ADC
Applaudo
Developers
Conference
2020

Powered by ApplaudoStudios

Angular Testing

The importance of testing your code

Carlos Lopez

Technologies

@clopez-app

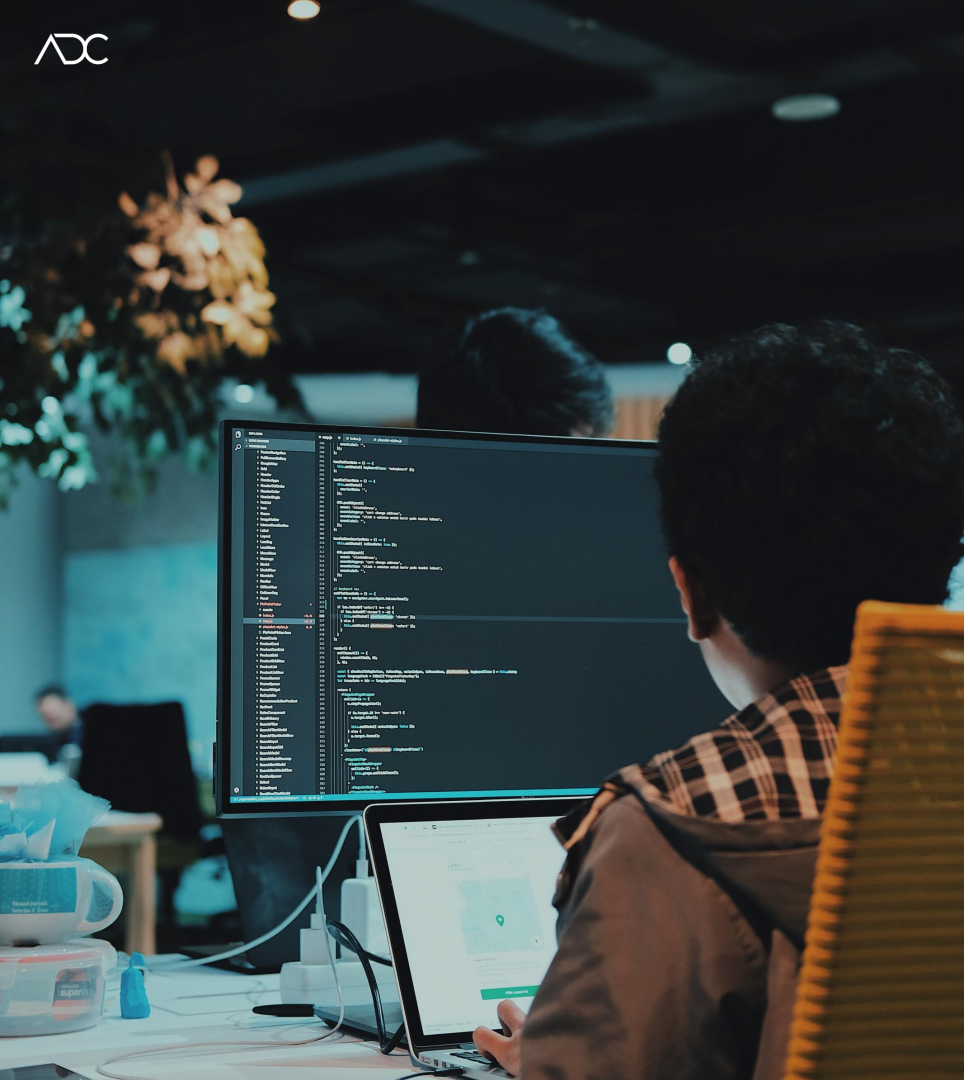
@carlos-alopez

Agenda

1. Importancía de realizar test.
2. Unit Tests / Pruebas unitarias.
3. Integration Tests / Pruebas de integración.
4. End-to-end Tests / Pruebas de extremo a extremo.
5. Live code.
6. QA

¿Por qué es importante?

Qué hace para nosotros y para el cliente.

