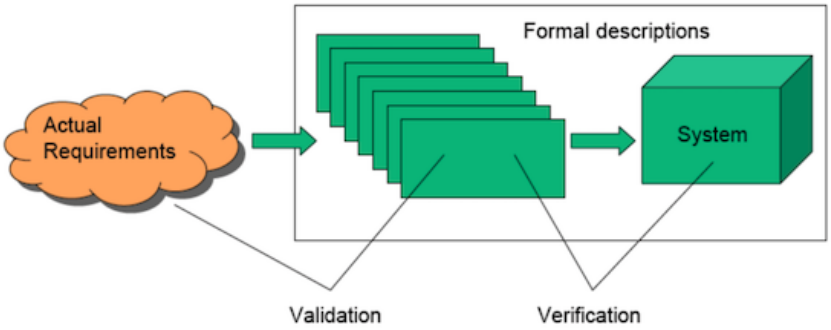
CHAPTER 12

Testing

En el capítulo sobre calidad hablamos de la necesidad de asegurar que el producto o servicio que hemos desarrollado sea el correcto: cumple con los requisitos funcionales y no funcionales. Con relación a ello distinguimos los conceptos de verificación (que el producto haga correctamente lo que se especificó) y validación (que el producto haga lo que el usuario o cliente realmente quería)



Existen varios caminos disponibles para lograrlo, de los cuales podemos usar algunos o todos los necesarios. Sin embargo, cualquiera sea la decisión, es altamente probable que en la batería se encuentren las técnicas de testing.

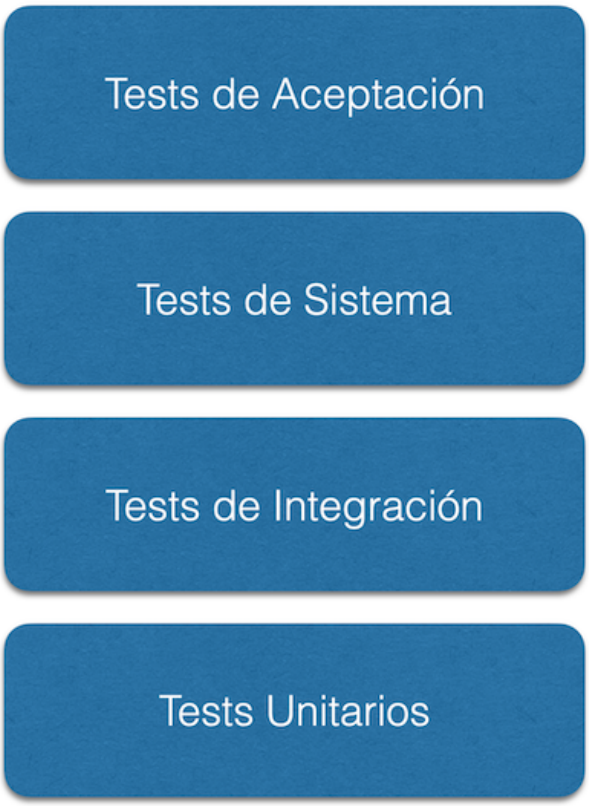
Recordemos todos los caminos que tenemos:

- No incorporar *bugs* al momento de construir el software - esto está asociado a lo que se denomina métodos formales, pero es impracticable en productos grandes y muy complejos
- Análisis estático del código - examinar el código en forma automatizada (con ayuda de herramientas de software) en búsqueda de potenciales problemas

- Inspección formal del código - examinar el código por un equipo de personas buscando posibles errores
- *Testing* - diseñar casos de prueba del software y someter al producto a ellos

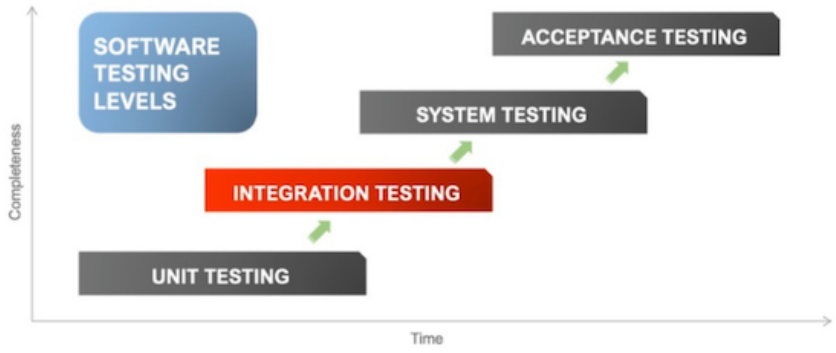
En este capítulo nos concentraremos en este último (*Testing*) por su enorme importancia y su amplia utilización en la industria.

