

Aseguramiento de Calidad (QA)

Definición de Calidad I

- Conformancia total a los requerimientos
 - problema: muchos de los defectos tienen que ver con errores en requerimientos (que no pueden ser detectados por testing)
 - problema: a veces los requerimientos no son comprendidos hasta después de completar el proyecto

Definición de Calidad II

Factores de Calidad

Augmentability	Portability
Compatibility	Reliability
Expandability	Scalability
Flexibility	Survivability
Interoperability	Understandability
Maintainability	Usability
Manageability	Testability
Modifiability	Traceability
Operability	Verifiability

- problema: la mayoría tiene poco que ver con calidad percibida por el usuario
- problema: la mayoría tiene poco que ver con defectos o potencial de defectos

Definición de Calidad III

- Ausencia de defectos que causarían que la aplicación deje de funcionar o entregue resultados erróneos
- Nivel de severidad del defecto
 - severidad 1 - la aplicación no funciona
 - severidad 2 - mayoría de funciones no habilitadas o resultados erróneos
 - severidad 3 - problemas menores
 - severidad 4 - problemas cosméticos que no afectan funcionalidad

Como identificar calidad

(Capers Jones)

- Quality implies **low levels of defects** when software is deployed, ideally approaching zero defects.
- Quality implies **high reliability**, or being able to run without stop or strange and unexpected results or sluggish performance.
- Quality implies **high levels of user satisfaction** when users are surveyed about software applications and its features.
- Quality implies a feature set that meets **the normal operational needs of a majority of customers or users**.
- Quality implies **a code structure and comment density that minimize bad fixes or accidentally inserting new bugs when attempting to repair old bugs**. This same structure will facilitate adding new features.
- Quality implies **effective customer support when problems do occur**, with minimal difficulty for customers in contacting the support team and getting assistance.
- Quality implies **rapid repairs of known defects**, and especially so for high-severity defects.
- Quality should be supported by **meaningful guarantees and warranties** offered by software developers to software users.
- Effective definitions of quality should lead to quality improvements. This means that **quality needs to be defined rigorously enough so that both improvements and degradations can be identified**, and also averages. If a definition for quality cannot show changes or improvements, then it is of very limited value.

Defecto: origen, severidad, causa

- Origen : requerimientos, diseño, código, documentos de usuario, arreglo defectuoso
- Severidad : 1 al 4 de acuerdo a lo que vimos
- Causas :

Errors of omission:	Something needed was accidentally left out
Errors of commission:	Something needed is incorrect
Errors of ambiguity:	Something is interpreted in several ways
Errors of performance:	Some routines are too slow to be useful
Errors of security:	Security vulnerabilities allow attacks from outside
Errors of excess:	Irrelevant code and unneeded features are included
Errors of poor removal:	Defects that should easily have been found

Eliminación de Defectos

- static analysis
 - structural problems such as boundary conditions, duplications, failures of error-handling, and branches to incorrect routines. Static analysis can also find security flaws.
- code review (formal inspections)
 - complicated and subtle problems that require human intelligence and insight. Formal inspections are also best at finding errors of omission and errors of ambiguity.
- testing
 - problems that occur when software is executing, such as performance issues, usability issues, and security issues

Static Analysis

- Análisis del código (sin correrlo) mediante herramientas que permiten detectar elementos sospechosos
 - más usado con lenguajes como C/C++, Java que con lenguajes dinámicos (Ruby, Python)
 - encuentra algunos problemas fácilmente: buffer overflow, SQL injection, etc
 - muchos falsos positivos
 - muchos problemas importantes no son detectados

Code Review

- Forma demostradamente efectiva para producir código de calidad
- Código es revisado por equipo de pares con el objetivo de detectar la mayor cantidad de defectos
- Equipo revisor se prepara antes de sesión de revisión
- Pueden usarse checklists para búsqueda mas dirigida

No se trata de esto ...

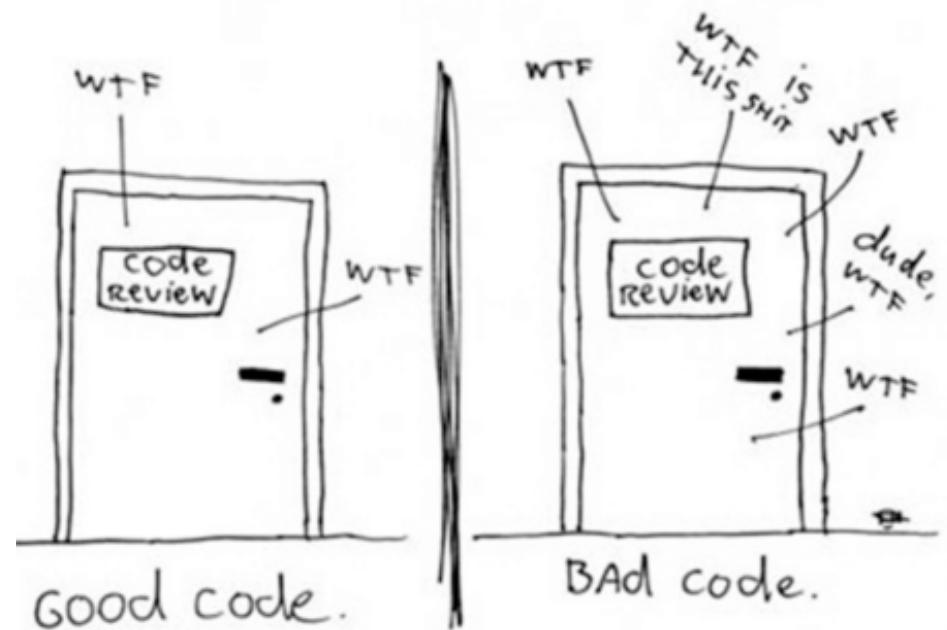
The Peer Code Review



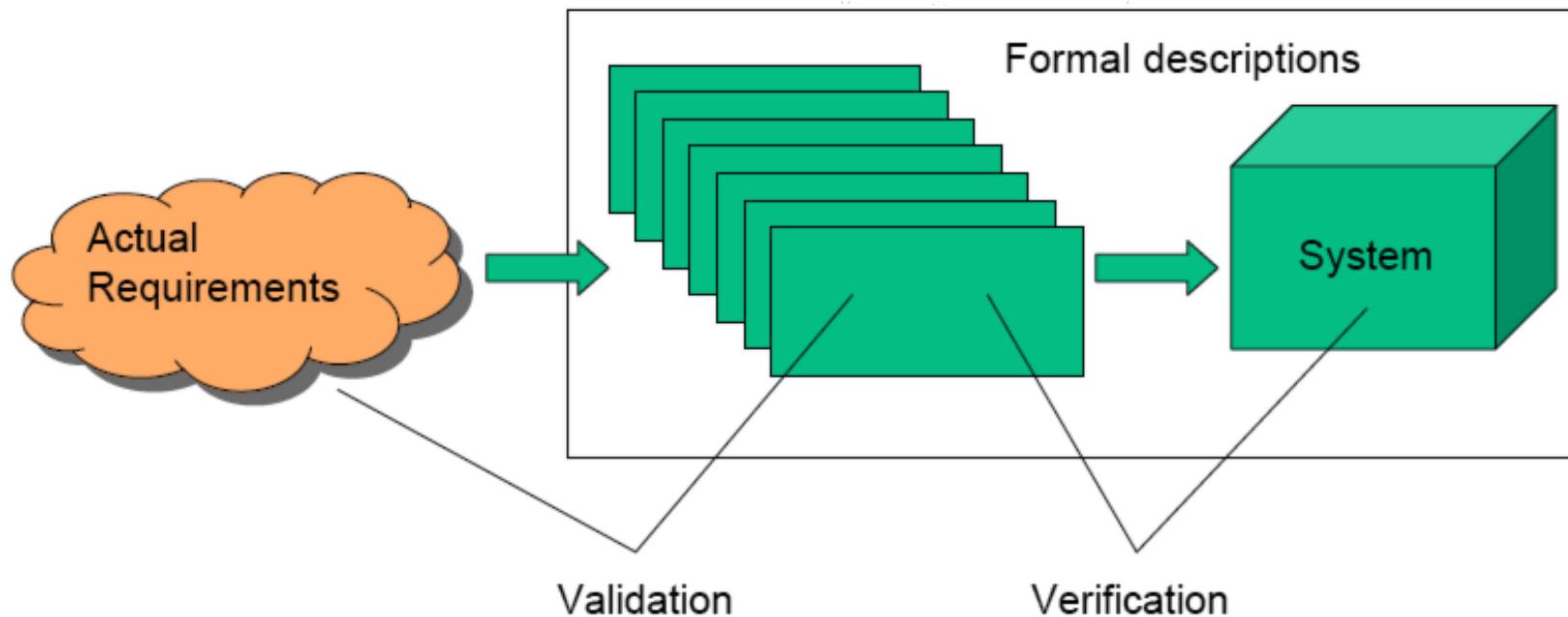
Sesión de Inspección

- un facilitador dirige al grupo sobre el código
- al llegar a un bloque en que hay observaciones los revisores plantean sus dudas para ver si se trata en verdad de un problema
- Si es un problema se registra (no se corrige)
- Aprovechar de recolectar datos (defectos por grado de severidad, por página, por número de líneas, etc)

