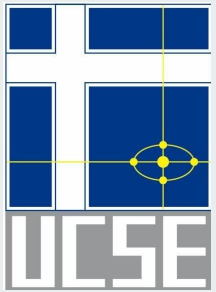
Unit testing, TDD, CI

UCSE - SEIA



Repaso de Unit Testing

- Al escribir unit tests automatizados, estamos automatizando la validación de que el código hace lo que queremos que haga.
- Cada vez que alguien hace un cambio, puede ejecutar los unit tests, y ellos le dicen si rompió algo de lo que ya existía.
- Lleva tiempo, no es gratis, pero es una ventaja muy grande!
- Bien hecho nos da una super tranquilidad al hacer cambios sobre el código existente. Impide que muchos errores lleguen a producción.

Repaso de Unit Testing

Más ventajas: tests como docs

- Tests bien hechos, nos sirven como otra cosa: documentación!
- Cada test de unidad nos muestra cómo usar un fragmento de código, y qué esperar como resultado.
- Unit tests claros sirven como una gran librería de ejemplos de uso de nuestro código.

Repaso de Unit Testing

Más ventajas: código testeable es código ordenado

- Para poder escribir buenos unit tests, nuestro código tiene que estar poco acoplado, bien estructurado en unidades independientes, testeables, con pocos side effects y dependencias de estado global.

- Escribir unit tests muchas veces nos fuerza a escribir buen código para que sea fácil de testear.

TDD

TDD es una práctica que a partir de estas ventajas, propone escribir primero los tests, y luego el código.

De esa forma:

- Vamos a primero diseñar cómo queremos usar el código, cómo sería práctico que funcione.
- Vamos a forzarnos desde el inicio a que el código tenga pocos side effects, esté bien desacoplado, etc, para poder escribir los tests.
- Como resultado, el código que escribamos después, va a ser más mantenible.
- No es mágico. Programadores con malas prácticas van a escribir malos tests y luego mal código.
- Fomenta, no garantiza.

Integración continua

- Teniendo unit tests (sin importar si usamos o no TDD), tenemos una herramienta super poderosa para chequear código automáticamente... por qué depender de que los usuarios la usen? Podemos automatizar su ejecución también!
- Es decir, no solo esperar a que los usuarios ejecuten los tests. Podemos hacer que se ejecuten solos!
- Podemos tener un servidor que esté mirando nuestro repositorio de código, y cada vez que ve nuevos commits, corre los tests.
- Si algún test falló, que envíe un mail al equipo avisando: "El commit que John Doe agregó al repo, rompe los tests!".
- No dependemos de que la gente recuerde ejecutar los tests.
- Nos enteramos inmediatamente cuando alguien rompe lo que tenemos todos para trabajar.
- Sabemos inmediatamente quién fue, para que pushee cambios rotos y se borre por tres días!
- Nos enteramos a tiempo, antes de empezar a trabajar sobre cosas rotas con nuevos cambios no relacionados.

Integración continua 2.0: Branches y merge/pull requests

- Si el servidor puede saber que esos cambios rompen el código... por qué permitir que sean parte de la rama oficial de desarrollo??
- Podemos trabajar con un nuevo paso intermedio: pull requests o merge requests.
- Cuando un programador trabaja, lo hace en un branch (una rama) de git diferente a master.
- Cuando termina sus cambios, pushea su branch al servidor (pero no a master!), y "propone" los cambios por medio de una web (crea un pull o merge request), donde los demás pueden verlos. Esa propuesta básicamente dice "propongo mergear este branch", solo que en la web se ve el diff más lindo, se puede discutir, etc.
- Y en ese momento, la integración continua corre los tests en ese branch y valida si pasan. Si los tests no pasan, no se permite mergear esos cambios a master!
    - No dependemos de que la gente recuerde ejecutar los tests.
    - Nunca entran a master cambios que rompan el código. Solo entran cosas que los tests validaron que no rompen nada.
    - El desarrollador tiene chances de arreglar los problemas sin molestar a los demás. El que rompió, no está presionado a arreglarlo cuanto antes! Un alivio gigante. Se puede ir de fiesta el finde y verlo el lunes.

Code reviews (i)

Dijimos que solo se puede mergear un merge/pull request, si los tests pasan.

Pero quién lo mergea? Cuándo?

Una buena práctica es aprovechar ese paso para hacer code review!