Como sabemos, se trata de un proceso que busca encontrar problemas a través de someter a nuestro producto o servicio a casos de prueba cuidadosamente diseñados para maximizar la probabilidad de que sirvan para encontrar un problema.



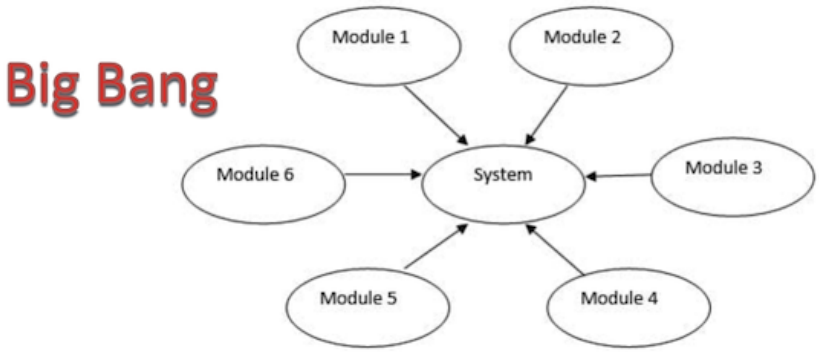
El testing se lleva a cabo en varios niveles, lo que constituye una jerarquía de *tests*. En el nivel más bajo están los *tests* unitarios. Estos se realizan a nivel de componentes elementales (clases, módulos) usando herramientas como RSpec en la plataforma Ruby u otras similares. El siguiente nivel comprende los *tests* de integración en que se trata de asegurar que dos o más componentes funcionan correctamente al ser unidas. El siguiente nivel corresponde a los *tests* de sistema que ponen el sistema completo como un todo a pruebas de distinto tipo (corresponde a una verificación completa con respecto a los requisitos). Finalmente, llegamos a los *tests* de aceptación que están asociados a la validación por parte de los usuarios de nuestro producto de software.

12.1 *Tests* de Integración

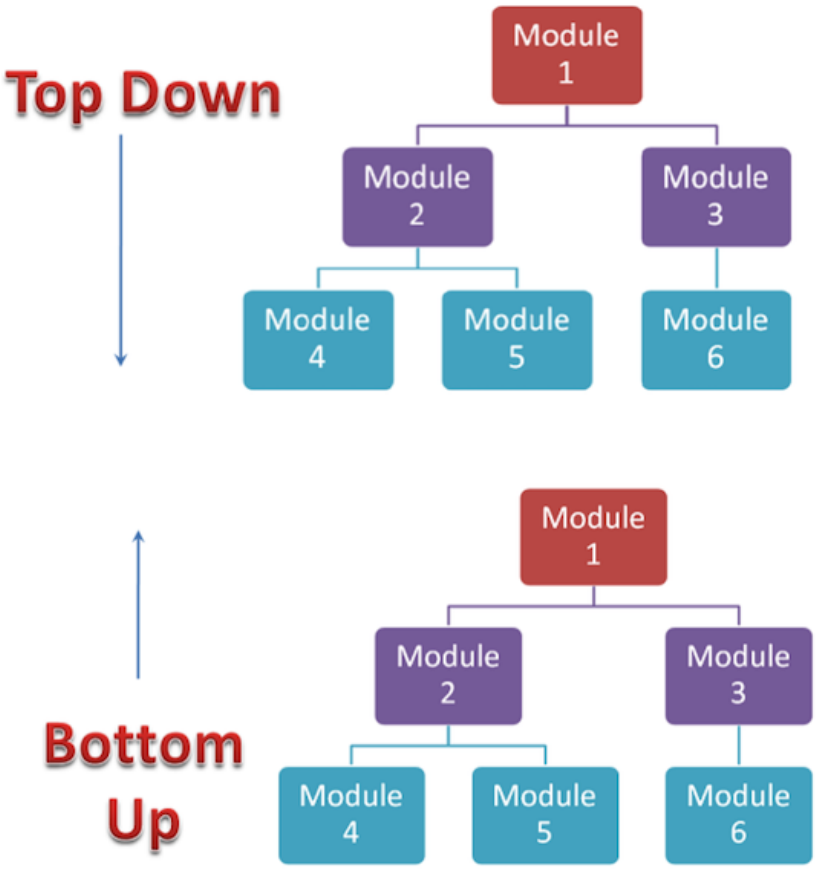


Este es el nivel que sigue inmediatamente después de los *tests* unitarios. El objetivo de los *tests* de integración es detectar problemas que no tienen las unidades en forma individual, pero que sí aparecen cuando se integran dichas unidades para formar una unidad mayor.

