# Capítulo 7: Pruebas

Víctor Iranzo

26 de julio de 2018

## 1. Tipos de pruebas

El siguiente esquema es una modificación del cuadrante de Marick empleada para catalogar pruebas en distintos tipos:

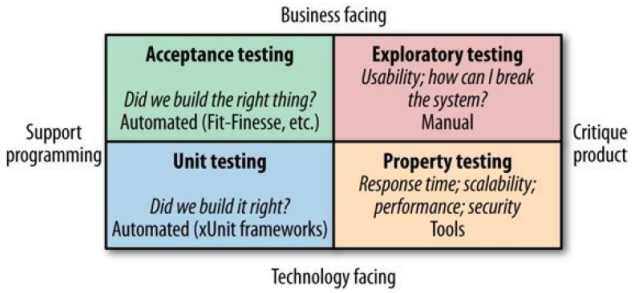


Figura 1: Modificación del cuadrante de Marick hecha por L.Crispin y J.Gregory en su libro Agile Testing.

Las pruebas situadas en la mitad inferior están relacionadas con la tecnología empleada, como son las pruebas unitarias o las de rendimiento, que deben ser implementadas por los desarrolladores y que se pueden automatizar. Las pruebas situadas en la mitad superior están orientadas a que los stakeholders conozcan cómo funciona el sistema y que no guardan relación con aspectos tecnológicos. Entran dentro de esta mitad las pruebas de aceptación o las pruebas manuales.

La proporción necesaria de cada tipo de test depende del sistema a desarrollar. La tendencia actual es en favor de los tests de pequeña escala y automatizados. En un sistema basado en microservicios, son esenciales las pruebas de este tipo ya que validarán que no se despliegue a producción código defectuoso.

Podemos clasificar las pruebas en los microservicios en 3 tipos según su alcance: unitarias, de servicios y de extremo a extremo. Estos términos pueden llevar a discusión: por ejemplo, en una prueba unitarias se puede incluir para algunas personas la validación de diferentes clases o funciones mientras que para otras no.

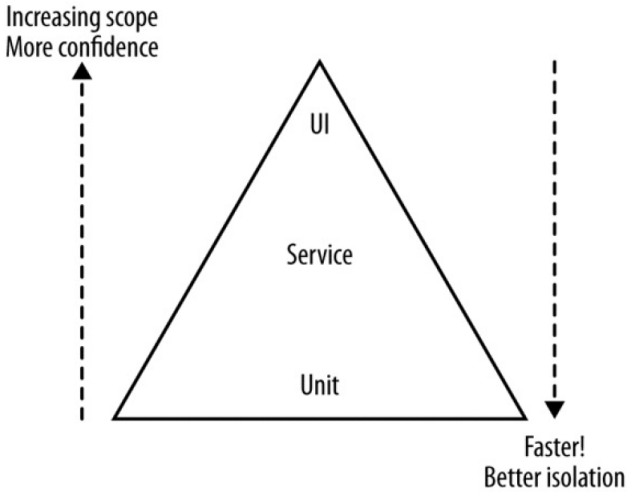


Figura 2: Pirámide de pruebas diseñada por Mike Cohn en su libro Succeding with Agile.

## 1.1. Pruebas unitarias

Son pruebas que validan una única función. Se pueden generar fruto de procesos de desarrollo como el test-driven development (TDD). En general, se prefiere tener un gran número de este tipo de pruebas por su rapidez, porque pueden ayudar a la refactorización del código y porque es donde mayor cantidad de defectos se suele capturar.

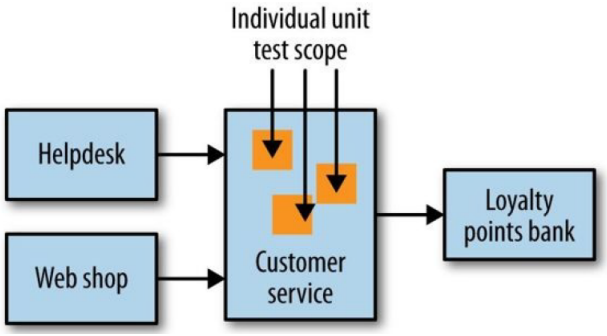


Figura 3: En las pruebas unitarias no se ejecuta ningún servicio, sino que se prueban métodos específicos de ellos.

