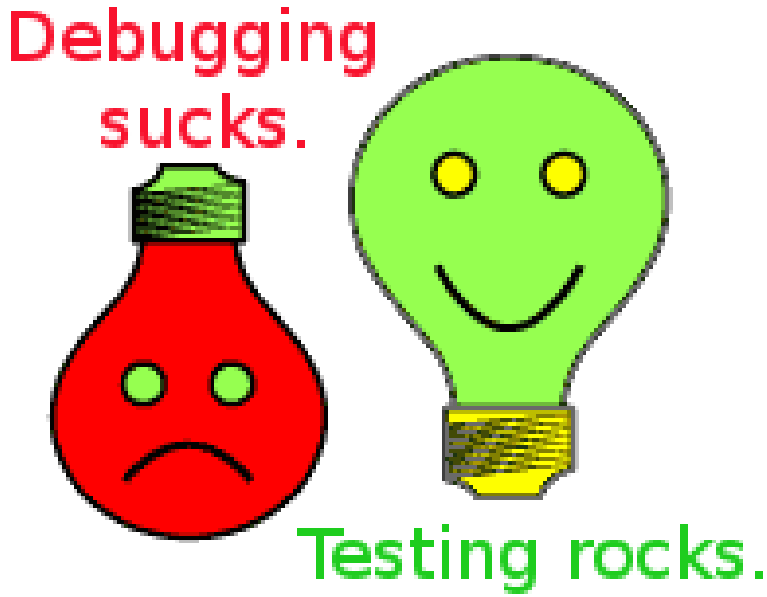
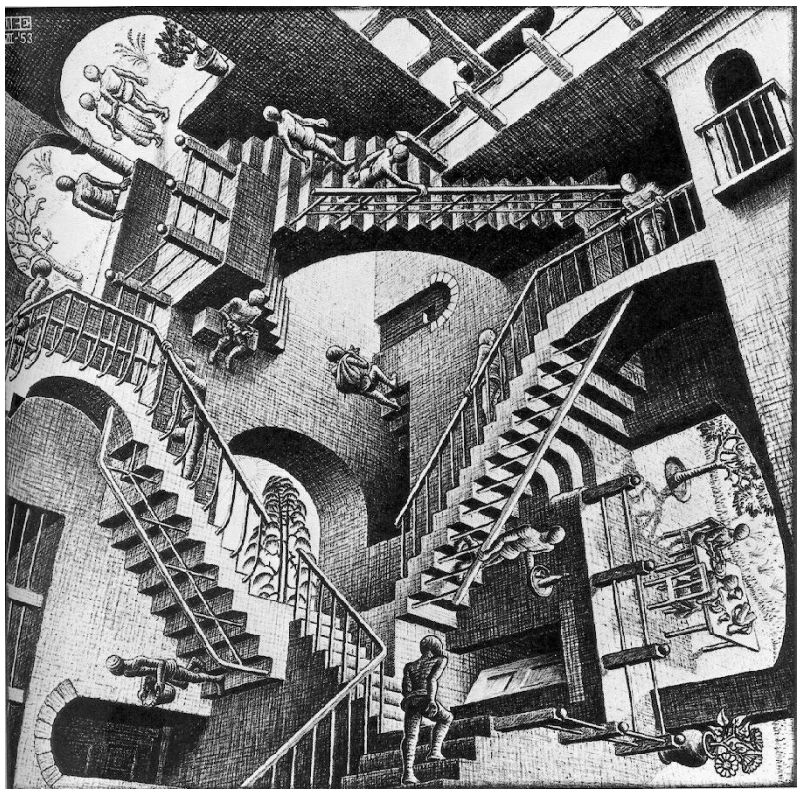


Testing y Mocking: Probando Aplicaciones

Outline

Outline

Cuando escribimos software



- falta tiempo
- hacemos suposiciones
- el código queda feo
- proyectos legacy sin andar
- falta tiempo de refactor

Por todo esto la calidad se degrada

Si bien no hay silver bullets

Veremos como, testeando, el proyecto sale mejor. Los tests:

- generan confianza
- permiten la detección temprana de bugs
- ayudan a detectar inconsistencias en RQs
- mejoran la productividad (a largo plazo)

Si bien no hay silver bullets

Veremos como, testeando, el proyecto sale mejor.
Siempre considerando que, los tests . . .