Recordemos que un producto de software se compone de muchas partes que son desarrolladas en forma independiente y que eventualmente deben ser ensambladas. La integración entonces es una etapa fundamental en el desarrollo de un producto de software.



Bajo un modelo de desarrollo más tradicional, la integración se llevaba a cabo cuando la mayoría de las componentes menores estaban ya desarrolladas y, por lo tanto, muchos defectos e incompatibilidades se descubrían muy tardíamente. El caso extremo de ello es la estrategia de integración que se conoce como de "*big bang*" que consiste en integrar todo al final.



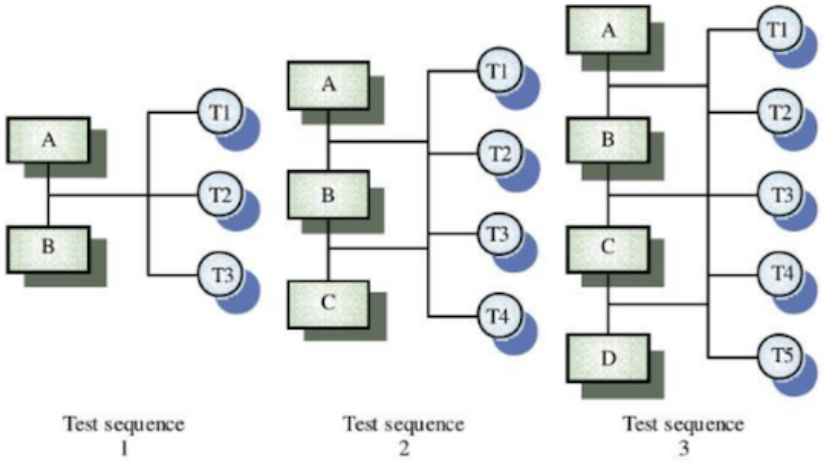
Una forma menos extrema es no esperar hasta el final, sino integrar en unidades mayores cuando se disponga de todas las que forman parte de él. Esta estrategia se conoce como "*bottom up*", porque el armado se hace desde unidades más chicas a más grandes. La estrategia opuesta "*top down*" requiere que mientras las componentes que constituyen una unidad mayor no estén disponibles, se usen "*stubs*" o componentes ficticias que simplemente llenan el espacio de las reales.

La popularidad de los métodos ágiles, y últimamente el movimiento *devops*, ha hecho que muchos equipos de desarrollo hayan adoptado un esquema de integración distinto denominado integración continua.

Bajo el enfoque de integración continua, los nuevos incrementos se integran al resto del producto tan pronto como estén terminados. En algunas organizaciones se realiza un "*build*" cada día. Por "*build*" se entiende una versión ejecutable del producto completo, en que se ha incorporado todo lo que se ha completado. Otros equipos pueden optar por hacer esto semanalmente o cada dos semanas (por ejemplo cuando se completa un sprint).

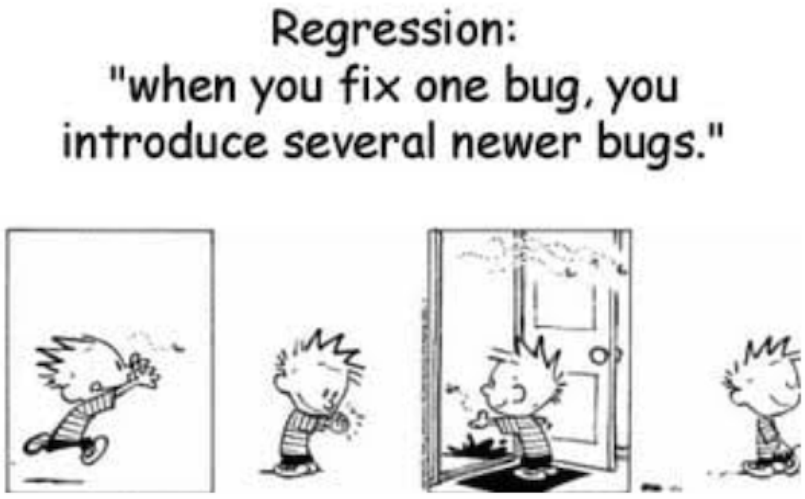
En cualquier caso, es fundamental que inmediatamente integrado un nuevo incremento o funcionalidad se hagan los tests de integración necesarios. La figura siguiente muestra una secuencia de desarrollo en que se integra

primero a con B, luego se integra C y luego D. Después de cada uno de estos incrementos es necesario correr tests que verifiquen una integración correcta.



