|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| PSL – LCD Display  |  | | --- | |  | | [MAURICIO.BEDOYA@GMAIL.COM](mailto:MAURICIO.BEDOYA@GMAIL.COM) | |  | | 3004802278 | | MEDELLÍN | |  | |  | | --- | | ELABORADO POR: mAURICIO BEDOYA LONDOÑO Objetivo: El objetivo de este documento es la conceptualización del diseño de la prueba LCD Display como proceso de selección para PSL. | |

LISTADO DE VERSIONES

|  |  |  |  |
| --- | --- | --- | --- |
| Descripción | Versión | Fecha | Creado por |
| Versión inicial que muestra cómo se implementará la solución. | 1.0 | 2017/04/20 | Mauricio Bedoya |
|  |  |  |  |
|  |  |  |  |

## LISTADO DE CONTENIDO

[DEFINICIONES DE ARQUITECTURA 4](#_Toc480444659)

[a. Estilo Arquitectónico 4](#_Toc480444660)

[b. Patrones de diseño 4](#_Toc480444661)

[c. Rationale 4](#_Toc480444662)

[DISEÑO DE ALTO NIVEL 6](#_Toc480444663)

[DISEÑOS DETALLADOS 8](#_Toc480444664)

[OTROS 11](#_Toc480444665)

# DEFINICIONES DE ARQUITECTURA

* 1. Estilo Arquitectónico: Tanto los patrones de Arquitectura como los estilos Arquitectónicos dependen de 3 factores:
     1. Requerimientos funcionales
     2. Restricciones técnicas
     3. Atributos de calidad

Para el caso específico del test LCD Display para PSL, sólo se tienen requerimientos funcionales, no hay restricciones técnicas o atributos de calidad, así que la decisión para esta prueba será usar modelo vista controlador (MVC2 - Web) con .Net, algo simple.

* 1. Patrones de diseño:
     1. Se usará “Simple Factory” para crear los objetos que harán la conversión de dígito al formato de display deseado, que para este caso es HTML, pero se deja abierto a nuevos tipos de display.
  2. Rationale:

|  |  |  |
| --- | --- | --- |
| Decisión | Sustentación | Tomada por |
| Se implementarán los patrones arquitectónicos MVC | Se requiere separar responsabilidades y desacoplar elementos. | Mauricio Bedoya |
| El Factory usado, en realidad no será un patrón formal, se podría Factory Method para tener diferentes versiones de Displays: HTML, Console, HTML 5 (canvas), etc, pero como no se sabe si se necesitarán nuevas implementaciones en el futuro, se deja una diseño simple y cumpliendo con la necesidad conocida y no basado en supuestos. |  |  |
| MVC no estándar en .Net | Se usará un MVC en el cual la vista no usará elementos .Net (razor y otros), será una vista con Html, Javascript, Css y Bootstrap más estándar y totalmente desacoplado de .Net, esto para que la vista en caso de ser necesaria portarla a otro backend y para que el backend soporte cualquier otro cliente. Se harán llamados HTTP Post al .Net para la interacción cliente-servidor. | Mauricio Bedoya |
| Se usará .Net para Controlador y Modelo | No hay restricción y es el lenguaje en el cual se tiene mayor experiencia | Mauricio Bedoya |
|  |  |  |

# OTROS

1. Control de código fuente: Se usará github para controlar los fuentes y documentación del proyecto: https://github.com/mbedoya/LcdDisplay.

Se usa también el control de código para demostrar el uso de TDD, que se iniciará por definir una pequeña prueba (Unitaria), luego se procederá a hacer la implementación de esa prueba y así hasta finalizar, cabe anotar que el control de código fuente nunca se verán las pruebas en rojo, es una práctica recomendada sólo subir código cuando todas las pruebas están aceptadas.

1. Interfaz: Esta es la interfaz gráfica usada
2. Pruebas Unitarias: Para estas se usan herramientas entregadas por Microsoft (Visual Studio: UnitTestFramework)