The ONLY VALID MEASUREMENT
OF CODE QUALITY: WTFs/MINUTE



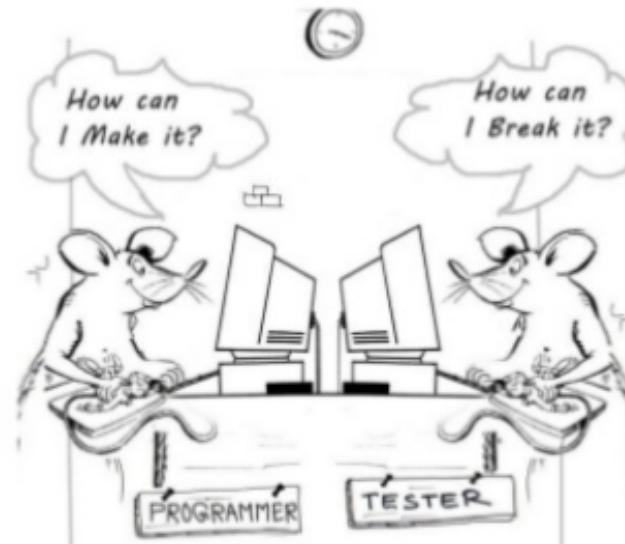
Verificación vs Validación



- Verificación - el sistema hace correctamente lo que se especificó
- Validación - el sistema hace lo que realmente se necesitaba que hiciera

Testing

- Proceso de ejecutar un programa con el objetivo de encontrar un error
- un buen caso de prueba es uno con una alta probabilidad de encontrar un error oculto
- un test exitoso es aquel que descubre un error que no se conocía

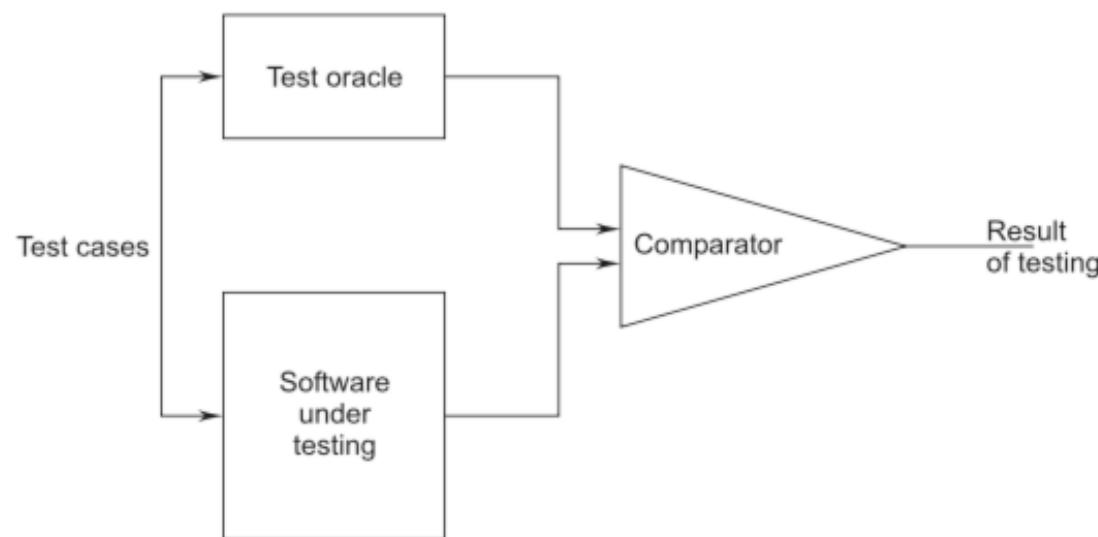


Dificultad en Testing

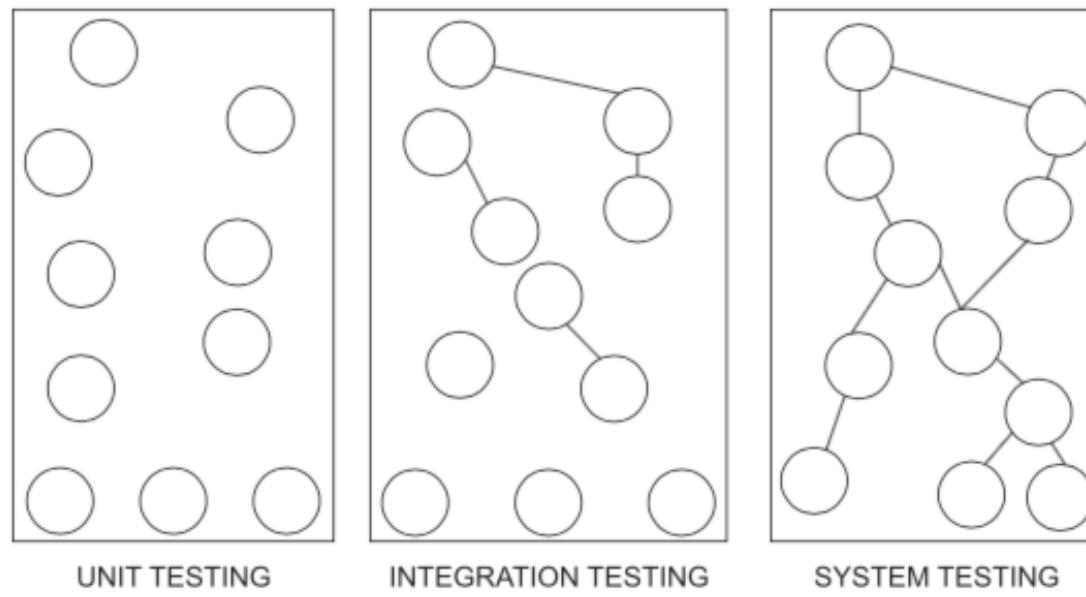


Oráculos

- Mecanismo diferente al programa mismo que permite chequear la correctitud del caso de test
- Por lo general basado en personas