12.2 *Tests* de Regresión

Cuando se usa una estrategia de integración continua es común hablar de *tests* de regresión. Básicamente, un *test* de regresión tiene por objetivo asegurar que después de agregar algo el sistema sigue funcionando como lo hacía antes. A pesar cd que ello es precisamente lo que necesitamos hacer cuando integramos un nuevo elemento al producto, la idea de test de regresión puede ser usada en otros escenarios. Por ejemplo, hemos reparado un error en una componente y queremos asegurar que lo que hemos arreglado no haya roto otras cosas



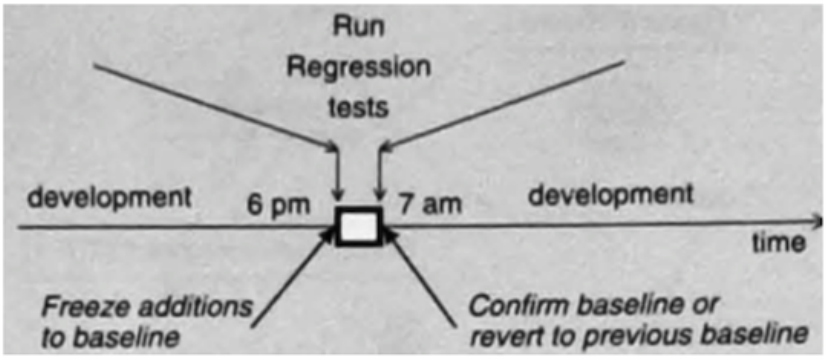
Un *test* de regresión consiste en la ejecución automatizada de un set de *tests* orientado a asegurarnos que podemos seguir adelante después del cambio

que hemos realizado, ya sea esté la corrección de un problema o la integración de una nueva funcionalidad.

Una parte clave de esto es la selección del set de regresión. Mientras más grande hay más seguridad, pero más tiempo tomará llevar a cabo todos los *tests*. Algunas opciones que suelen ser usadas son por ejemplo:

- incluir *tests* sobre nuevas funcionalidades
- incluir una muestra representativa que ejercite todas las funciones
- *tests* que se centren en las componentes modificadas
- las funcionalidades más visibles para el usuario
- las funcionalidades "clave:
- realizar todos los *tests* "complejos"
- realizar todos los *tests* de integración
- un porcentaje de *tests* deben fallar y otros deben pasar

Entre los desafíos adicionales está por ejemplo, determinar con qué frecuencia correr los *test* de regresión. El *set* de regresión se irá inflando a medida que avanza el desarrollo, por lo que va a tomar cada vez más tiempo. Es claro que al integrar una nueva funcionalidad debe hacerse, pero no necesariamente después de corregir un pequeño error.



Otro aspecto importante es que la realización de *tests* de regresión debe coordinarse y supone que existe un control de configuración que permita revertir los cambios en caso de que el *test* no pase. Por ejemplo, en una organización que use una estrategia de *builds* diarios puede congelar todos los aportes a la línea de base a una determinada hora para realizar los *tests* de regresión. Si pasan los *tests* se conforma una nueva línea de base, pero si fallan se revierte a la situación previa.

12.3 *Test* de humo (*Smoke test*) y *test* de sanidad (*Sanity test*)

Cuando el *test* de regresión ha crecido mucho, hay dos técnicas que permiten no tener que correr la suite completa innecesariamente. Por ejemplo, imaginemos que hacemos el *build* y notamos que ahora la aplicación no parte. Es

evidente que algo anduvo mal y no vale la pena correr toda la batería de *tests* completa.

La idea detrás del *smoke test* es precisamente esa. Si nos damos cuenta de alguna forma que no se ve bien parar de inmediato. Piensa en una placa de circuito a la que acabas de agregar una componente, lo energizas y observas humo. No necesitas saber nada más para detenerlo.

