Desarrollo guiado por pruebas

Electrónica IV

Mg.Ing. Esteban Volentini (evolentini@herrera.unt.edu.ar)

https://facetvirtual.facet.unt.edu.ar/course/view.php?id=165

¿Que es el testing de software?

- Es el proceso centrado en el objetivo de encontrar defectos en un sistema.
- Prueba Estáticas (Verificación)
  - El código no es ejecutado
  - Verificar el cumplimiento del estándar
- Pruebas Dinámicas (Test)
  - El código es ejecutado
  - Verificar el cumplimiento de los requisitos

Coding Gidelines

- Es un conjunto de convenciones sobre como escribir el código:
  - Aumentan la legibilidad
  - Disminuyen los errores
- Hay Guidelines publicas y pagas
  - MISRA -C
  - NASA SEL-94-003

MISRA 2004

- Rule 1.1 Code shall be conform to C90
- Rule 6.1 The plain char type shall be used only for the storage and use of character values.
- Rule 8.1 Functions shall have prototype declarations and the prototype shall be visible at both the function definition and call.
- Rule 8.8 An external object or function shall be declared in one and only one file.
- Rule 9.1 All automatic variables shall have been assigned a value before being used.

Verificación

- Pueden ser realizadas por herramientas automáticas (Linter)
  - Verifican cumplimiento de los prototipos
  - Eliminan situaciones ambiguas
- También pueden ser realizadas por otra persona (Review)
  - Mejora la calidad del software
  - Mejora la calidad del programador
  - Mejora el trabajo en equipo

Clang-Format

- Es una herramienta para ajustar y/o verificar el formato de código fuente en C.
- Soporta varios estilos conocidos como Google, LLVM, Chromium, Mozilla, WebKit, Microsoft y GNU.
- Sobre los formatos de base se pueden ajustar muchos parámetros.
- La configuración se puede almacenar en un archivo dentro del repositorio.

Cppcheck

```
1  #include <stdio.h>
2  int main()
3  {
4    char c;
5    while (c != 'x');
6    {
7      c = getchar();
8      if (c = 'x')
9      return 0;
10     switch (c) {
11       case '\n':
12       case '\r':
13         printf("NewLine\n");
14       default:
15         printf("%c", c);
16     }
17   }
18   return 0;
19 }
```

- Variable c used before definition
- Suspected infinite loop. No value used in loop test (c) is modified by test or loop body.
- Assignment of int to char: c =getchar()
- Test expression for if is assignment expression: c = 'x'
- Test expression for if not boolean, type char: c = 'x'
- Fall through case (no preceding break)