# proceso tdd

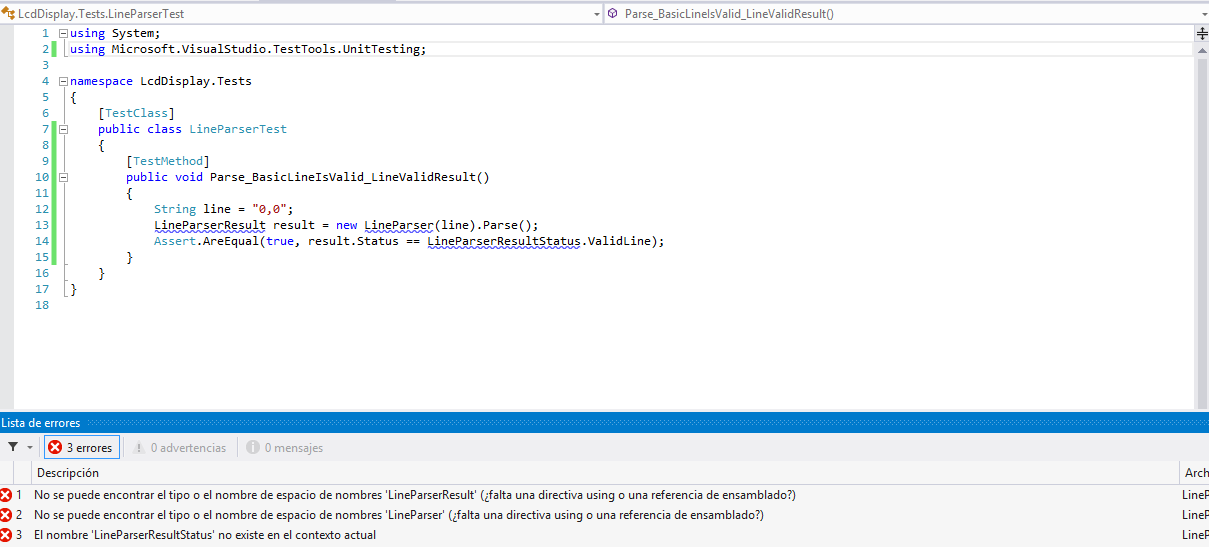
Esta es la forma como se ha llevado el proceso TDD:

Vamos desde la parte interna y más sencilla del programa, parser de líneas, verificar que cada línea es correcta, iniciamos con una prueba muy sencilla, que ni va a compilar porque las clases que vamos a probar aún ni existen.

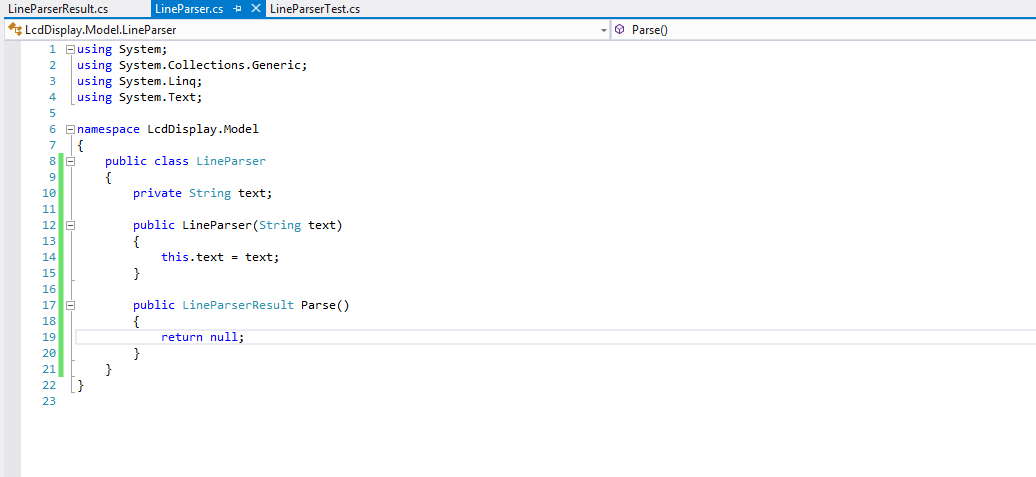
Como primera prueba será lo más básico, enviamos una línea (0,0) y esperamos que el resultado de ejecutar el Parse sea correcto.

Para el nombre de cada prueba seguiremos el estándar:

