VORTEX

EMBEDDED MAKERS

SASE 2017 SIMPOSIO ARGENTINO DE SISTEMAS EMBEBIDOS

Prueba de máquinas de estados

Ing. Leandro Francucci (lf@vortexmakes.com)

9 de Agosto 2017

Objetivo

Presentar estrategias en lenguaje C para la prueba de máquinas de estados planas y anidadas

- Aplicando conceptos como el desacople de módulos, el uso de stub, spy y mock, fases de casos de prueba unitarios, entre otros.
- Utilizando los frameworks Unity y Cmock para las pruebas unitarias.
- Utilizando el modelo de SM, sobre un **proceso de desarrollo evolutivo e incremental dirigido por pruebas**
- En el caso de máquinas de estados anidados complejas, se presentará la resolución mediante el framework RKH.
- Fomentar la alta calidad de nuestro código fuente mediante el uso de modelos y TDD.
- Basado en el artículo "Prueba de máquinas de estados planas y jerárquicas mediante casos de prueba" de EmbeddedExploited

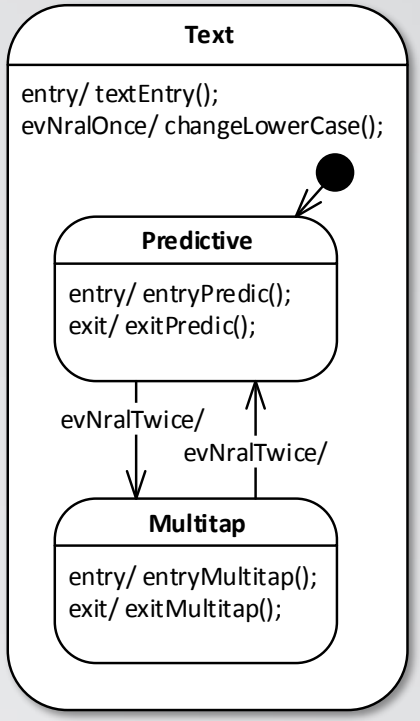
Agenda

- Introducción
  - Estructura, acciones y semántica de una máquina de estados (SM)
  - Fases de un caso de prueba aplicados a una SM
- Estrategia para probar la estructura de una máquina de estados (SM) tradicional y de estados anidados (Statechart)
- Estrategias para probar el comportamiento de una SM tradicional y de estados anidados