- no garantizan el éxito de un proyecto
- no aseguran 100% de calidad
- no son aplicables a todo proyecto
- obligan a pensar antes de programar! [a]

[a] Queda como ejercicio determinar si esto es bueno o no

Pero, ¿Qué es testing?

Outline

¿Qué es testing?

Testear el software es:

- validar su comportamiento
- validar un set de datos
- validar requerimientos
- una forma de medir la calidad

Testear el software no es:

- asegurar el éxito del proyecto
- afirmar que no hay bugs
- "a silver bullet"

Algunos "extras"

El testing (o, mejor, TDD) ayuda a:

- especificar requerimientos
- especificar comportamiento
- evitar regresiones
- refactorizar

Cómo testear

Vimos muchos motivos, muy dispares . . .
¿todo eso con un test?

No, hay distintos tipos. Veamos algunos.

Outline

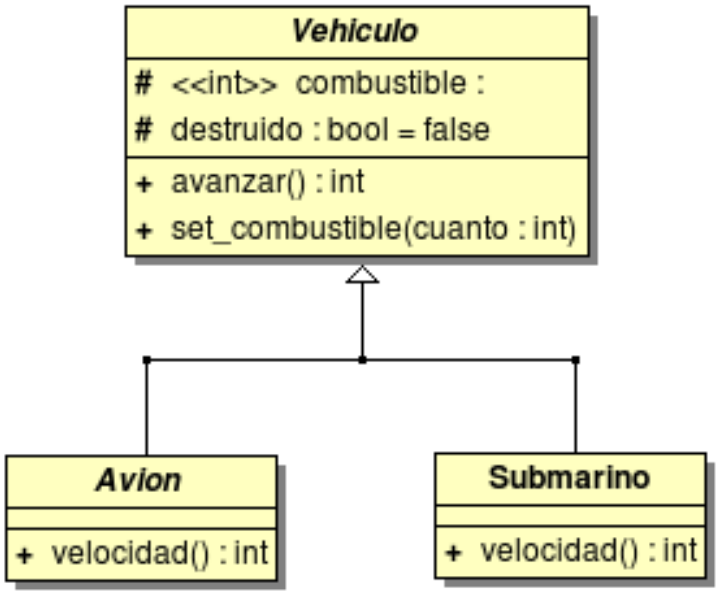
Outline

Tests unitarios: el tipo de test mas básico

- Prueba sólo una clase (aislada)
- repetibles y deterministicos
- automáticos
- se componen de varios tests cases
- cada test case prueba un aspecto de funcionalidad . . .
- que es un requerimiento (de la clase)

Veamos un ejemplo...

```
1  def Vehiculo.avanzar
2    if combustible > 0 then
3      combustible=combustible-1
4      return self.velocidad
5    else
6      destruido = true
7      return 0
8    end
9  end
10 def Avion.velocidad() return 20 end
11 def Submarino.velocidad() return 10 end
```



Para el ejemplo

Armemos un test unitario que verifique que los vehículos se destruyen si no tienen combustible

```
1  def test_destruccion
2    avion = Avion.new
3    submarino = Submarino.new
4
5    {avion, submarino}.
6      each.set_combustible 0
7
8    Expect{ avion.avanzar == 0 }
9    Expect{ avion.destruido == true }
10   Expect{ submarino.avanzar == 0 }
11   Expect{ submarino.destruido == true }
12 end
```