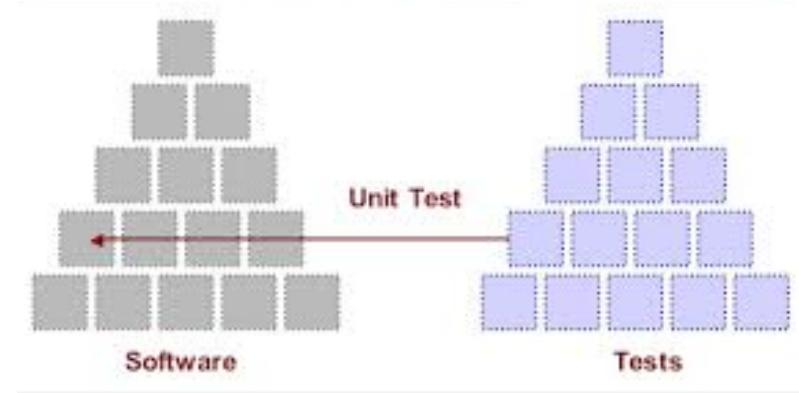


Niveles de Tests



Tests Unitarios

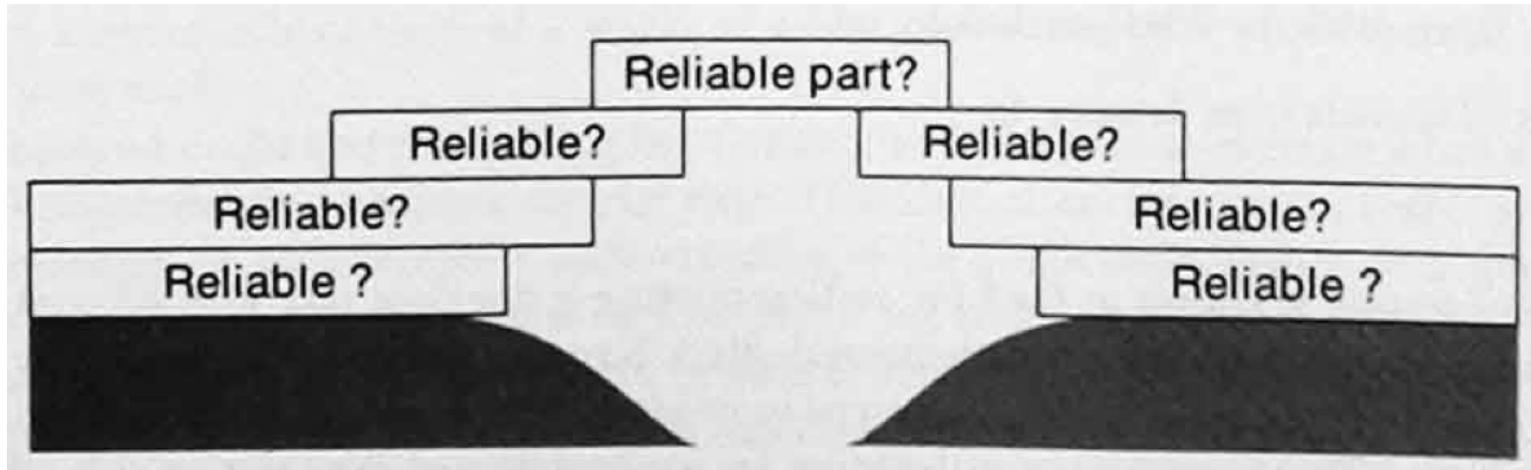
- Realizado por los desarrolladores en paralelo con la implementación o inmediatamente después de terminar la parte a probar
- Pueden usarse tests de black box o white box
- A veces hay que crear inputs y outputs (unidades fuera de análisis)



I'm a programmer and #iwritebugs. All the time. Embarrassingly obvious bugs. Brain dead stupid. Fortunately, I also write tests.

Howard Lewis Ship

Importancia de Test Unitarios



- Al igual que un puente, confiabilidad del todo depende de confiabilidad de sus partes
- Test parte desde unidades
- Una unidad típicamente es un método o una clase

Blackbox vs Whitebox



- blackbox test - no se conocen detalles del software, solo se considera input y output
- whitebox test - se conocen los detalles del software y se puede utilizar en diseño del test

```
using namespace std;
int main()
{
    const double THIRDPi = 3.14
    float radius;
    float height;
    double volume;

    cout << "Enter Radius" << e
    cin >> radius;
    cout << "Enter height" << e
    cin >> height;
```

Estrategias según tipo de test

```
using namespace std;
int main()
{
    const double THIRDPI = 3.14
    float radius;
    float height;
    double volume;
    cout << "Enter Radius" << endl;
    cin >> radius;
    cout << "Enter height" << endl;
    cin >> height;
    volume = radius * radius * height * THIRDPI;
    cout << "Volume is " << volume;
}
```

White Box

- cobertura de instrucciones
- cobertura de caminos
- cobertura de decisiones

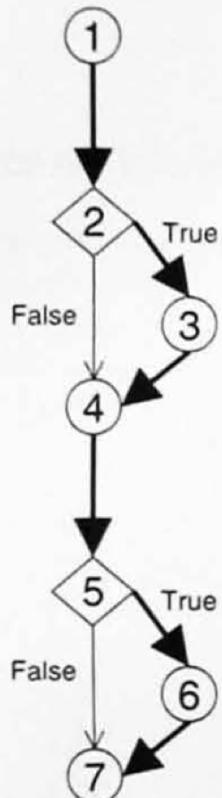


Black Box

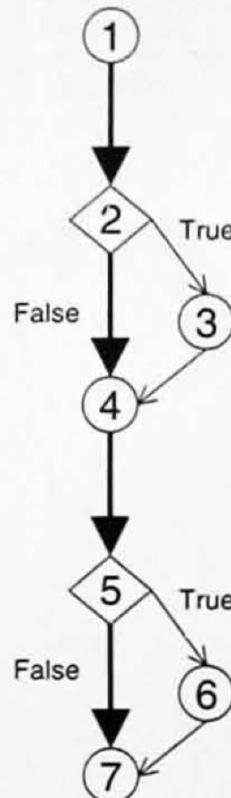
- clases de equivalencia
- análisis de valores frontera

Cobertura de Caminos (white box)

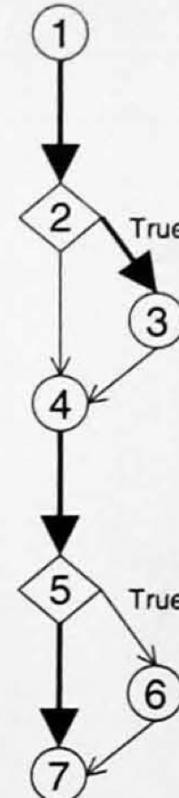
Path #1



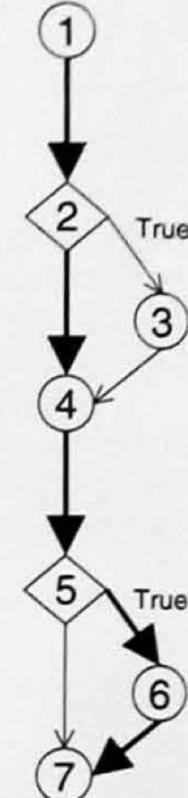
Path #2



Path #3

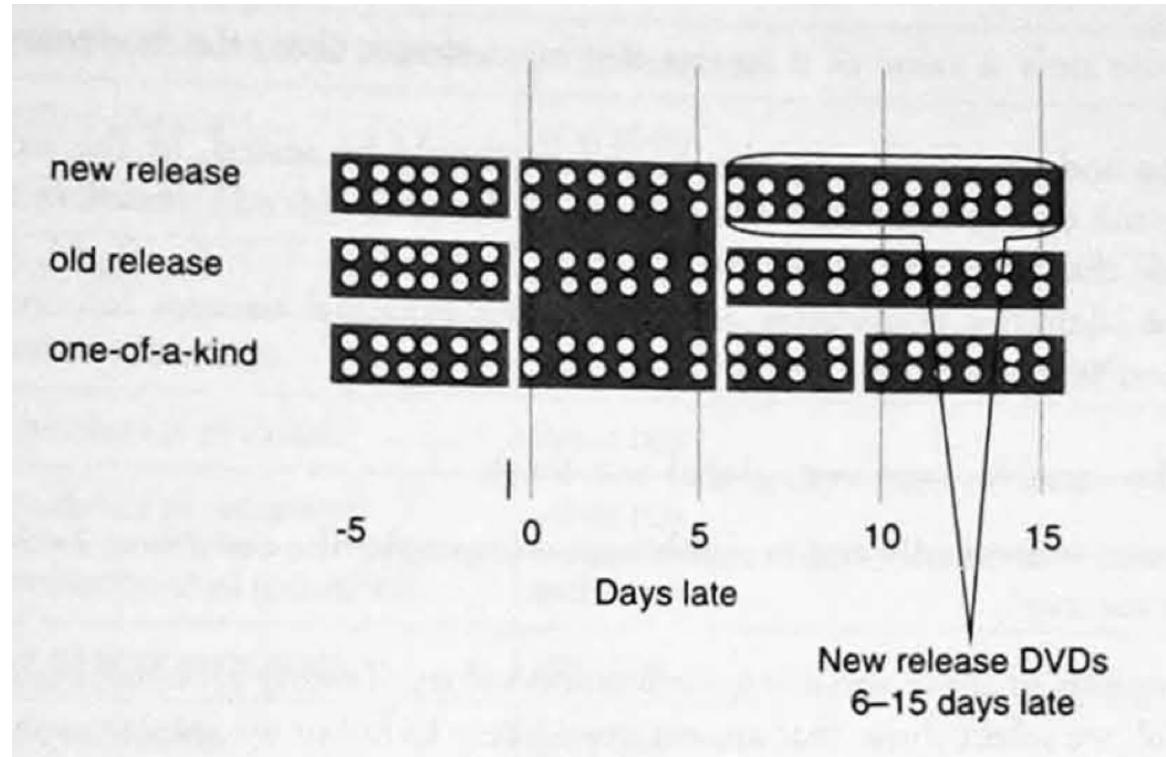


Path #4



Clases de Equivalencia (black box)