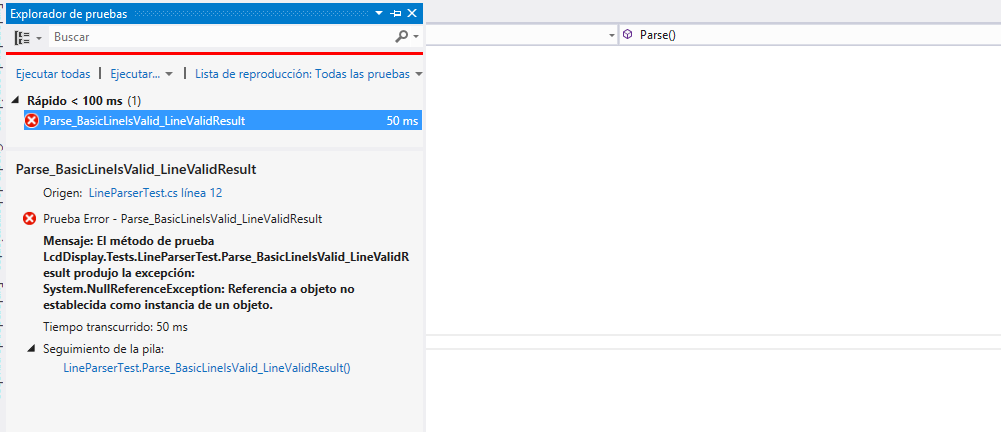
MétodoPrueba\_Contexto\_ResultadoEsperado



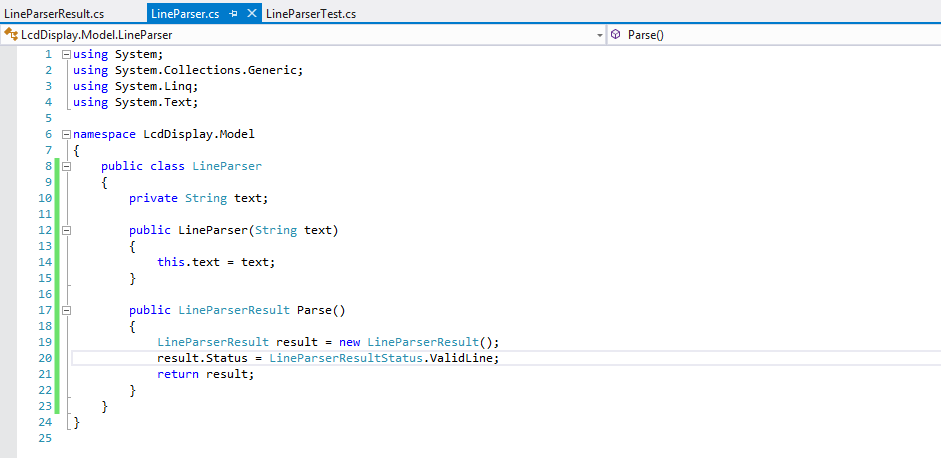
Luego vamos a crear las clases necesarias para que la solucione compile, pero sin lógica alguna, así vamos a obtener el “Rojo”, test fallido



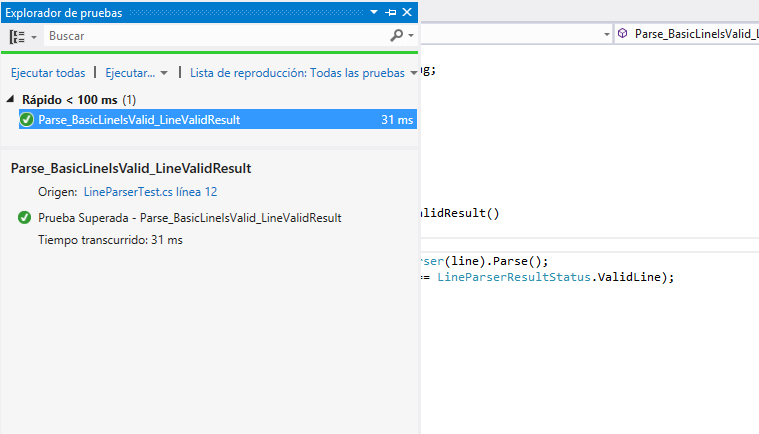
y acá test fallido: Rojo



Vamos entonces a mejorar el método Parse para pasar el Test (la lógica no es la correcta, pero no importa en este punto, acá lo importante es pasar este test, luego iremos paso a paso adicionando nuevas pruebas que van a ir convirtiendo nuestra clase LineParser en un elemento totalmente funcional)