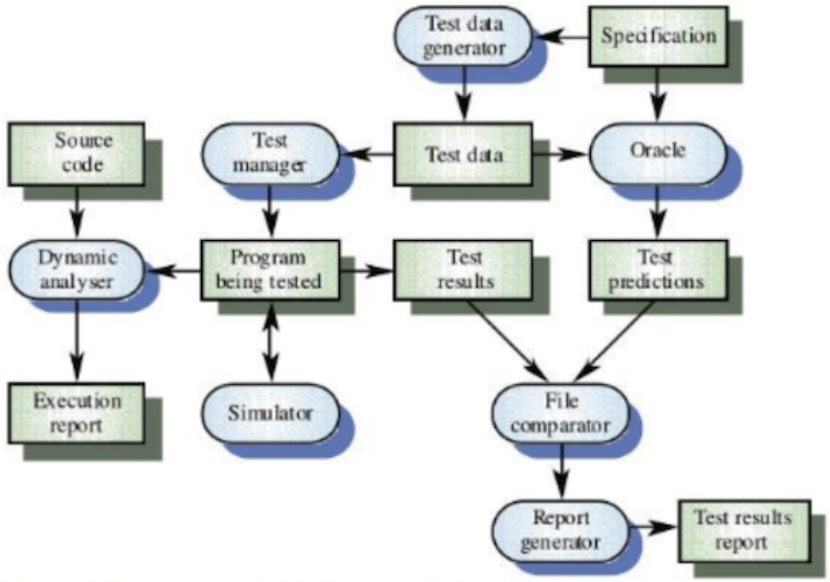
El *sanity test*, que se suele confundir con el *smoke test*, también sirve para ahorrarse el correr los *test* de regresión completo, pero es distinto. En este caso el *test* está asociado a la corrección de un error y asegurarse que por lo menos eso se hizo bien antes de correr los *tests* de regresión completos. Por ejemplo, había un error de precisión que hacía que 6*(1000/6) entregara 5.99. Antes de correr las pruebas para ver que no hemos roto otras cosas deberíamos asegurarnos que lo que estaba malo se arregló.

12.4 Automatización en *testing*

Ya hemos dicho que es imposible hacer *tests* de regresión a menos que tengamos ese trabajo automatizado. Es mucha pega que se debe hacer en forma muy frecuente, por lo que de hacerlo manual requeriría de muchos recursos humanos. Afortunadamente, existen muchas herramientas que permiten llevar a cabo la ejecución de un *test* y la verificación de sus resultados en forma completamente automatizada.

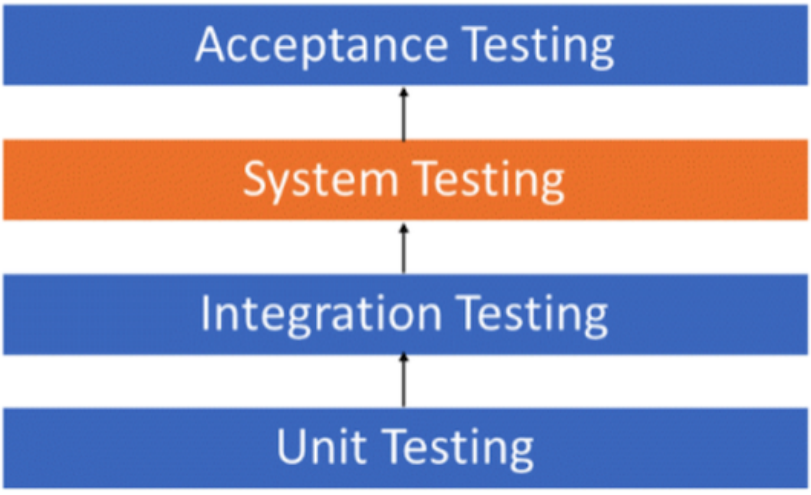
Por ejemplo, una herramienta encargada de testear el funcionamiento de una página web, es capaz de cargar la página, activar botones, ingresar texto en elementos de *input*, etc, e ir registrando todo lo que sucede para compararlo con lo que se espera que suceda. De la misma forma, podría someter a pruebas a una API con un *script* que llama una a una a todas las funcionalidades con *inputs* preparados registrando los resultados.

Una de las herramientas más ampliamente utilizadas con este fin es Selenium, especialmente si se trata de probar aplicaciones Web que incorporan una interfaz de usuario interactiva. Los *scripts* pueden ser escritos en distintos lenguajes de programación como Ruby, Python o C#. Alrededor de esta herramienta, que es *open source*, se ha creado todo un ecosistema de proyectos que incluyen soporte para los distintos *browsers*, *drivers* para distintos lenguajes e incluso *frameworks* de *testing* completos alrededor de Selenium.