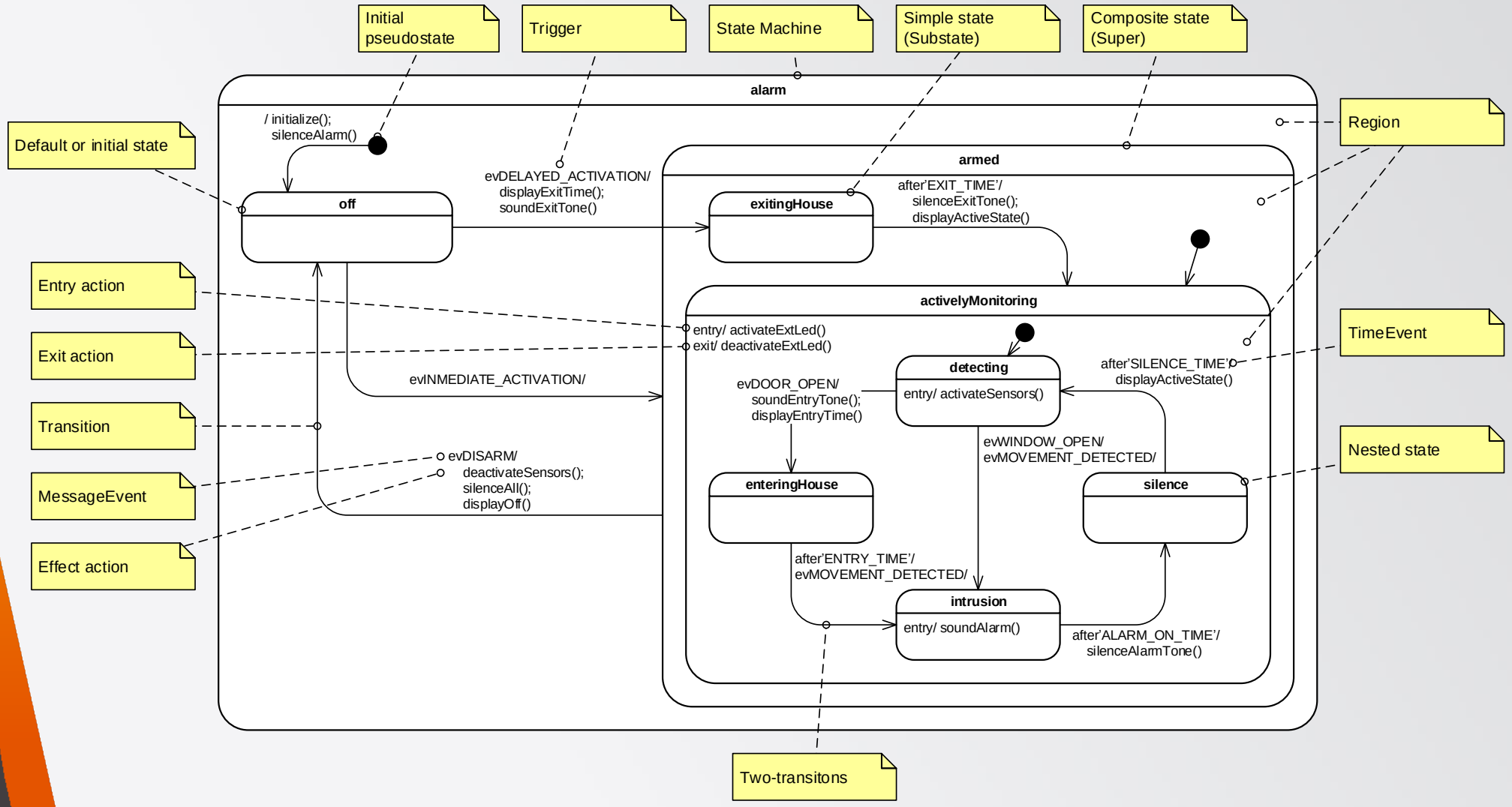
Introducción

Máquina de estados

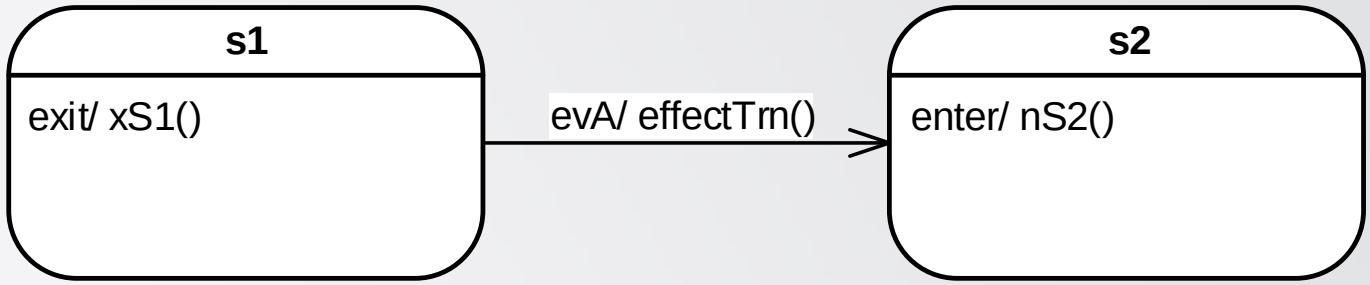
- Se compone de una estructura y sus acciones (o comportamiento)
  - La estructura se constituye por estados, eventos y transiciones
  - Las acciones pueden ser de efecto, entrada y salida de estados, inicialización, evaluación de condiciones, y actividades
- El software que representa una SM implica la representación de:
  - La estructura que respeta la semántica del modelo (Mealy, Moore, Statechart u otra)
  - Las acciones
  - Por lo tanto, ambas cuestiones deben probarse para verificar el correcto funcionamiento de la SM, ya sea en conjunto o de manera independiente
- Su modelo de ejecución es discreto, Run-to-Completion (RTC)



Máquina de estados
Elementos



Máquina de estados
Estrategia de prueba



- ¿Cómo realizaríamos una prueba para determinar que la SMUT (SM Under Test) realiza la transición de estados esperada, es decir: *cuando ocurra el evento* `evA` *en el estado* `s1`*, transite al estado* `s2` *ejecutando ordenadamente las acciones* `xS1()`, `effectTrn()` *y* `nS2()`?

Máquina de estados
Estrategia y fases de una prueba

- De acuerdo con el patrón de prueba de xUnit
  - **Fase de establecimiento** (*setup*): establece las precondiciones a la prueba
  - **Fase de ejercitación** (*exercise*): hace algo contra el sistema
  - **Fase de verificación** *(verify)*: verifica el resultado esperado
  - **Fase de limpieza** (*cleanup*): luego de la prueba vuelve el sistema bajo prueba a su estado inicial
- En principio descomponemos la prueba en partes bien marcadas:
  - las **precondiciones** como el estado origen `s1` y el resultado esperado de la prueba, en este caso el estado destino `s2` y la lista ordenada de acciones a ejecutar `xS1()`, `effectTrn()` y `nS2()`,
  - la **excitación** de la SM, y por supuesto,
  - la **verificación** de la misma.

Probar la estructura de una máquina de estados (SM) tradicional

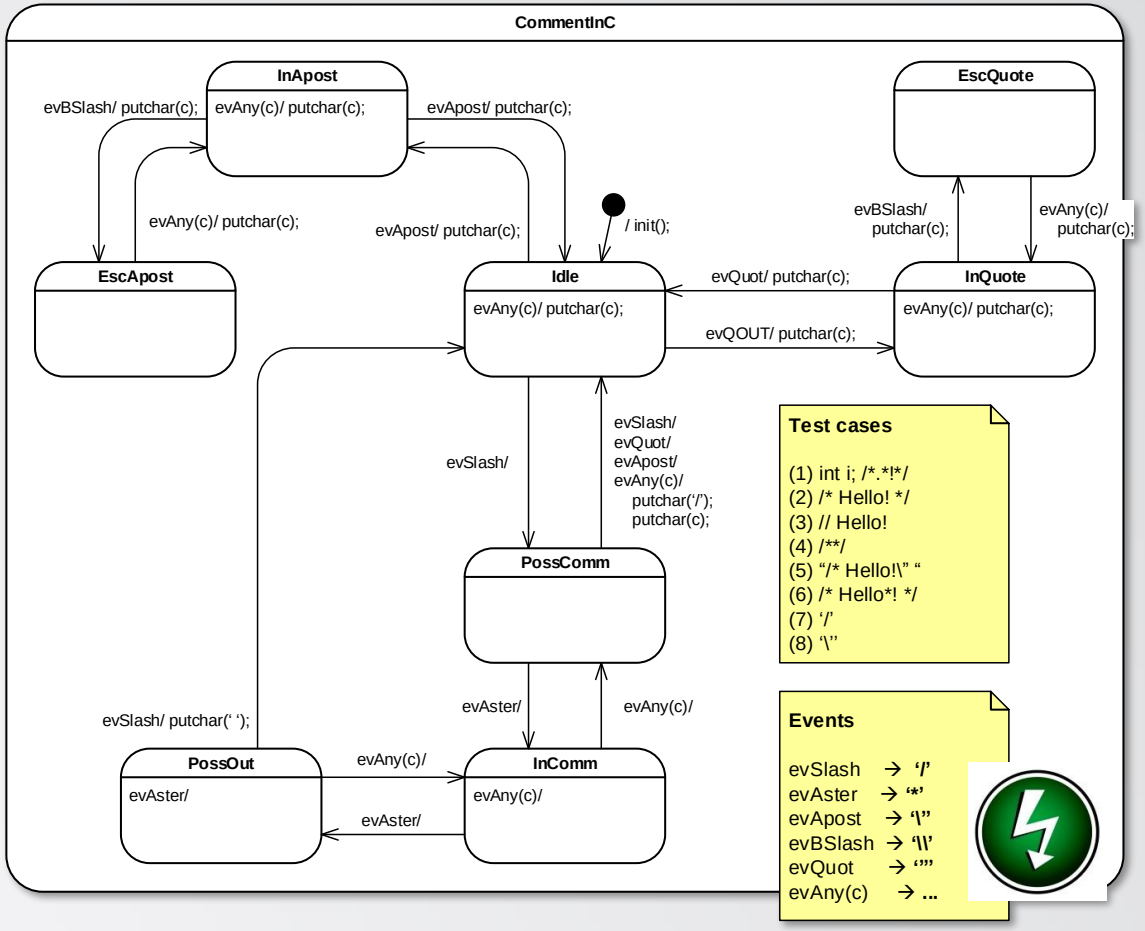
Objetivo y estrategia propuesta

- Verificar la tabla de transición de estados (estructura) de una SM plana (sin anidamiento de estados)
- La estrategia consiste en:
  1. Realizar un caso de prueba para cada estado de la SMUT
  2. estimulándola en el estado en cuestión con los eventos de su alfabeto de entrada (disparadores),
  3. para luego verificar que el estado destino de la transición y sus acciones asociadas, sean los esperados, de acuerdo con su diagrama de estados (o tabla de transición de estados)

