que se necesite consumir una cantidad excesiva de recursos. Toda la disciplina de pruebas debe moverse, por lo tanto, en un equilibrio entre la disponibilidad de recursos y la confianza que aportan los casos para descubrir los defectos existentes.

Técnicas de diseño

Enfoques de pruebas de código

Las pruebas de código consisten en la ejecución de un programa con el objetivo de encontrar errores. El programa o parte de él, se va a ejecutar bajo unas condiciones previamente especificadas, para una vez observados los resultados, estos serán registrados y evaluados.

Para llevar a cabo las pruebas, es necesario definir una serie de casos de prueba, que van a ser un conjunto de entradas, de condiciones de ejecución y resultados esperados, desarrollados para un objetivo particular. Para el diseño de casos de prueba de código, se suelen utilizar tres enfoques: el enfoque estructural, el funcional y el aleatorio.

- **El enfoque funcional o de caja negra** se centra en las funciones, entradas y salidas. Se aplican los valores límite y las clases de equivalencia
- **El enfoque estructural o de caja blanca**, se centra en la estructura interna del programa, analizando los caminos de ejecución. Lo aplicamos con el cubrimiento.

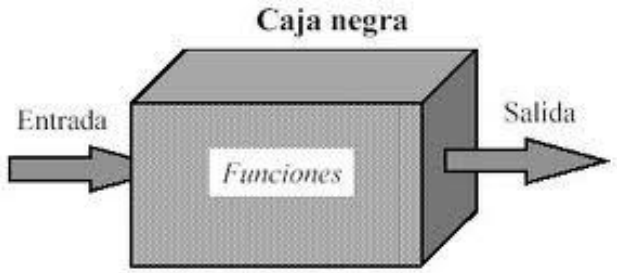
Técnicas de diseño

Técnicas de diseño en casos de prueba

La idea fundamental para el diseño de casos de prueba consiste en elegir algunas pruebas que, por sus características, se consideren representativas del resto. La dificultad de esta idea es saber elegir los casos que se deben ejecutar. De hecho, una elección puramente aleatoria no proporciona demasiada confianza en que se puedan detectar todos los defectos existentes.

Existen tres enfoques principales para el diseño de casos de prueba:

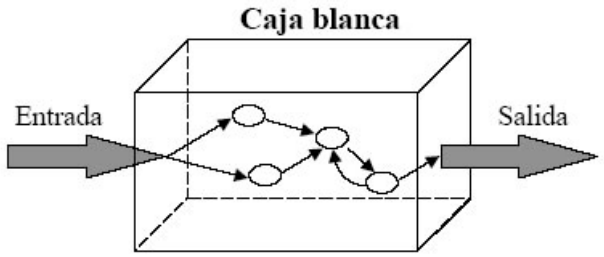
- **Enfoque funcional o de caja negra**. En este tipo de pruebas, nos centramos en que el programa, o parte del programa que estamos probando, recibe una entrada de forma adecuada y se produce una salida correcta, así como que la integridad de la información externa se mantiene. La prueba no verifica el proceso, solo los resultados. La prueba ideal consistiría en probar todas las posibles entradas y salidas del programa.



Técnicas de diseño

Técnicas de diseño en casos de prueba

- **Enfoque estructural o caja blanca**. En este tipo de pruebas, debemos centrar en la estructura interna (implementación) del programa. Se trata de comprobar que la operación interna se ajusta a las especificaciones. En esta prueba, se deberían de probar todos los caminos que puede seguir la ejecución del programa.



- **Enfoque aleatorio**. A partir de modelos obtenidos estadísticamente, se elaboran casos de prueba que prueben las entradas del programa. La prueba exhaustiva consistiría en probar todas las posibles entradas del programa.

Notar que los enfoques no son excluyentes entre sí, ya que se pueden combinar para conseguir una detección de defectos más eficaz.

Técnicas de diseño

Pruebas de caja negra o funcionales

Se trata de probar, si las salidas que devuelve la aplicación, o parte de ella, son las esperadas, en función de los parámetros de entrada que le pasemos. Este tipo de prueba no consideraría, en ningún caso, el código desarrollado, ni el algoritmo, ni la eficiencia, ni si hay partes del código innecesarias, etc.

No nos interesa la implementación del software, solo si realiza las funciones que se esperan de él.

Comprenderían aquellas actividades cuyo objetivo sea verificar una acción específica o funcional dentro del código de una aplicación. Las pruebas funcionales intentarían responder a las preguntas ¿puede el usuario hacer esto? o ¿funciona esta utilidad de la aplicación?

Con estas pruebas se intenta encontrar errores en las categorías siguientes:

- Funciones incorrectas o faltantes.
- Errores de interfaz.
- Errores en las estructuras de datos o en el acceso a bases de datos externas.
- Errores de comportamiento o rendimiento.
- Errores de inicialización o finalización.