Existen en el mercado además herramientas que caen bajo el concepto de "*testing workbench*", que corresponde a un ambiente completo para hacer *testing* que puede ser configurado con herramientas específicas y de acuerdo a las necesidades de cada organización.

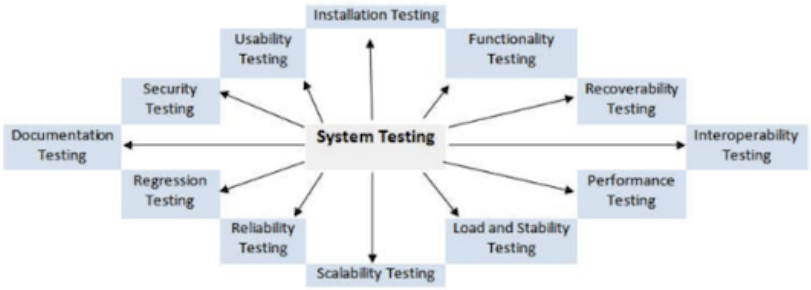
12.5 *Test* de Sistema



El *test* de sistema, también llamado *end-to-end*, está orientado a verificar el producto como un todo. Piensa en una línea de producción de automóviles donde se van incorporando las distintas partes del auto a medida que avanza en la línea. Hay muchos *tests* en las distintas etapas, pero una vez que el auto está completo se saca de la línea y es instrumentalizado y sometido a pruebas con conductores profesionales.

La idea en el *test* de sistema es la misma. Uno o más profesionales toman el producto construido en forma completa y lo someten a una gran cantidad de pruebas que tienen que ver con asegurar que el producto satisface los requerimientos funcionales y especialmente los no funcionales. Debido a que hay distintos tipos de requerimientos no funcionales (desempeño, usabilidad, etc), existen muchos tipos de pruebas a nivel de sistema completo.

La figura muestra algunos posibles tipos de *tests* a que puede ser sometido el producto en revisión



Por ejemplo, una prueba de sistema para un sitio web de un *e-commmerce* puede incluir los *tests* que muestra la figura.

- If the site launches properly with all the relevant pages, features, and logo
- If the user can register/login to the site
- If the user can see products available, he can add products to his cart can do payment and can get the confirmation via e-mail or SMS or call.
- If the major functionality like searching, filtering, sorting, adding, changing, wishlist, etc work as expected
- If the number of users (defined as in requirement document) can access the site simultaneously
- If the site launches properly in all major browsers and their latest versions
- If the transactions are being done on the site via a specific user are secure enough
- If the site launches properly on all the supported platforms like Windows, Linux, Mobile, etc.
- If the user manual/guide return policy, privacy policy and terms of using the site are available as a separate document and useful to any newbie or first time user.
- If the content of pages is properly aligned, well managed and without spelling mistakes.
- If session timeout is implemented and working as expected
- If a user is satisfied after using the site or in other words user does not find it difficult to use the site.

Es importante destacar aquí que el *test* de sistema no es lo mismo que un *test*

de aceptación. La diferencia fundamental es que en el caso del *test* de sistema estamos hablando de verificación, en cambio, con validación en el caso del *test* de aceptación. Por ello, el *test* de sistema es realizado por profesionales o gente especialmente preparada para ello, a diferencia del *test* de aceptación que se lleva a cabo con usuarios o clientes.

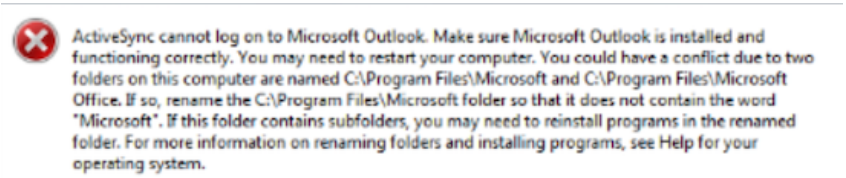
12.6 *Test* de Carga y *Test* de Esfuerzo (*Load Test* y *Stress Test*)

El *test* de carga es fundamental. Si yo construyo un puente que debe resistir hasta 10 toneladas, entonces hay que asegurar que resista 10 toneladas.