Ejemplo de una SM bajo prueba (SMUT)

- Para demostrar la aplicación de las estrategias propuestas, se utiliza un ejemplo simple, la SM `CommentInC`, la cual elimina los comentarios en C de un archivo, basado en el enunciado del ejercicio 1-23 del libro “El lenguaje de programación C” de K&R.

State/Event	evAny(c)	evApost	...
Idle	printChar(c)/Idle	printChar('\'')/InApost	...
PossComm	printSlashAndNextChar()/Idle	printSlashAndApos()/Idle	...
...	...	...	...



Prueba

```
1  /**
2   * \file test_CommentInC.c
3   */
4  ...
5  #include "unity_fixture.h"
6  #include "CommentInC.h"
7  #include "Mock_CommentInCAct.h"
8  ...
9  TEST_GROUP(Structure);
10 ...
11 static void
12 setProfile(int currState, int nextState, int signal, char
13 {
14     CommentInC_setState(currState);
15     event.signal = signal;
16     event.inputChar = evParam;
17     expectedNextState = nextState;
18 }
19
20 TEST_SETUP(Structure)
21 {
22     Mock_CommentInCAct_Init();
23     CommentInC_init();
24 }
25
26 TEST_TEAR_DOWN(Structure)
27 {
28     Mock_CommentInCAct_Verify();
29     Mock_CommentInCAct_Destroy();
30 }
```

```
32 TEST(Structure, DefaultStateAfterInit)
33 {
34     TEST_ASSERT_EQUAL(Idle, CommentInC_getState());
35 }
36
37 TEST(Structure, stateTransitionTableForIdle)
38 {
39     setProfile(Idle, Idle, evAny, 'x');
40     CommentInC_printChar_Expect('x');
41     state = CommentInC_dispatch(&event);
42     TEST_ASSERT_EQUAL(expectedNextState, state);
43
44     setProfile(Idle, InApost, evApost, '\'');
45     CommentInC_printChar_Expect('\'');
46     state = CommentInC_dispatch(&event);
47     TEST_ASSERT_EQUAL(expectedNextState, state);
48
49     ...
50 }
```

Archivo de prueba de CommentInC, test_CommentInC.c

Notas

- Las pruebas están escritas basadas en el framework Unity con el módulo fixture, para lograr una mejor legibilidad, ya que se asemeja al framework CPPUTest, por lo cual se incluye el archivo `unity_fixture.h`.
- El nombre del archivo de prueba de la estructura de la SM adopta la forma `test_<SM>.c`
- Cada caso de prueba comienza con la inicialización de la SM, invocada desde `TEST_SETUP()` del grupo `Structure`, el cual ha sido instanciado mediante `TEST_GROUP()`
- El primer ensayo, `DefaultStateAfterInit`, consiste en verificar el estado por defecto (o inicial) luego de iniciar la SM, en el ejemplo estado `Idle`. En términos de UML o Statechart es la transición emergente de un pseudoestado Initial, o bien transición por defecto de la SM.
- Observar la correlación en los archivos incluidos entre el archivo de producción y su correspondiente archivo de prueba.

Notas

- El caso de prueba `stateTransitionTableForIdle` del grupo X, estimula la SM CommentInC en el estado `Idle`, con todos los eventos de su alfabeto de entrada para luego verificar que el estado destino de la transición y su acción asociada sean las esperadas, de acuerdo con su diagrama de estados (o tabla de transición de estados).
- El alfabeto de entrada también incluye aquellos eventos que no son disparadores en el estado bajo prueba, por lo tanto esta prueba podría dividirse en dos partes, aquella que recibe los disparadores del estado (la cual denominamos "*sunny day*") y otra que recibe el resto de los eventos que no son parte de sus disparadores ("*rainy day*").
- La función de ayuda `setProfile()` establece el estado actual de la SM de acuerdo con el estado bajo prueba, el estado destino esperado, y el evento de entrada.
- El mock de las acciones permite, antes de realizar el ensayo, es decir, estimular la SM, esperar el orden correcto de las llamadas a las acciones, esto se realiza mediante las llamadas a `CommentInC_<action>_Expect()`. Esto se verifica al finalizar cada ensayo, mediante `Mock_CommentInCAct_Verify()`, invocada desde `TEST_TEAR_DOWN()`.

Especificación e implementación

```c
/**
 * \file CommentInC.c
 */
...
#include “CommentInC.h”
#include “CommentInCAct.h”

static int state;
...
int
CommentInC_dispatch(Event *event)
{
    switch (state)
    {
        case Idle:
            switch (event->signal)
            {
                case evAny:
                    CommentInC_printchar(event->inputChar);
                    break;
                case evApost:
                    CommentInC_printchar('\'');
                    state = InApost;
                    break;
                case evQuot:
                    CommentInC_printchar('"');
                    state = InQuote;
                    break;
                ...
            }
            break;
        case PossComm:
          ...
    }
    return state;
}
```

Archivo de implementación, CommentInC.c

```c
/**
 * \file CommentInC.h
 */
#ifdef __COMMENTINC_H__
#define __COMMENTINC_H__

/* States */
enum
{
    Idle, PossComm, InComm, PossOut,
    InApost, EscApost, InQuote, EscQuote
};

/* Event signals */
enum
{
    evAny, evSlash, evAster, evApost,
    evBSlash, evQuot
};

typedef struct Event Event;
struct Event
{
    int signal;
    char inputChar;
};

void CommentInC_init(void);
int CommentInC_dispatch(Event *event);

void CommentInC_setState(int currState);
int CommentInC_getState(void);

#endif
```

