Testing Software

Pruebas y mantenimiento de software

Lunes de 8:00 a 10:00

en CReCE

Prof. José Antonio Cervantes Alvarez

antonio.alvarez@academicos.udg.mx

Unidad II. Pruebas durante el ciclo de vida del software

2.1 El modelo general en V.

2.2 Pruebas unitarias o de componentes.

2.3 Pruebas de integración.

2.4 Pruebas de sistema.

2.5 Pruebas de aceptación.

2.6 Pruebas para nuevas versiones del producto.

2.7 Tipos genéricos de pruebas.

Objetivo

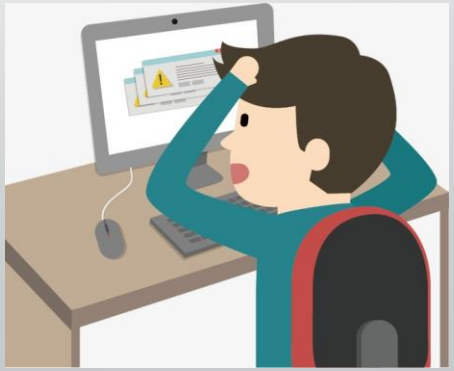
En esta unidad el alumno conocerá a través del modelo en V, el rol del probador de software durante todo el ciclo de vida de un sistema.

Anécdota de dos alumnos

- **Luis y Pedro**
  - Tienen el mismo plazo para terminar un proyecto de programación.
  - La misma edad y experiencia con python.

- **Luis**

  - Produce código rápidamente, en poco tiempo tiene mucho código.
  - Sus compañeros se ponen nervioso porque no tienen tanto código como él.
  - De vez en cuando se asegura que su código compila y se ejecuta.
  - Dos días antes del plazo de entrega ha terminado y ejecuta todo el código para demostrar que funciona.
  - El sistema no funciona y tiene que utilizar un debugger para encontrar el o los errores.
  - ¿Cómo puede ser que esta variable sea cero aquí?
  - Eso es imposible……. @C#J?/!&!
  - Luis entrega un trabajo que presenta fallas…..



- **Pedro**

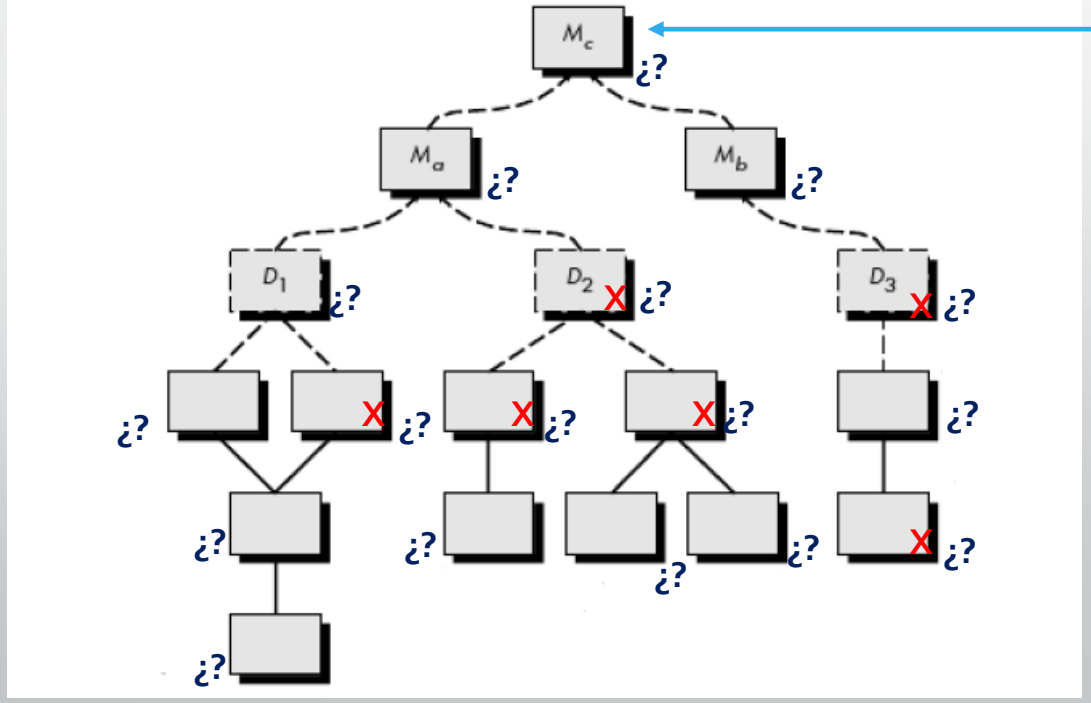
  - No produce código tan rápido.
  - Escribe un test por cada método que programa para ver si el método hace lo que él quiere que haga.
  - Adicionalmente escribe un test por cada clase que va integrando al sistema para ver que la integración sea correcta.
  - Termina su proyecto una noche antes del día de la entrega, y ejecuta todo el código para demostrar que funciona.
  - Se presenta un error pero lo puede arreglar en dos minutos por que los tests le permite saber dónde está el error.
  - Ni si quiera tiene que utilizar el debugger.
  - Entrega un trabajo perfecto.

La diferencian entre Luis y Pedro son las pruebas.

- **Luis**

  - Estaba construyendo un castillo de naipes (error sobre error sobre error)
  - Es difícil encontrar los errores.
  - No podía confiar en su código.

Errores cometidos



Ejecutar la clase principal

El sistema presenta fallas

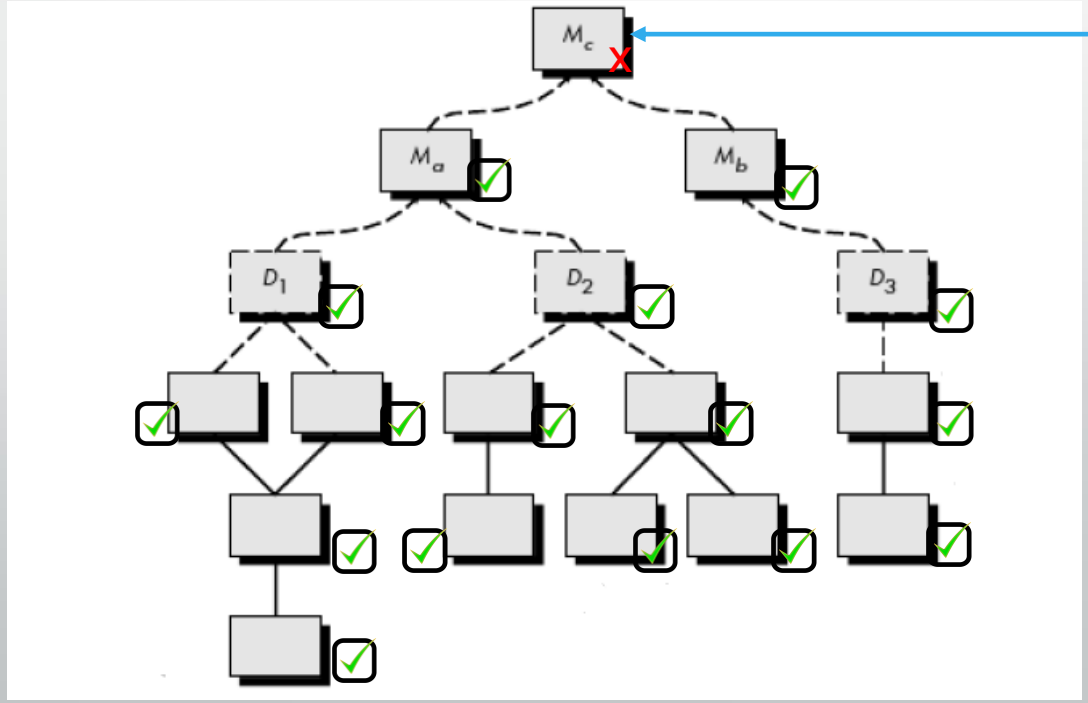
¿De dónde vienen las fallas?

Hay que debuggear todo el sistema

¿Qué es lo que tenemos que hacer para mejorar la calidad del código que escribimos?

- **Pedro**
  - Constantemente se preguntaba ¿qué es lo que hace mi código? ¿hace lo que espero?
  - Construye código nuevo sobre código correcto.
  - Podía confiar en su código.
  - Es muy fácil encontrar errores.