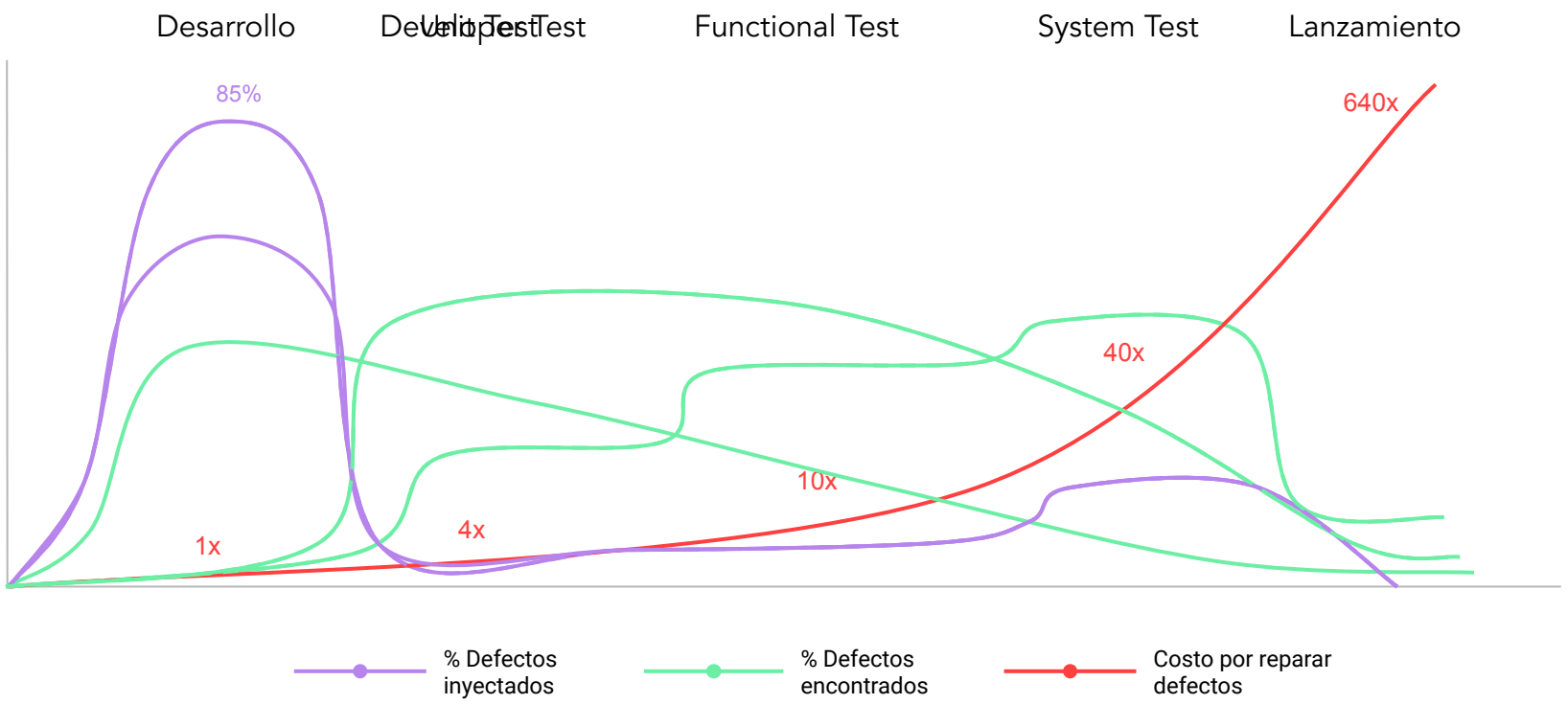


Beneficios personales

- Fortalece el trabajo en equipo.
- Mejora el diseño y calidad de código.
- Incrementa la velocidad de desarrollo.
- Se evita código innecesario.
- Provee documentación.
- Facilita refactorización de código.
- Permite tranquilidad.
- Promueve el aprendizaje.
- Genera satisfacción para quienes usan el código.

Análisis global de productividad y calidad

Applied Software Measurement: Global Analysis of Productivity and Quality. Jones, Capers.



Unit Test / Pruebas Unitarias

Unidades individuales de código.

Los errores detectados se localizan y solucionan fácilmente

¿Cómo funciona una prueba unitaria?

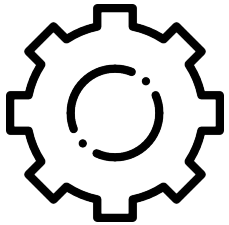
Consiste en comprobar si unidades individuales de código se encuentran correctamente implementadas para su uso. Estas unidades individuales están definidas como el menor componente o fragmento de código que sea totalmente funcional y al mismo tiempo pueda ser probado, usualmente realizando una sola función cohesiva.