Los hook de Git

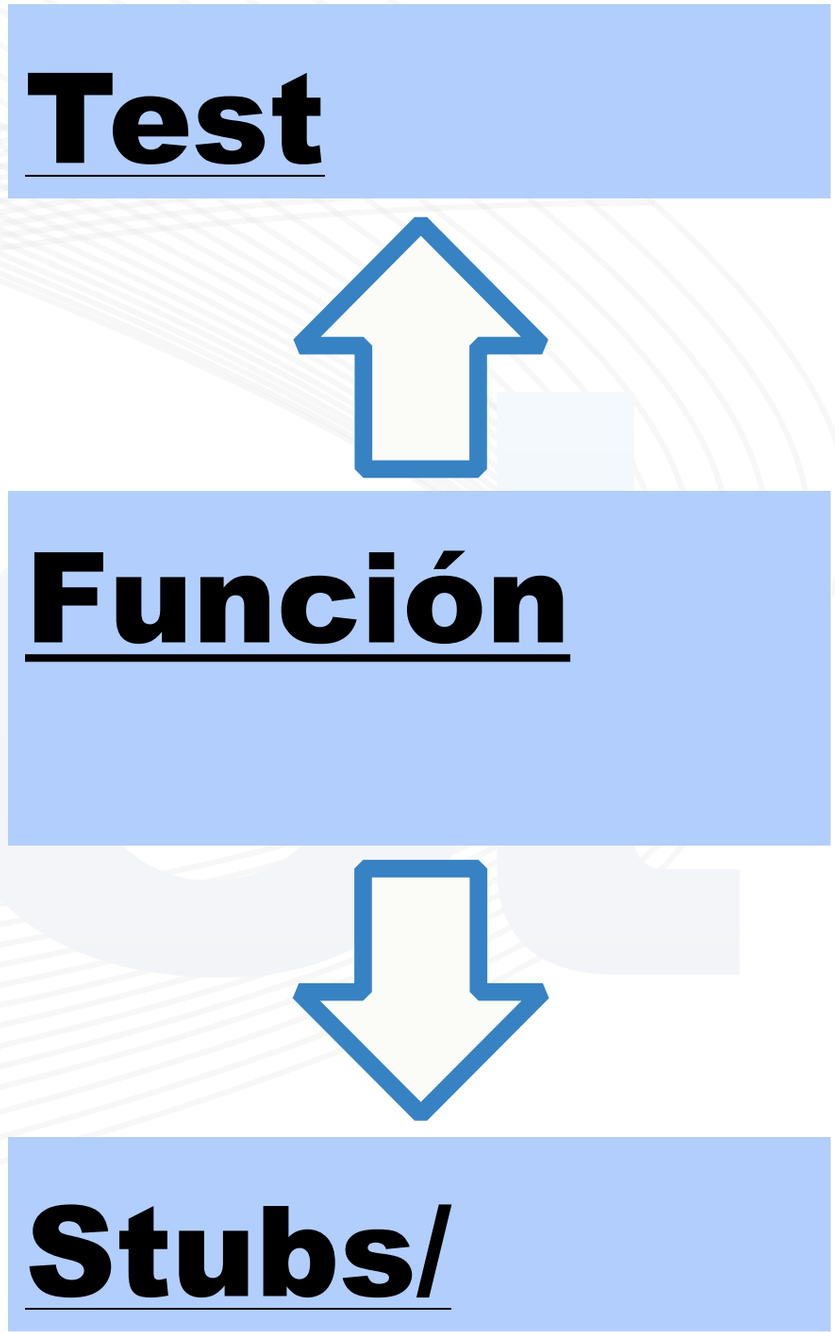
- Git permite definir funciones de usuario que se ejecutan al realizar un commit o un push.
- En los hooks se pueden instalar scripts que revisen o cambien el formato el código.
- Un ejemplo es https://pre-commit.com/
- Esta herramienta tambien guarda la configuración en un archivo dentro del mismo repositorio.

Testing

- Los tests son pruebas dinámicas
  - Verifican el cumplimento de los requisitos
- Pruebas unitarias
  - Verifica un único archivo C o una clase
- Pruebas de modulo
  - Verifica el conjunto de archivos que forma un modulo
- Pruebas del sistema
  - Verifica el sistema completo

Pruebas unitarias

- Prueban un archivo c
- Se prueban únicamente las funciones externas
- Todas las funciones utilizadas de otros archivos utilizadas por el código se reemplazan por funciones Stub o Mock



Pruebas de módulos

- Prueban todo el conjunto de archivos que conforman un modulo
- Se deben reemplazar las funciones provistas por otros módulos por stubs o mocks
- Son muy similares a las pruebas unitarias pero incluyen los posibles problemas de acoplamiento entre los componentes del módulo archivos
- Se utilizan las mismas herramientas que para los test unitarios