tested



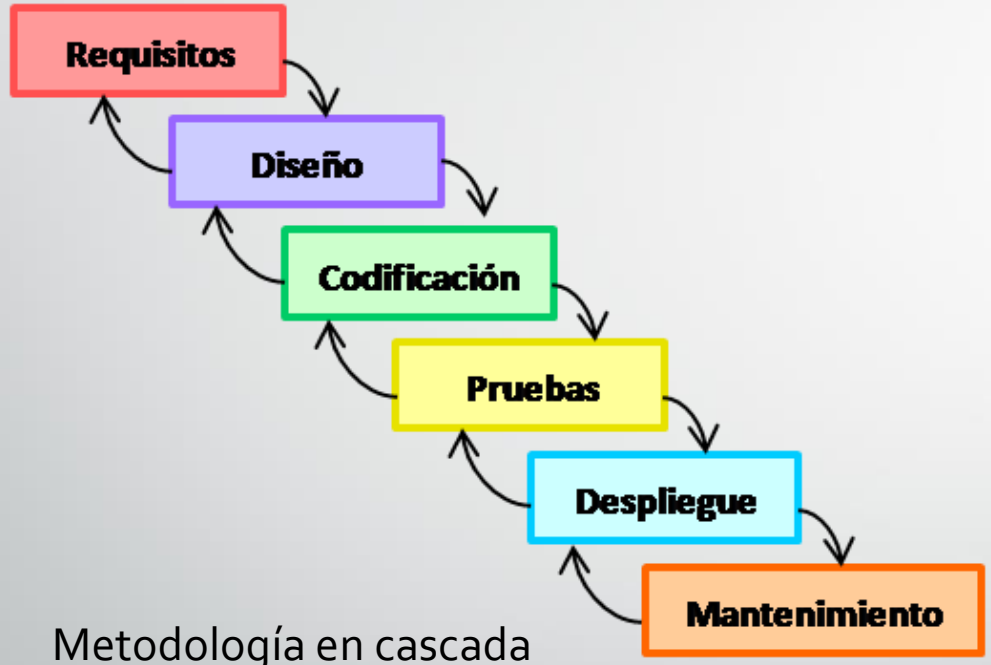
Ejecutar la clase principal

El sistema presenta una falla

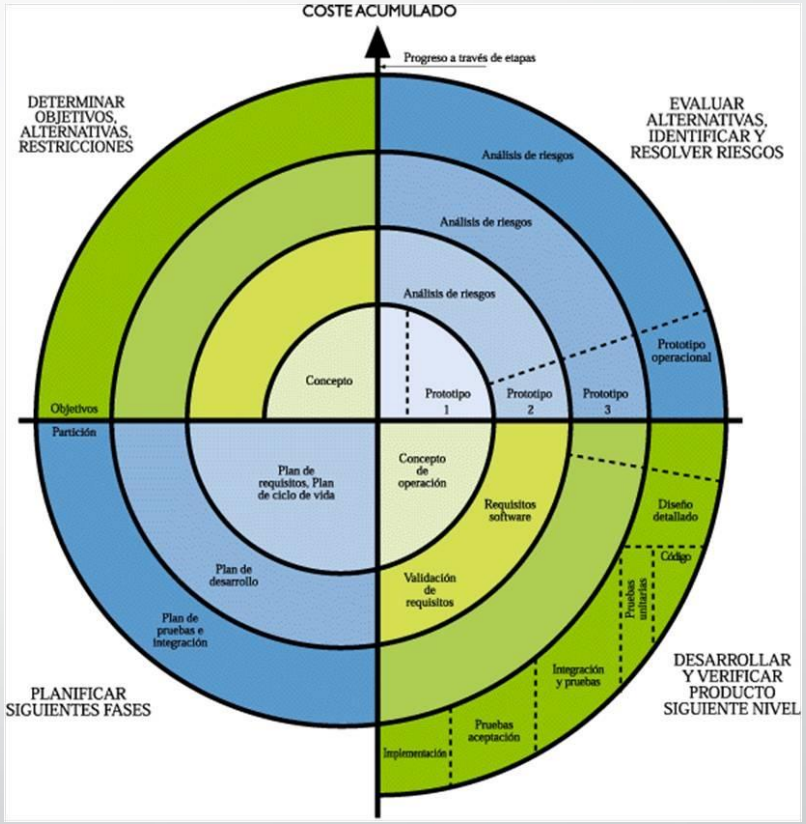
¿De dónde viene la falla?

Rápidamente encuentra el error y lo corrige

Metodologías de desarrollo de Software



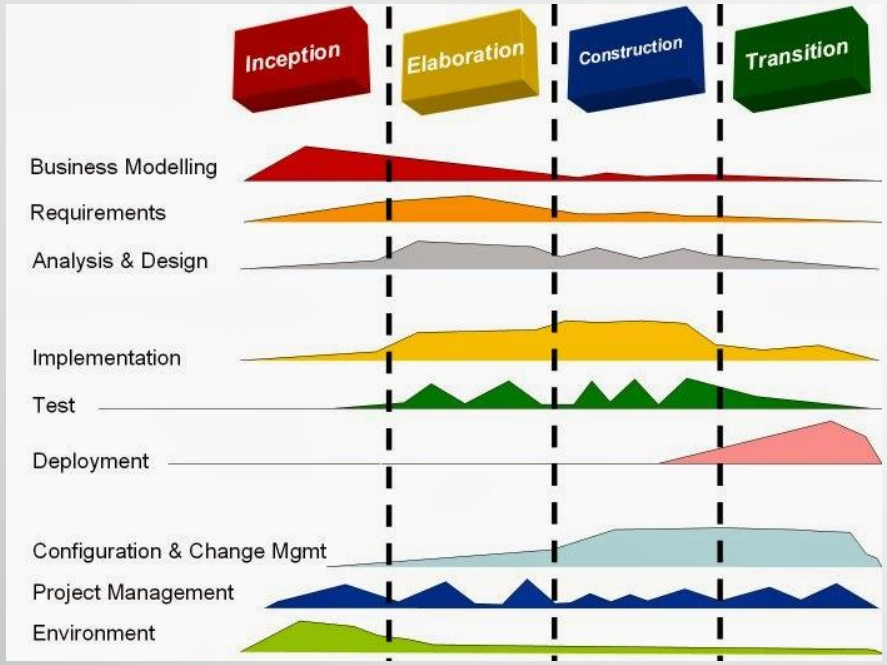
Metodología en cascada



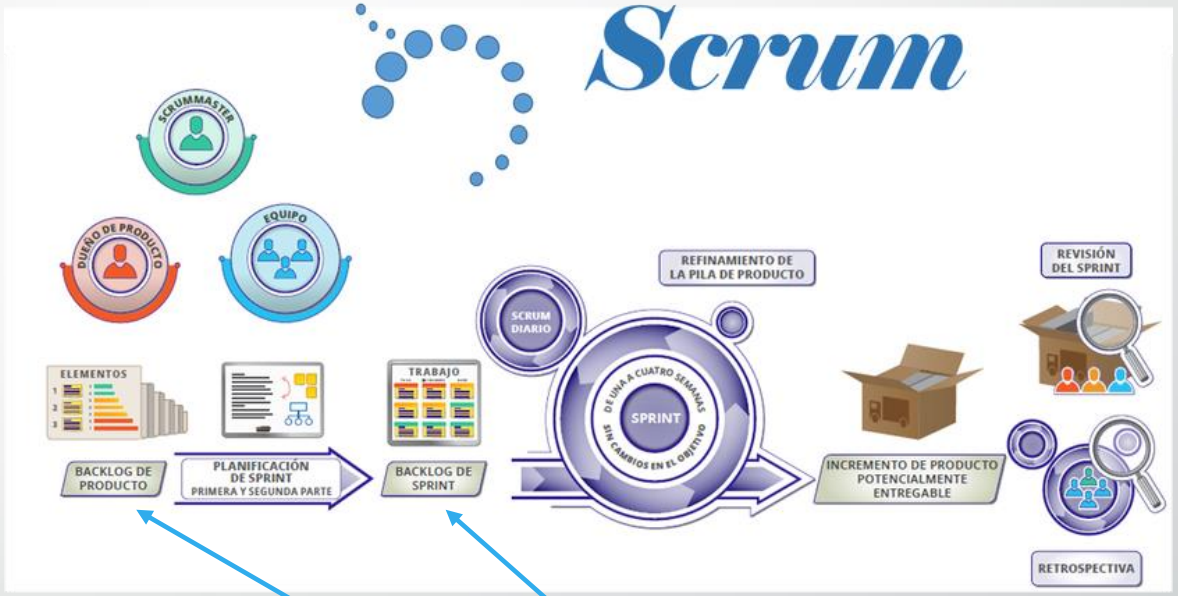
Metodología en espiral

¿En qué momento se realizan las pruebas de software y para qué?