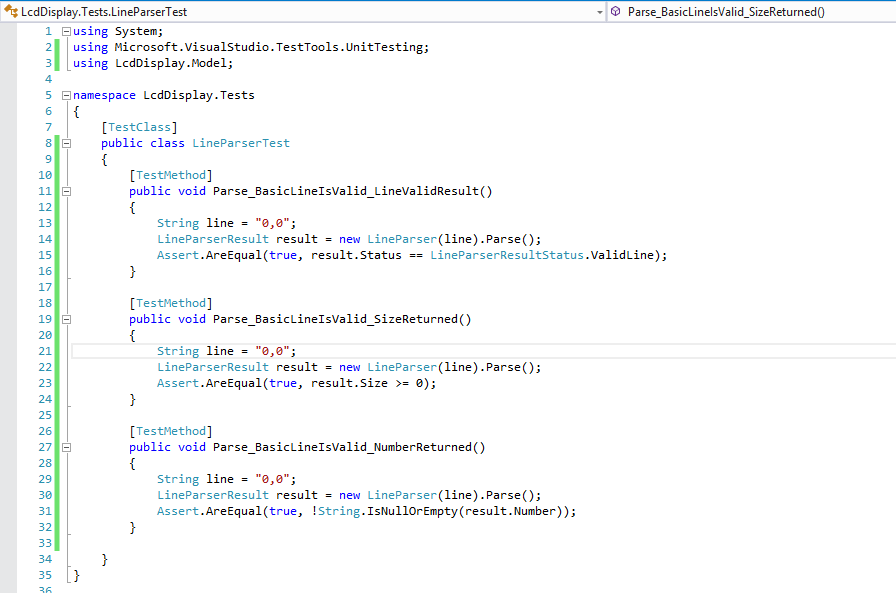


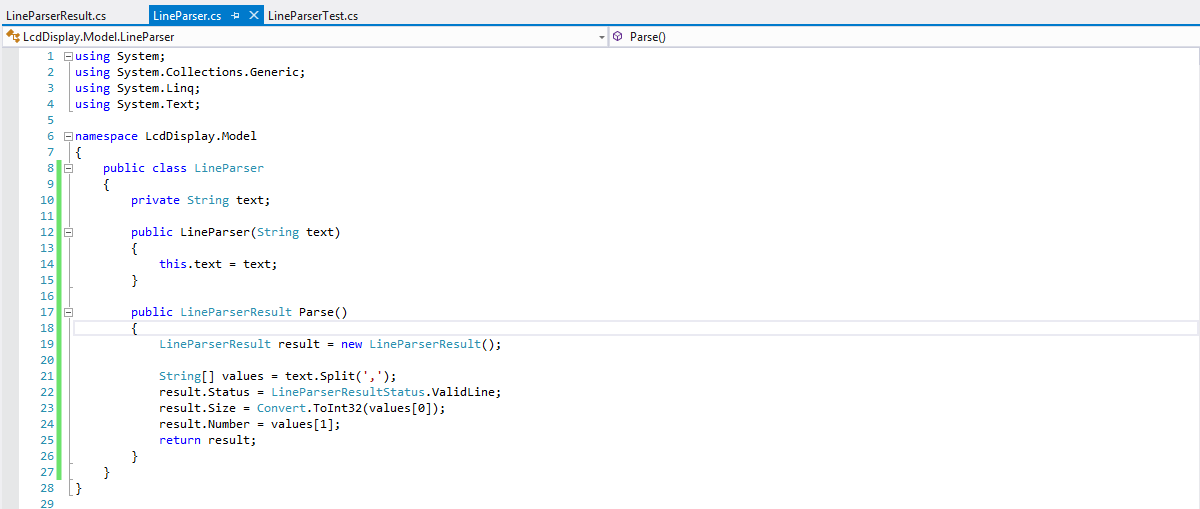
Y ahora obtenemos verde :)



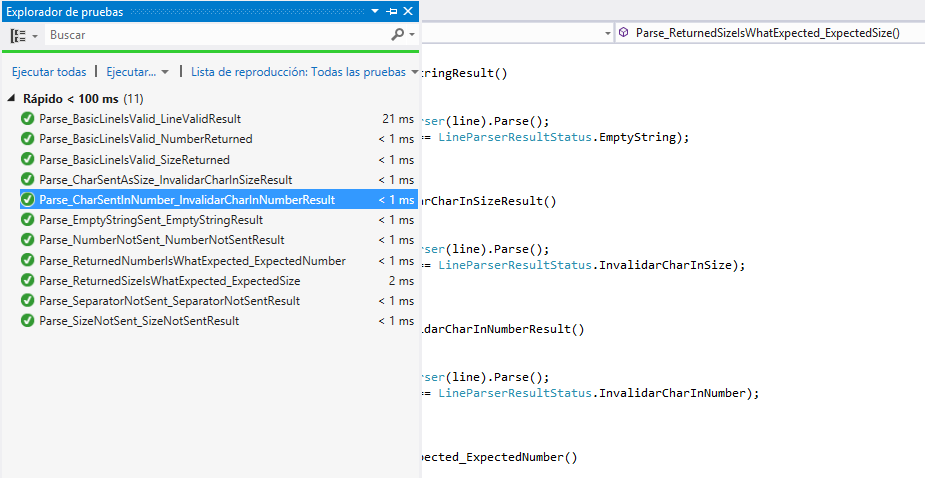
y ahora las nuevas pruebas, y avancemos hasta mejorar nuestra clase ☺