El *test* de *stress* no siempre se hace, pero es muy recomendable hacerlo y en este caso se trata de someter a nuestro sistema a una carga muy por encima de la carga para la que fue diseñado. Un ingeniero de puentes puede estar abriendo los ojos al máximo, pero en el caso de productos de software tenemos la gran ventaja de que el costo de "romperlo" es muy bajo. Dado que no importa romperlo puede resultar muy útil saber qué pasaría en condiciones extremas.

Otra diferencia fundamental en el diseño de un puente y un sistema computacional es que en el primer caso no va a ocurrir nunca que se cargue un puente con 10 veces la carga máxima para la que fue diseñado. En el caso de un producto de software, esto no es tan poco frecuente. Por ejemplo, una promoción lanzada por el área de marketing o un rumor en las redes sociales puede elevar rápidamente el tráfico en un sitio web mucho más de 10 veces lo normal.

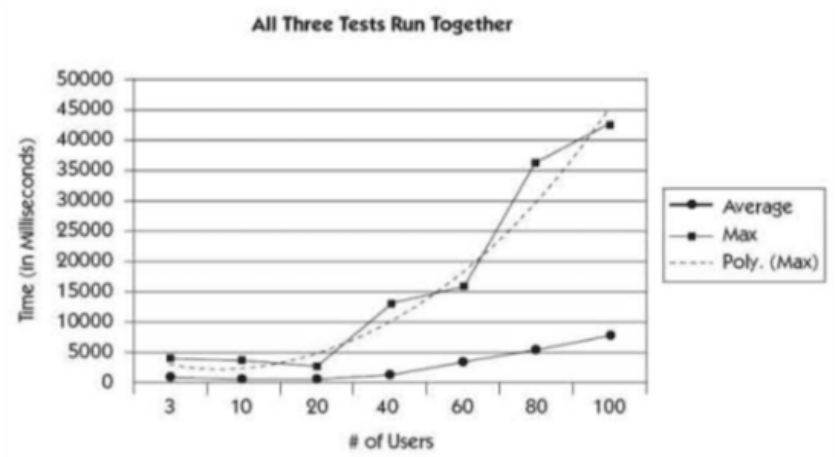
Hay muchas maneras en que una aplicación va a responder a una carga para la que no fue diseñada. Por ejemplo, podría bloquearse completamente, caerse con una pantalla azul y un mensaje incomprensible o hacerlo de un modo más "amable". El ver que es lo que sucede en estos casos nos permite evitar daños en la reputación de la empresa (caídas feas), detectar mensajes poco apropiados (a veces el programador lo cree tan poco probable que pone cualquier cosa)



Los objetivos del *test* de *stress* son entonces:

- Observar que el sistema falle "graciosamente"
- Observar la capacidad de recuperación del sistema

Aunque el *test* de *stress* es distinto al *test* de carga, pueden usarse las mismas herramientas para construir los tests. Lo único diferente son las condiciones de la prueba. En un *test* de carga estaremos registrando por ejemplo el desempeño del sistema bajo condiciones de carga máxima.



Para poder hacer este tipo de pruebas se requieren herramientas de software que permitan simular los escenarios que queremos probar: un número de usuarios simultáneos, un número de transacciones por segundo, etc. Estas herramientas no solo permiten definir el escenario y ejecutar el *test*, sino que además permiten registrar los resultados en forma de reportes (ver figura).

Ejemplo de estas herramientas son LoadRunner, Jmeter, y Neoload. La siguiente tabla muestra un comparativo de algunas de las características de estas herramientas.

Neoload vs LoadRunner vs Jmeter

Let's explore the features of Neoload, LoadRunner, and JMeter using the below table.

Feature	Neoload	LoadRunner	Jmeter
Latest Version	6.2	12.53	3.3
Scripting Language	Java script/java	C/Java/Java script	Groovy/Java
Script maintenance	Easy to use user path maintenance feature which reduces redesigning/re-scripting effort by 40 to 50%	Not available	Not available
Correlation	Automatic and easy - Session and cookies handling is not required - Frameworks for all protocols - No scripting involved	Automatic - Rules need to be established first - Manually set parameters using scripting language	No Automatic Manual correlation only
Enable/Disable Script Sections	Select any element(s) with a right-click	Type characters "/" to be entered or blocks of script with "*/"	Select any element(s) with a right-click
Continuous Integration	Jenkin- with custom graphs Hudson, Bamboo & Team city	Jenkin No custom graphs	Jenkin, Bamboo and Hudson No custom graphs