Técnicas de diseño

Pruebas de caja negra o funcionales

Dentro de las pruebas funcionales, podemos indicar cuatro tipos de pruebas:

- **Particiones equivalentes:** La idea de este tipo de pruebas funcionales, es considerar el menor número posible de casos de pruebas, para ello, cada caso de prueba tiene que abarcar el mayor número posible de entradas diferentes. Lo que se pretende, es crear un conjunto de clases de equivalencia, donde la prueba de un valor representativo de la misma, en cuanto a la verificación de errores, sería extrapolable al que se conseguiría probando cualquier valor de la clase.
- **Análisis de valores límite:** En este caso, a la hora de implementar un caso de prueba, se van a elegir como valores de entrada, aquellos que se encuentra en el límite de las clases de equivalencia.
- **Pruebas aleatorias:** Consiste en generar entradas aleatorias para la aplicación que hay que probar. Se suelen utilizar generadores de prueba, que son capaces de crear un volumen de casos de prueba al azar, con los que será alimentada la aplicación. Este tipo de pruebas, se suelen utilizar en aplicaciones que no sean interactivas, ya que es muy difícil generar las secuencias de entrada adecuadas de prueba, para entornos interactivos.
- **Hipótesis de errores:** se basan en la experiencia de dónde aparecen los errores más habituales en las fases de análisis y desarrollo. Los errores se buscan dónde se prevé que suelen presentarse. Es un método fuertemente dependiente de la experiencia.

Pruebas de código

Clases de equivalencia

Las clases de equivalencia, es un tipo de prueba funcional, en donde cada caso de prueba, pretende cubrir el mayor número de entradas posible.

El dominio de valores de entrada, se divide en número finito de clases de equivalencia. Como la entrada está dividida en un conjunto de clases de equivalencia, la prueba de un valor representativo de cada clase, permite suponer que el resultado que se obtiene con él, será el mismo que con cualquier otro valor de la clase.

Cada clase de equivalencia debe cumplir:

- Si un parámetro de entrada debe estar comprendido entre un determinado rango, hay tres clases de equivalencia: por debajo, en y por encima.
- Si una entrada requiere un valor entre los de un conjunto, aparecen dos clase de equivalencia: en el conjunto o fuera de él.
- Si una entrada es booleana, hay dos clases: sí o no.
- Los mismos criterios se aplican a las salidas esperadas: hay que intentar generar resultados en todas y cada una de las clases.

Pruebas de código

Clases de equivalencia

```
public double funcion (double x) {
  if (x>0 && x<100)
    return x+2;
  else
    return x-2;
}
```

Las clases de equivalencia serían:

- Por debajo: x<=0
- En: x>0 y x<100
- Por encima: x>=100

y los respectivos casos de prueba, podrían ser:

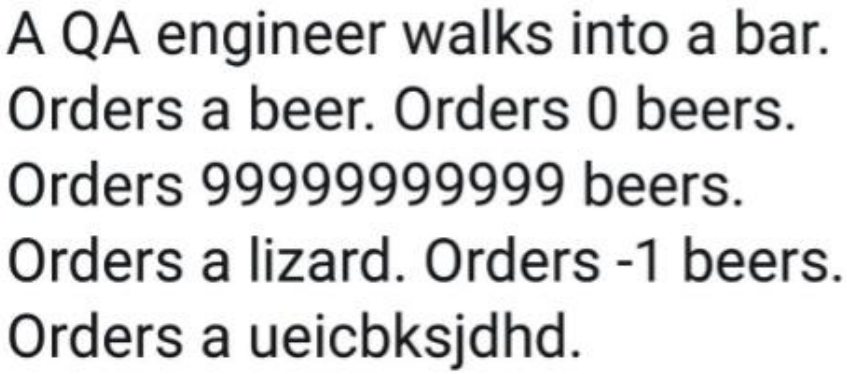
- Por debajo :x=0
- En: x=50
- Por encima: x=100

Pruebas de código

Valores límite

Esta técnica, se suele utilizar como complementaria de las particiones equivalentes, pero se diferencia, en que se suelen seleccionar, no un conjunto de valores, sino unos pocos, en el límite del rango de valores aceptado por el componente a probar.

Cuando hay que seleccionar un valor para realizar una prueba, se escoge aquellos que están situados justo en el límite de los valores admitidos.



Pruebas de código

Valores límite

Dado el siguiente fragmento de código:

```
public double funcion1 (double x) {
  if (x>5)
    return x;
  else
    return -1;
}
```

```
public int funcion2 (int x)
  {
  if (x>5)
    return x;
  else
    return -1;
}
```

Vemos que aparecen dos funciones que reciben el parámetro x. En la función1, el parámetro es de tipo real y en la función2, el parámetro es de tipo entero.

Como se aprecia, el código de las dos funciones es el mismo, sin embargo, los casos de prueba con valores límite va a ser diferente:

- Si el parámetro x de entrada tiene que ser mayor estricto que 5, y el valor es real, los valores límite pueden ser 4,99 y 5,01.
- Si el parámetro de entrada x está comprendido entre -4 y +4, suponiendo que son valores enteros, los valores límite serán -5, -4, -3,3, 4 y 5.

Técnicas de diseño

Pruebas de caja blanca o estructurales

Para ver cómo el programa se va ejecutando, y así comprobar su corrección, se utilizan las pruebas estructurales, que se fijan en los caminos que se pueden recorrer

Con este tipo de pruebas, se pretende verificar la estructura interna de cada componente de la aplicación, independientemente de la funcionalidad establecida para el mismo.

Este tipo de pruebas no pretende comprobar la corrección de los resultados producidos por los distintos componentes, su función es comprobar que se van a ejecutar todas las instrucciones del programa, que no hay código no usado, comprobar que los caminos lógicos del programa se van a recorrer, se utilizan las decisiones en su parte verdadera y en su parte falsa, etc.

Este tipo de pruebas, se basan en unos criterios de cobertura lógica, cuyo cumplimiento determina la mayor o menor seguridad en la detección de errores.