1. Otra persona, no el programador original, revisa si los cambios le parecen buenos.
2. Se puede discutir en el merge/pull request, y seguir agregando cambios. Ej: "me parece que esto es mejor de otra forma", "ok! ya lo cambio y pusheo!".
3. Solo cuando la segunda persona considera que está todo ok (y si pasan los tests!), se mergea a master.

Code reviews (ii)

En muchos lados, se exige que N personas lo revisen y aprueben antes de ser mergeado. Depende de cuánto estamos dispuestos a gastar, y qué tan delicado sea.

Ventajas:

- Ayuda a encontrar cosas temprano: varios programadores ven más que uno.
- Ayuda a repartir ownership!! todo el código fue visto por al menos dos personas diferentes, ya no hay un "eso solo lo conoce pepito".
- Ayuda a transmitir conocimiento a los nuevos! Los fuerza a estar mirando cómo hace las cosas el resto del equipo.

Problemas:

- Hacerlo de verdad lleva tiempo, trabajo.
- Hay que saber dar feedback respetuoso y constructivo. Si no, es fuente de peleas.
- Hay gente que por desinterés va a aprobar cosas sin mirar, hay que educar para que no pase.

Integración continua revolutions: automatizando de todo

- Puede utilizar herramientas que detecten errores de código automáticamente, o que chequeen el estándar de estilo de código que usemos (ej en Python: pyflakes, pep8, etc).
- Puede compilar y publicar la documentación, si la tenemos como parte del repo!
- Puede incluso... deployar nuestro sitio web!
- Si tenemos automatizado el deploy de nuestro sitio o app (es decir: que el deploy sea un programa que ejecutamos, no pasos que un humano tiene que hacer a mano)...
- Cada vez que alguien mergea cosas nuevas a master, podría deployar a un servidor de pruebas automáticamente. Cada vez que etiquetamos en el repo un release nuevo, podría deployar automáticamente a producción.
- DEPLOY AUTOMÁTICO A PRODUCCIÓN???!! :O
- Sí! Esto se llama Continuous Deployment, y se hace en muchos lados.

Automatizando el deploy

Cómo hacemos esto?. Básicamente necesitamos un programa que pueda hacer todo esto sin intervención humana:

- Instalación de dependencias (motor de base de datos, servicios, libs, etc)
- Configuración de servicios (ej: crear usuario en la db, abrir puertos, etc)
- Iniciar y reiniciar nuestras apps (levantar el servidor web, etc)
- Si tenemos más de una app, entender de las dependencias entre sí (no levantar el server web sin el de la API de la que depende, etc).
- No es una locura, es accesible. Hay todo un mundo de herramientas alrededor de esto: Ansible, Docker, Kubernetes, Cheff, Puppet, ...
- Y no se puede obviar...: en Linux es mucho más fácil automatizar que en Windows (pero están mejorando).
- Y hay empresas que proveen esto como servicio (Amazon, Heroku, etc).

Automatizando la infraestructura

Hasta ahora asumimos que el server "existe", solo estamos automatizando la instalación y config de software.

Pero hoy, a escala, también se tiende a automatizar la infraestructura! Por ejemplo:

- Que se levanten más máquinas virtuales si se detecta carga muy grande (por ejemplo por tráfico de un evento en vivo).
- Que se den de baja máquinas si la carga es baja.
- Que al actualizar, se vayan reemplazando de a una máquina con la versión nueva, progresivamente.
- ...

Las máquinas son claramente descartables, se crean y destruyen todo el tiempo.

La data no va en esas máquinas!

Herramientas como Docker y Kubernetes nos permiten resolver incluso a este nivel.

Resumiendo

1. Programador crea un branch. Agrega código y tests. Quizás hace TDD, quizás no, pero tests escribe.
2. Programador pushea branch, y crea un merge/pull request en el repo.
3. El server de CI corre los tests, valida calidad, y compila doc de ese branch.
4. Otros programadores hacen code review, quizás hay varias idas y vueltas de actualizar y agregar más cambios.
5. Cuando tenga suficientes +1 y los tests pasen, se mergea!
6. Al mergear a master, el server de CI vuelve a correr los chequeos y tests en master, y además deploya master al server de pruebas.
7. Cuando alguien quiera, marca un tag en master como un release, y el server de CD deploya eso a producción.

Unit testing, TDD, CI

UCSE - SEIA