Archivo de especificación, CommentInC.h

Notas

- **Primeramente, se recomienda que la implementación de la SM se desarrolle de manera incremental y evolutiva, guiada por las pruebas.**
- El módulo bajo prueba (`CommentInC.c`) no debe utilizar explícitamente especificaciones que provengan de archivos que no hayan sido incluidos explícitamente, porque de lo contrario la dependencia es confusa y compleja de desacoplar.
- La función `CommentInC_dispatch()` despacha un evento a la SM, la cual devuelve el siguiente estado de la transición. Desde esta se invocan las acciones.
- A diferencia del diagrama de estados, el nombre de las funciones que implementan las acciones, posee el prefijo "`CommentInC_`", indicando que pertenecen al módulo CommentInC.
- El evento, representado por el tipo `Event`, se constituye por la señal que efectivamente es parte del alfabeto de entrada (`evAny`, `evSlash`, etc) de la SM y su parámetro, que transporta información asociada con la señal.
- Si bien las funciones `CommentInC_setState()` y `CommentInC_getState()` pertenecen a la especificación de CommentInC, no se utilizan fuera de sus pruebas. Por dicha razón podrían ocultarse mediante un spy de CommetInC.

Desacoplando las acciones

```
1  /**
2   *    \file CommentInCAct.h
3   */
4  #ifdef __COMMENTINCACT_H__
5  #define __COMMENTINCACT_H__
6
7  #include "CommentInC.h"
8
9  /* Effect actions */
10 void CommentInC_printChar(char c);
11 void CommentInC_space(void);
12 ...
13 #endif
```

Archivo de especificación de acciones, CommentInCAct.h

- Para lograr la estrategia de prueba propuesta, es fundamental desacoplar las acciones de la estructura de la SM.
- Independizando así la prueba de sus acciones, y así probar la estructura de la SM de manera unitaria.
- Esto permite cambiar la implementación de las acciones para controlarlas desde la prueba, sin cambio alguno en la estructura que representa la SM.
- Las acciones se concentran en funciones contenidas en un archivo específico, definidas en `<SM>Act.h` e implementadas en `<SM>Act.c`

Mock de las acciones

- Esta estrategia implementa las acciones como un objeto simulado o mock.
- El cual permite determinar desde la prueba no sólo las llamadas a las funciones (acciones) sino también el orden de ejecución de estas.
- En este caso, utilizamos Cmock para generar automáticamente el mock de las acciones a partir de `CommentInCAct.h`
- Utilizando las opciones de configuración de Cmock, a través del archivo `comment.yml`

Probar el comportamiento de una SM tradicional

Alternativa 1
Utilizando un mock de las acciones

```
1  /**
2   * \file test_CommentInCBehavior.c
3   */
4  ...
5  #include "unity_fixture.h"
6  #include "Mock_CommentInCAct.h"
7  ...
8  TEST_SETUP(Behavior)
9  {
10     Mock_CommentInCAct_Init();
11     CommentInC_init();
12 }
13
14 TEST_TEAR_DOWN(Behavior)
15 {
16     Mock_CommentInCAct_Verify();
17     Mock_CommentInCAct_Destroy();
18 }
```

```
20 TEST(Behavior, CommentWithAsterics)
21 {
22     Event inEvents[] = {{evAny, 'i'},
23                         {evAny, 'n'},
24                         {evAny, 't'},
25                         {evAny, ' '},
26                         {evAny, 'i'},
27                         {evAny, ';'},
28                         {evAny, ' '},
29                         {evAny, '/'},
30                         {evAny, '*'},
31                         {evAny, '.'},
32                         {evAny, '*'},
33                         {evAny, '!'},
34                         {evAny, '*'},
35                         {evAny, '/'},
36                         {evAny, 'n'}};
37     Event *pInEvent;
38
39     CommentInC_printChar_Expect('i');
40     CommentInC_printChar_Expect('n');
41     CommentInC_printChar_Expect('t');
42     CommentInC_printChar_Expect(' ');
43     CommentInC_printChar_Expect('i');
44     CommentInC_printChar_Expect(';');
45     CommentInC_printSpace_Expect();
46
47     for (pInEvent = inEvents; ...; ++pInEvent)
48     {
49         CommentInC_dispatch(&(*pInEvent));
50     }
51 }
```

Archivo de prueba del comportamiento de CommentInC, test_CommentInCBehavior.c

- El objetivo es probar, de manera simple, el comportamiento de la SM por medio de un mock del módulo de sus acciones.
- Estimulando la SM con diversos patrones de entrada, los cuales constituyen los casos de prueba, y verificando luego su comportamiento mediante las acciones ejecutadas.
- Por ejemplo, para el patrón "`int i; /*.*!*/`" la salida esperada es "`int i; `"

Notas

- La prueba `CommentWithAsterics` estimula CommentInC con el patrón de entrada `"int i; /*.*!*/"` cuya respuesta se espera a través de las llamadas a las acciones, de acuerdo con el diagrama de estados.
- Más patrones de entrada
  - `int i; /*.*!*/`
  - `/* Hello! */`
  - `// Hello!`
  - `/**/`
  - `"/* Hello!\"`
  - `/* Hello*! */`
  - `'/'`
  - `'\'`

Alternativa 2
Utilizando un spy/stub de las acciones

```c
/**
 * \file test_CommentInC.c
 */
...
#include "unity_fixture.h"
#include "CommentInCActSpy.h"
...
TEST_SETUP(Behavior)
{
    CommentInCActSpy_init();
    CommentInC_init();
}

TEST_TEAR_DOWN(Behavior)
{
}
```

```c
TEST(Behavior, CommentWithAsterics)
{
    const char *expectedOutput = "int i; ";
       Event inEvents[] = {{evAny, 'i'},
                           {evAny, 'n'},
                           {evAny, 't'},
                           {evAny, ' '},
                           {evAny, 'i'},
                           {evAny, ';'},
                           {evAny, ' '},
                           {evAny, '/'},
                           {evAny, '*'},
                           {evAny, '.'},
                           {evAny, '*'},
                           {evAny, '!'},
                           {evAny, '*'},
                           {evAny, '/'},
                           {evAny, 'n'}};
    Event *pInEvent;

    for (pInEvent = inEvents; ...; ++pInEvent)
    {
        CommentInC_dispatch(&(*pInEvent));
    }

    TEST_ASSERT_EQUAL_STRING(expectedOutput, CommentInCSpy
}
```

```c
TEST_ASSERT_EQUAL_STRING(expectedOutput,
                         CommentInCSpy_getBuffer());
```

Archivo de prueba del comportamiento de CommentInC, test_CommentInCBehavior.c

- El objetivo es probar, de manera simple, el comportamiento de la SM, pero esta vez utilizando un spy/stub del módulo de acciones CommentInCAct, en lugar de un mock.
- La estrategia utiliza el stub para resolver fácilmente la realimentación que se necesita para determinar cuál es el resultado de la prueba y así verificarlo con lo esperado.
- Requiere desacoplar las acciones de la estructura de la SM.
- Las pruebas se realizan estimulando la SM con diversos patrones de entrada, los cuales constituyen los casos de prueba, por ejemplo, el patrón “int i; /*.*!*/”, para luego verificar si la salida es la esperada.