Técnicas de diseño

Pruebas de caja blanca o estructurales

Algunos de los criterios de cobertura lógica propuestos son:

- **Cobertura de sentencias:** se han de generar casos de pruebas suficientes para que cada instrucción del programa sea ejecutada, al menos, una vez.
- **Cobertura de decisiones:** se trata de crear los suficientes casos de prueba para que cada opción resultado de una prueba lógica del programa, se evalúe al menos una vez a cierto y otra a falso.
- **Cobertura de condiciones:** se trata de crear los suficientes casos de prueba para que cada elemento de una condición se evalúe al menos una vez a falso y otra a verdadero.
- **Cobertura de condiciones y decisiones:** consiste en cumplir simultáneamente las dos anteriores.
- **Cobertura de caminos:** establece que se debe ejecutar al menos una vez cada secuencia de sentencias encadenadas, desde la sentencia inicial del programa, hasta su sentencia final. La ejecución de este conjunto de sentencias se conoce como camino. Como el número de caminos que puede tener una aplicación, puede ser muy grande, para realizar esta prueba, se reduce el número a lo que se conoce como camino prueba.

Pruebas de código

Cobertura

Esta tarea la realiza el programador o programadora y consiste en comprobar que los caminos definidos en el código se pueden llegar a recorrer.

Al ser un tipo de **prueba de caja blanca**, lo que se pretende, es comprobar que todas las funciones, sentencias, decisiones, y condiciones, se van a ejecutar.

La siguiente función que forma parte de un programa mayor:

```
function int prueba(int x, int y){
    int z=0;
    if((x>0) && (y>0))
    x=z;
    return z;
}
```

- Si durante la ejecución del programa, la función es llamada, al menos una vez, el cubrimiento de la función es satisfecho.
- El cubrimiento de sentencias para esta función, será satisfecho si es invocada, por ejemplo como prueba(1,1), ya que en esta caso, cada línea de la función se ejecuta, incluida z=x;
- Si invocamos a la función con prueba(1,1) y prueba(0,1), se satisfará el cubrimiento de decisión. En el primer caso, la if condición va a ser verdadera, se va a ejecutar z=x, pero en el segundo caso, no.

Buenas prácticas

Nafarroako Gobernua
Gobierno de Navarra

Buenas prácticas

Code Smells

```
41 @    static MappedField validateQuery(final Class clazz, final Mapper mapper, final StringBuilder origProp, final FilterOperator op, final
42          MappedField mf = null;
43          final String prop = origProp.toString();
44          boolean hasTranslations = false;
45          if (!origProp.substring(0, 1).equals("$")) {
46              final String[] parts = prop.split( regex: "\\.");
47              if (clazz == null) { return null; }
48              MappedClass mc = mapper.getMappedClass(clazz);
49              //CHECKSTYLE:OFF
50              for (int i = 0; ; ) {
51                  //CHECKSTYLE:ON
52                  final String part = parts[i];
53                  boolean fieldIsArrayOperator = part.equals("$");
54                  mf = mc.getMappedField(part);
55                  //translate from java field name to stored field name
56                  if (mf == null && !fieldIsArrayOperator) {
57                      mf = mc.getMappedFieldByJavaField(part);
58                      if (validateNames && mf == null) {
59                          throw new ValidationException(format("The field '%s' could not be found in '%s' while validating - %s; if you wis
60                      }
61                      hasTranslations = true;
62                      if (mf != null) {
63                          parts[i] = mf.getNameToStore();
64                      }
65                  }
66                  i++;
67                  if (mf != null && mf.isMap()) {
68                      //skip the map key validation, and move to the next part
69                      i++;
70                  }
71                  if (i >= parts.length) {
72                      break;
73                  }
74                  if (!fieldIsArrayOperator) {
75                      //catch people trying to search/update into @Reference/@Serialized fields
76                      if (validateNames && !canQueryPast(mf)) {
77                          throw new ValidationException(format("Cannot use dot-notation past '%s' in '%s'; found while validating - %s", pa
78                      }
79                      if (mf == null && mc.isInterface()) {
80                          break;
81                      } else if (mf == null) {
82                          throw new ValidationException(format("The field '%s' could not be found in '%s'", prop, mc.getClazz().getName()))
83                      }
84                      //get the next MappedClass for the next field validation
85                      mc = mapper.getMappedClass((mf.isSingleValue()) ? mf.getType() : mf.getSubClass());
86                  }
87              }
88
89              //record new property string if there has been a translation to any part
90              if (hasTranslations) {
91                  origProp.setLength(0); // clear existing content
92                  origProp.append(parts[0]);
93                  for (int i = 1; i < parts.length; i++) {
```

What's a prop?

What's a part?

Eeek!

Why all the null checks?