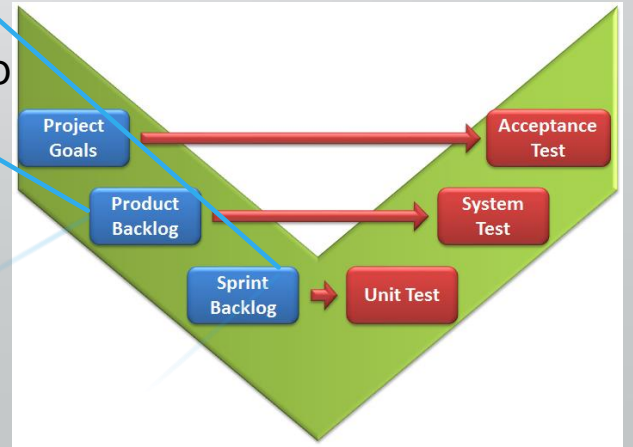
Metodologías de desarrollo de Software



Metodología RUP



Metodo

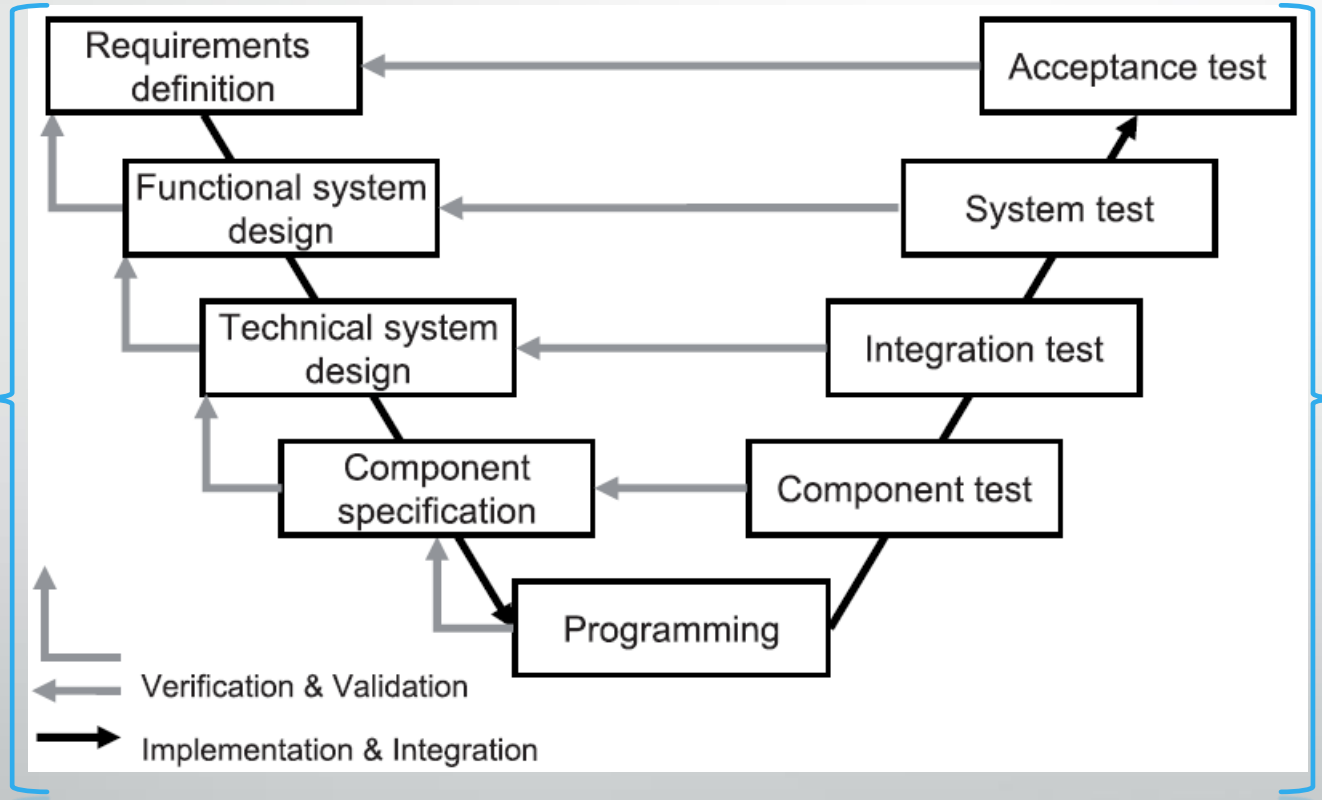


¿En qué momento se realizan las pruebas de software y para qué?

El modelo general en V.

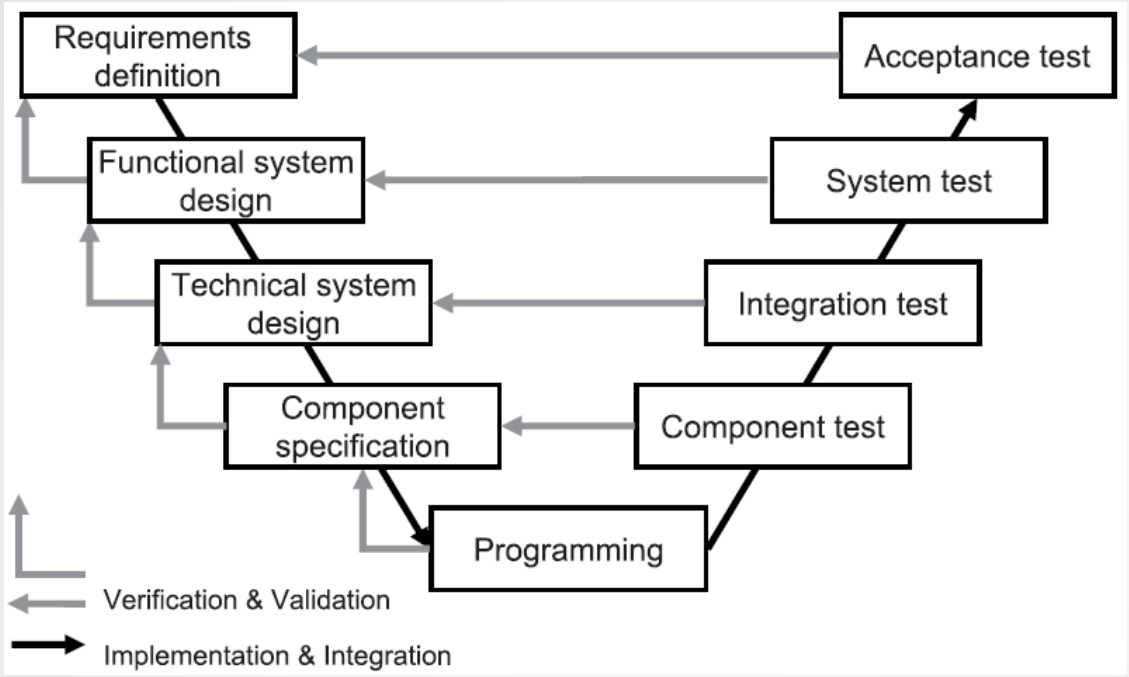
- La principal idea de tras del modelo general en V es que **las actividades de desarrollo y pruebas son actividades de igual importancia.**

Esta parte del modelo representa el proceso de desarrollo. Durante el desarrollo, el sistema es gradualmente diseñado hasta concluir con la programación.

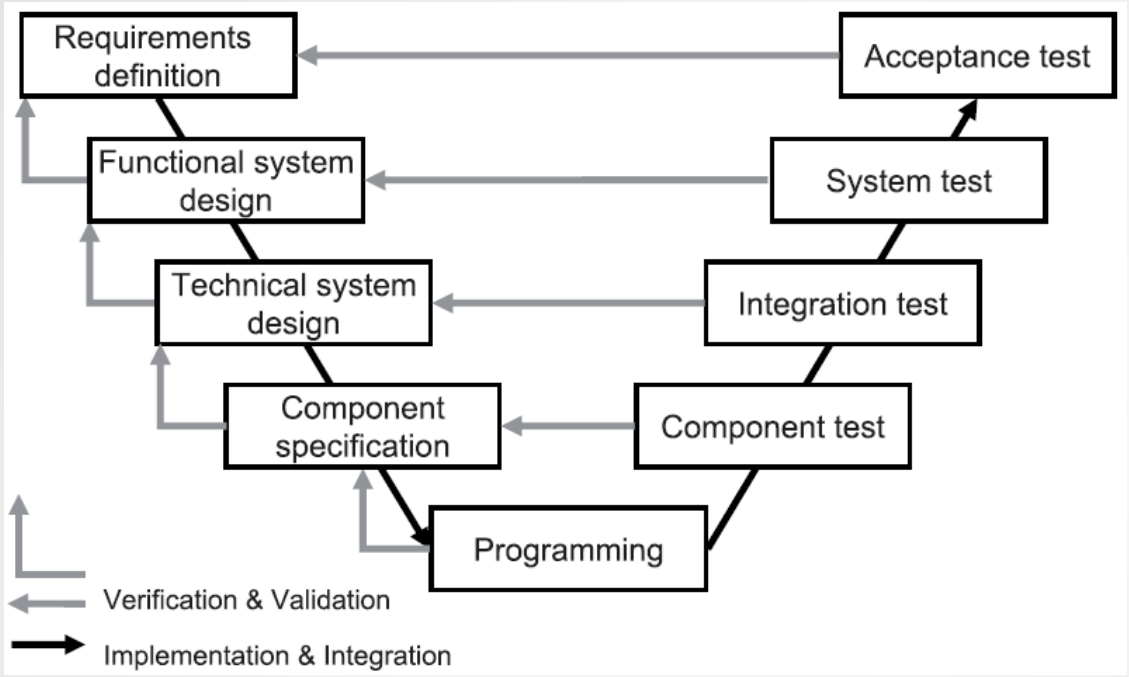


Esta parte del modelo representa el proceso de pruebas e integración. Los componentes del sistema se integran en subsistemas más grandes, mientras que su funcionalidad es probada.

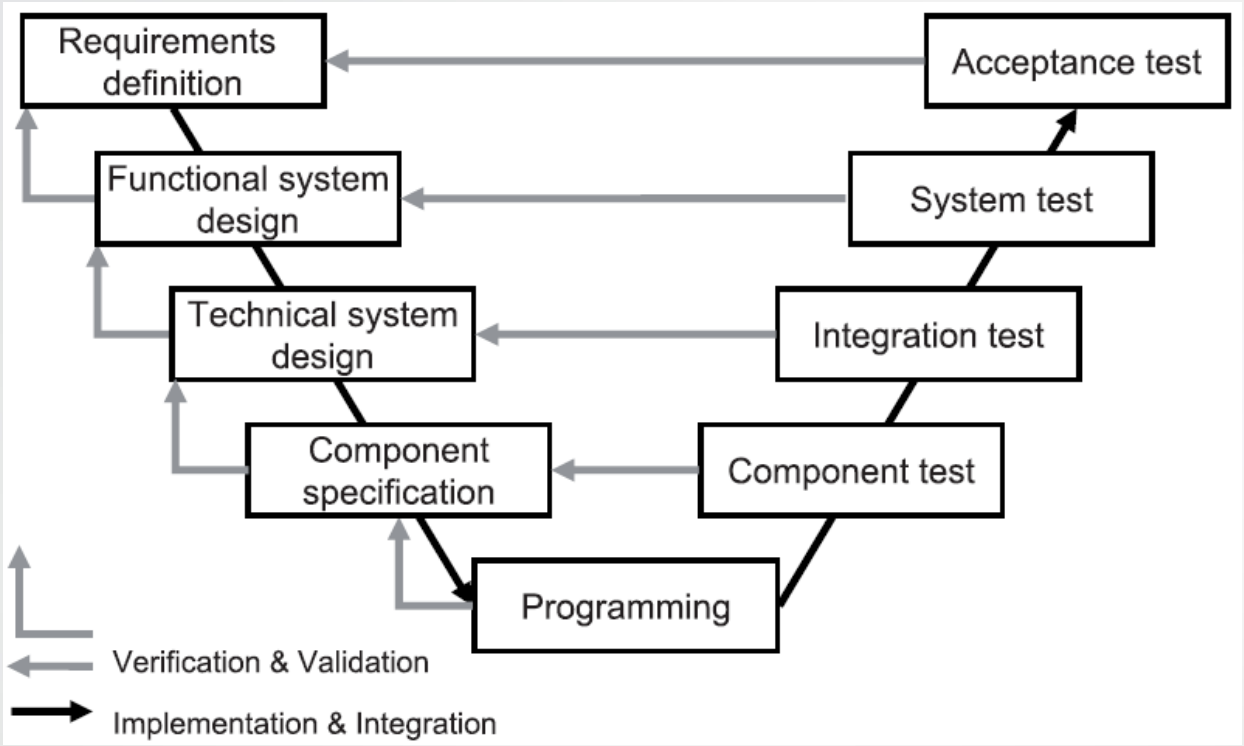
- La integración y las pruebas concluyen cuando las pruebas de aceptación de todo el sistema son completadas.