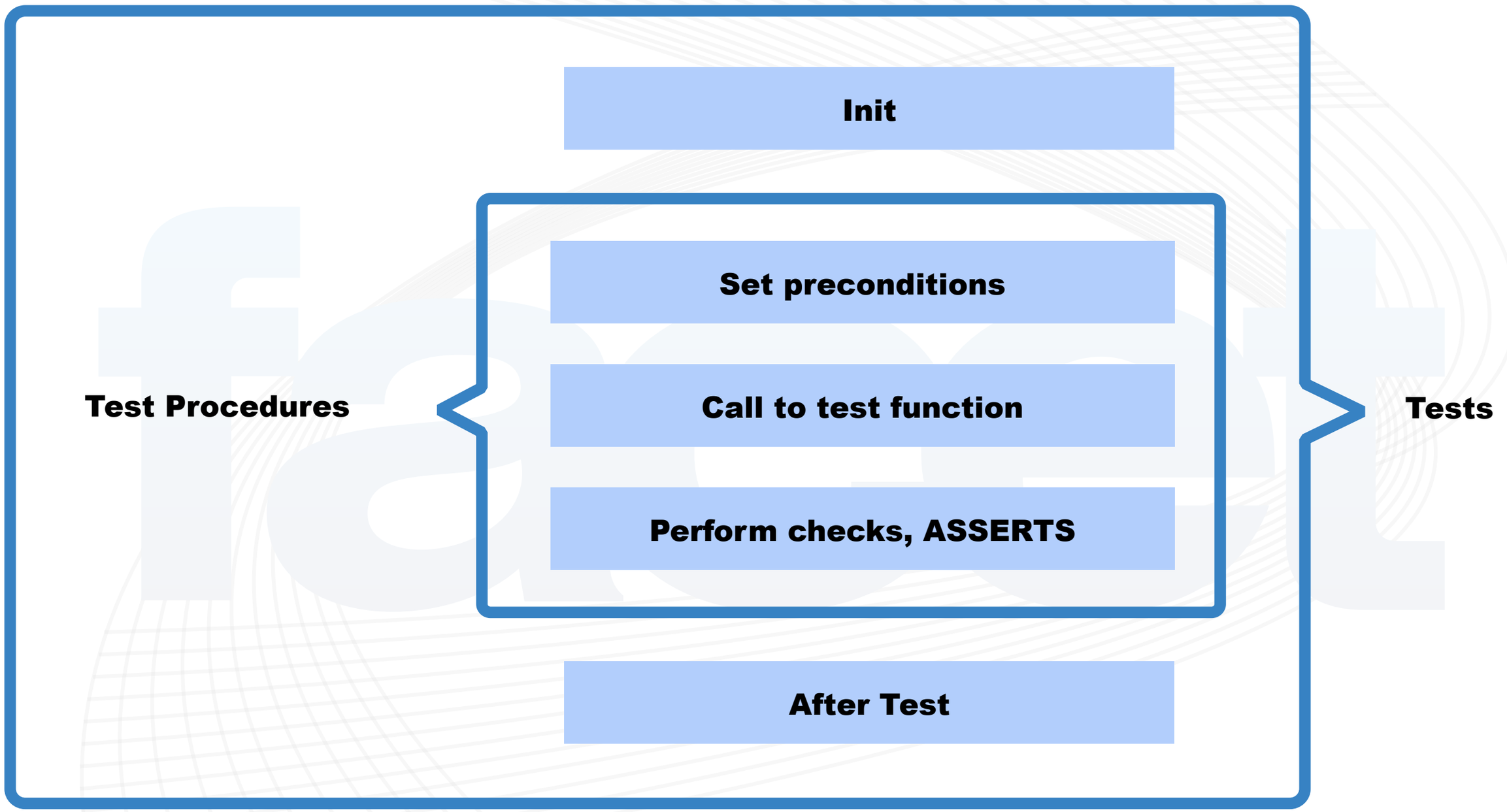
¿Como programamos?

- Obtenemos un sub-conjunto de requerimientos
- Definimos una o mas funciones
  - Definimos los parámetros y el comportamiento
- Escribimos cada una de las funciones
- Escribimos un programa que prueba la función
  - Ingresa valores determinados a la función
  - Comparamos el resultado obtenido con el esperado
- Conservamos la función
- Tiramos el programa de prueba

¿Y si guardamos las pruebas?