Control the loop

Comments, because code is unclear

Parameter mutation!

https://blog.jetbrains.com/idea/2017/09/code-smells-too-many-problems/

Buenas prácticas

Code Smells

```
static ValidatedField validateQuery(final Class clazz, final Mapper mapper,
                                    final String propertyPath, final boolean validateNames) {
    final ValidatedField validatedField = new ValidatedField(propertyPath);
    if (clazz == null || isOperator(propertyPath)) {
        return validatedField;
    }

    final String[] pathElements = propertyPath.split("\\.");
    final List<String> databasePathElements = new ArrayList<>(asList(pathElements));
    validatedField.mappedClass = mapper.getMappedClass(clazz);

    for (int i = 0; i < pathElements.length; i++) {
        final String fieldName = pathElements[i];
        if (isArrayOperator(fieldName)) {
            continue;
        }

        validatedField.mappedField = getAndValidateMappedField(fieldName, validatedField, propertyPath,
                                                                validateNames, databasePathElements, i);

        if (isMap(validatedField.mappedField)) {
            //skip the map key validation, and move to the next fieldName
            i++;
        }

        if (hasMoreElements(pathElements, i)) {
            if (validatedField.mappedField.isPresent()) {
                final MappedField mappedField = validatedField.mappedField.get();
                //catch people trying to search/update into @Reference/@Serialized fields
                if (validateNames && cannotQueryPastCurrentField(mappedField)) {
                    throw cannotQueryPastFieldException(propertyPath, fieldName, validatedField);
                }

                //get the next MappedClass for the next field validation
                validatedField.mappedClass = getMappedClass(mapper, mappedField);
            } else if (validatedField.mappedClass.isInterface()) {
                break;
            } else {
                throw fieldNotFoundException(propertyPath, validatedField);
            }
        }
    }
    validatedField.databasePath = databasePathElements.stream().collect(joining("."));
    return validatedField;
}
```

Buenas prácticas

SOLID Principles

Los 5 principios que ayudan a desarrollar software de mejor calidad

- *Single Responsibility Principle*
- *Open/Closed Principle*
- *Liskov Substitution Principle*
- *Interface Segregation Principle*
- *Dependency Inversion Principle*

Buenas prácticas

SOLID Principles – Single Responsability

A class/method should have one, and only one, reason to change

```
class User
{
    void CreatePost(Database db, string postMessage)
    {
        try
        {
            db.Add(postMessage);
        }
        catch (Exception ex)
        {
            db.LogError("An error occured: ", ex.ToString());
            File.WriteAllText("\LocalErrors.txt", ex.ToString());
        }
    }
}
```

Buenas prácticas

SOLID Principles – Single Responsability

A class/method should have one, and only one, reason to change

```
class Post
{
    private ErrorLogger errorLogger = new ErrorLogger();

    void CreatePost(Database db, string postMessage)
    {
        try
        {
            db.Add(postMessage);
        }
        catch (Exception ex)
        {
            errorLogger.log(ex.ToString())
        }
    }
}
```

```
class ErrorLogger
{
    void log(string error)
    {
      db.LogError("An error occured: ", error);
      File.WriteAllText("\LocalErrors.txt", error);
    }
}
```

Buenas prácticas

SOLID Principles – Single Responsability

A class/method should have one, and only one, reason to change

```
public Producto GetProductoPorId(int idProducto)
{
    Producto producto = null;

    var command = _dataBaseManager.GetStoredProcedureCommand(GET_PRODUCTOS_BY_ID);
    _dataBaseManager.AddInInteger32Parameter(command, "IdProducto", idProducto);
    using (var dataReader = _dataBaseManager.ExecuteReader(command))
    {
        if (dataReader.Read())
        {
            producto = _productoMapper.MapearProducto(dataReader);
        }
    }
    return producto;
}

2 referencias
public IList<Producto> GetProductosPorCategoria(int idCategoriaProducto)
{
    var listaProductos = new List<Producto>();

    var command = _dataBaseManager.GetStoredProcedureCommand(GET_PRODUCTOS_BY_IDCATEGORIAPRODUCTO);
    _dataBaseManager.AddInInteger32Parameter(command, "IdCategoriaProducto", idCategoriaProducto);
    using (var dataReader = _dataBaseManager.ExecuteReader(command))
    {
        while (dataReader.Read())
        {
            var producto = _productoMapper.MapearProducto(dataReader);
            listaProductos.Add(producto);
        }
    }
```

```
as de "ExecuteReader"
olución    Agrupar por: Proyecto y, después, Definició    Mantener resultados    Vista de lista
Código
DataReader IDataBaseManager.ExecuteReader(DbCommand) (2)
    .Setup(x => x.ExecuteReader(dbCommand)).Returns(dataReader);
    return _dbManager.ExecuteReader(command);
emplo.Dal (7)
DataReader IDataBaseManager.ExecuteReader(DbCommand) (7)
    using (var dataReader = _dataBaseManager.ExecuteReader(command))
    using (var dataReader = _dataBaseManager.ExecuteReader(command))
    using (var dataReader = _dataBaseManager.ExecuteReader(command))
    using (var dataReader = _dataBaseManager.ExecuteReader(command))
    using (var dataReader = _dataBaseManager.ExecuteReader(command))
    using (var dataReader = _dataBaseManager.ExecuteReader(command))
    using (var dataReader = _dataBaseManager.ExecuteReader(command))
```

Buenas prácticas

SOLID Principles – Open/Closed

You should be able to extend a classes behavior, without modifying it

```
class Post
{
    void CreatePost(Database db, string postMessage)
    {
        db.Add(postMessage);
    }
}
```

```
class Post
{
    void CreatePost(Database db, string postMessage)
    {
        if (postMessage.StartsWith("#"))
        {
            db.AddAsTag(postMessage);
        }
        else
        {
            db.Add(postMessage);
        }
    }
}
```

Buenas prácticas

SOLID Principles – Open/Closed

You should be able to extend a classes behavior, without modifying it

```
class Post
{
    void CreatePost(Database db, string postMessage)
    {
        db.Add(postMessage);
    }
}

class TagPost : Post
{
    override void CreatePost(Database db, string postMessage)
    {
        db.AddAsTag(postMessage);
    }
}
```