¿Cuál es el objetivo?

Segregar cada parte del programa y asegurarse que las partes individuales estén funcionando correctamente.

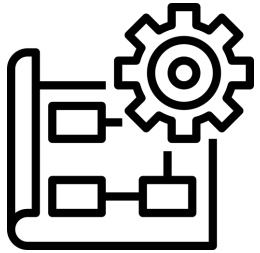
Tipos de pruebas unitarias

Existen dos tipos de Pruebas Unitarias: Prueba Unitarias Aisladas/ Isolated Unit Tests o Pruebas Unitarias poco profundas / Shallow Unit Tests



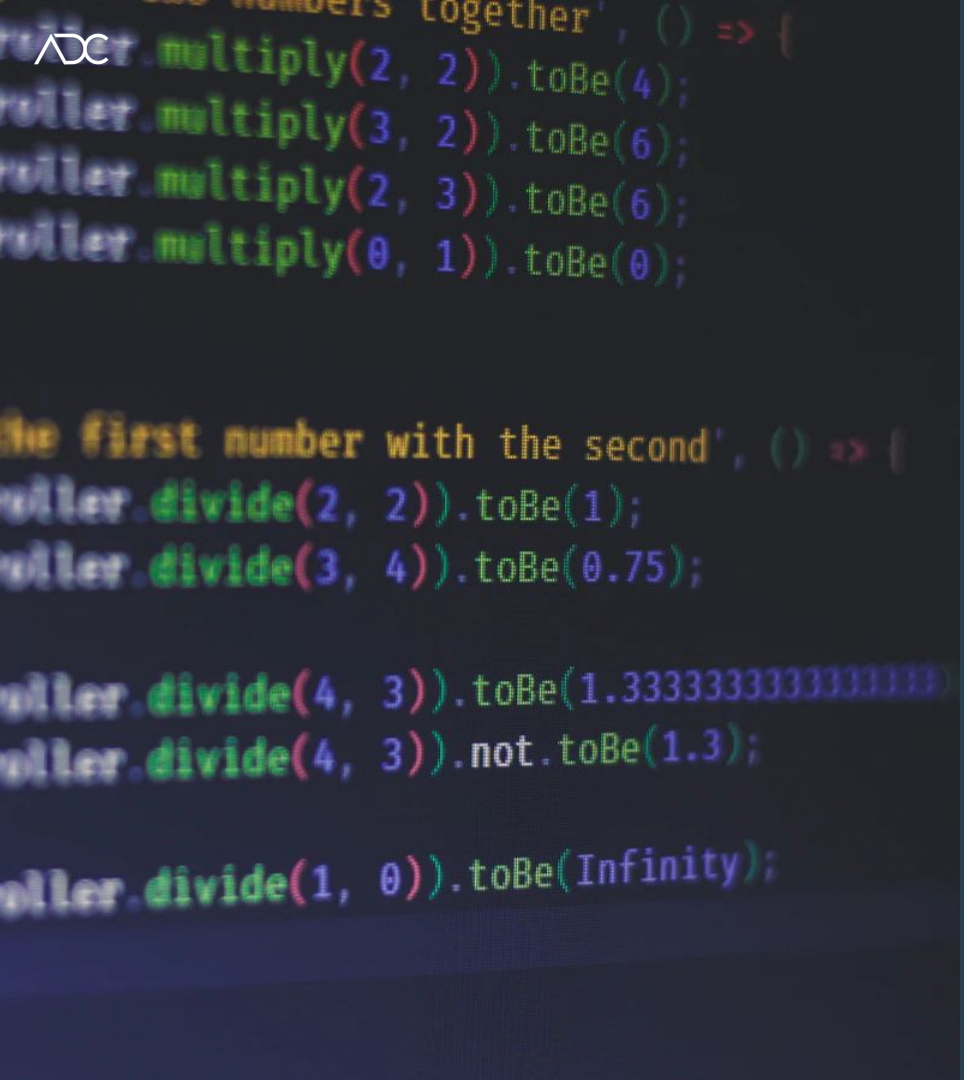
Isolated Unit Test

Solo se prueban las funciones del componente



Shallow Unit Test

Se prueba las funciones y el template del componente, pero evitando renderizar los componentes hijos.