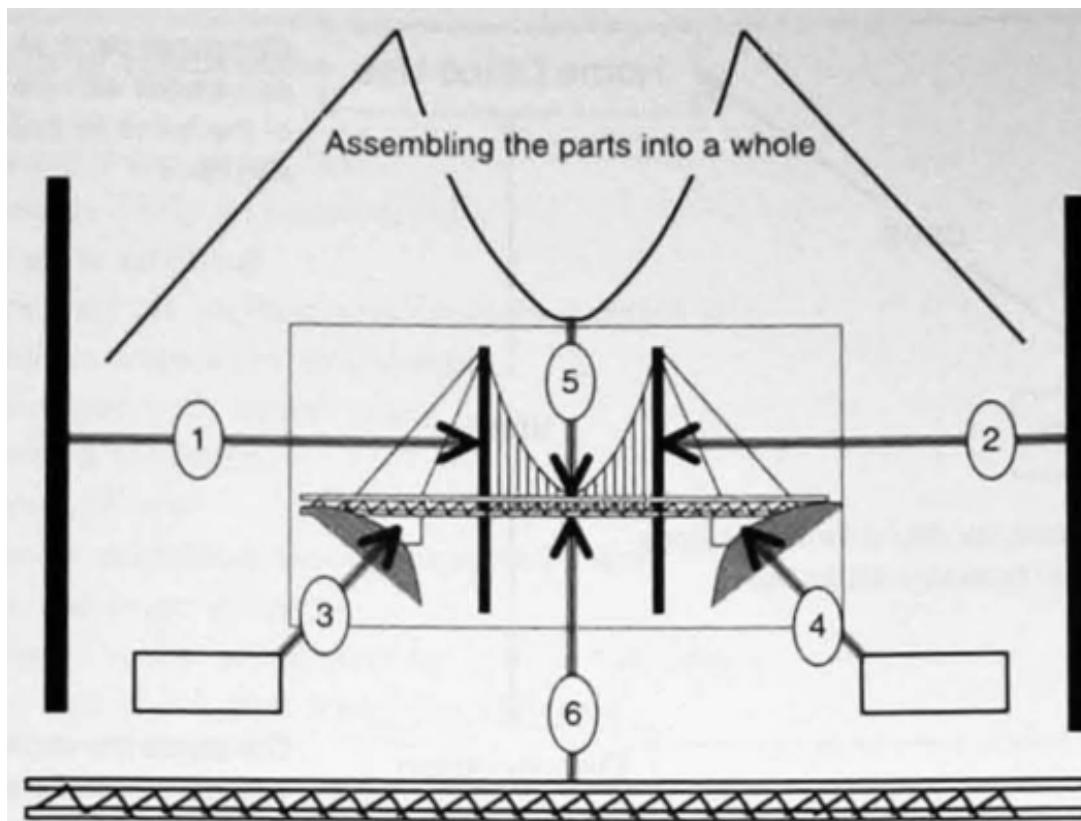
Ejemplo de Clases para probar clase que calcula las multas en arriendo de películas





Tests de Integración

- El software, al igual que un puente se construye en partes que son probadas en forma independiente



La necesidad

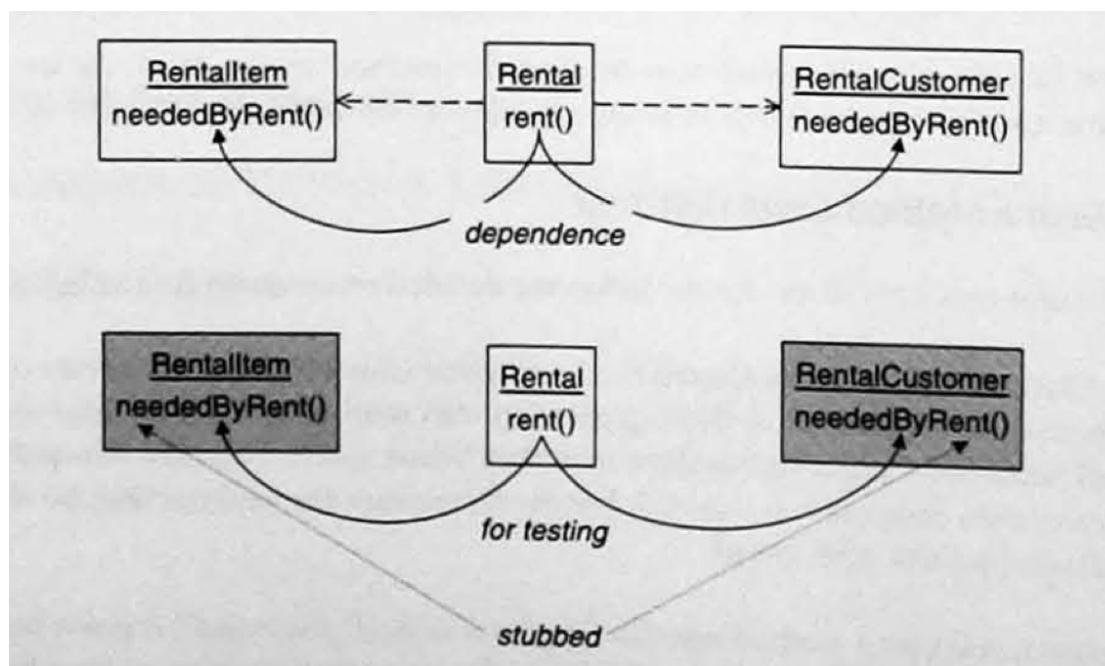


Estrategias para Tests de Integración

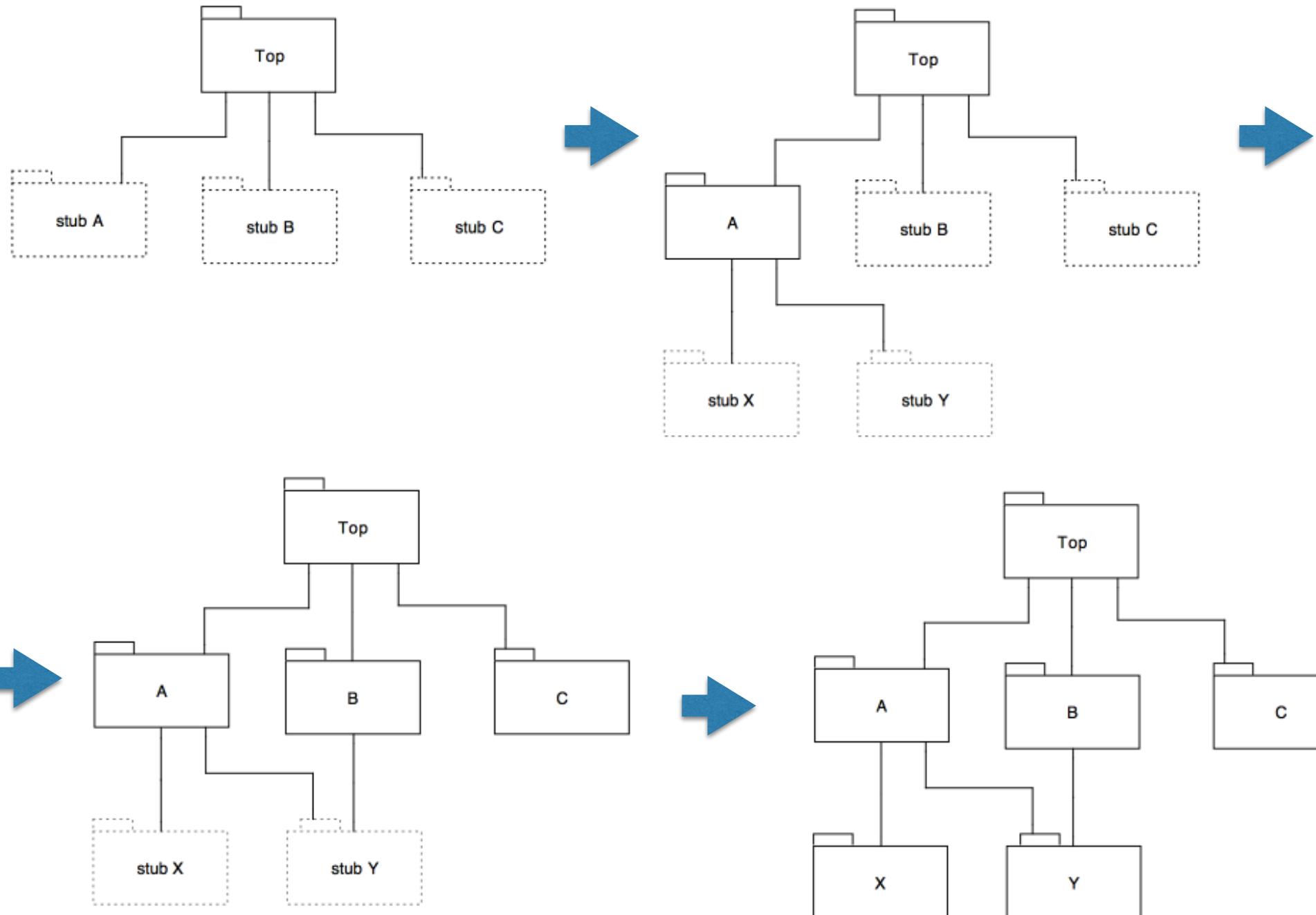
- Big Bang
- Top Down
- Bottom up
- Mixed