- Muchas veces al implementar una nueva funcionalidad rompemos otra
- Si guardamos las pruebas “junto” con la función:
  - Cada vez que algo cambia podemos verificar que lo que antes andaba sigue funcionado
  - Es un test unitario: prueba las funciones publicas de una clase.
- No compilo un programa sino muchos programas:
  - El programa que originalmente quería escribir
  - Uno por cada programa de prueba
- Los frameworks de testing para simplifican el proceso

Estructura de los test



Unity

- Es una librería para testing de código C a nivel de unidades y módulos
- Permite escribir Test muy rápidamente y en una forma muy conceptual
- Se expresan muy claramente los resultados esperados de una operación

```
void test_calcularCRC(void) {
    char data[11] = "0123456789";
    TEST_ASSERT_EQUAL(0x443D, calcularCRC(data, 10));
}
```

CMock

- Es una librería para generar funciones de Stub o de Mock
- Emulan a las funciones externas al archivo que se esta probando
  - Las funciones stub que no hacen nada y solo permiten la compilación del test
  - Las funciones mock son funciones verifican los parámetros y devuelven resultados predefinidos

Ceedling

- Es un framework que trabaja en conjunto con unity y cmock
- Automatiza las tareas de testing
  - Genera automaticamente los mock
  - Ejecuta los test
  - Recopila los resultados
- Automatiza también la creación del makefile

Coverage

- Un factor de calidad del software es el grado de cobertura de los test
- Existen distintas métricas
  - Function coverage
  - Statment coverage
  - Branch coverage
  - Condition coverage
- Cuanto mayor la cobertura de los test menor es la posibilidad de errores

Reportes de cobertura

GCC Code Coverage Report

			Exec	Total	Coverage
Directory:	sources/	**Lines:**	21	21	100.0 %
Date:	2021-07-15 18:22:39	**Branches:**	13	18	72.2 %
Legend:	low: < 75.0 % medium: >= 75.0 % high: >= 90.0 %				

File	Lines			Branches	
suma.c		100.0 %	12 / 12	75.0 %	9 / 12
promedio.c		100.0 %	9 / 9	66.7 %	4 / 6

Generated by: GCOVR (Version 4.2)

Reportes de cobertura

```
54          /* === Definiciones de funciones internas ========================================= */
55
56          /* === Definiciones de funciones externas ========================================= */
57
58      1   int acumular(int * acumulado, int operando) {
59              int resultado;
60              int suma;
61
62      1       suma = *acumulado + operando;
63
64  ✓✗✓✗ 1      if ((*acumulado > 0) && (operando > 0) && (suma < 0)) {
    ✗✓
65                  *acumulado = 0x7FFFFFFF;
66                  resultado = 1;
67  ✗✓✗✗ 1      } else if ((*acumulado < 0) && (operando < 0) && (suma > 0)) {
    ✗✗
68                  *acumulado = 0x80000000;
69                  resultado = 0;
70              } else {
71      1           *acumulado = suma;
72      1           resultado = 0;
73              }
74      1       return resultado;
75          }
76
77          /* === Ciere de documentacion ===================================================== */
78
79          /** @} Final de la definición del modulo para doxygen */
```

Generated by: GCOVR (Version 4.2)

Como funciona ceedling

- Escribimos una función de suma con saturación.
- La probamos de forma tradicional.
- Escribimos nuestro primer test unitario.
- Obtenemos nuestro primer informe de coverage.
- Agregamos pruebas para tener un coverage de 100%.
- Escribimos una función de promedio que usa la suma.
- Creamos nuestro primer Mock.
- Corremos nuestro primer test de modulo.

¿Como programamos?

- Obtenemos un sub-conjunto de requerimientos
- Definimos una o mas funciones
  - Definimos los parámetros y el comportamiento
- Escribimos cada una de las funciones
- Escribimos un programa que prueba la función
  - Ingresa valores determinados a la función
  - Comparamos el resultado obtenido con el esperado
- Conservamos la función
- Tiramos el programa de prueba

¿Y si cambiamos la forma?