Outline

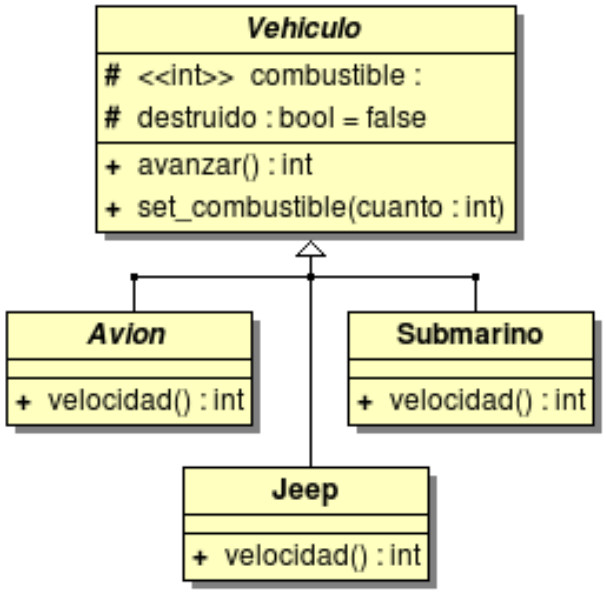
Tests de regresión

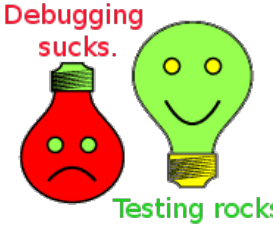
Los tests de regresión en verdad son tests unitarios

- que ya funcionan
- además de requerimientos puede validar "bugs"
- se corren luego de cada
  - refactor
  - bugfix
  - hacer un deploy
  - en general, modificación
- verifican si existe una regresión

Siguiendo el ejemplo...

```
1  def test_destruccion
2     avion = Avion.new
3     submarino = Submarino.new
4     jeep = Jeep.new
5
6     {avion, submarino, jeep}.
7       each.set_combustible 0
8
9     Expect{ avion.avanzar == 0 }
10    Expect{ avion.destruido == true }
11    Expect{ submarino.avanzar == 0 }
12    Expect{ submarino.destruido == true }
13    Expect{ jeep.avanzar == 0 }
14    Expect{ jeep.destruido == false }
15 end
```



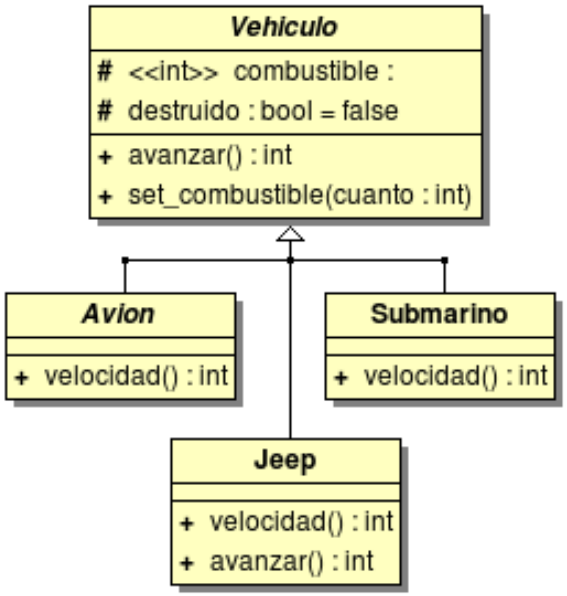


Y corremos el test...

¿Qué falló?

```
1  def Vehiculo.avanzar
2    if combustible > 0 then
3      combustible=combustible-1
4      return self.velocidad
5    else
6      destruido = true
7      return 0
8    end
9  end
10 def Avion.velocidad() return 20 end
11 def Submarino.velocidad() return 10 end
```

Corregimos el error, y ahora si . . .

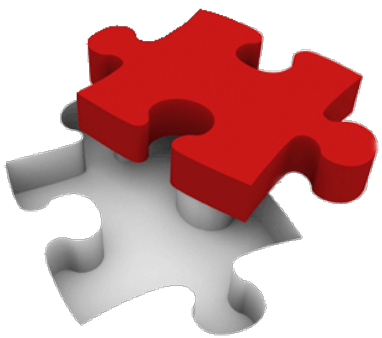


```
1  def Jeep.avanzar
2   if combustible > 0 then
3    combustible=combustible-1
4    return self.velocidad
5   else
6    return 0
7   end
8  end
```

Outline

Integración continua

Si un test que antes funcionaba ahora falla:



- existe una regresión
- entre el último commit testeado y el actual
- menos commits $\Longrightarrow$ mas fácil descubrir la causa
- uno por commit se llama integración continua!

Integración continua

Un servidor de IC

- requiere tests "rápidos"
- puede schedulear tests "lentos" diariamente
- provee feedback del commit en cuestión de minutos
- requiere tests deterministicos y repetibles!