Uso de Stubs

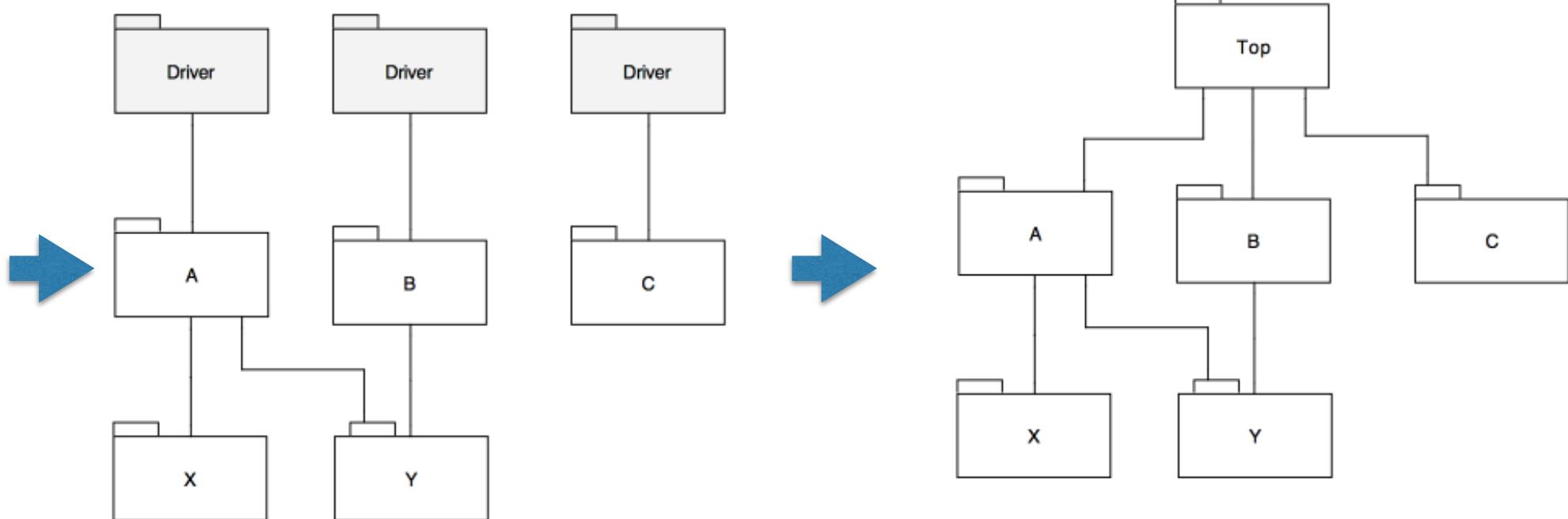
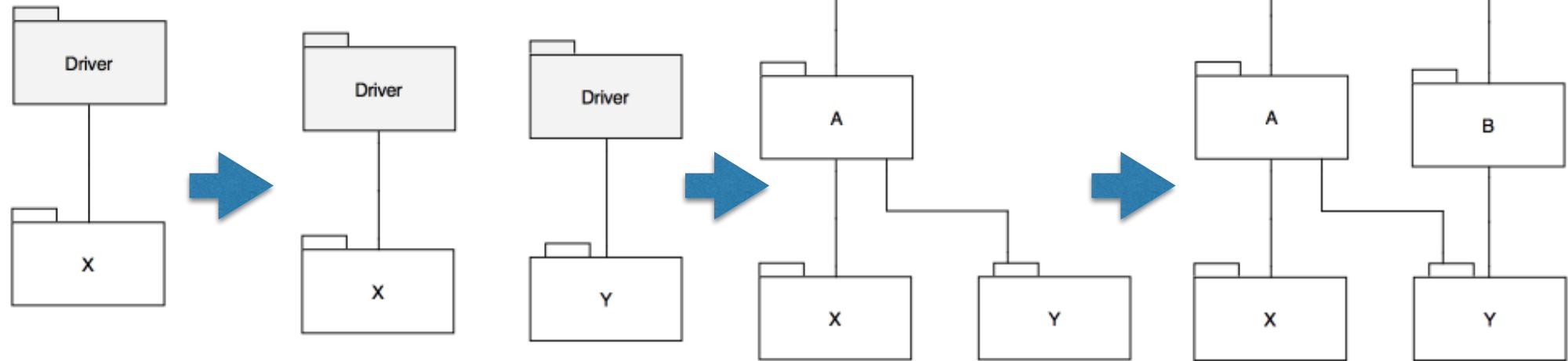
- Puede ser necesario escribir clases/métodos que no existen para probar el que nos interesa
- En ese caso se escribe lo mínimo necesario (stub)



Top Down

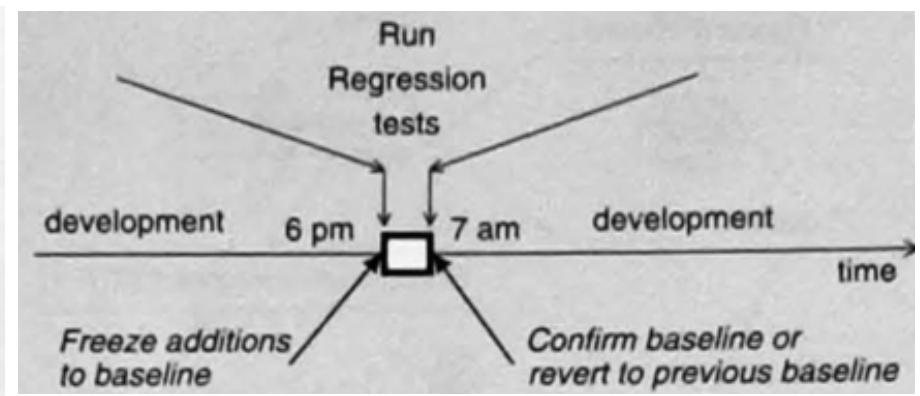
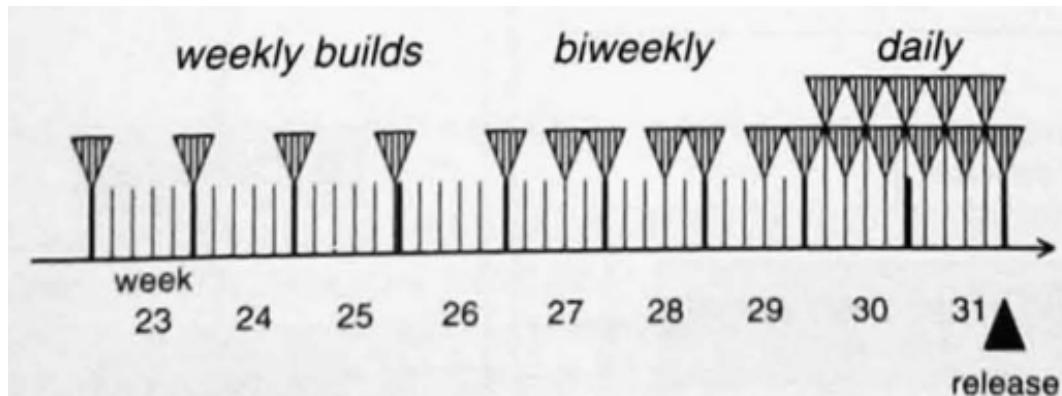