- Los requisitos son los que dirigen el desarrollo.
- Los test verifican el cumplimiento de requisitos.
- Si escribimos primero los test y después el programa:
  - Garantizamos el cumplimiento de los requisitos
  - Obligamos a realizar un correcto análisis de requisitos
- TDD: Test Driven Development
  - Primero escribimos los test
  - Después escribimos el código para que cumplan los test

Test Driven Development

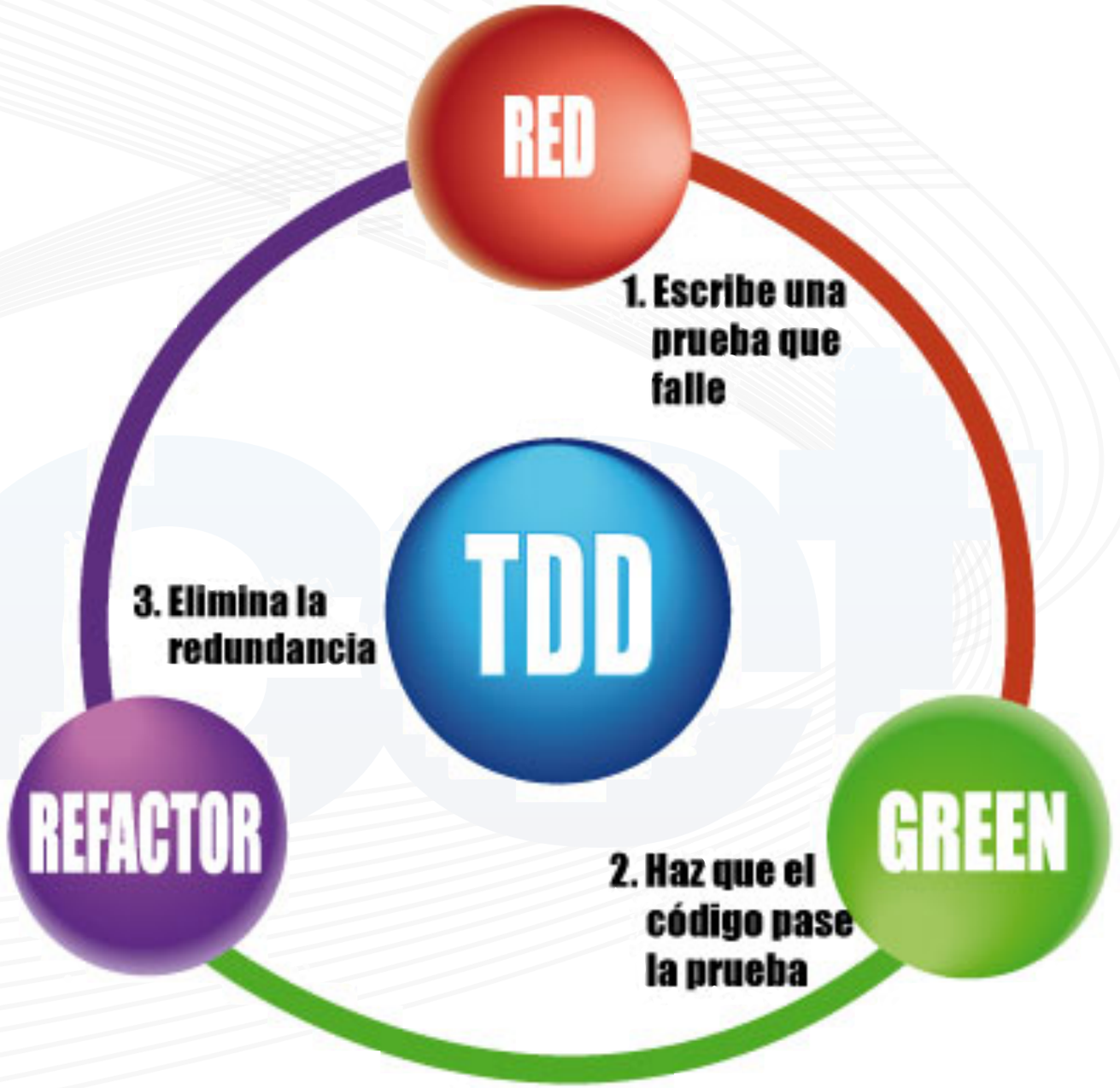
- Mejora la calidad del software desarrollado
- Diseño enfocado a las necesidades
- Diseño mas simple
- Mayor productividad
- Menor tiempo invertido en depuración de errores