Integración continua

Un servidor de IC

- requiere tests "rápidos"
- puede schedulear tests "lentos" diariamente
- provee feedback del commit en cuestión de minutos
- requiere tests deterministicos y repetibles!

Veamos una demo con el Cruise Control!

Outline

Tests de integración

Tests de Integración != Integración Continua

Los tests de integración:

- prueban mas de un componente
- pueden implementarse como tests unitarios
- no siempre son deterministicos o repetibles
- no siempre son "rápidos"

¿Por qué?

Tener solo tests de integración no es cost-effective:

```
1  def foo(x, y, z)
2    if z then
3      bar(x)
4    else
5      baz(y)
6    end
7  end
```

¿Cuántos tests unitarios son?
¿Cuántos tests de integración son?

Tests de integración

Integración

N	Función	x	y	z
1	foo	true	true	true
2	foo	true	true	false
3	foo	true	false	true
4	foo	true	false	false
5	foo	false	true	true
6	foo	false	true	false
7	foo	false	false	true
8	foo	false	false	false

Unitario

N	Función	x	y
1	bar	true	N/A
2	bar	false	N/A
3	baz	N/A	true
4	baz	N/A	false

Hacen falta 8 tests de integración y 4 unitarios
Entonces, ¿conviene usar tests de integración?

¿Por qué no?

Los tests para ambas funciones son simples
Pero en conjunto andan muy mal!

test_multiplicar_manzanas ()

test_get_peras ()

multiplicar_manzanas(get_peras ())

Outline

stress tests

¿qué son?

- verifican el funcionamiento bajo carga
- simulan un entorno de producción
- pocas veces son automáticos

Outline

Orientado a los datos

Cuando armarmos un test:

- Se toma un módulo
- Determinamos las entradas
- Evaluamos las salidas
- Buscamos casos límite
- Existe un contrato implicito

```
1  def test_division
2    Expect{1 == 2/2}
3    Expect{Err == 1/0}
4    Expect{-5 == -5/1}
5    Expect{-5 == 5/-1}
6    Expect{5 == -5/-1}
7    # Etc ...
8  end
```

Orientado al comportamiento

Cuando armarmos un test (en un proyecto testeable):

- Pensamos en término de objetos y relaciones
- . . . y no tanto en función de la entrada-salida
- Los objetos tienen interfaces
- . . . e interactuan con otros objectos
- Existe un contrato explícito
- No es necesario evaluar tantos casos limite

Outline

Outline

Empecemos con un ejemplo

¿Qué pasa cuando corremos el test?

```
1  def test_compra
2     item = Empanada.new
3     item.precio = 2
4     venta = Venta.new
5     venta.add item
6     Expect{ venta.comprar == true }
7     Expect{ venta.total == 2 }
8  end
```

Empecemos con un ejemplo

¿Qué pasa cuando corremos el test?

```
1  def test_compra
2     item = Empanada.new
3     item.precio = 2
4     venta = Venta.new
5     venta.add item
6     Expect{ venta.comprar == true }
7     Expect{ venta.total == 2 }
8  end
```

Null ptr exception

¿Qué hay en la clase Venta?

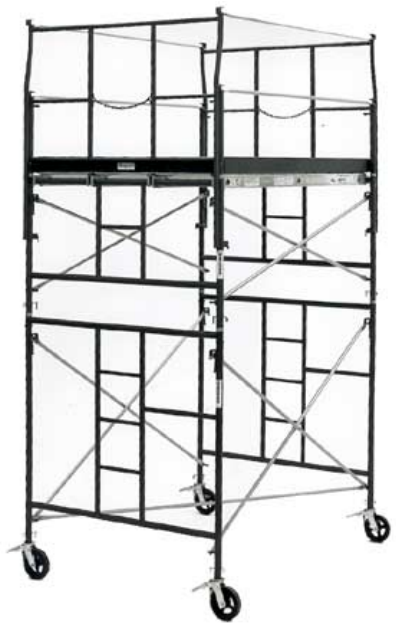
```
1  def Venta.new
2    db = ResourceLocator.DbSingleton.get_instance
3    cliente = db.get_instance.get_ultimo_cliente
4  end
5
6  def Venta.add item
7    raise "Te falta plata!"
8      if cliente.billetera.saldo < self.total
9
10   items.add item
11 end
12
13 def Venta.comprar
14   cliente.billetera.debitar self.total
15   return self.total
16 end
```