- Las actividades de la rama izquierda son las actividades definidas en el modelo de cascada.
  - **Definición de requerimientos.** Las necesidades y requerimientos del cliente son reunidas, especificadas y aprobadas.
  - **Diseño del sistema funcional.** Mapeo de los requerimientos sobre las funciones del nuevo sistema.
  - **Diseño del sistema técnico.** Consiste en el diseño de la implementación del sistema. Incluye la definición de interfaces, la descomposición del sistema en pequeños subsistemas.
  - **Especificación de componentes.** Define cada subsistema, incluyendo sus tareas, comportamiento, estructuras internas, e interfaces a otros subsistemas.
  - **Programación.** Cada componente especificado es programado.



- La rama derecha define un nivel de pruebas correspondiente a los niveles de la rama izquierda.
  - **Pruebas de componentes.** Verificar si cada componente del software satisface sus especificaciones.
  - **Pruebas de integración.** Verifica si el grupo de componentes interactúan de forma correcta en base al diseño técnico.
  - **Pruebas de sistema.** Verifica si el sistema (como un todo) reúne los requerimientos especificados.
  - **Pruebas de aceptación.** Verifica si el sistema cumple con los requerimientos del sistema especificados en el contrato.



- Los niveles de prueba mostrados en la rama derecha del modelo en V debe ser interpretados como niveles de la ejecución de pruebas.

- La preparación de pruebas (planeación de pruebas, análisis y diseño de pruebas) debe iniciar en las primeras etapas del ciclo de vida del software. Es decir, estas deben desarrollarse en paralelo a las fases de la rama izquierda del modelo en V.

Pruebas unitarias

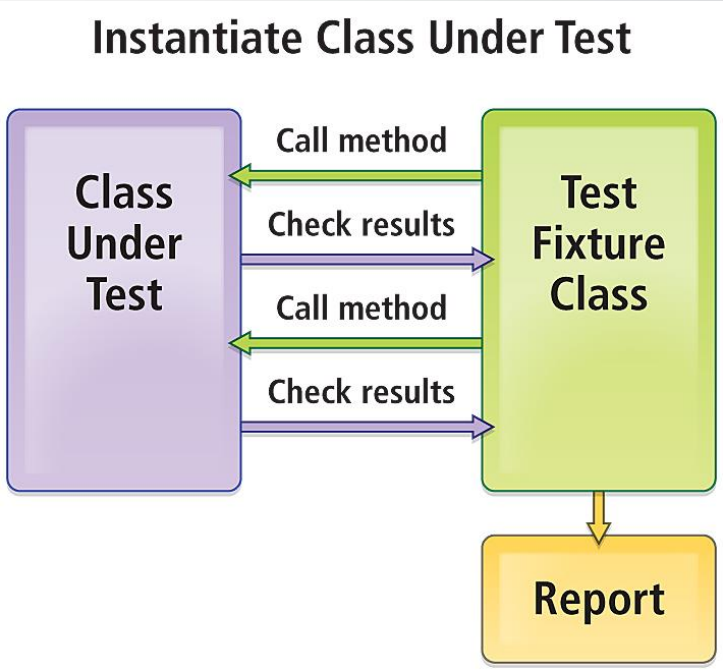
- En el primer nivel de pruebas, los componentes/unidades del software son probados sistemáticamente por primera vez.

- Dependiendo del lenguaje de programación utilizado, las unidades del software pueden ser llamadas por distintos nombres:
  - Módulos.
  - Unidades.
  - Clases.

- Sus pruebas respectivas suelen ser llamadas:
  - Pruebas de módulos.
  - Pruebas unitarias.
  - Pruebas de clases.

Pruebas unitarias

- La prueba unitaria se basan en los requerimientos del componente y su diseño.
  - Casos de prueba de caja blanca pueden ser desarrollados.
  - El código puede ser analizado.
  - El comportamiento del componente debe ser comparado con el comportamiento especificado.



Objetos de prueba

- Los objetos de prueba típicos son:
  - Módulos/unidades.
  - Clases.
  - Scripts.
  - Base de datos.
  - Otros componentes del software.
- La principal característica de las pruebas unitarias es que los componentes son probados individualmente y de manera aislada de los otros componentes del sistema.
- La ventaja de probar los componentes de manera aislada es que evita la influencia de otros componentes. Si la prueba detecta un problema, definitivamente el origen del problema se encuentra en el componente bajo prueba.

Entorno de las pruebas unitarias

- Las pruebas unitarias son el nivel de pruebas más bajo. Es obvio que en este nivel de pruebas debe existir una estrecha cooperación con el desarrollador para trabajar con los objetos de prueba.

- Es recomendable utilizar algún *framework* para reducir el esfuerzo requerido en la programación de las pruebas dirigidas. También ayuda a estandarizar la arquitectura de pruebas de los componentes del proyecto.

CppUnit

Ejemplo con python y unit

```python
# -*- coding: utf-8 -*-
"""
Created on Mon Sep 21 14:22:12 2020

@author: José Antonio cervantes
"""

class Calculadora:

    def suma(self,a,b):
        for n in (a,b):
            if not isinstance(n, int) and not isinstance(n, float):
                raise TypeError("parameter values must be int or float")
        return a+b

    def resta(self,a,b):
        for n in (a,b):
            if not isinstance(n, int) and not isinstance(n, float):
                raise TypeError("parameter values must be int or float")
        return a-b

    def devision(self,a,b):
        for n in (a,b):
            if not isinstance(n, int) and not isinstance(n, float):
                raise TypeError("parameter values must be int or float")
        return a/b

    def mutiplicación(self,a,b):
        for n in (a,b):
            if not isinstance(n, int) and not isinstance(n, float):
                raise TypeError("parameter values must be int or float")
        return a*b

    def exponenteX(self,a,b):
        for n in (a,b):
            if not isinstance(n, int) and not isinstance(n, float):
                raise TypeError("parameter values must be int or float")
        return a**b
```

```python
# -*- coding: utf-8 -*-
"""
Created on Mon Sep 21 14:29:15 2020

@author: josé Antonio Cervantes
"""

import unittest
from calculadora import Calculadora

class TestCalculadora(unittest.TestCase):

    def setUp(self):
        self.calc = Calculadora()

    def test_suma(self):
        self.assertEqual(self.calc.suma(7,5),12)

    def test_resta(self):
        self.assertEqual(self.calc.resta(7,5),2)

    def test_division(self):
        self.assertEqual(self.calc.devision(5,2),2.5)

    def test_multiplicacion(self):
        self.assertEqual(self.calc.mutiplicación(7,5),35)

    def test_exponenteX(self):
        self.assertEqual(self.calc.exponenteX(2,5),32)


if __name__ == "__main__":
    unittest.main()
```

Ejemplo. Probar un método de una clase.

En el subsistema *DreamCar* del sistema VSR, la especificación para calcular el precio de un vehículo está dada por:

- El punto de partida es *baseprice* menos *discount*, donde *baseprice* es el precio base general de un vehículo y *discount* es el descuento aprobado por el distribuidor.
- Un precio para un modelo especial (specialprice) y el precio para el equipamiento adicional (extraprice) pueden ser agregados.
- Si 3 o más equipos adicionales son seleccionados. Habrá un descuento del 10% sobre el precio de los componentes adicionales. Pero si son 5 o más elementos adicionales, el descuento será del 15%.
- El descuento otorgado por el distribuidor se aplica sólo al precio base (*baseprice*). Mientras que el descuento sobre los componentes adicionales aplica sólo sobre esos componentes.

La siguiente función calcula el precio total:

```
def calcularPrecio(baseprice,specialprice,extraprice,extras,discount):
    if extras >= 3:
        addon_discount = 10
    elif extras >= 5:
        addon_discount = 15
    else:
        addon_discount = 0
    result = baseprice/100.0*(100 - discount) + specialprice + extraprice/100.0*(100 - addon_discount)
    return result
```

¿El error fue encontrado con las pruebas?

Ejemplo de un test

```
bool test_calculate_price() {
  double price;
  bool test_ok = TRUE;

  // testcase 01
  price = calculate_price(10000.00,2000.00,1000.00,3,0);
  test_ok = test_ok && (abs (price-12900.00) < 0.01);

  // testcase 02
  price = calculate_price(25500.00,3450.00,6000.00,6,0);
  test_ok = test_ok && (abs (price-34050.00) < 0.01);

  // testcase ...

  // test result
  return test_ok;
}
```

En este ejemplo podría no aplicar. Sin embargo, es importante considerar que un valor flotante no debería ser directamente comparado, porque podrían existir imprecisiones de redondeo. Puede ser que se generen resultados aceptables con un margen de error despreciable. Esto esta en fusión al tipo de sistema y la precisión que se busca obtener.