Beneficios:

- Crea un proceso ágil .
- Buena calidad de código.
- Detecta errores de manera temprana.
- Facilita cambios y simplifica la integración.
- Proporciona documentación.
- Facilita el proceso de depuración.
- Mayor claridad de diseño.
- Reduce costos.
- Permite mayor comunicación de equipo.

```
@Component({
  selector:
'lightswitch-comp',
  template: `
    <button
(click)="clicked()">Clic
k me!</button>

<span>{{message}}</span>
`
})
export class
LightswitchComponent {
  isOn = false;
  clicked() { this.isOn
= !this.isOn; }
  get message() { return
`The light is
${this.isOn ? 'On' :
'Off'}`; }
}
```

```
describe('LightswitchComp', () => {
  it('#clicked() should toggle #isOn', () => {
    const comp = new LightswitchComponent();
    expect(comp.isOn).toBe(false, 'off at first');
    comp.clicked();
    expect(comp.isOn).toBe(true, 'on after click');
    comp.clicked();
    expect(comp.isOn).toBe(false, 'off after second click');
  });

  it('#clicked() should set #message to "is on"', () => {
    const comp = new LightswitchComponent();
    expect(comp.message).toMatch(/is off/i, 'off at first');
    comp.clicked();
    expect(comp.message).toMatch(/is on/i, 'on after clicked');
  });
});
```

Integration Test / Pruebas de Integración

Modulos de código. Interacción de unidades.

Detecta errores en la interfaz del programa, asegurando la calidad y desempeño del producto.

¿Cómo funciona una prueba de integración?

Se incluyen unidades de código, previamente probadas, con el fin de completar por lo menos un módulo de funcionalidad dentro del programa. Luego se realizan pruebas para comprobar que el resultado de las funciones en conjunto sea el esperado.

¿Cuál es el objetivo?

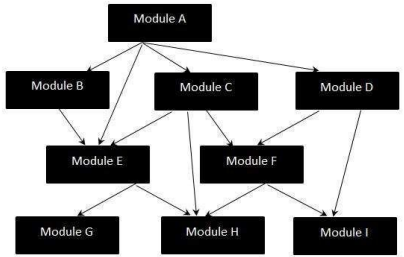
Tomar unidades individuales y comprobar la funcionalidad de su interacción.

Características generales

- Se realiza después de completar las pruebas unitarias.
- La integración de cada módulo está definida en un plan de prueba

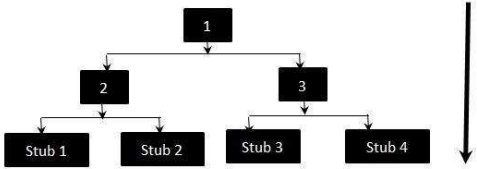
Tipos de pruebas de integración

Las pruebas de integración pueden clasificarse en 4 tipos diferentes, enfocados en las diferentes maneras de comprobar la funcionalidad de un programa.



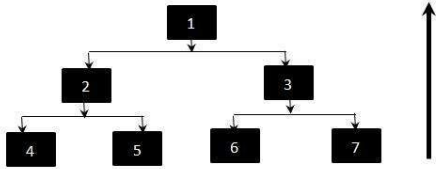
Big Bang

Conveniente para aplicaciones pequeñas. Detecta errores de lógica fácilmente.



Descendente

Módulos cruciales para el programa son priorizados. Detecta errores de manera eficiente.