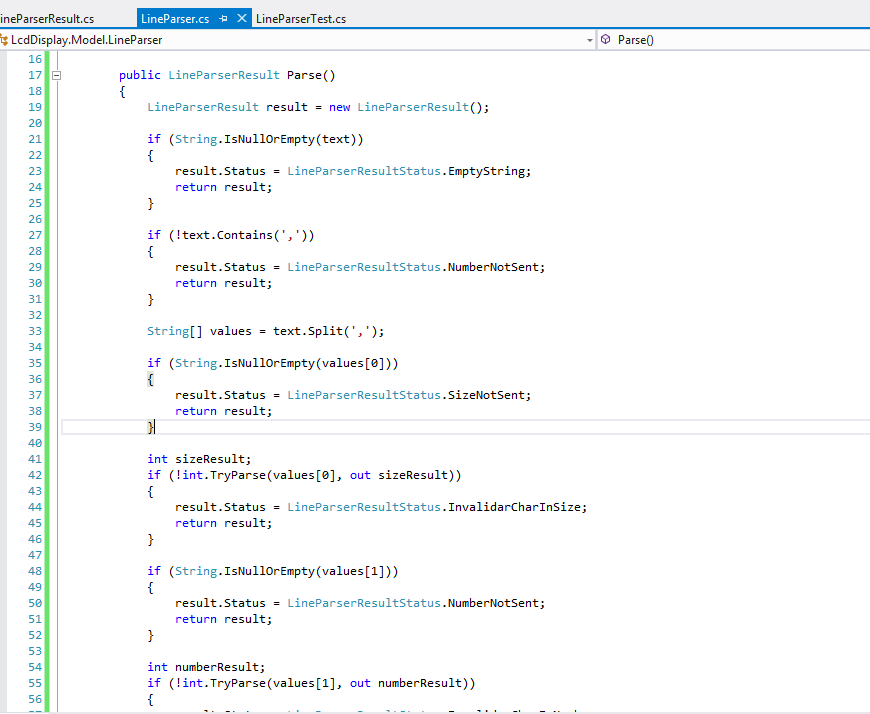
seguimos con el proceso hasta llegar hasta una versión sin validaciones, pero que cumple con la función básica del parser



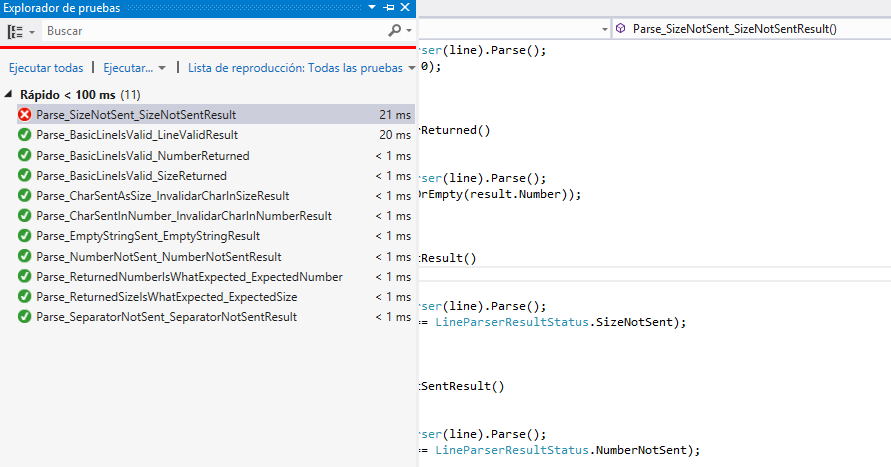


Momento de adicionar validaciones o reglas sobre la cadena de entrada y también refactoring a nuestro modelo que se empieza a poner spaghetti, luego de 11 pruebas así nos vemos:





Luego de hacer recfactoring, TDD en acción, hicimos un daño ☹, pero detectado por las pruebas:



Ahora tenemos un código funcional y refactorizado, sabemos que está acorde a lo esperado y sin ejecutar una vez aún el proyecto completo.