Las tres reglas del TDD

- No puede escribir ningún código de producción a menos que sea para hacer pasar una prueba unitaria que falla.
- No escriba más de un prueba unitaria que falle, y una falla de compilación es también una falla.
- No escriba más código de producción que el estrictamente necesario para hacer pasar la prueba unitaria que falla.

El ciclo de desarrollo de TDD

- Escribir una prueba que falle
- Escribir el código de producción mínimo para que la prueba pase
- Mejorar el código de producción para cumplir con los estándares y facilitar el mantenimiento



Biblioteca para el reloj: requisitos

- Gestionar la hora actual en modo de 24 horas con precisión de segundos a partir de un pulso de reloj configurable
- Considerar que la hora del reloj es invalida hasta que el usuario la fije por primera vez
- Configurar una alarma con precisión de horas y minutos
- Habilitar e inhabilitar la alarma sin cambiar la hora
- Posponer la alarma una cantidad arbitraria de minutos
- Las horas se deben manejar en formato BCD sin compactar

Biblioteca para el reloj: pruebas

- Al inicializar el reloj está en 00:00 y con hora invalida.
- Al ajustar la hora el reloj queda en hora y es válida.
- Después de n ciclos de reloj la hora avanza un segundo, diez segundos, un minutos, diez minutos, una hora, diez horas y un día completo.
- Fijar la hora de la alarma y consultarla.
- Fijar la alarma y avanzar el reloj para que suene.
- Fijar la alarma, deshabilitarla y avanzar el reloj para no suene.
- Hacer sonar la alarma y posponerla.
- Hacer sonar la alarma y cancelarla hasta el otro dia..

Algo parece no andar bien

- Siguiendo la metodología al pie de la letra.
  - Tenemos una función mal escrita.
  - Pero tenemos un test que funciona bien.
  - Un test posterior debería detectar el error.
- Al tener un test que detecta un cambio
  - Tenemos un “ancla”que fija el comportamiento del código.
  - Hacemos los cambios en un comportamiento por vez.
- En esencia vamos colocando “referencias” en el comportamiento para ir “llevando” el código por el camino correcto.

FIRST

- Fast: Los test deben correr rápido.
- Isolated: No debe haber dependencia entre los test.
- Repeteable: Se debe poder repetir el test n veces y obtener el mismo resultado.
- Self-Checking: El propio test debe determinar si el resultado es correcto o no.
- Timely: Se deben escribir al mismo tiempo que el código de producción.

En Resumen

- Como se usa en general:
  - Se captura la funcionalidad requerida.
  - Se escriben las pruebas de aceptación del mismo.
  - Se escribe el código para pasar las pruebas.
- En el caso particular de embebidos:
  - Dual targetting
  - Mock del hardware
  - Inyección de dependencias
- Cuidado con los errores comunes
  - Evitar que los test corran despacio
  - No testear detalles de implementación
  - Evitar la dependencia entre los test

Bibliografía

- Test-Driven Development for Embedded C
  - James W. Grenning
- The OLD Object Mentor Blog Site
  - http://butunclebob.com
- Ejemplo simple de testing con ceedling
  - https://bitbucket.org/labmicro/testing
- Ejemplo simple de TDD con leds
  - https://bitbucket.org/labmicro/tdd_leds
- Ejemplo completo de un proyecto con testing
  - https://bitbucket.org/equiser/punku.firmware