## 1.2. Pruebas de servicios

En estas pruebas se verifica la funcionalidad de los servicios sin emplear las interfaces de usuario. Cada prueba verifica una de las funcionalidades que el servicio expone. Se pretende comprobar la separación entre los servicios y para solventar las dependencias que el servicio bajo pruebas tiene sobre otros, se reemplazan los servicios colaboradores por stubs o fakes.

Los stubs y los fakes imitan el comportamiento real del servicio al que representan. Los stubs contienen menos lógica que los fakes y ante una petición responden siempre de la misma manera. Por su parte, los fakes pueden emplearse para comprobar que los efectos secundarios de la interacción entre los servicios ocurre. Por ejemplo, si al invocar a un servicio el comportamiento esperado es que este añada una tupla en base de datos, un stub podría responder simplemente con un 200 OK mientras que un fake tendría más lógica para reproducir esto.

Este tipo de pruebas pueden ser igual de rápidas que las unitarias siempre que no se tengan que emplear un gran número de infraestructuras como bases de datos o redes.

## 1.3. Pruebas de extremo a extremo

Son pruebas que se ejecutan sobre todo el sistema. Cubren gran parte de código, por lo que su correcta ejecución dan mucho grado de confianza. En su ejecución se levantan varios servicios diferentes.

De los servicios que se prueban, ¿qué versión se debe emplear de cada uno? ¿Quién debe poder ejecutar esta prueba? Cada vez que se realice un

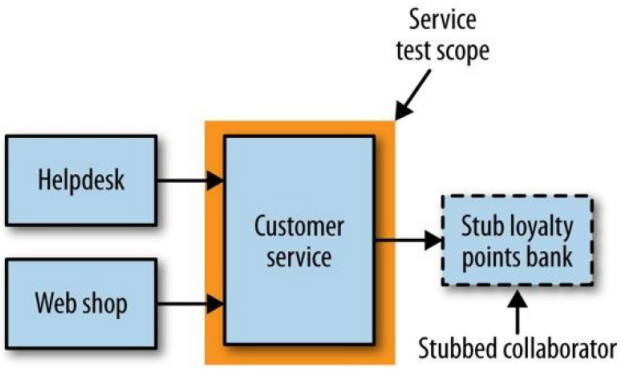


Figura 4: Las pruebas de servicio se ejecutan sobre un único servicio y emplean fakes para reemplazar a los colaboradores de este.

cambio en alguno de los participantes deberían ejecutarse todas las pruebas en las que este se emplea. Una manera elegante de conseguir esto es hacer que en la compilación de la integración continua, la fase de pruebas de extremo a extremo sea compartida entre todos los involucrados para que las pruebas se ejecuten cada vez que cambie uno de ellos y la versión a probar de todos sea la asociada al cambio.

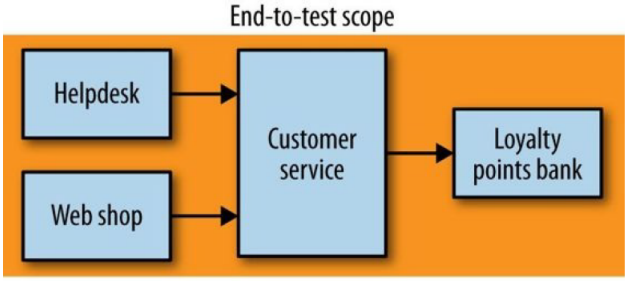


Figura 5: Las pruebas de extremo a extremo cubren una funcionalidad del sistema ejecutando los servicios que se involucran para ofrecerla.

### 1.3.1. Desventajas de las pruebas de extremo a extremo

- Pruebas frágiles: conforme el alcance del test aumenta mayor son las partes bajo prueba sobre las que no se puede tener control. Estas partes pueden introducir fallos que no demuestran que la funcionalidad tenga un defecto, sino fallan porque otro problema ha ocurrido como puede ser un fallo en la red. Las pruebas así son menos deterministas y el desarrollador puede acostumbrarse a que fallen si consigue hacerlas pasar al ejecutarlas por segunda vez, lo que se conoce como normalizar

la desviación. Este tipo de pruebas han de ser eliminadas cuanto antes de la suite de pruebas y refactorizadas en pruebas de menor alcance.