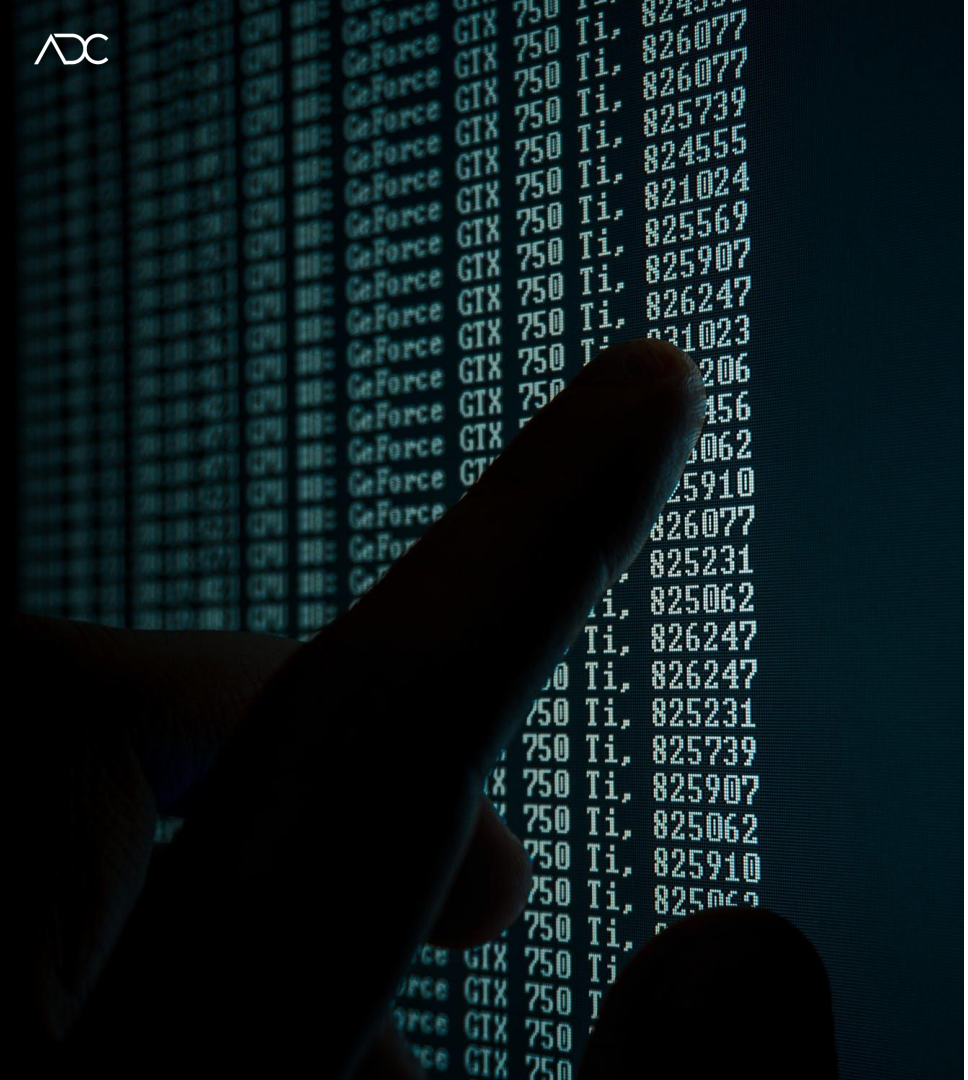
Ascendente

Preferible para el desarrollo de un módulo individual. Requiere menos tiempo.

Ascendente
+
Descendente

Sandwich

Utilizado para resolver las complejidades de programas grandes.



Beneficios:

- Permite integrar módulos diferentes con mayor facilidad.
- La cobertura del código es alta y fácil de seguir.
- Proporcionan mayor claridad con casos de uso real.
- Se realizan con mayor velocidad que los test de extremo a extremo.
- Las pruebas ascendentes y/o descendentes pueden ser utilizadas desde las primeras etapas del desarrollo, encontrando errores rápidamente.
- Las pruebas son fáciles de realizar ya que previamente se ha confirmado la funcionalidad de las unidades.
- Permite encontrar errores por nivel, errores de base de datos, errores de integración y más.

Ejemplo

List of users, component and service.

```
@Injectable({ providedIn: 'root' })
export class UserService {
  constructor(private http: HttpClient) {}

  get(): Observable<User[]> {
    return
this.http.get<User[]>('https://jsonplaceholder.typicode.com/users');
  }
}
```

```
@Component({
  selector: 'app-users',
  template: `
    <h3>Users</h3>
    <ul *ngIf="users$">
      <li *ngFor="let user of users$ | async">{{ user.name }} ({{
user.email }})</li>
    </ul>

    <button id="load" (click)="load()">Load</button>
  `
})
export class UsersComponent {
  users$: Observable<User[]>;

  constructor(private userService: UserService) {}

  load() {
    this.users$ = this.userService.get();
  }
}
```

```
it('loads the users and displays them if user clicks on button,' () => {
    // acquire services and set up spies
    const fixture = TestBed.createComponent(UsersComponent);
    const userService: UserService = TestBed.get(UserService);
    spyOn(userService, 'get').and.callThrough();
    const httpController: HttpTestingController = TestBed.get(HttpTestingController);

    // click on Load button
    const button = fixture.nativeElement.querySelector('#load');
    button.click();

    // validate outgoing request to API and provide dummy data
    expect(userService.get).toHaveBeenCalled();

    expect(fixture.componentInstance.users$).toBeTruthy();

    fixture.detectChanges();

    const testRequest = httpController.expectOne('https://jsonplaceholder.typicode.com/users');
    expect(testRequest.request.method).toEqual('GET');
    testRequest.flush(dummyUsers);

    // validate presentational changes
    fixture.detectChanges();

    const listItems = fixture.nativeElement.querySelectorAll('li');
    expect(listItems.length).toEqual(dummyUsers.length);
  });
```