Desglosar el IVA de un producto:

Valor sin IVA = Costo total / 1.16

Costo total = 100

Costo sin IVA = 100/1.16 = **86.2068966**

IVA = 13.7931034

Costo total = 86.2068966 + 13.7931034 = 100

Sin embargo, para cuestiones prácticas los sistemas de cobro sólo muestran dos dígitos después del punto. ¿Cómo debería ser el redondeo?

86.2068966 → 86.21
86.2068966 → 86.20

13.7931034 → 13.79

Objetivo de las pruebas unitarias

- El objetivo principal de las pruebas unitarias es probar la funcionalidad del objeto de pruebas. Verificando que funcione correctamente en base a las especificaciones de requerimientos.

- **En este tipo de pruebas la funcionalidad se refiere al comportamiento que tiene el objeto de pruebas con las entradas y salidas.**

- Para comprobar la exactitud y completitud de la implementación, el componente debe ser probado con una serie de casos de prueba, donde cada caso de prueba debe cubrir una combinación particular de entradas y salidas.

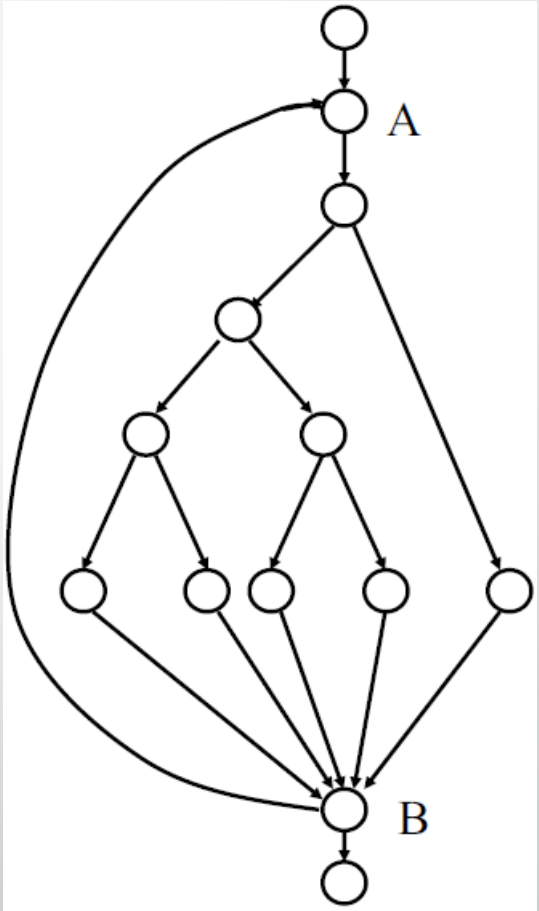
Estrategia de pruebas unitarias

- Las pruebas unitarias están muy relacionadas con el desarrollador. Usualmente el probador (tester) tiene acceso al código fuente de los componentes. Esto permite crear pruebas de caja blanca.

- El probador debe desarrollar casos de prueba en base a su conocimiento de la programación del componente bajo prueba. Es decir, en base a la estructura del programa, funciones y variables.

- Con la ayuda de algunas herramientas (debugger) es posible observar el contenido de las variables durante la ejecución de las pruebas.

- Es importante que las pruebas permitan ejecutar los distintos caminos de flujo que puede seguir un programa. En el grafo se puede observar que del punto A al B existen 5 posibles caminos a seguir.



Estrategia de pruebas unitarias

- Por lo general, los sistemas reales están formados por un gran número de componentes (funciones). Esto hace imposible poder analizar todo el código para realizar casos de estudio para todos los componentes.

- Es importante considerar que algunos componentes básicos pueden ser integrados en otros componentes principales.

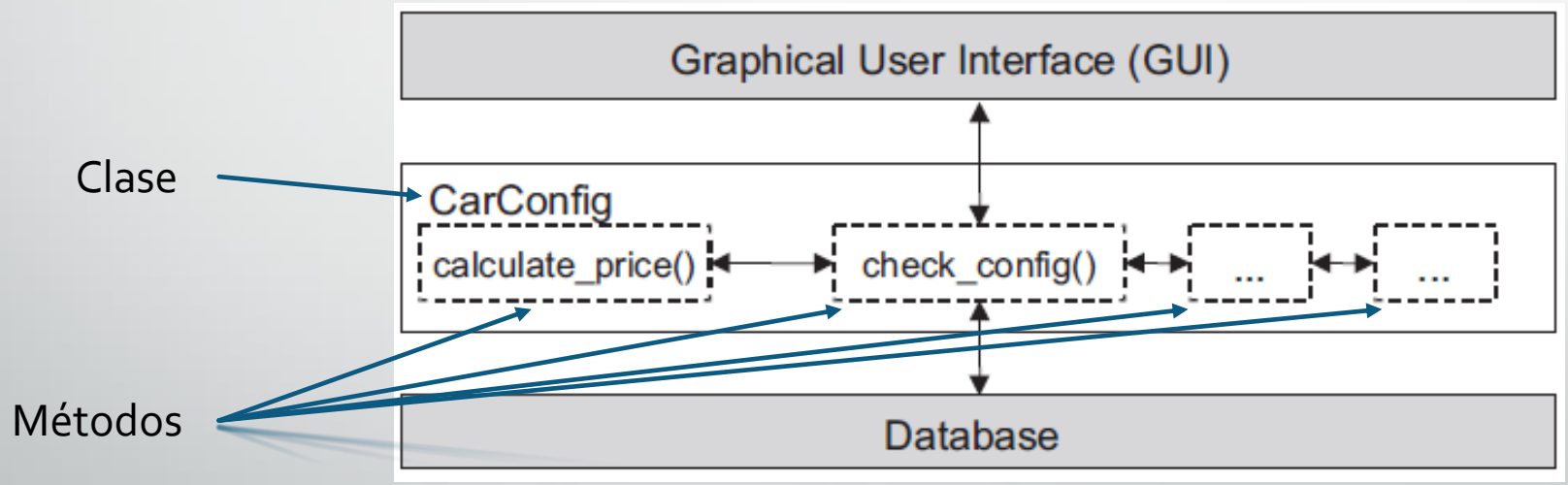
- Por lo general el probador se debe enfocar a probar los componentes principales. Los cuales generalmente hacen uso de otros componentes básicos.

Pruebas de integración

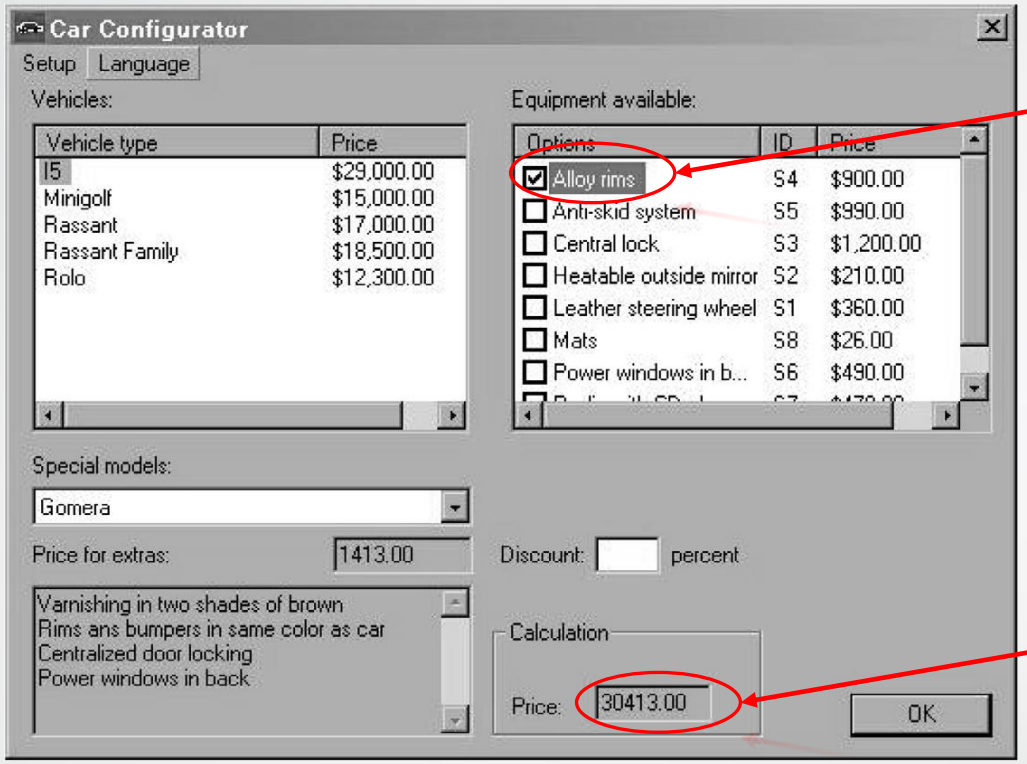
- Las pruebas de integración son el segundo nivel de pruebas en el modelo en V.
- **Una precondición para el desarrollo de las pruebas de integración es que los objetos de prueba (componentes) hayan sido probados.**
- Desarrolladores, probadores o especialistas del grupo de integración agrupan estos componentes en unidades estructurales más grandes o subsistemas. ***Esta conexión de componentes es denominada integración.***
- Las pruebas de integración son utilizadas para probar las unidades estructurales o subsistemas con la finalidad de asegurar que todos los componentes colaboran de forma correcta.
- **Las base de las pruebas de integración puede ser el diseño del sistema, la arquitectura del sistema o el flujo de trabajo a través de distintas interfaces y casos de uso.**

¿Por qué son necesarias las pruebas de integración si cada componente ha sido probado de manera individual?

- Las pruebas de integración buscan aislar las causa de problemas relacionados con la colaboración e interoperabilidad.



Componentes básicos del subsistema *DreamCar*.



El sistema no está incluyendo el costo del accesorio.

Falla del Sistema al calcular el precio

Ejemplo de la interfaz grafica para el usuario.