Notas

- El stub de las acciones agrega funciones específicas para determinar el comportamiento de la SM, estas se nombran con el prefijo "`CommentInCSpy_`" y las utiliza únicamente la prueba, es decir, son transparentes a la implementación de CommentInC.
- La función `CommentInCSpy_getBuffer()` permite obtener el resultado del procesamiento de CommentInC a un patrón de entrada, para luego compararlo con la salida esperada en `expectedOutput`. Esta función podríamos llamarla "espía".
- Siguiendo la estructura del ensayo `CommentWithAsterics`, las pruebas restantes necesitarán una función de ayuda común, que permita ingresar el patrón de entrada y la salida esperada.

Stub

```c
/**
 * \file CommentInCActSpy.h
 */
#ifdef __COMMENTINCACTSPY_H__
#define __COMMENTINCACTSPY_H__

#include "CommentIncAct.h"

char *CommentInCSpy_getBuffer(void);
Void CommentInCSpy_init(void);

#endif
```

```c
/**
 * \file CommentInCActSpy.c
 */
...
static char buffer[MAX_BUFFER_SIZE];
static char pBuffer;

void
CommentInC_printChar(char c)
{
    if (...) /* is there room? */
    {
        *pBuffer++ = c;
    }
    else
    {
        /* assertion() */
    }
}
...
/* Spy functions */
void
CommentInCSpy_init(void)
{
    pBuffer = buffer;
}

char *
CommentInCSpy_getBuffer(void)
{
    return buffer;
}
```

Archivo spy/stub de CommentInCAct.c, CommentInCActSpy.c

- El stub provee una implementación de las acciones de `CommentInC` de forma tal que pueda verificarse fácilmente su comportamiento a los eventos de entrada.
- Las acciones no imprimen los caracteres sino más bien los almacenan durante toda la prueba.
- Luego, terminado el ensayo, la prueba consulta los caracteres almacenados para determinar si `CommentInC` responde como se espera.

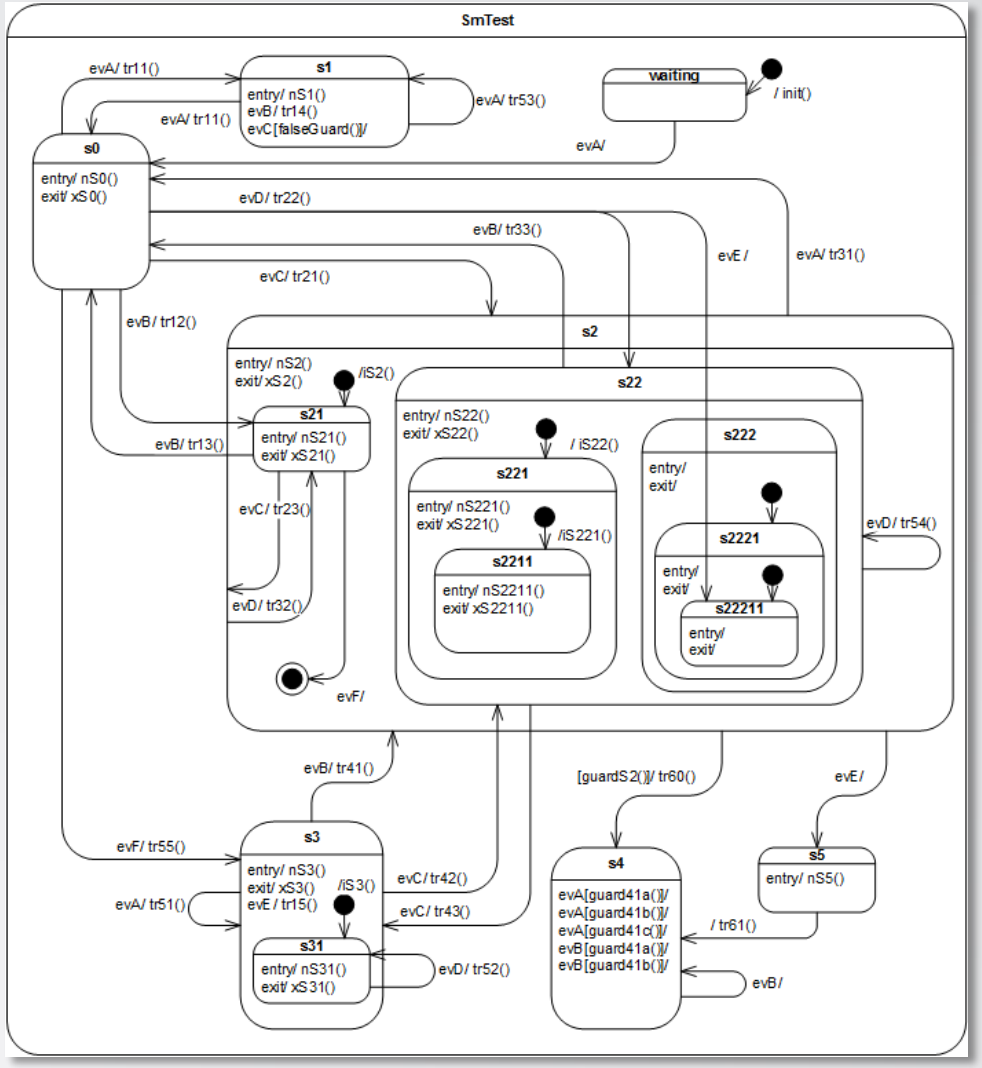
Alternativa 3
Probando las acciones independientemente de la SM

- Si la SM es lo suficientemente compleja, los métodos anteriores pueden no ser apropiados,
- en cuyo caso las acciones podrían probarse de manera independiente a la SM, es decir, fuera del contexto de esta última,
- es decir, las pruebas no utilizarían la función de despacho de eventos `CommentInC_dispatch()`.