End-to-end Test / Pruebas de extremo a extremo

Flujo de interacción completos.

Asegura la funcionalidad y la salud de la aplicación.

¿Cómo funciona una prueba de extremo a extremo?

Evalúa el programa de principio a fin para asegurarse que el flujo de la aplicación se comporta como lo esperado.

¿Cuál es el objetivo?

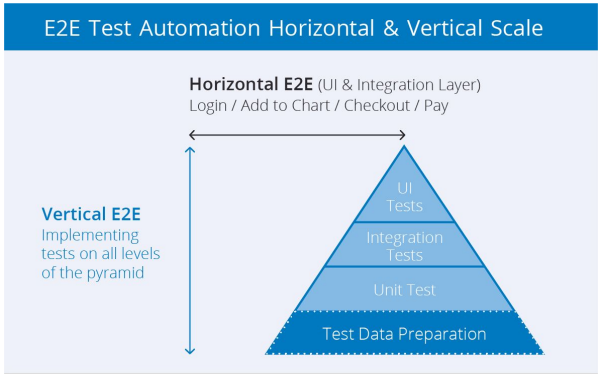
Simular la experiencia del usuario en un escenario real. Validar la integración de los componentes y la información.

Ciclo de vida:

- Planeación de las pruebas.
- Diseño de pruebas.
- Ejecución de pruebas.
- Análisis de resultados.

Tipos de pruebas de extremo a extremo

Existen dos tipos de maneras para este tipo de pruebas.



Horizontal

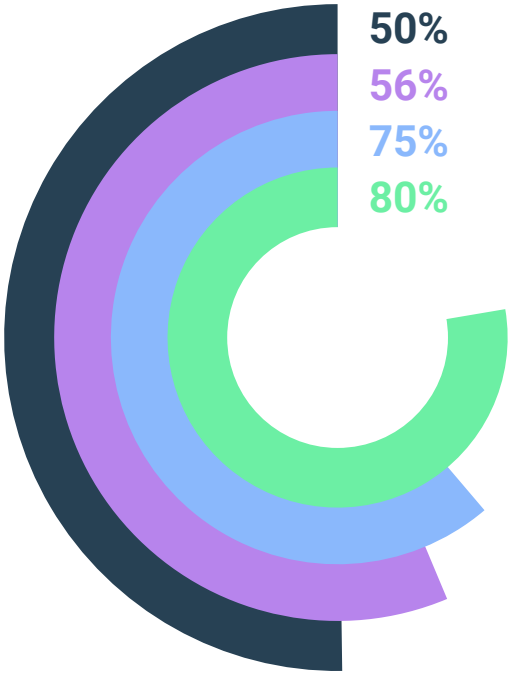
- Se enfoca en realizar test desde la perspectiva de un usuario.
- Previene que los errores sucedan en producción.
- Asegura que exista cobertura para los requisitos de la lógica de la empresa.