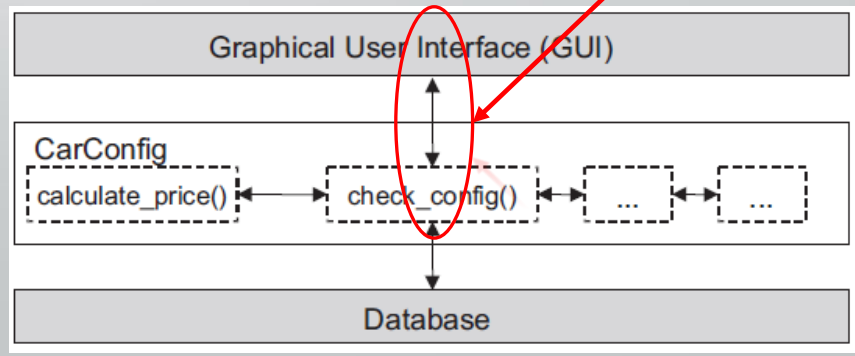
Defecto entre la interfaz y el método *check_config*.

En este ejemplo, el precio final debería ser $31,313 dado que:

Tipo de vehículo I5 → $29,000.00

Precio por extras → $1,413.00

Equipamiento disponible → $900.00



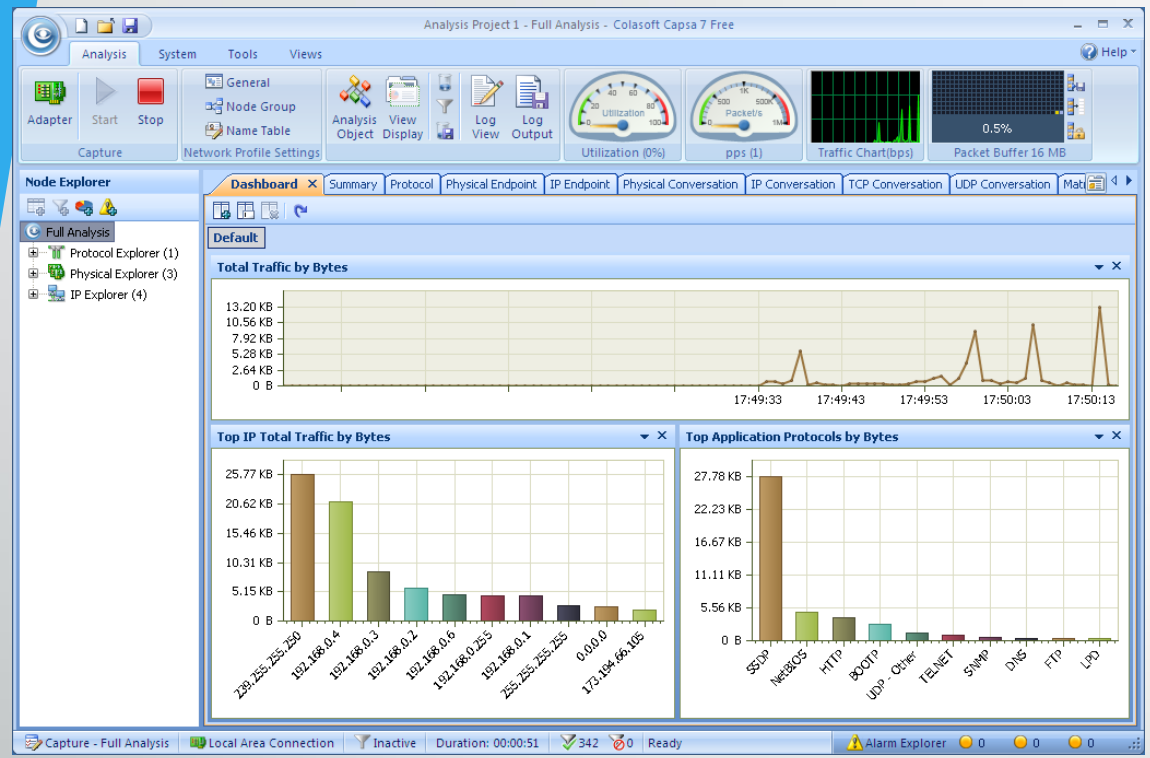
Objetos de prueba

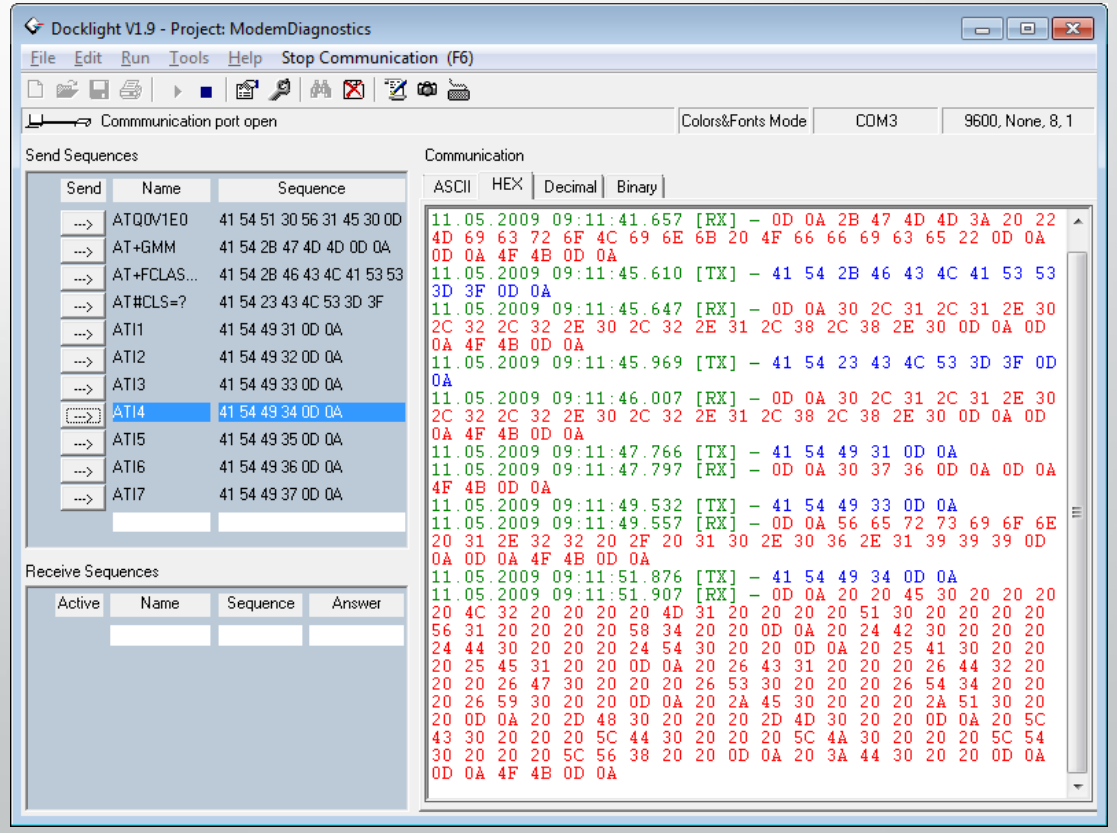
- Los objetos de prueba más importantes en las pruebas de integración son:
  - Las interfaces internas entre los componentes.
  - La configuración de programas y configuración de datos.
  - El correcto acceso a las bases de datos.
  - El correcto uso de otros componentes que forman parte de la infraestructura del sistema.

Entorno de las pruebas de integración

- Al igual que en las pruebas unitarias, en las pruebas de integración son necesario el uso de controladores de pruebas que permitan:
  - Enviar los datos de prueba a los objetos de prueba.
  - Recibir y registrar los resultados obtenidos.
- Dado que los objetos de prueba son componentes ensamblados que no tienen una interfaz hacia el exterior, es normal reutilizar los controladores de pruebas disponibles en las pruebas unitarias.
- Otras herramientas denominadas monitores pueden ser requeridas durante las pruebas de integración.
  - Los monitores son programas que leen y registran el trafico de datos entre los componentes.
  - Existen monitores para los protocolos estándar.

Entorno de las pruebas de integración





- Monitoreo de trafico.
- Análisis de paquetes para identificar errores en ellos.
- Estadísticas de las tasas de trafico y aplicaciones.
- Etc.

Objetivo de las pruebas de integración

- Los principales objetivos de estas pruebas son:
  - Identificar problemas de interfaces relevantes.
    - Tipos de datos no compatibles.
    - Número de argumentos distintos.
    - Problemas de consistencia de la información en los componentes que la utilizan.
  - Identificar conflictos entre las partes integradas.
    - Formato de los datos (ejemplo fecha).
    - Valores fuera de rango o con una precisión distinta para uno de los dos módulos integrados.

Estrategias de integración

- Como parte de la estrategia de integración, es necesario considerar ¿en qué orden deben integrarse los componentes para que las pruebas puedan funcionar de manera eficiente?

- La eficiencia es la relación entre el costo de las pruebas y su beneficio.
  - **Costo.** Se obtiene en base al número de probadores involucrados, las herramientas a utilizar, el tiempo dedicado a realizar las pruebas, entro otros aspectos.
  - **Beneficio.** Puede ser calculado en base al número y severidad de los problemas encontrados.

- El administrador de pruebas (test manager) es quien decide la estrategia de integración del proyecto.
  - Los diferentes componentes del software pueden ser terminados en tiempos distintos (días, semanas, meses).