- Responsable de las pruebas: si los servicios que participan en las pruebas son mantenidos por distintos equipos, es necesario que todos ellos se hagan responsables de la suite de pruebas. Diferentes situaciones pueden ocurrir sino:
  - Explosión de pruebas: si nadie controla la salud de la suite, los equipos pueden añadir pruebas descontroladamente que no aporten valor.
  - Ignorar resultados de las pruebas: cuando una prueba falla, nadie trata de arreglarla porque cree que la responsabilidad la tiene otro.
  - Equipos especializados en pruebas: esto puede alargar el tiempo de desarrollo de una funcionalidad por requerir mayor coordinación entre equipos. Además, con esta metodología los desarrolladores se involucran menos en las pruebas, haciendo su código más difícil de probar e ignorando los resultados de las pruebas.
- Tiempo de ejecución mayor: al levantar un mayor número de servicios y cubrir más código se alarga el tiempo de ejecución de la prueba. Esto puede ser sobrellevado a través de la ejecución en paralelo de pruebas. Esto combinado con la fragilidad de los tests puede desencadenar que encontrar la causa del fallo de una prueba requiera de un mayor esfuerzo.
- Mayor tiempo de retroalimentación: si las pruebas tardan más en ejecutarse, el desarrollador tardará más tiempo en darse cuenta de que en los cambios realizados ha introducido un defecto. Este hecho puede repercutir en el resto del equipo, ya que si la compilación está rota debido a una prueba que no pasa, los cambios que otros suben pueden apilarse, dificultando su futura integración.

## 2. Balance de pruebas a realizar

Si volvemos a la pirámide de Cohn, uno de los conceptos clave es que a medida que aumenta el alcance las pruebas lo hace el nivel de confianza que las pruebas dan sobre la ausencia de defectos. Por otro lado, cuanto más arriba en la pirámide más tiempo tardará una prueba en ejecutarse y dar retroalimentación al desarrollador. Además, determinar el motivo de fallo de una prueba será más costoso cuanto mayor sean las líneas de código probadas.

El número de pruebas que se aconseja tener de cada tipo aumenta conforme descendemos por la pirámide. Un antipatrón sería seguir como convención la misma pirámide invertida. En este caso, nuestra cobertura de pruebas se basaría en pruebas de largo alcance y larga ejecución, nada deseable según los principios de integración continua.

En cuanto a las pruebas de extremo a extremo, no podemos añadir una prueba de este tipo por cada nueva funcionalidad que el sistema ofrezca. El número de servicios que participan para ofrecer una funcionalidad puede ser muy alto y en una prueba no se deberían de levantar más de 3 o 4 servicios para no potenciar las desventajas que hemos mencionado. Si es necesario, se pueden emplear fakes o stubs para no levantar a todos los servicios en la prueba.

## 3. Pruebas dirigidas por el consumidor

Durante las pruebas queremos asegurarnos que desplegar un cambio en un microservicio no rompe ninguno de sus consumidores. Esto se puede verificar mediante contratos o consumer-driven contract (CDC). Un contrato representa las expectativas que un consumidor tiene sobre un servicio. La manera de asegurarnos de que no se va a romper ningún consumidor es no desplegando cambios que puedan romper el contrato entre ambos.

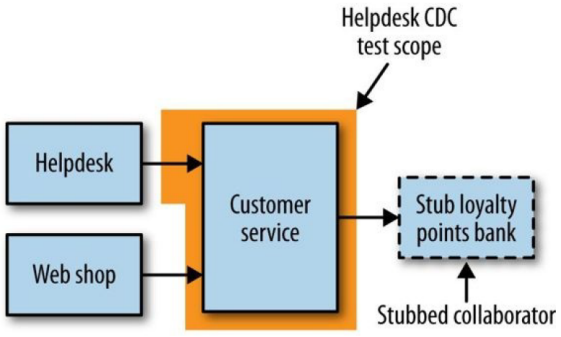


Figura 6: Dentro de la pirámide de Cohn, las pruebas dirigidas por el consumidor se sitúan al mismo nivel que las pruebas de servicio.

Para la elaboración de un contrato se puede contar con el equipo del servicio que va a consumir este. La colaboración entre ambos equipos también es fundamental si es necesario hacer un cambio que rompe el contrato para que el consumidor este preparado. En casos donde el servicio con el que se integre sea de terceros, la comunicación cuando se rompe un contrato no podrá ser tan frecuente como la deseada.