Arreglemos el test...

```
1  def test_compra
2    # Scaffolding
3    dagoberto = Cliente.new
4    dagoberto.billetera = Billetera.new
5    dagoberto.billetera.saldo = 2
6
7    singleton = Singleton.new
8    singleton.instance = DB_Factory.new
9    ResourceLocator.DbSingleton = singleton
10   db = ResourceLocator.DbSingleton.get_instance
11   db.set_cliente = dagoberto
12
13   item = Empanada.new
14   item.precio = 2
15
16   # Test real - Dependencias ocultas
17   venta = Venta.new
18   venta.add item
19   Expect{ venta.comprar == true }
20   Expect{ venta.total == 2 }
21 end
```

Arreglemos el test...

En el test se reflejan varios problemas



- Hay mucho "scaffolding"
- El sentido del test se pierde
- Hay dependencias ocultas
- Se accede a mas objetos que los necesarios
- El scope demasiado grande

Outline

Podemos mejorar el diseño con DI

La inyección de dependencias

- separa la construcción del uso
- hace explicitas las dependencias
- ayuda a simplificar la lógica
- limita el scope de un objeto
- reduce el factor sorpresa!

Dependency Injection Demystified

La inyección de dependencias es en realidad muy simple

Ask for what you need

- A un objeto se le dan sus dependencias
- El objeto inyectado no crea sus propias dependencias
- No mas singletons ni ServiceLocators
- Chau estado global

Outline

Resultados

```
1  def Venta.new cuenta
2    self.cuenta = cuenta
3  end
4
5  def Venta.add item
6    raise "Te falta plata!"
7      if cuenta.saldo < self.total
8
9    items.add item
10 end
11
12 def Venta.comprar
13   cuenta.debitar self.total
14   return self.total
15 end
```

Resultados

No solo el código es mas prolijo, el test también

```
1  def test_compra
2     item = Empanada.new
3     item.precio = 2
4     cuenta = Cuenta.new
5     cuenta.saldo = 2
6
7     # Test real - la dependencia es clara
8     venta = Venta.new cuenta
9     venta.add item
10    Expect{ venta.comprar == true }
11    Expect{ venta.total == 2 }
12 end
```

Outline

Mocking: seguimos el ejemplo

Sigamos el ejemplo de DI. ¿Podemos mejoralo mas?

- ¿Qué estamos testeando?
- ¿Queremos testear el objeto "Cuenta"?
- ¿Y si Cuenta aún no está implementado?
- ¿O tiene muchos muchos bugs?
- ¿O es muy lento? (P. ej: accede a la BD)

Mocking es la solución!

Mocking

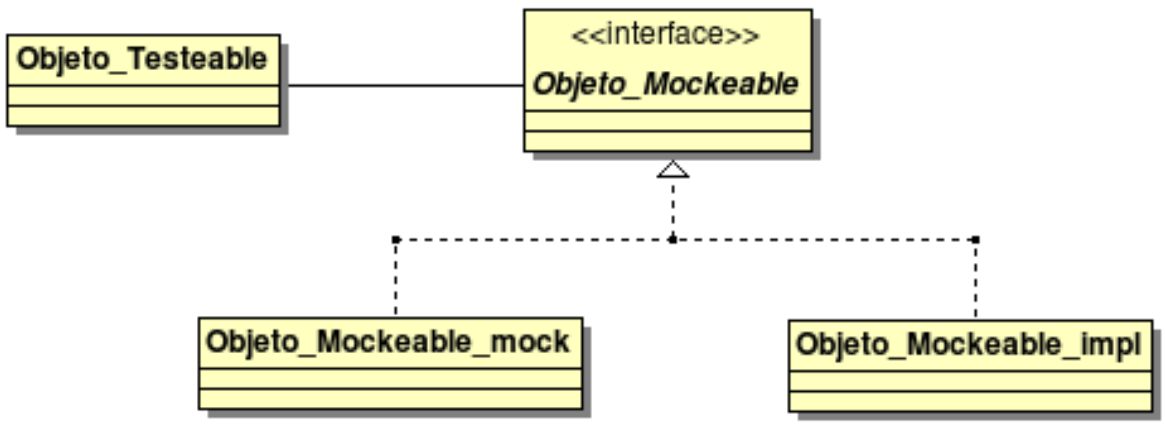
¿Pero qué es "mocking"?