Probar la estructura de una SM de estados anidados (Statechart)

Ejemplo de una SMUT jerárquica

- Probar una SM de estados anidados, jerárquica o Statechart es una tarea más compleja que probar una tradicional. Esto se debe a su semántica.
- Una manera particular para probar Statecharts lo explora el framework RKH, que permite representar Statecharts y objetos activos en C/C++.
- RKH utiliza varias estrategias para verificar que la representación de las SM es la esperada.
- Por ejemplo, para corroborar el correcto funcionamiento de todos los tipos de transiciones en una SM jerárquica, RKH utiliza el diagrama de estados mostrado, del cual se asocian una serie de casos de prueba.



Estrategia 1
Utilizando un mock de las acciones

```
1  TEST_SETUP(trnWoutUnitrazer)
2  {
3      Mock_smTestAct_Init();
4      sm_ignore();
5  }
6
7  TEST_TEAR_DOWN(trnWoutUnitrazer)
8  {
9      Mock_smTestAct_Verify();
10     Mock_smTestAct_Destroy();
11 }
12
13 TEST(trnWoutUnitrazer, simpleToCompositeFromHighToLowLevel)
14 {
15     expectedState = RKH_STATE_CAST(&s2211);
16
17     smTest_init_Expect(RKH_CAST(SmTest, smTest));
18     smTest_xS0_Expect(RKH_CAST(SmTest, smTest));
19     smTest_tr22_Expect(RKH_CAST(SmTest, smTest), &evD);
20     smTest_nS2_Expect(RKH_CAST(SmTest, smTest));
21     smTest_nS22_Expect(RKH_CAST(SmTest, smTest));
22     smTest_iS22_Expect(RKH_CAST(SmTest, smTest));
23     smTest_nS221_Expect(RKH_CAST(SmTest, smTest));
24     smTest_iS221_Expect(RKH_CAST(SmTest, smTest));
25     smTest_nS2211_Expect(RKH_CAST(SmTest, smTest));
26     setProfileWoutUnitrazer(smTest,
27                             RKH_STATE_CAST(&s0),
28                             RKH_STATE_CAST(&s0),
29                             expectedState,
30                             INIT_STATE_MACHINE);
31
32     result = rkh_sm_dispatch((RKH_SM_T *)smTest, &evD);
33
34     TEST_ASSERT_TRUE(expectedState == getState(smTest));
35     TEST_ASSERT_EQUAL(RKH_EVT_PROC, result);
36 }
```

Fragmento del caso de prueba SimpleToCompositeFromHighToLowLevel

- Para probar la transición de un estado simple a uno compuesto, donde el estado fuente tiene mayor jerarquía que el estado destino, RKH aplica el caso de prueba `SimpleToCompositeFromHighToLowLevel` del grupo `TrnWoutUnitrazer`
- El cual se basa en el mock de las acciones de la SMUT.
- Con lo cual puede verificarse la estructura de la SMUT **indirectamente**, mediante la ejecución de sus acciones.

Notas

- A diferencia del diagrama de estados SmTest, la implementación agrega a las acciones el prefijo "`smTest_`", indicando que pertenecen al módulo smTest.
- `smTest_<action>_Expect()` pertenecen al mock de las acciones de la SM
- `smTest_tr<x>()` son las acciones de transición
- `smTest_n<state>()` son las acciones de entrada al estado
- `smTest_x<state>()` para las acciones de salida del estado
- `smTest_i<composite_state>()` son las acciones por defecto de los estados compuestos
- La función `setProfileWoutUnitrazer()` inicializa la SM, establece el estado actual y el estado fuente de la prueba, como así también el estado destino esperado
- La función `rkh_sm_dispatch()` despacha un evento a la SM en cuestión
- Finalmente se verifica que:
  - el estado destino sea el esperado,
  - el evento despachado haya sido procesado
  - y por último que se hayan llamado a las acciones esperadas (inicial, transición, entrada y salida) en el orden preestablecido, de acuerdo con el diagrama de estados de SmTest.

Notas

- El resto de los casos de prueba

```
TEST_GROUP_RUNNER(trnWoutUnitrazer)
{
    RUN_TEST_CASE(trnWoutUnitrazer, firstStateAfterInit);
    RUN_TEST_CASE(trnWoutUnitrazer, simpleToSimpleAtEqualLevel);
    RUN_TEST_CASE(trnWoutUnitrazer, simpleToSimpleFromHighToLowLevel);
    RUN_TEST_CASE(trnWoutUnitrazer, simpleToSimpleFromLowToHighLevel);
    RUN_TEST_CASE(trnWoutUnitrazer, simpleToCompositeAtEqualLevel);
    RUN_TEST_CASE(trnWoutUnitrazer, simpleToCompositeFromHighToLowLevel);
    RUN_TEST_CASE(trnWoutUnitrazer, simpleToCompositeFromLowToHighLevel);
    RUN_TEST_CASE(trnWoutUnitrazer, compositeToSimpleAtEqualLevel);
    RUN_TEST_CASE(trnWoutUnitrazer, compositeToSimpleFromHighToLowLevel);
    RUN_TEST_CASE(trnWoutUnitrazer, compositeToSimpleFromLowToHighLevel);
    RUN_TEST_CASE(trnWoutUnitrazer, compositeToCompositeAtEqualLevel);
    RUN_TEST_CASE(trnWoutUnitrazer, compositeToCompositeFromHighToLowLevel);
    RUN_TEST_CASE(trnWoutUnitrazer, compositeToCompositeFromLowToHighLevel);
    RUN_TEST_CASE(trnWoutUnitrazer, loopSimpleStateOnTop);
    RUN_TEST_CASE(trnWoutUnitrazer, loopNestedSimpleState);
    RUN_TEST_CASE(trnWoutUnitrazer, loopCompositeStateOnTop);
    RUN_TEST_CASE(trnWoutUnitrazer, loopNestedCompositeState);
    RUN_TEST_CASE(trnWoutUnitrazer, internalInSimpleState);
    RUN_TEST_CASE(trnWoutUnitrazer, internalInCompositeState);
    RUN_TEST_CASE(trnWoutUnitrazer, fails_EventNotFound);
    RUN_TEST_CASE(trnWoutUnitrazer, fails_GuardFalse);
    RUN_TEST_CASE(trnWoutUnitrazer, fails_ExceededHierarchicalLevel);
    RUN_TEST_CASE(trnWoutUnitrazer, multipleEnabledTrn_FiringFirstTrueGuard);
    RUN_TEST_CASE(trnWoutUnitrazer, multipleEnabledTrn_FiringFirstEmptyGuard);
    RUN_TEST_CASE(trnWoutUnitrazer, defaultTrnWithAssociatedEffect);
    RUN_TEST_CASE(trnWoutUnitrazer, generatedCompletionEventBySimpleState);
    RUN_TEST_CASE(trnWoutUnitrazer, generatedCompletionEventByFinalState);
    RUN_TEST_CASE(trnWoutUnitrazer, syncDispatchingToStateMachine);
}
```