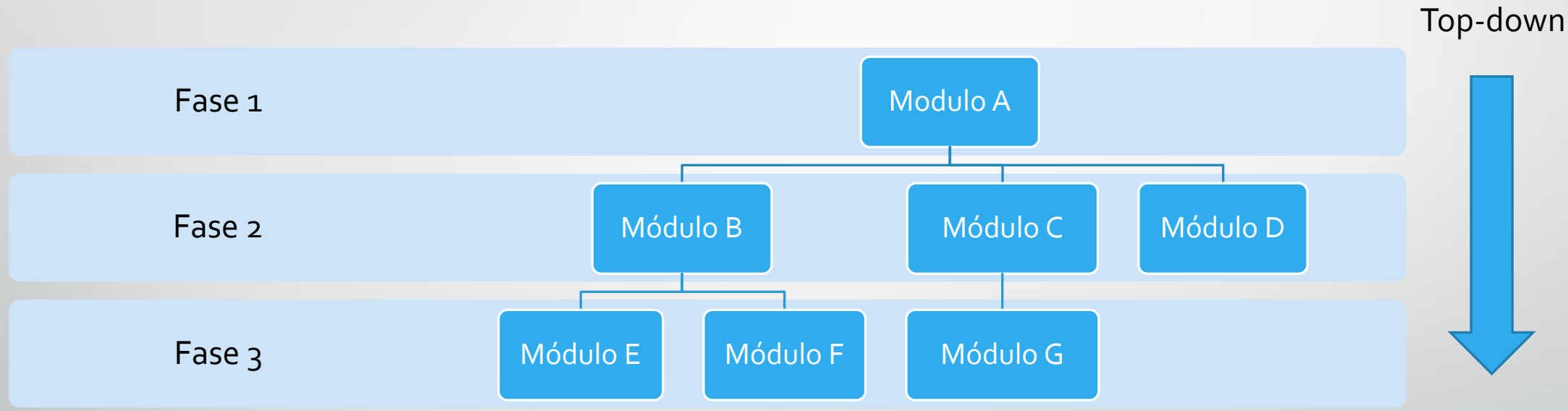
- Sin embargo, el probador no puede quedarse sin hacer nada, mientras espera a que todos los componentes estén desarrollados para iniciar con el proceso de integración.
  - Una estrategia básica para resolver este problema es iniciar integrando los componentes con forme se van desarrollando.

Estrategias de integración

Algunas estrategias que se pueden utilizar en la integración son:

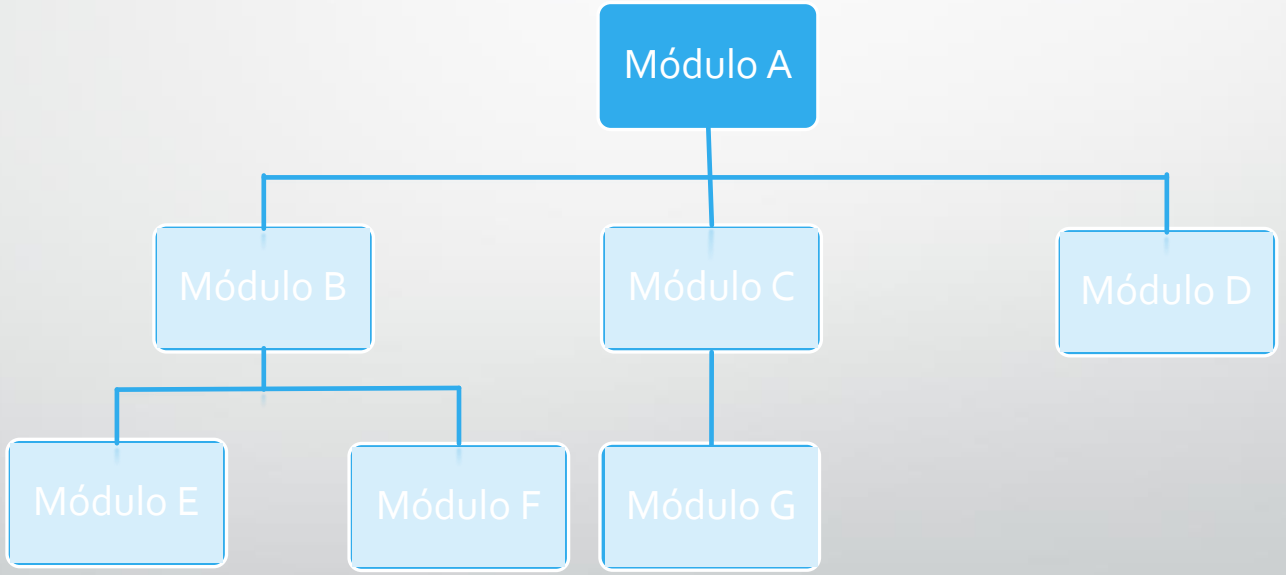
- **Integración Top-down.** Las pruebas inician con los componentes de alto nivel del sistema que invocan otros componentes.
  - Anchura.
  - Profundidad.
- **Integración Bottom-up.** Las pruebas inician con los componentes de bajo nivel del sistema que no invocan otros componentes.
- **Integración ad hoc.** Los componentes son integrados y probados conforme su desarrollo va concluyendo.
- **Integración backbone.** Un esqueleto o columna vertebral es definida y los componentes son gradualmente integrados a él.

Estrategia top-down



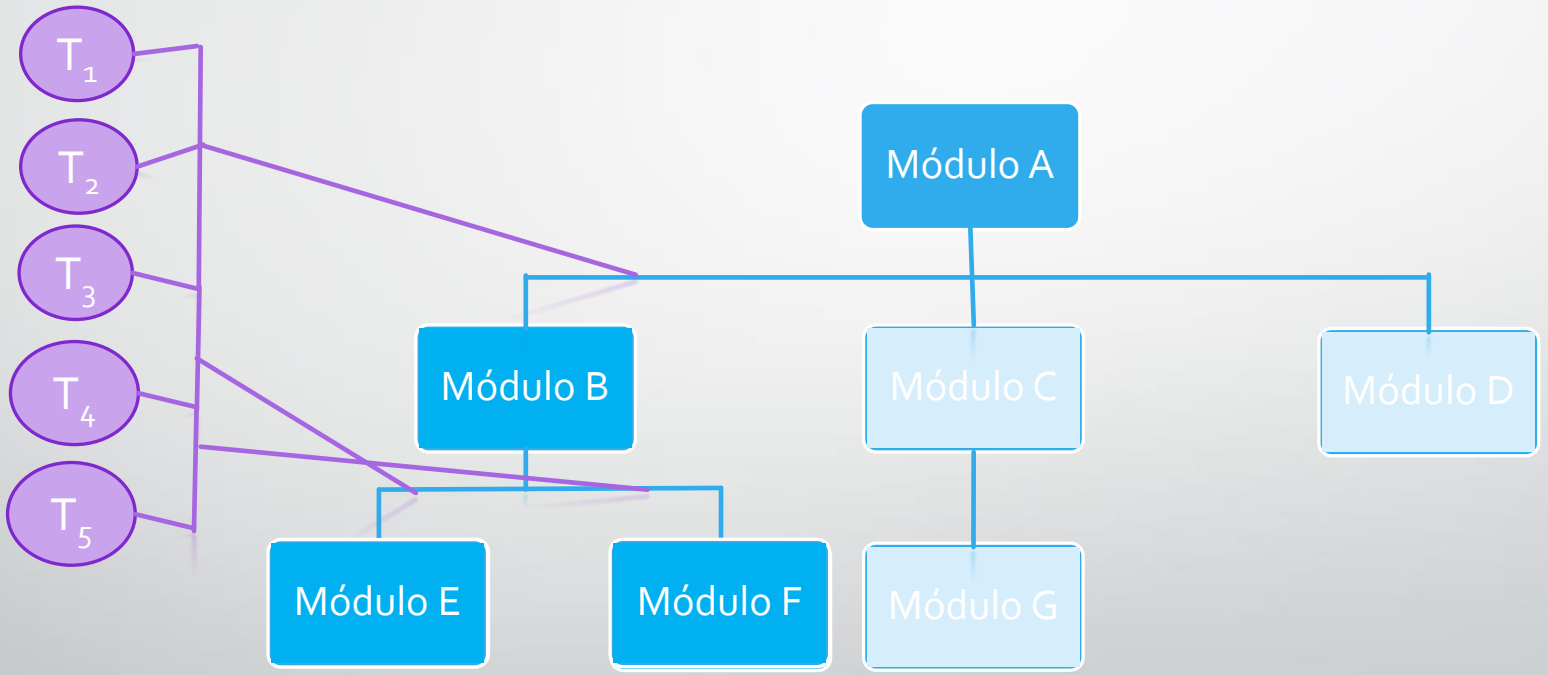
Estrategia top-down

El módulo principal es usado como controlador y todos sus módulos subordinados son remplazados por módulos simulados (stubs)