Buenas prácticas

SOLID Principles – Open/Closed

You should be able to extend a classes behavior, without modifying it

```
namespace Ssid.Ejemplo.FrontEnd.Common.Infrastructure.Ssid.Utilities.Attributes
{
    0 referencias
    public class AutorizacionRolCarAttribute : AutorizacionRolAbstractAttribute
    {
        /// <summary>
        /// Retornamos todos los roles que tiene el usuario de CAR
        /// </summary>
        2 referencias
        public override string[] ObtenerRolesUsuario(IPrincipal user)
        {
            return (string[])((UserPrincipal)user).Roles.ToArray(typeof(string));
        }
    }
}
```

Buenas prácticas

SOLID Principles – Liskov Subtitution

Derived classes must be substitutable for their base classes

```
class Post
{
    void CreatePost(Database db, string postMessage)
    {
        db.Add(postMessage);
    }
}

class TagPost : Post
{
    override void CreatePost(Database db, string postMessage)
    {
        db.AddAsTag(postMessage);
    }
}
```

```
class MentionPost : Post
{
    void CreateMentionPost(Database db, string postMessage)
    {
        string user = postMessage.parseUser();

        db.NotifyUser(user);
        db.OverrideExistingMention(user, postMessage);
        base.CreatePost(db, postMessage);
    }
}
```

Buenas prácticas

SOLID Principles – Liskov Subtitution

Derived classes must be substitutable for their base classes

```
class MentionPost : Post
{
    void CreateMentionPost(Database db, string postMessage)
    {
        string user = postMessage.parseUser();

        db.NotifyUser(user);
        db.OverrideExistingMention(user, postMessage);
        base.CreatePost(db, postMessage);
    }
}
```

```
class PostHandler
{
    private database = new Database();

    void HandleNewPosts() {
        List<string> newPosts = database.getUnhandledPostsMessages();

        foreach (string postMessage in newPosts)
        {
            Post post;

            if (postMessage.StartsWith("#"))
            {
                post = new TagPost();
            }
            else if (postMessage.StartsWith("@"))
            {
                post = new MentionPost();
            }
            else {
                post = new Post();
            }

            post.CreatePost(database, postMessage);
        }
    }
}
```

Buenas prácticas

SOLID Principles – Liskov Subtitution

Derived classes must be substitutable for their base classes

```
class MentionPost : Post
{
    override void CreatePost(Database db, string postMessage)
    {
        string user = postMessage.parseUser();

        NotifyUser(user);
        OverrideExistingMention(user, postMessage)
        base.CreatePost(db, postMessage);
    }

    private void NotifyUser(string user)
    {
        db.NotifyUser(user);
    }

    private void OverrideExistingMention(string user, string postMessage)
    {
        db.OverrideExistingMention(_user, postMessage);
    }
}
```

```
class PostHandler
{
    private database = new Database();

    void HandleNewPosts() {
        List<string> newPosts = database.getUnhandledPostsMessages();

        foreach (string postMessage in newPosts)
        {
            Post post;

            if (postMessage.StartsWith("#"))
            {
                post = new TagPost();
            }
            else if (postMessage.StartsWith("@"))
            {
                post = new MentionPost();
            }
            else {
                post = new Post();
            }

            post.CreatePost(database, postMessage);
        }
    }
}
```

Buenas prácticas

SOLID Principles – Liskov Subtitution

Derived classes must be substitutable for their base classes

```
public class CarUserContext : AbstractUserContext
{
    7 referencias
    private GdN.Car.Core.UserPrincipal CarUser
    {
        get
        {
            var carUser = HttpContext.Current.User as GdN.Car.Core.UserPrincipal;

            if (carUser == null)
            {
                throw new CustomException("El usuario CAR no está establecido");
            }

            return carUser;
        }
    }
```

Buenas prácticas

SOLID Principles – Interface Segregation

Make fine grained interfaces that are client specific

```
interface IPostNew
{
    void CreatePost();
    void ReadPost();
}
```

Buenas prácticas

SOLID Principles – Interface Segregation

Make fine grained interfaces that are client specific

```
interface IPostCreate
{
    void CreatePost();
}

interface IPostRead
{
    void ReadPost();
}
```

Buenas prácticas

SOLID Principles – Interface Segregation

Make fine grained interfaces that are client specific

```
namespace Ssid.Ejemplo.Dal.Interfaces
{
    6 referencias
    public interface IPedidoRepository
    {
        2 referencias
        Pedido GetPedidoPorId(int idPedido);

        4 referencias
        IList<Pedido> GetPedidosPorFiltro(FiltroPedido filtro, int numMaximoResultados);

        2 referencias
        Pedido InsertPedido(Pedido pedido);

        2 referencias
        Pedido UpdatePedido(Pedido pedido);
    }
}
```

Buenas prácticas

SOLID Principles – Dependency Inversion

Depend on abstractions, not on concretions

```
class Post
{
    private ErrorLogger errorLogger = new ErrorLogger();

    void CreatePost(Database db, string postMessage)
    {
        try
        {
            db.Add(postMessage);
        }
        catch (Exception ex)
        {
            errorLogger.log(ex.ToString())
        }
    }
}
```