Estrategia 2
Utilizando Tracer

```
TEST(transition, simpleToCompositeFromHighToLowLevel)
{
    UtrzProcessOut *p;
    const RKH_ST_T *targetStates[] =
    {
        RKH_STATE_CAST(&s22), RKH_STATE_CAST(0)
    };
    const RKH_ST_T *entryStates[] =
    {
        RKH_STATE_CAST(&s2), RKH_STATE_CAST(&s22),
        RKH_STATE_CAST(&s221), RKH_STATE_CAST(&s2211),
        RKH_STATE_CAST(0)
    };
    const RKH_ST_T *exitStates[] =
    {
        RKH_STATE_CAST(&s0), RKH_STATE_CAST(0)
    };

    smTest_xS0_Expect(RKH_CAST(SmTest, smTest));
    smTest_tr22_Expect(RKH_CAST(SmTest, smTest), &evD);
    smTest_nS2_Expect(RKH_CAST(SmTest, smTest));
    smTest_nS22_Expect(RKH_CAST(SmTest, smTest));
    smTest_iS22_Expect(RKH_CAST(SmTest, smTest));
    smTest_nS221_Expect(RKH_CAST(SmTest, smTest));
    smTest_iS221_Expect(RKH_CAST(SmTest, smTest));
    smTest_nS2211_Expect(RKH_CAST(SmTest, smTest));

    smTest_init_Expect(RKH_CAST(SmTest, smTest));
    setProfile(smTest,
               RKH_STATE_CAST(&s0),
               RKH_STATE_CAST(&s0),
               targetStates, entryStates, exitStates,
               RKH_STATE_CAST(&s2211), 1,
               TRN_NOT_INTERNAL, INIT_STATE_MACHINE,
               &evD, RKH_STATE_CAST(&s0));

    rkh_sm_dispatch((RKH_SM_T *)smTest, &evD);

    p = unitrazer_getLastOut();
    TEST_ASSERT_EQUAL(UT_PROC_SUCCESS, p->status);
}
```

Fragmento del caso de prueba SimpleToCompositeFromHighToLowLevel

- RKH posee un módulo nativo, Tracer, que permite almacenar diferentes rastros durante la ejecución de una SM.
- Estos contienen información relevante para determinar en tiempo de ejecución:
  - El estado y pseudoestado destino de una transición
  - Las acciones ejecutadas
  - La lista de estados de salida y entrada de una transición
  - El estado inicial
  - Resultado de una transición condicionada
  - Evento no reconocido
  - Excepciones y errores
  - Entre otros
- RKH permite establecer las precondiciones de una prueba, instruyendo que rastros espera obtener y en qué orden, durante el ejercicio de la misma.
- También permite ignorar rastros particulares y argumentos de rastros para flexibilizar las pruebas

Notas

- Utiliza el mock de las acciones de la SM de igual manera que la estrategia anterior.
- La función `setProfile()` establece las precondiciones de la prueba, instruyendo qué **rastros** espera obtener y en qué orden.
- La función `rkh_sm_dispatch()` despacha un evento a la SM en cuestión
- Finalmente, verifica que se cumplan las precondiciones esperadas

Estrategia 2
Estableciendo las precondiciones

```
void
setProfile(…)
{
    int nEntryStates, nExitStates;

    if (initStateMachine)
    {
        sm_init_expect(RKH_STATE_CAST(RKH_SMA_ACCESS_CONST(me, istate)));
        sm_enstate_expect(RKH_STATE_CAST(RKH_SMA_ACCESS_CONST(me, istate)));
    }
    sm_dch_expect(event->e, RKH_STATE_CAST(dispatchCurrentState));
    sm_trn_expect(RKH_STATE_CAST(sourceState), RKH_STATE_CAST(*targetStates));

    if (kindOfTrn == TRN_NOT_INTERNAL)
    {
        executeExpectOnList(targetStates, EXPECT_TS_STATE);
        nExitStates = executeExpectOnList(exitStates, EXPECT_EXSTATE);
    }
    sm_ntrnact_expect(nExecEffectActions, 1);

    if (kindOfTrn == TRN_NOT_INTERNAL)
    {
        nEntryStates = executeExpectOnList(entryStates, EXPECT_ENSTATE);
        sm_nenex_expect(nEntryStates, nExitStates);
        sm_state_expect(RKH_STATE_CAST(mainTargetState));
    }
    sm_evtProc_expect();

    if (initStateMachine)
    {
        rkh_sm_init((RKH_SM_T *)me);
    }
    if (currentState)
    {
        setState(me, RKH_STATE_CAST(currentState));
    }
}
```

Función de ayuda setProfile()

Ejemplo

Control básico sobre la inyección del motor (multipunto)

Requerimientos
De acuerdo con AUTO-ER-0001

[AUTO-ER-0001-REQ0004] El software debe variar el ciclo de trabajo de la señal PWM de 1 KHz de frecuencia, que enviará hacia el inyector en función de los valores leídos de: el sensor de revoluciones por minuto, el sensor de posición de mariposa de admisión, y el sensor de temperatura de motor.

[AUTO-ER-0001-REQ0005] El software debe fijar un ciclo de trabajo de 50% en el PWM que envía hacia el inyector, durante 2 segundos luego de la señal de arranque del motor.

[AUTO-ER-0001-REQ0006] El software debe fijar un ciclo de trabajo mínimo de entre el 20% y el 30% en el PWM que envía hacia el inyector cuando el motor se encuentra “regulando”, para asegurar que el motor genere 2000 RPM +/- 20 RPM.