# 4. Pruebas antes de producción

Muchas organizaciones invierten en pruebas durante los días previos a publicar una versión a producción. Los usuarios usan nuestro sistema en formas que no podemos preveer, por lo que no siempre podemos detectar todos los errores de nuestro código. La reacción habitual cuando se produce un fallo es añadir más pruebas para evitar que vuelva a ocurrir. Sin embargo, es imposible garantizar una aplicación real libre de fallos.

Una forma de capturar más errores es extender los pasos del despliegue donde tradicionalmente las pruebas se ejecutan. Un ejemplo es una suite de pruebas humo (en inglés, smoke tests), una colección de pruebas de rápida ejecución que se ejecuta en cada nuevo componente desplegado para garantizar que el despliegue ha ido correctamente. Otras aproximaciones son los despliegues azul y verde y las canary releases.

## 4.1. Despliegues azul y verde

Este proceso consiste en contar con dos copias del software desplegadas al mismo tiempo, aunque solo una de ellas recibe realmente peticiones. Cuando se despliega una nueva versión de un servicio, se realizan pruebas de humo sobre la nueva versión y una vez se verifica que las pruebas pasan, se redirige el tráfico de la vieja versión a la nueva. La vieja versión se mantiene desplegada pero sin aceptar peticiones por si se tiene que redirigir de nuevo el tráfico debido a un error en la nueva versión.

Para seguir esta aproximación se debe tener la infraestructura suficiente para dirigir el tráfico hacia diferentes hosts y tener los suficinetes nodos para ejecutar diferentes versiones de un mismo servicio. Como ventajas encontramos que reduce el riesgo del despliegue porque nos permite revertir un cambio si se detecta un problema, proceso que se puede automatizar para mayor eficacia.

## 4.2. Canary releases

Empleando esta aproximación verificamos la nueva versión de un servicio, dirigiendo hacia él algunas de las peticiones que se realizan a la versión en producción para ver si se comporta de la manera esperada. El comportamiento esperado se puede medir a través de métricas como el ratio de errores. El mayor reto de este modelo es el enrutamiento del tráfico a una u otra versión.

Si la nueva versión no funciona correctamente, el cambio se puede revertir rápidamente. Sino, se puede incrementar progresivamente el tráfico real que atiende. Netflix emplea esta aproximación en muchos servicios.

Existen dos vertientes posibles: dirigir parte del tráfico real a la nueva versión o duplicar el tráfico que el servicio recibe, de tal forma que la nueva versión recibe peticiones pero sus respuestas no son vistas por el invocador.

### 4.3. Tiempo medio entre fallos y tiempo medio entre reparaciones

Muchas organizaciones invierten mucho esfuerzo en producir un gran número de pruebas funcionales y ningún esfuerzo en mecanismos para monitorizar o revertir el sistema cuando un fallo ocurre.

Se debe hacer un balance entre optimizar la media de tiempo entre fallos (MTBF) y la media de tiempo de reparación (MTTR) para alcanzar un equilibro. Mientras que el primero puede reducirse aunmentando el número de pruebas, el segundo puede mejorar con las técnicas que hemos citado antes y buenos sistemas de monitorización.

## 5. Pruebas de requisitos no funcionales

Los requisitos no funcionales representan aquellas características que el sistema exhibe pero que no pueden contabilizarse como una funcionalidad. Las pruebas que sobre estas características se hacen encajan en el cuadrante de Marick de pruebas de propiedades.

Su implementación puede hacerse de forma unitaria o de extremo a extremo, pero se aconseja seguir la relación de la pirámide de Cohn sobre el alcance y número de pruebas.

### 5.1. Pruebas de rendimiento

Las pruebas de rendimiento pertenecen al grupo de pruebas de requisitos no funcionales. Este tipo de pruebas es más importante en los sistemas basados en microservicios que en los monolíticos porque las integraciones entre los servicios (por ejemplo, invocaciones HTTP) pueden introducir un sobrecoste en el rendimiento si hay muchos servicios colaborando.

En estas pruebas se busca simular un entorno lo más cercano posible al

de producción, por lo que puede resultar útil contar con diferentes máquinas con las que ejecutar las pruebas. Debido al tiempo que pueden tardar en ejecutarse, no suele incluirse un estado para este tipo de pruebas en la compilación de cada cambio. En su lugar, suelen ejecutarse periódicamente. Esto puede producir que no se observen los resultados de las pruebas y esto puede resultar un problema si se pierde trazabilidad hacia el cambio que empeoró el rendimiento.