Buenas prácticas

SOLID Principles – Dependency Inversion

Depend on abstractions, not on concretions

```
class Post
{
    private Logger _logger;

    public Post(Logger injectedLogger)
    {
        _logger = injectedLogger;
    }

    void CreatePost(Database db, string postMessage)
    {
        try
        {
            db.Add(postMessage);
        }
        catch (Exception ex)
        {
            logger.log(ex.ToString())
        }
    }
}
```

Buenas prácticas

SOLID Principles – Dependency Inversion

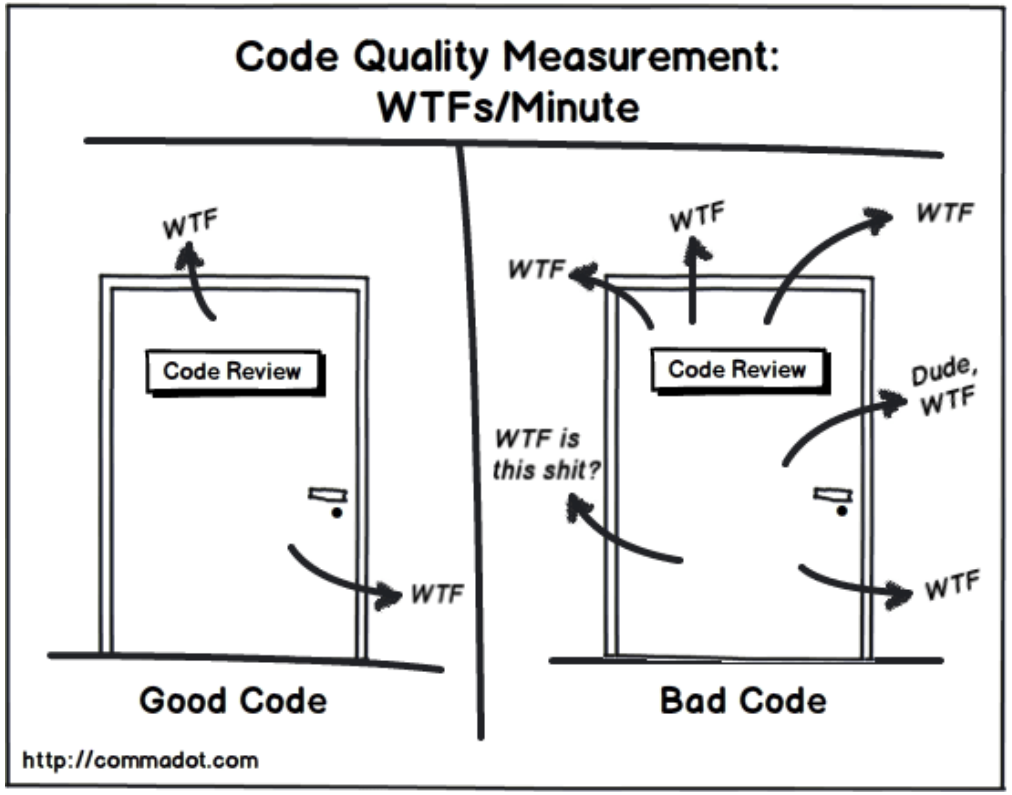
Depend on abstractions, not on concretions

```
public class PedidoService : IPedidoService
{
    private const int GET_PEDIDOS_NUM_MAXIMO_RESULTADOS = 100;
    private readonly IPedidoRepository _pedidoRepository;

    0 referencias
    public PedidoService(IPedidoRepository pedidoRepository)
    {
        _pedidoRepository = pedidoRepository;
    }

    3 referencias
    public Result<IList<Pedido>> GetPedidosPorFiltro(FiltroPedido filtroPedido)
    {
        var listaPedidos = _pedidoRepository.GetPedidosPorFiltro(filtroPedido, GET_PEDIDOS_NUM_MAXIMO_RESULTADOS);

        var resultado = new Result<IList<Pedido>>(listaPedidos);

        if (listaPedidos.Count == GET_PEDIDOS_NUM_MAXIMO_RESULTADOS)
        {
            resultado.AddWarningMessage("Se han obtenido los primeros " + GET_PEDIDOS_NUM_MAXIMO_RESULTADOS + " resultados. Revise
        }

        return resultado;

    }


    1 referencia
    public string prueba()
    {
        return Resource.Prueba;
    }
}
```

Buenas prácticas

SOLID Principles

"No se trata de reglas, ni leyes, ni verdades absolutas, sino más bien soluciones de sentido común a problemas comunes. Son heurísticos, basados en la experiencia." (Uncle Bob)



Buenas prácticas

DRY: Don’t Repeat Yourself

```java
public class Calculator {

    public int total(int a, int b) {
        return a + b;
    }

    public double average(int a, int b) {
        int sum = a + b;

        return sum / 2;
    }
}
```

```java
public class Calculator {

    public int total(int a, int b) {
        int sum =  a + b;
        System.out.println("Total=" + sum);

        return sum;
    }

    public double average(int a, int b) {
        int sum = a + b;
        System.out.println("Total=" + sum);

        return sum / 2;
    }
}
```