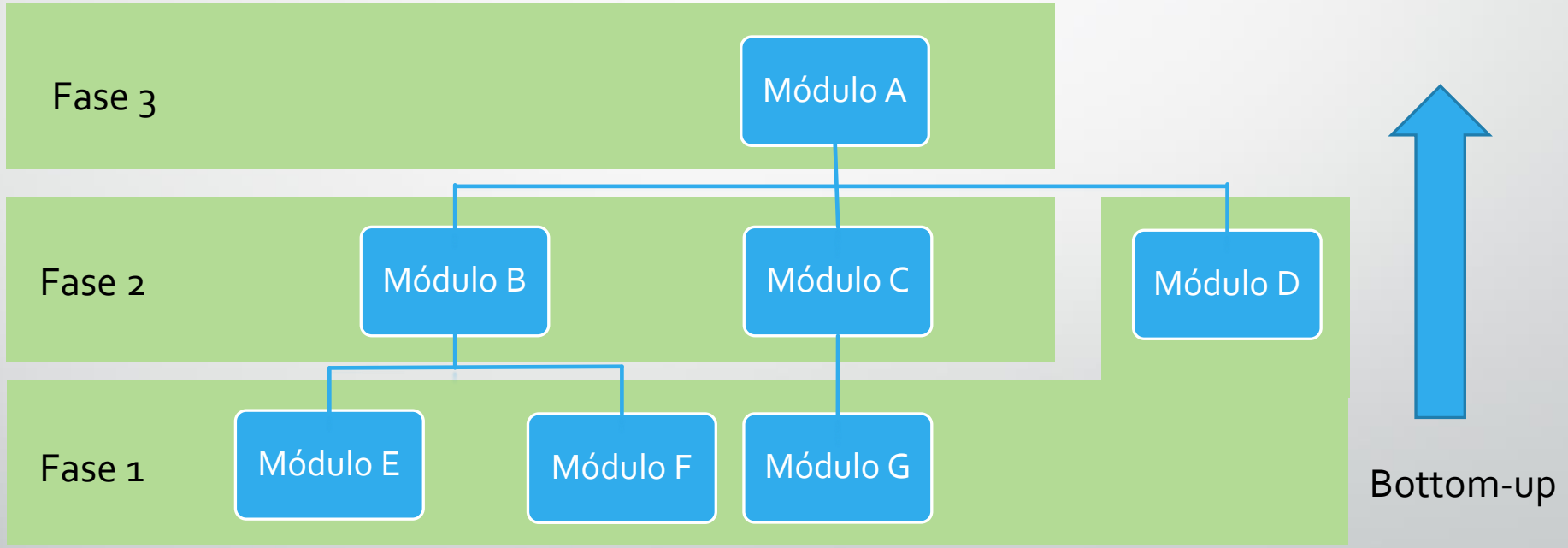
Estrategia top-down

Los módulos simulados se remplazan uno a la vez con los componentes reales (en profundidad) y se van probando.



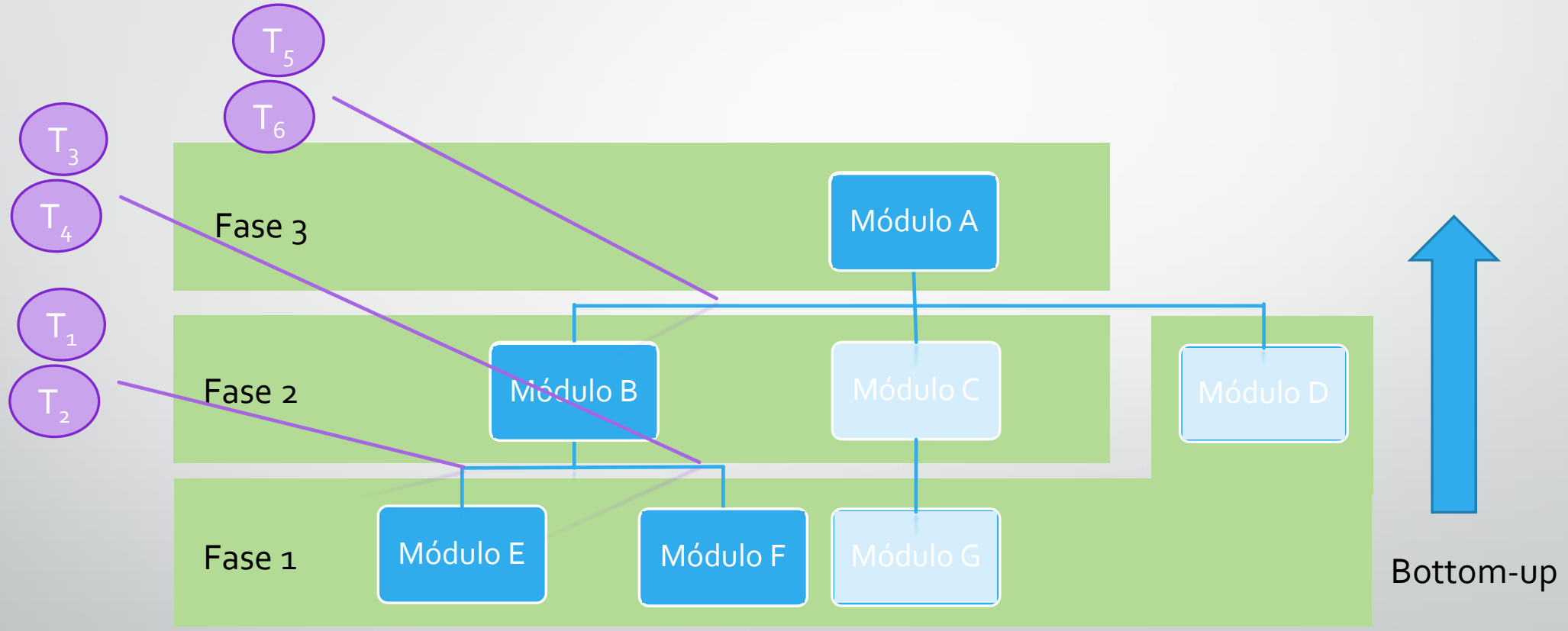
Estrategia bottom-up

A diferencia del enfoque descendente, éste inicia la construcción y prueba de los módulos en los niveles más bajos de la estructura del programa.

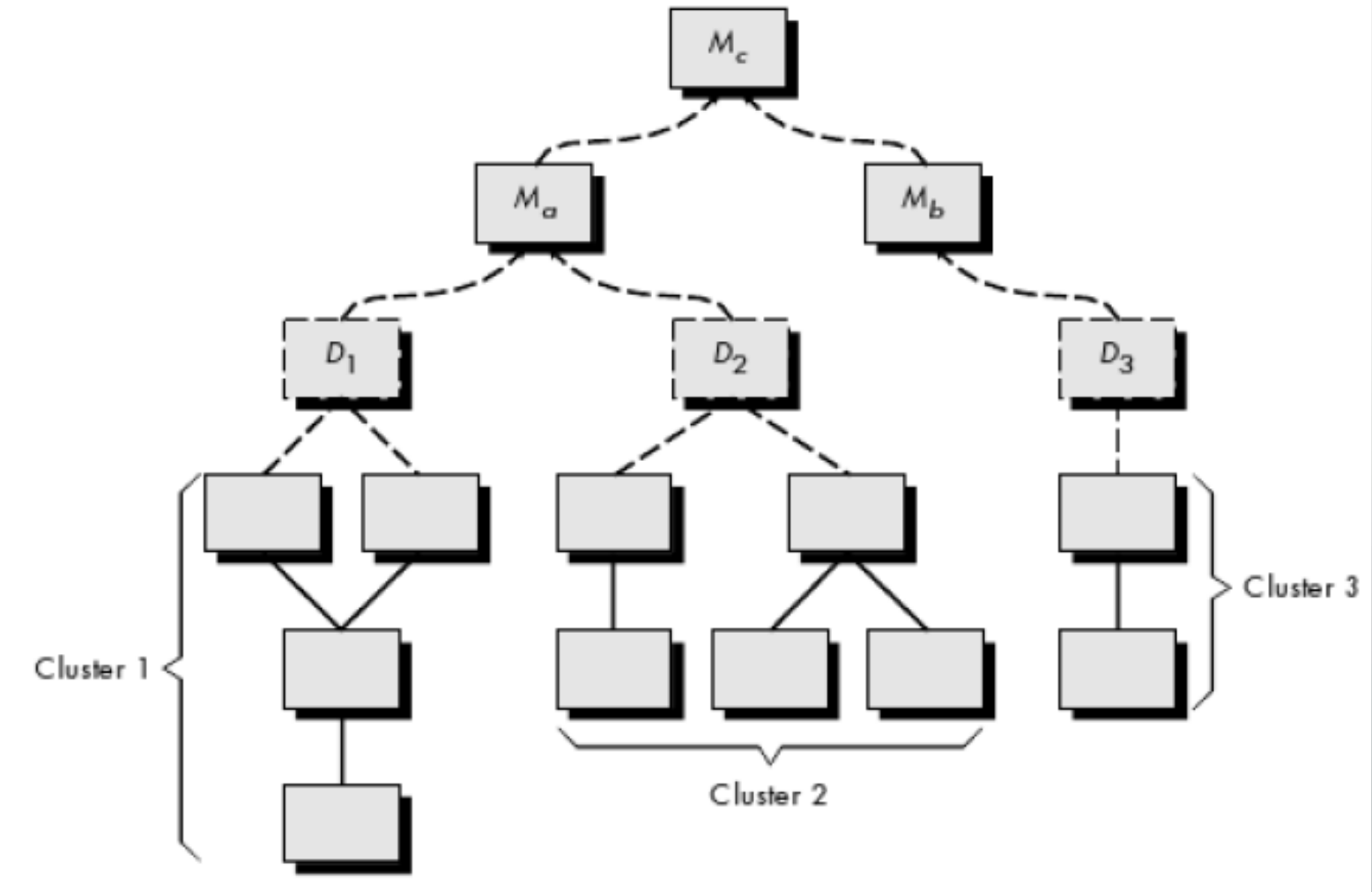


Estrategia bottom-up

A diferencia del enfoque descendente, éste inicia la construcción y prueba de los módulos en los niveles más bajos de la estructura del programa.



Estrategia bottom-up



Tarea 4

- Realizar una exposición en equipo de:

1

  - Las ventajas y desventajas relacionadas con la automatización de pruebas.
  - Criterios y características para determinar si una prueba puede (y debe) ser automatizada o no.
  - Clasificación (tipos) de pruebas automatizadas.
  - Describir algunas herramientas para la automatización de pruebas.
  - Ofrecer algunos ejemplos (mínimo 10) de pruebas automatizadas utilizando el proyecto.
  - La tarea deberá dar respuesta a las siguientes preguntas:
    - ¿Qué es la automatización de las pruebas de software?
    - Beneficios de la automatización.
    - ¿Qué tipo de pruebas se recomiendan automatizar?
    - ¿Cuándo no es recomendable hacer automatización de pruebas?

- Subir la presentación a Moodle.