Bottom up



Integración Contínua

- Tipo de integración incremental en que se agregan piezas en forma muy frecuente (diario, semanal)
- El incremento ha sido probado en forma individual, se integra y si la línea de base no se daña queda



Test de Regresión

- puede ser visto como combinación de test unitarios y de integración
- asegurar que los cambios no introduzcan nuevos errores
- una suite de tests que puede incluir
 - tests de nuevas funcionalidades
 - una muestra representativa de tests que ejerciten todas las funciones
 - tests que se centren en las componentes modificadas

Desafíos de tests de regresión

- mantenimiento de la suite de tests
 - cuantos y cuales casos deben correrse al cambiar feature X
 - hay que eliminar, agregar o modificar tests de la suite
- cambio pequeño puede ocasionar costo alto de test
- test suites deben hacerse modulares
- automatización es un "must"

Test de Sistema y Test de Aceptación

- Sistema
 - Verificación (no validación) del sistema completo
 - Contra requisitos funcionales y no funcionales
 - Dos tipos
 - Alpha test - lo hace cliente en el sitio de desarrollo bajo dirección del jefe del proyecto
 - Beta test - lo hace un número de usuarios en su propio ambiente (registran cualquier problema)
- Aceptación - validación, efectuada por el cliente con el objeto de aceptar o rechazar el producto (puede durar semanas o meses)

Tests de Sistema, de aceptación y de regresión

	System	Acceptance	Regression
Test for ...	Correctness, completion	Usefulness, satisfaction	Accidental changes
Test by ...	Development test group	Test group with users	Development test group
	Verification	<i>Validation</i>	Verification

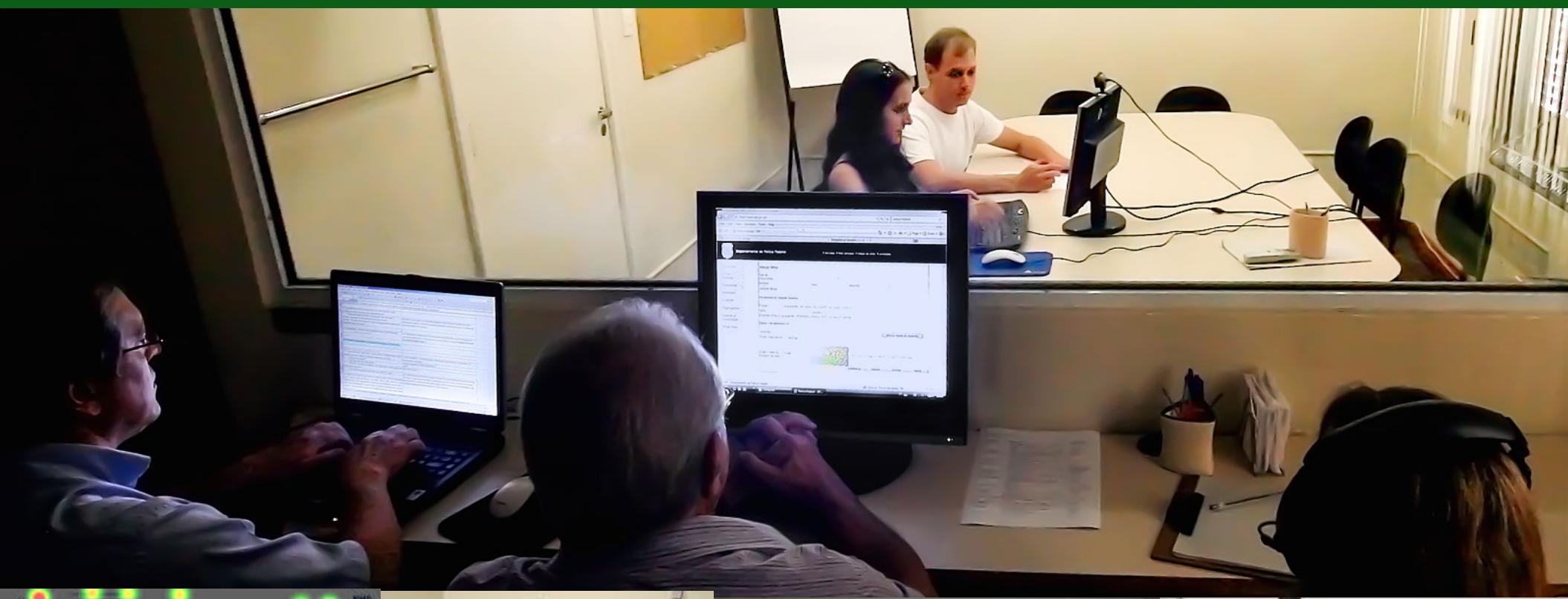
Tests No Funcionales

- Performance - se prueba que el sistema satisfaga los requerimientos de desempeño
- stress (carga) - se prueba que tan bien el sistema responde a una carga mayor de la planeada
- confiabilidad - se mide el tiempo medio entre fallas (MTBF)
- usabilidad - se mide tiempo promedio en completar tareas, tasa de errores del usuario, etc
- seguridad - hackers

Tests de Usabilidad



- observación de personas reales usando el software (5 a 8)
 - pueden usarse cámaras, trackers de mirada, etc
 - se registra todo
- se hacen mediciones sobre
 - tiempo necesario para llevar a cabo ciertas tareas específicas
 - errores en proceso (repeticiones, correcciones)
- opinión de los testers es menos importante (subjetivo)



A/B Testing (no funcional)

- Ampliamente usado con sitios Web
- Se ponen a prueba dos versiones de la página y se ve cual funciona mejor



Desempeño, carga, stress, escalabilidad

- tests de desempeño ayudan a detectar cuellos de botella
- hay muchas herramientas de software que ayudan a esta tarea (open source y comerciales)