Buenas prácticas

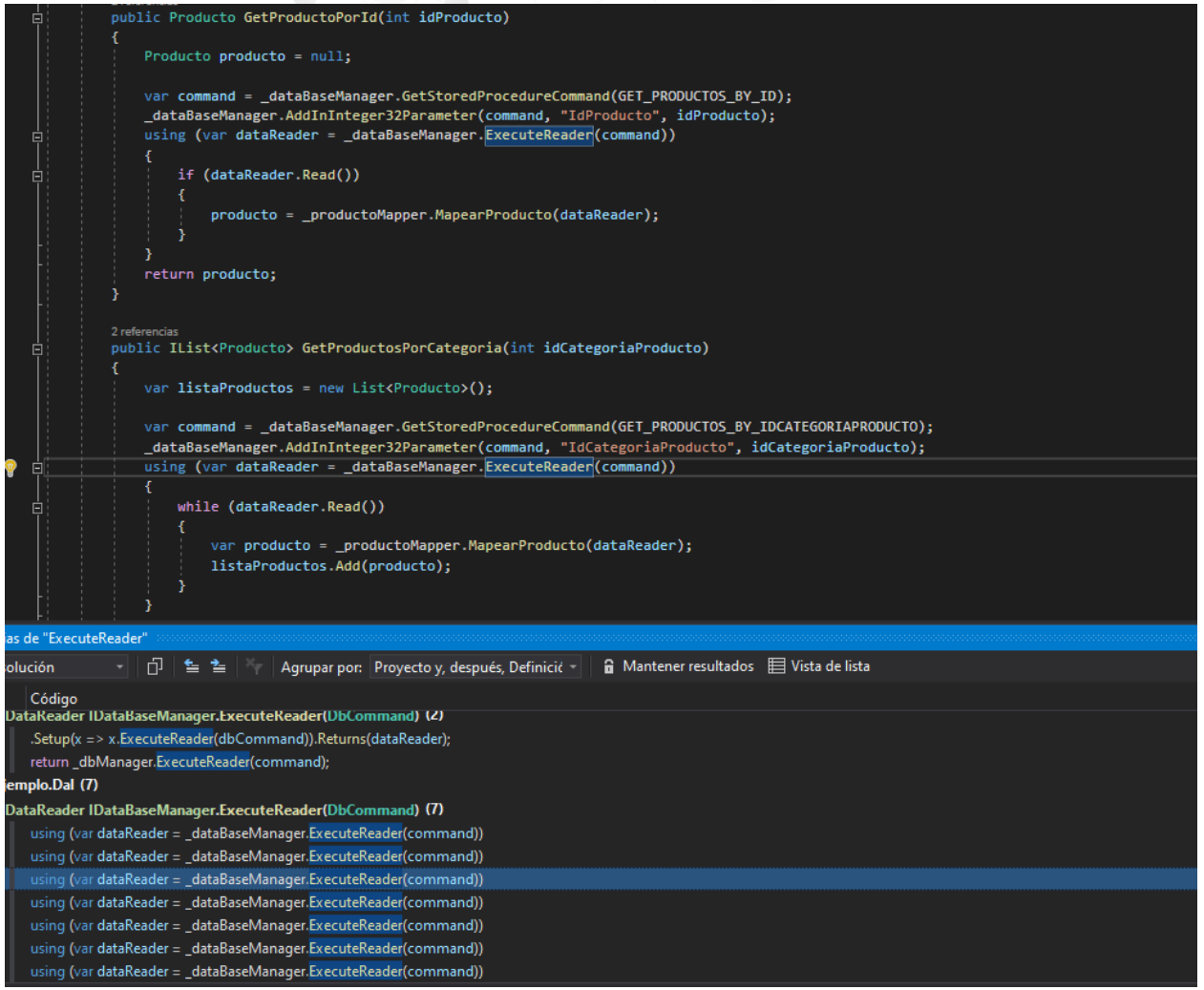
DRY: Don't Repeat Yourself

```java
public class Calculator {

    public int total(int a, int b) {
        return a + b;
    }

    public double average(int a, int b) {
        int sum = total(a, b);

        return sum / 2;
    }
}
```

```java
public class Calculator {

    public int total(int a, int b) {
        int sum =  a + b;
        System.out.println("Total=" + sum);

        return sum;
    }

    public double average(int a, int b) {
        int sum = total(a, b);

        return sum / 2;
    }
}
```

Buenas prácticas

DRY: Don’t Repeat Yourself



```
public Producto GetProductoPorId(int idProducto)
{
    Producto producto = null;

    var command = _dataBaseManager.GetStoredProcedureCommand(GET_PRODUCTOS_BY_ID);
    _dataBaseManager.AddInInteger32Parameter(command, "IdProducto", idProducto);
    using (var dataReader = _dataBaseManager.ExecuteReader(command))
    {
        if (dataReader.Read())
        {
            producto = _productoMapper.MapearProducto(dataReader);
        }
    }
    return producto;
}

2 referencias
public IList<Producto> GetProductosPorCategoria(int idCategoriaProducto)
{
    var listaProductos = new List<Producto>();

    var command = _dataBaseManager.GetStoredProcedureCommand(GET_PRODUCTOS_BY_IDCATEGORIAPRODUCTO);
    _dataBaseManager.AddInInteger32Parameter(command, "IdCategoriaProducto", idCategoriaProducto);
    using (var dataReader = _dataBaseManager.ExecuteReader(command))
    {
        while (dataReader.Read())
        {
            var producto = _productoMapper.MapearProducto(dataReader);
            listaProductos.Add(producto);
        }
    }
```

```
as de "ExecuteReader"
olución    Agrupar por: Proyecto y, después, Definici    Mantener resultados    Vista de lista

Código
DataReader IDataBaseManager.ExecuteReader(DbCommand) (2)
    .Setup(x => x.ExecuteReader(dbCommand)).Returns(dataReader);
    return _dbManager.ExecuteReader(command);
emplo.Dal (7)
DataReader IDataBaseManager.ExecuteReader(DbCommand) (7)
    using (var dataReader = _dataBaseManager.ExecuteReader(command))
    using (var dataReader = _dataBaseManager.ExecuteReader(command))
    using (var dataReader = _dataBaseManager.ExecuteReader(command))
    using (var dataReader = _dataBaseManager.ExecuteReader(command))
    using (var dataReader = _dataBaseManager.ExecuteReader(command))
    using (var dataReader = _dataBaseManager.ExecuteReader(command))
    using (var dataReader = _dataBaseManager.ExecuteReader(command))
```