Requerimientos
De acuerdo con AUTO-ER-0001

[AUTO-ER-0001-REQ0007] El software debe fijar un ciclo de trabajo entre 30% y 80% en el PWM que envía hacia el inyector en función proporcionalmente lineal en que la mariposa de admisión se encuentre abierta. (Nota: El pedal de aceleración del conductor se conecta mecánicamente con la mariposa de admisión. Cuando el pedal se encuentra sin presionar, la mariposa queda un 30% abierta. Cuando se presiona a fondo el pedal, la mariposa se abre un 80%. El ciclo de trabajo del PWM del inyector debe ser linealmente proporcional con el grado de apertura de dicha mariposa).

[AUTO-ER-0001-REQ0008] Si el software detecta que la temperatura del motor es menor a 70C, incrementará en 10% el ciclo de trabajo en el PWM que ha determinado aplicar al inyector.

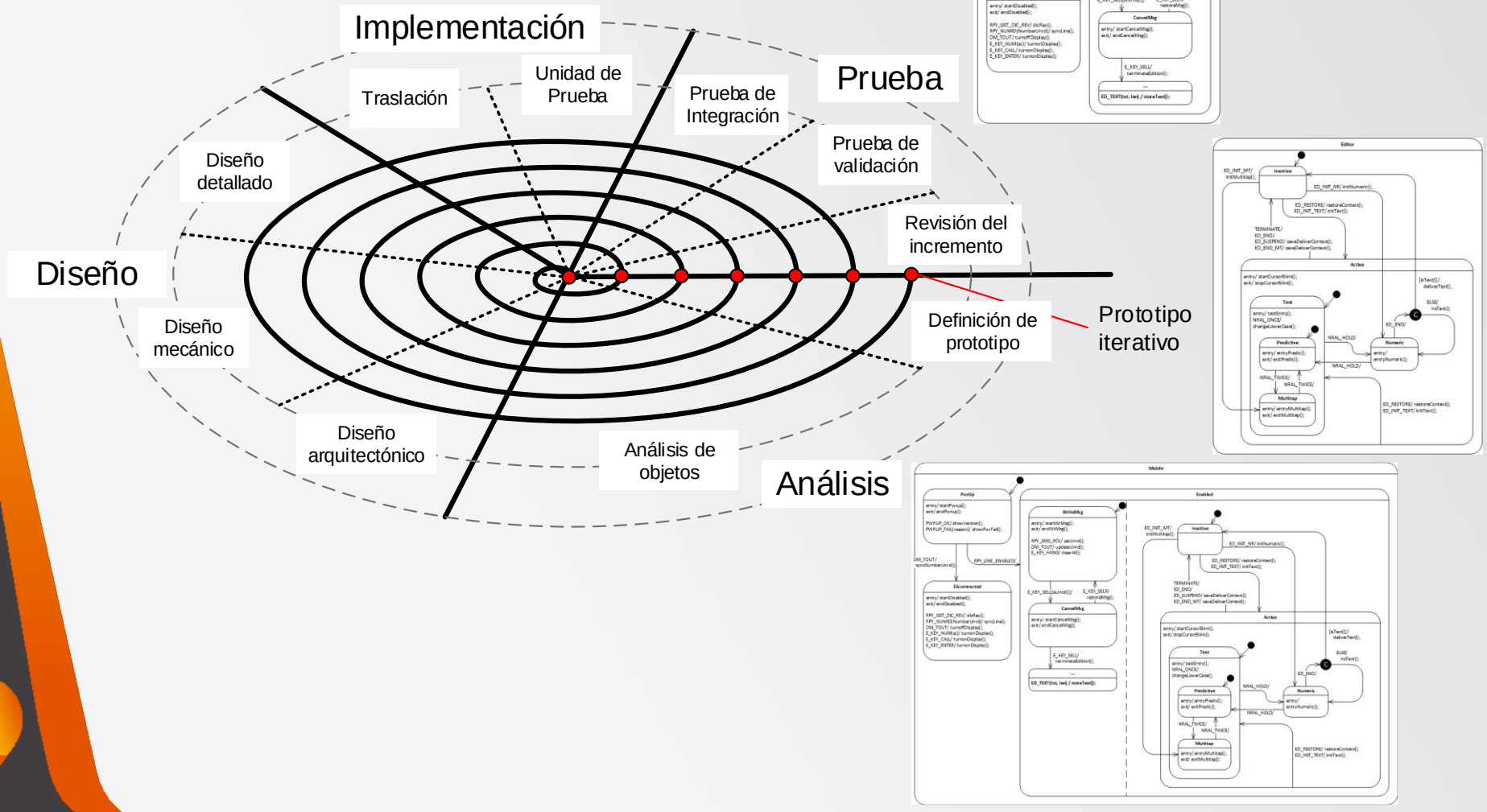
Desarrollo evolutivo e incremental

El objetivo de esta práctica es:

Desarrollar el software que cumple con los requerimientos expuestos, mediante el modelado de software con máquinas de estados y el paradigma del desarrollo guiado por pruebas

- Siendo este del tipo iterativo e incremental, el cual permite al software evolucionar creciendo funcionalidad por funcionalidad durante sus iteraciones.
- El resultado de cada iteración se lo denomina **prototipo iterativo**, siendo este un sistema funcional de alta calidad, parcialmente completo respecto del sistema final. No obstante, implementa y ejecuta correctamente una porción de los requerimientos. Además, contiene el código real que se distribuirá con el producto una vez completo.

Desarrollo evolutivo e incremental



Proceso de desarrollo propuesto

- Para construir una SM se propone realizar una serie de iteraciones incrementales las cuales tienen por tarea:
  - Diseñar un modelo con una SM que cumpla con una única funcionalidad
  - Desarrollar el software que representa el modelo mediante TDD, aplicando las estrategias expuestas
- Iterar hasta cumplir con los correspondientes requerimiento

Preguntas

Referencias

[1] Leandro Francucci, “Prueba de Máquina de estados planas y jerárquicas mediante casos de prueba”, EmbeddedExplited, 2017

[2] James W. Grenning, “Test Driven Development for Embedded C”, PragmaticBookshelf, 2011

[3] Robert C. Martin, “CleanCode”, Prentice Hall, 2009

[4] Kernighan & Ritchie, "C Programming Language (2nd Edition)", April 1, 1988

[5] RKH, “RKH Sourceforge download site,”, http://sourceforge.net/projects/rkh-reactivesys/, August7, 2010

[6] D. Harel, Statecharts: A Visual FormalismforComplexSystems, Sci. Comput. Programming8 (1987), pp. 231–274

[7] Gerard Mezaros, xUnit Test Patterns: Refactoring Test Code, Financial Times Prentice Hall