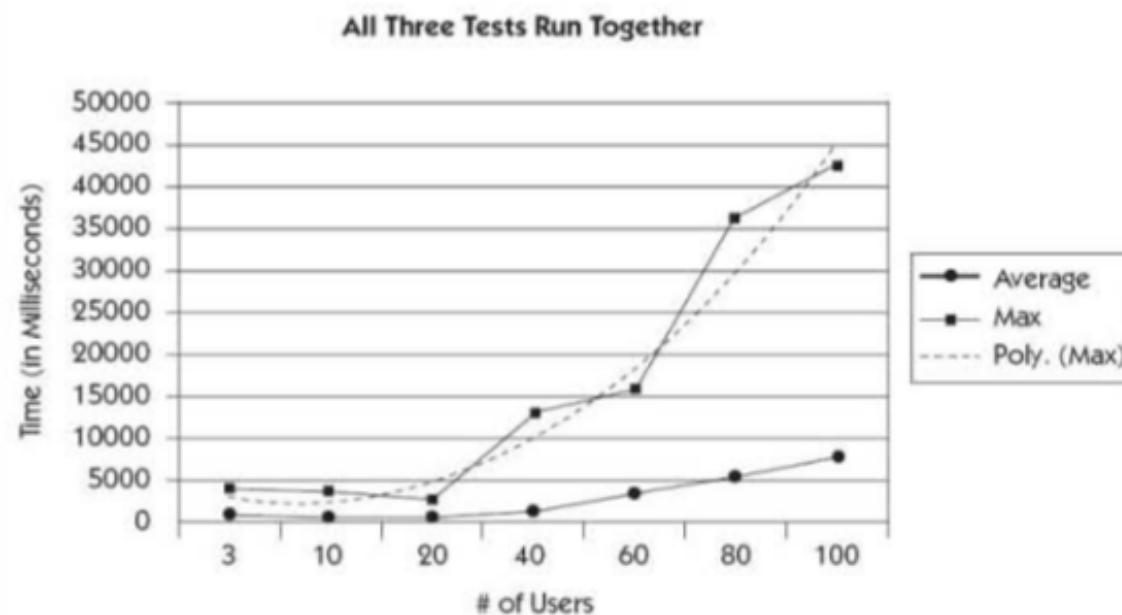
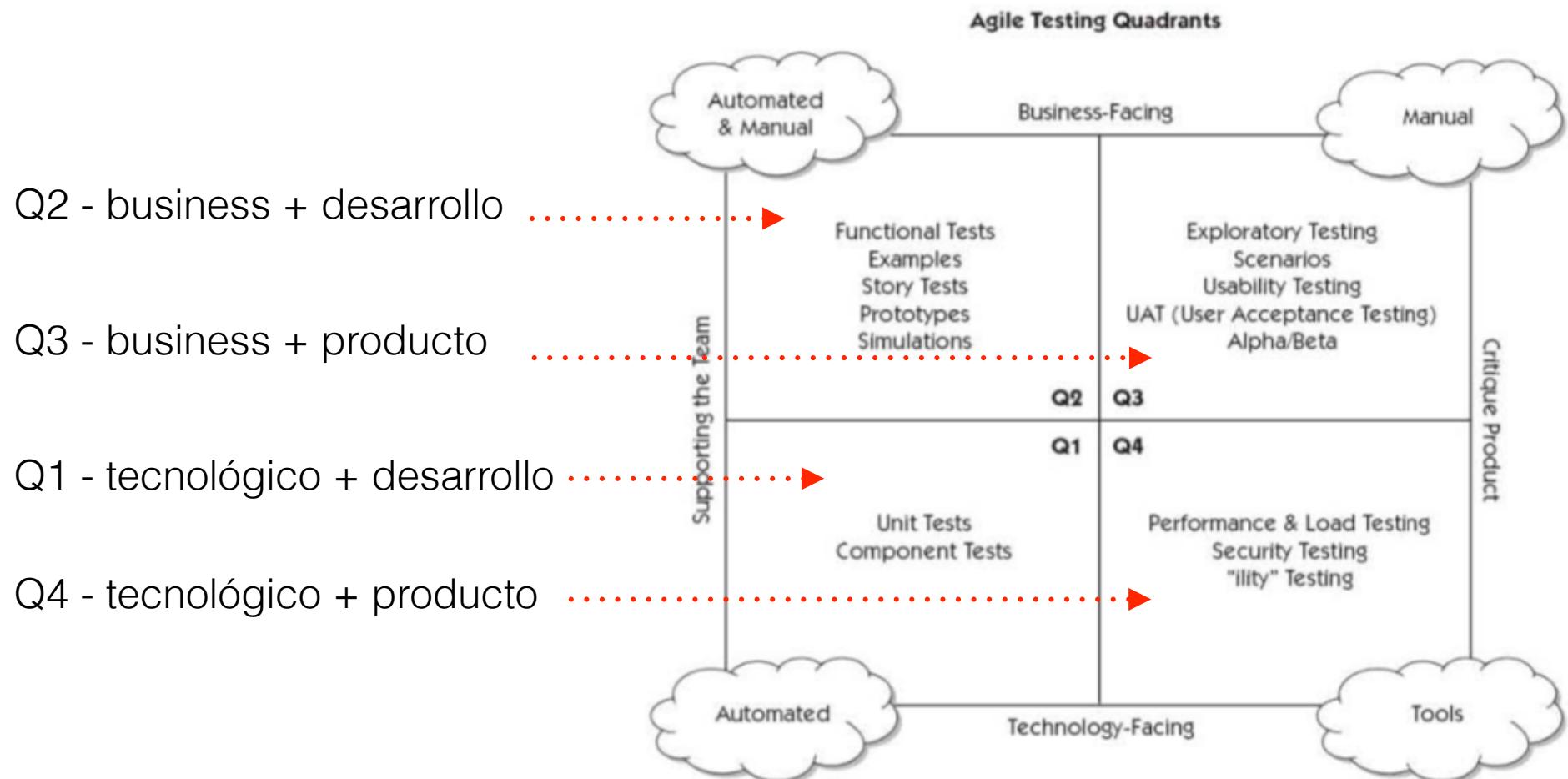


Figure 11-2 Max and average transaction times at different user loads.

Testing Agil

Eje X - apoyo al desarrollo vs crítica al producto

Eje Y - tests tecnológicos vs tests orientados al negocio



Testing como apoyo al desarrollo

- TDD - test driven development
- escribir los tests asociados a un trozo de código ANTES de escribir el código
- probar el código (que no existe) - fails
- escribir código para que pueda superar el test
- refactor
- escribir el siguiente test
- ciclo red/green/refactor

Tests Automatizados

- dependiendo de la plataforma hay herramientas que facilitan el desarrollo de los tests
- Rails desde sus inicios ha incluido herramientas para hacer TDD
 - test::unit
 - minitest (reemplaza al anterior a partir de 1.9.3, default)
 - gema rspec (recomendable)

RSpec

- Creada por Steven Baker en 2005
- Se escriben "specs" que son ejemplos ejecutables del comportamiento esperado en un cierto contexto dado
- Ejemplo: un spec con un ejemplo

```
describe MovieList do
  context "when first created" do
    it "is empty" do
      movie_list = MovieList.new
      movie_list.should be_empty
    end
  end
end
```

Terminología

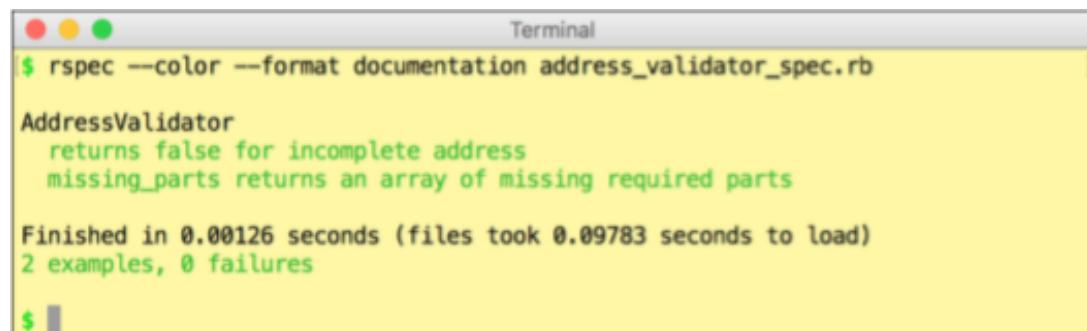
- Un test se llama un spec (especificación)
- En un spec pueden haber varios ejemplos (tests)
- Ejemplos evaluan el comportamiento esperado de la unidad en prueba

Ejemplo

address_validator.rb → address_validator_spec.rb

```
module AddressValidator
  FIELD_NAMES = [:street, :city, :region, :postal_code, :country]
  VALID_VALUE = /^[A-Za-z0-9\.\#]+$/  
class << self
  def valid?(o)
    normalized = parse(o)
    FIELD_NAMES.all? do |k|
      v = normalized[k]
      !v.nil? && v != "" && valid_part?(v)
    end
  end
  def missing_parts(o)
    normalized = parse(o)
    FIELD_NAMES - normalized.keys
  end
  private
  def parse(o)
    if (o.is_a?(String))
      values = o.split(",").map(&:strip)
      Hash[ FIELD_NAMES.zip(values) ]
    elsif (o.is_a?(Hash))
      o
    else
      raise "Don't know how to parse #{o.class}"
    end
  end
  def valid_part?(value)
    value =~ VALID_VALUE
  end
end
```

```
require 'rspec'  
require_relative 'address_validator'  
describe AddressValidator do
  it "returns false for incomplete address" do
    address = { street: "123 Any Street", city: "Anytown" }
    expect(AddressValidator.valid?(address))
    ).to eq(false)
  end
  it "missing_parts returns an array of missing required parts" do
    address = { street: "123 Any Street", city: "Anytown" }
    expect(AddressValidator.missing_parts(address))
    ).to eq([:region, :postal_code, :country])
  end
end
```



A screenshot of a terminal window titled "Terminal". The command entered is "\$ rspec --color --format documentation address_validator_spec.rb". The output shows two green test results: "AddressValidator returns false for incomplete address" and "missing_parts returns an array of missing required parts". At the bottom, it says "Finished in 0.00126 seconds (files took 0.09783 seconds to load)" and "2 examples, 0 failures".