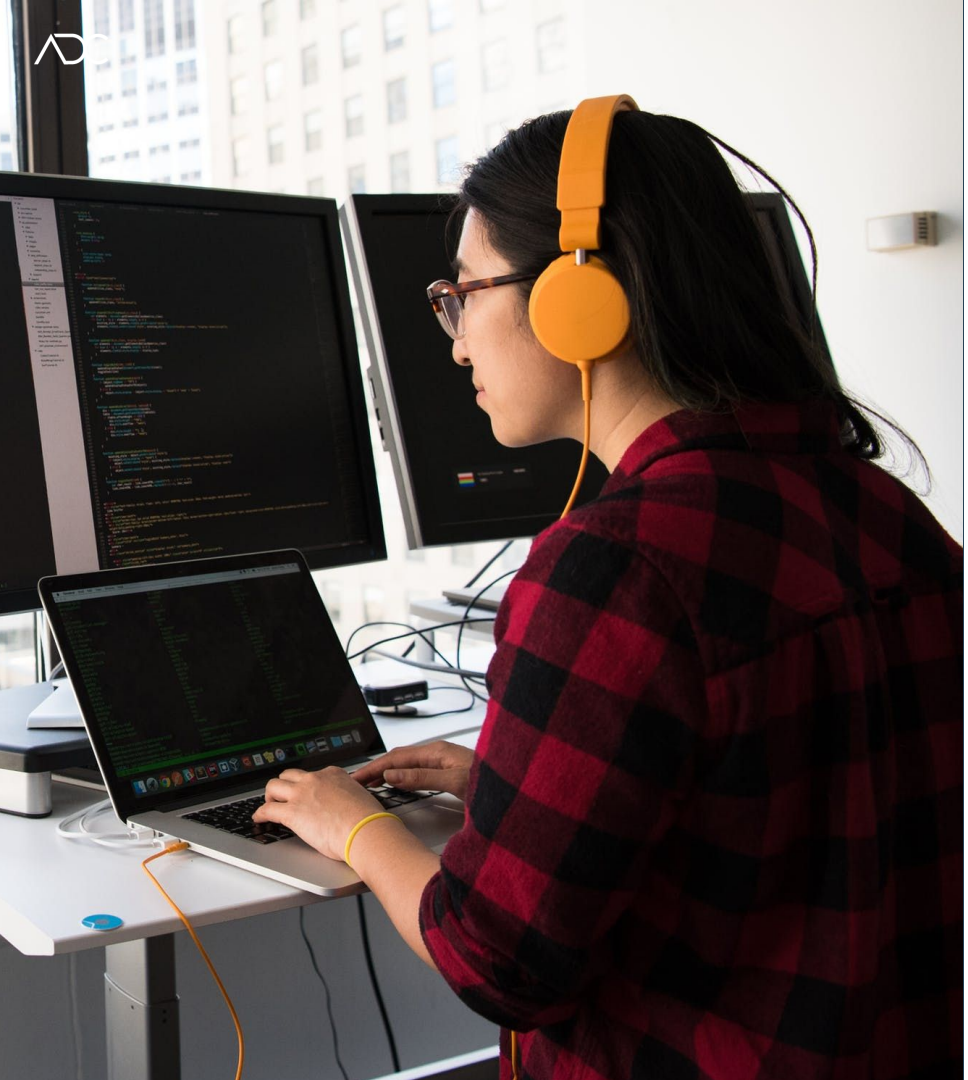
Vertical

- Asegura un alto nivel de cobertura.
- Los test se ejecutan con mayor velocidad.
- Mayor enfoque para los test.
- Muy útil para revisar partes críticas del programa.

Métricas importantes

- Estado de preparación de los casos de prueba.
- Seguimiento del proceso de prueba.
- Estado de defectos y detalles.
- Disponibilidad del ambiente.





Beneficios:

- Aumenta la cobertura de las pruebas.
- Asegura la funcionalidad y la salud de la aplicación.
- Reduce el tiempo para ser lanzado al mercado.
- Reduce costos y tiempo.
- Detecta errores.
- Incrementa la confianza en el producto.
- Reduce riesgos futuros.

```
exports.config = {
  seleniumAddress:
'http://localhost:4444/wd
/hub',
  specs: ['todo-spec.js']
};
```

```
  it('should add a todo', function() {
    page.navigateTo();

    element(by.css('.todoInput')).sendKeys('write first
protractor test');
    element(by.css('.btn-primary')).click();

    let todoList = element.all(by.css('.todo-element'));
    expect(todoList.count()).toEqual(3);
    expect(todoList.get(2).getText()).toEqual('write first
protractor test');

    // You wrote your first test, cross it off the list
    todoList.get(2).element(by.css('input')).click();
    let completedAmount = element.all(by.css('.done-true'));
    expect(completedAmount.count()).toEqual(2);
  });
```

Angular Test - Live code

¿Preguntas?

Unidades individuales de código.

Applaudo
Developers
Conference
2020

Powered by ApplaudoStudios