- técnica para testear
- un mock es un objeto "dummy"
- son "fakes" de partes del sistema
- definen su comportamiento "on the fly"
- usan la interfaz de un objeto real

Outline

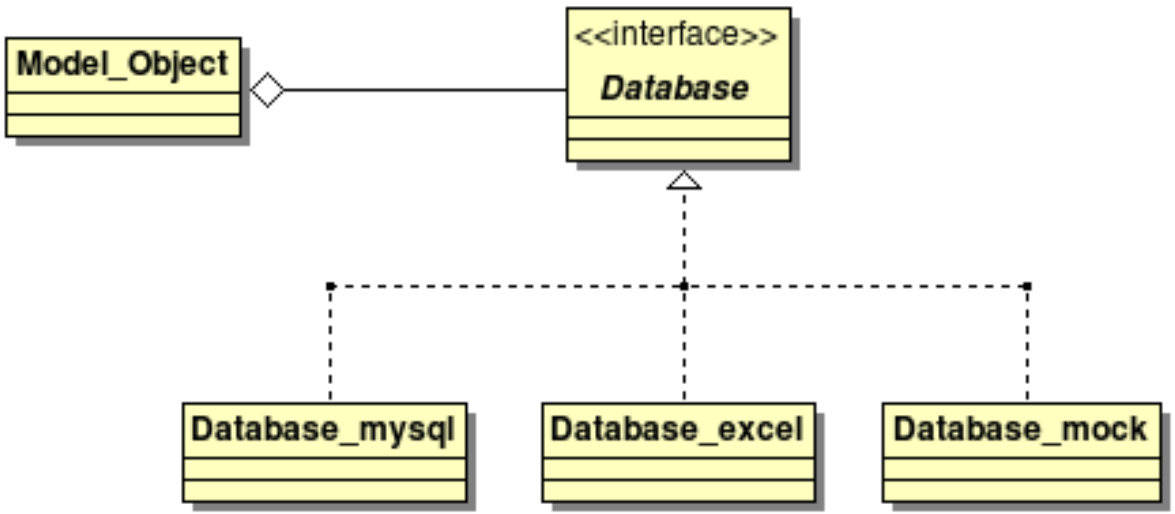
Cómo funciona

- Se basa en el polimorfismo
- El Objeto testeable usa un objeto "mockeable"
- El mock hereda de una interfaz común al objeto real
- Al objeto testeable no le "importa" cual de los dos usa



Cómo funciona

- El objeto testeado no sabe si es una implementación real
- Tampoco importa, el mock usa la interfaz conocida
- Es un concepto similar al de usar varias implementaciones



Outline

Por qué DI + Mocking

- DI obliga a separar componentes
- Para testear hace falta aislar módulos
- Aislar módulo $\Longrightarrow$ sacar dependencias
- Mocking permite simular otros módulos
- Ergo . . .

Mocking + DI r0ck$!!!!!11uno1

Outline

Mocking: seguimos el ejemplo II

Veamos como queda el ejemplo

```
1  def test_compra
2     item = MockItem.new
3     cuenta = MockCuenta.new
4     item.expect_call("precio").returns( 2 )
5     cuenta.expect_call("saldo").returns( 2 )
6     cuenta.expect_call("debitar", 2).returns( true )
7
8     venta = Venta.new cuenta
9     venta.add item
10    Expect{ venta.comprar == true }
11    Expect{ venta.total == 2 }
12 end
```

Mocking: ejemplo explicado

Algunos detalles del ejemplo:

```
1 item.expect_call("precio").returns( 2 )
2 cuenta.expect_call("saldo").returns( 2 )
3 cuenta.expect_call("debitar", 2).returns( true )
```

- El comportamiento se define con el mismo test
- Se define un set de "expectations"
- Las expectations son tests
  - cada una debe cumplirse
  - verifican el comportamiento del objeto
- Usando DI podemos inyectar una cuenta falsa
- . . . sería posible de no usar DI?

¿Preguntas?

42

¿Preguntas?