Estructura de un archivo spec

```
require 'rspec'  
require_relative 'address_validator'  
describe AddressValidator do  
  it "returns false for incomplete address" do  
    address = { street: "123 Any Street", city: "Anytown" }  
    expect(AddressValidator.valid?(address))  
      .to eq(false)  
  end  
  it "missing_parts returns an array of missing required parts" do  
    address = { street: "123 Any Street", city: "Anytown" }  
    expect(AddressValidator.missing_parts(address))  
      .to eq([:region, :postal_code, :country])  
  end  
end
```

The diagram illustrates the structure of a RSpec example block. It features a large rectangular box labeled 'examples' at the top right. Inside this box, there is a smaller rectangular box labeled 'matchers' at its top right. Within the 'matchers' box, an arrow points from the word 'expect' to the code 'expect(AddressValidator.valid?(address))'. Another arrow points from the variable 'address' to the assignment 'address = { street: "123 Any Street", city: "Anytown" }'.

Ejemplo TDD : Gatherer

- app destinada a llevar un control de las tareas de un proyecto
- para cada tarea, status, desarrolladores asignados, etc
- interesa llevar registros de tiempos para completar las tareas

Primer test

- un proyecto sin tareas pendientes está terminado
- estado inicial al crear proyecto sin tareas debería ser "done"
- el modelo del project va en app/models/project.rb
- el spec (example) va en spec/models/project_spec.rb

spec/models/project_spec.rb

```
require 'rails_helper'
```

```
RSpec.describe Project do
```

```
  it "considers a project with no tasks to be done" do
```

```
    project = Project.new
```

```
    expect(project.done?).to be_truthy
```

```
  end
```

```
end
```

Corriendo el test

```
$ rspec  
gatherer/spec/models/project_spec.rb:3:in `<top (required)>':  
  uninitialized constant Project
```

El test falla porque aún no hemos definido Project
La corrección mas simple sería agregar archivo app/models/project.rb con

```
class Project  
end
```

Corriendo nuevamente el test

```
$ rspec
```

```
F
```

Failures:

- 1) Project considers a project with no tasks to be done

Failure/Error: expect(project.done?).to be_truthy

NoMethodError:

undefined method `done?' for #<Project:0x00000107ce67d0>

./spec/models/project_spec.rb:6:in `block (2 levels) in <top (required)>'

Finished in 0.00104 seconds (files took 1.29 seconds to load)

1 example, 1 failure

Failed examples:

```
rspec ./spec/models/project_spec.rb:4 #
```

Project considers a project with no tasks to be done

Nuevos arreglos

```
class Project
  def done?
    true
  end
end
```

```
$ rspec
```

```
.
```

Finished in 0.00105 seconds (files took 1.2 seconds to load)
1 example, 0 failures

Agregamos un Segundo Test a project_spec.rb

```
require 'rails_helper'
```

```
RSpec.describe Project do
  it "considers a project with no tasks to be done" do
    project = Project.new
    expect(project.done?).to be_truthy
  end
```

```
  it "knows that a project with an incomplete task is not done" do
    project = Project.new
    task = Task.new
    project.tasks << task
    expect(project.done?).to be_falsy
  end
end
```

Debemos Modificar el modelo ...

```
class Task  
end
```

← app/models/task.rb

```
class Project  
attr_accessor :tasks  
def initialize  
  @tasks = []  
end
```

← app/models/project

```
def done?  
  tasks.empty?  
end  
end
```

Aplicando principio DRY con let

```
require 'rails_helper'
RSpec.describe Project do
  describe "initialization" do
    let(:project) { Project.new }
    let(:task) { Task.new }

    it "considers a project with no test to be done" do
      expect(project).to be_done
    end

    it "knows that a project with an incomplete test is not done" do
      project.tasks << task
      expect(project).not_to be_done
    end
  end
end
```

Falta diferenciar entre tareas completas e incompletas

Introducimos un test en spec/models/task_spec.rb

```
require 'rails_helper'  
RSpec.describe Task do  
  it "can distinguish a completed task" do  
    task = Task.new  
    expect(task).not_to be_complete  
    task.mark_completed  
    expect(task).to be_complete  
  end  
end
```

y el modelo de tarea ...

```
class Task
  def initialize
    @completed = false
  end
  def mark_completed
    @completed = true
  end
  def complete?
    @completed
  end
end
```

Proyecto completado cuando las tareas lo están ...

Agregamos un nuevo test a spec de project

```
it "marks a project done if its tasks are done" do
  project.tasks << task
  task.mark_completed
  expect(project).to be_done
end
```

y en el modelo modificamos método done?

```
def done?
  tasks.reject(&:complete?).empty?
end
```

Cálculos de Tiempos

Queremos saber cuánto falta de un proyecto y también el tiempo de completación

Agregamos a project_spec.rb lo siguiente

```
describe "estimates" do
  let(:project) { Project.new }
  let(:done) { Task.new(size: 2, completed: true) }
  let(:small_not_done) { Task.new(size: 1) }
  let(:large_not_done) { Task.new(size: 4) }
  before(:example) do
    project.tasks = [done, small_not_done, large_not_done]
  end
  it "can calculate total size" do
    expect(project.total_size).to eq(7)
  end
  it "can calculate remaining size" do
    expect(project.remaining_size).to eq(5)
  end
end
```

y en modelo ...

```
class Task
  attr_accessor :size, :completed
  def initialize(options = {})
    @completed = options[:completed]
    @size = options[:size]
  end
  def mark_completed
    @completed = true
  end
  def complete?
    @completed
  end
end
```

y project

```
class Project
  attr_accessor :tasks
  def initialize
    @tasks = []
  end
  def incomplete_tasks
    tasks.reject(&:complete?)
  end
  def done?
    incomplete_tasks.empty?
  end
  def total_size
    tasks.sum(&:size)
  end
  def remaining_size
    incomplete_tasks.sum(&:size)
  end
end
```

ahora sí

```
Tatooine:gatherer jnavon$ rspec
Running via Spring preloader in process 99216
.....
Finished in 0.00627 seconds (files took 2.97 seconds to load)
5 examples, 0 failures
```

Introduzcamos un error

- Supongamos que alguien modifica el código de total_size a ...

```
def total_size
  tasks.sum(&:size) - 1
end
```

```
Tatooine:gatherer jnavon$ rspec
Running via Spring preloader in process 99248
...F.
```

Failures:

- 1) Project estimates can calculate total size

```
Failure/Error: expect(project.total_size).to eq(7)
```

```
expected: 7
      got: 6
    (compared using ==)
# ./spec/project_spec.rb:37:in `block (3 levels) in <top (required)>'
```

```
Finished in 0.02347 seconds (files took 1.95 seconds to load)
5 examples, 1 failure
```

Failed examples:

```
rspec ./spec/project_spec.rb:36 # Project estimates can calculate total size
Tatooine:gatherer jnavon$
```

BDD

- Lo que el objeto hace es lo importante y no su estructura interna
- Lo mismo es válido a nivel de aplicación (no interesa si se usa MySQL o SQLite)
- El foco de los tests es el comportamiento del software

Ejemplo BDD (Cucumber)

Feature: Address Validation

As a postal customer,
In order to ensure my packages are delivered,
I want to validate addresses

Scenario: Invalid address

Given I enter "Seoul, USA"
When I validate the address
I should see the error message, "City and Country do not match"

¿ Es realmente costosa la calidad ?

- Costo promedio de corregir un error (waterfall)
 - durante la generación de código - US\$977
 - durante pruebas de sistema - \$7136
- Prevenir es mas barato que curar



"Good enough" software

- Si produces software de muy mala calidad nadie lo comprará o usará
- Si gastas una infinita cantidad de tiempo o dinero en hacerlo perfecto estarás fuera del negocio
- La industria trata de ubicarse en algún punto "mágico" intermedio
- Cuanto es "good enough" depende del producto y de la compañía

Organización de personal SQA

- en 50% de empresas - encargada de todo tipo de tests, reporta a vicepresidente de SE o CIO (hasta 25% del total de ingenieros)
- en 35% de empresas - responsables de estimar y medir calidad y adherencia a estándares, separada del área de testing y no es parte de grupo de desarrollo
- en 10% de empresas - no hay SQA, solo un grupo de testing