Best practices

KISS: Keep It Simple (, Stupid)

```
public static bool IsInt32IsDigit(string input)
{
    foreach (Char c in input)
    {
        if (!Char.IsDigit(c))
        {
            return false;
        }
    }
    return true;
}
```

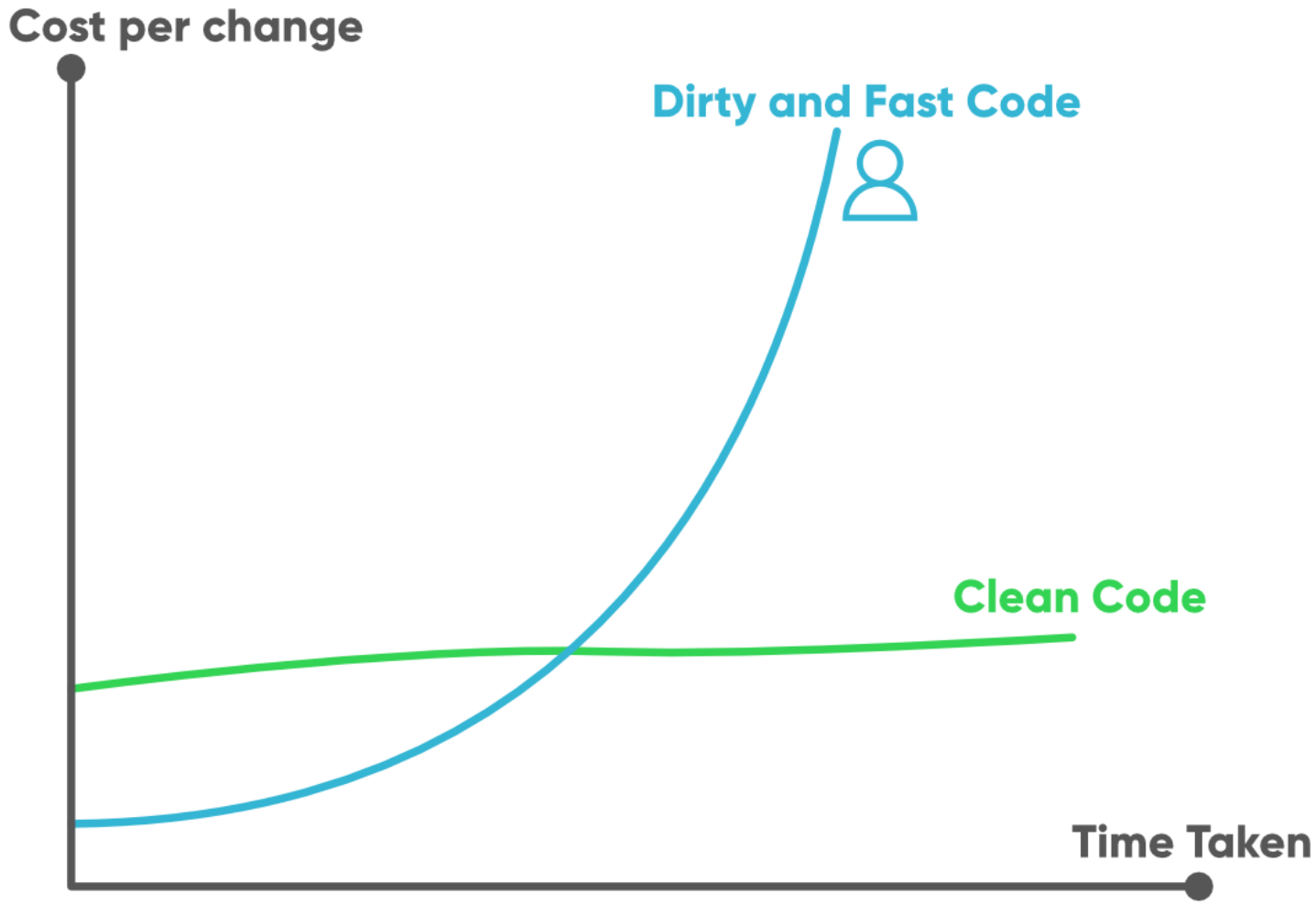
```
public static bool IsInt32TryParse(string input)
{
    int i;
    return Int32.TryParse(input, out i);
}
```

```
public static bool IsInt32RegEx(string input)
{
    return Regex.IsMatch(input, @"^\d+$");
}
```

https://shitcode.net/best/language/csharp

Beneficios de buenas prácticas

Agilidad y mantenibilidad



hiberus© TECNOLOGIA

La compañía hiperespecializada en las TIC

www.hiberus.com