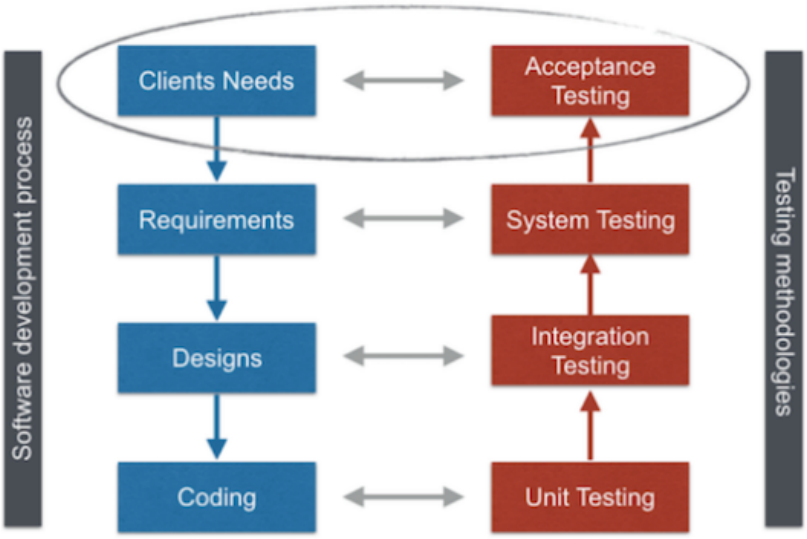
12.7 *Tests* de Usabilidad

Los *test* de usabilidad corresponden también a *tests* de sistemas porque son realizados por profesionales al producto ya terminado. Básicamente, se trata de observar y registrar la forma en que un grupo de usuarios lleva a cabo algunas tareas específicas utilizando el software. Para la observación puede usarse una sala que tiene un vidrio que es espejo por el lado interno y transparente por el otro (como en las películas de policías) o también cámaras, *trackers* de donde pone la mirada la persona, etc.

La sesión debe ser planificada con anticipación y todo debe ser registrado para estudiarlo posteriormente. No se trata de hacerles una encuesta a los participantes para que le pongan una nota al software o si digan si es "amistoso". Se trata de que tan fácil les resultó llevar a cabo las tareas para las cuales el software fue pensado. Por ejemplo, cuantos *clicks* les tomo hacer una compra a través de la página, cuantas veces hizo *click* en lugares incorrectos, cuanto se demoró en total, etc. Puede también registrarse una opinión general de los usuarios, pero es mucho menos relevante que la data objetiva recogida al usarlo.

Se ha estudiado el número de usuarios que debe usarse para estudiar usabilidad y se ha llegado a la conclusión que un número relativamente pequeño (5 a 8) es suficiente para tener una buena idea del grado de usabilidad y también de algún problema serio que debería ser corregido.

12.8 *Test* de Aceptación



El último nivel en la jerarquía de los *tests* y el único que permite hacer una validación del producto porque implica satisfacción del usuario.

Este *test* permite identificar problemas que no se han detectado antes porque el sistema fue probado por profesionales o ver si la usabilidad evaluada como buena por los *tests* corresponden a la percepción de los usuarios reales del producto.

Existen distintos tipos de *test* de aceptación:

- *Alpha Testing* - en el ambiente de desarrollo realizado por staff interno o grupos seleccionados

- *Beta Testing* - en el ambiente del cliente realizado por los mismos clientes (un grupo de ellos)
- *Contract Acceptance Testing* - se chequea contra lo especificado o acordado en un contrato
- *Regulation Acceptance Testing* - satisface las regulaciones legales o gubernamentales
- *Operational Acceptance Testing* - están todos los procedimientos asociados para la operación (*backups*, entrenamiento, mantención, etc)

La siguiente tabla permite comparar los *test* 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	*Validation*	Verification

12.9 Otros *tests*: *A/B Testing*

Este es un *test* que podría ser considerado como un *test* de aceptación, pero por la orientación y el uso cae en una categoría especial. El nombre A/B se debe a que en esencia se evalúan dos versiones distintas del producto (versión A y versión B) con un grupo de usuarios típicos y se observa cuál resulta más efectivo de acuerdo a alguna métrica. La decisión final del producto es netamente objetiva: aquella en que la métrica marque mayor puntaje.

Este tipo de *test* es muy utilizado con páginas web donde se quiere lograr por ejemplo, que la mayoría de las visitas se traduzcan en compras (factor de conversión). Se pone en línea entonces las dos versiones y se permite que un 50% de los usuarios tenga acceso a la versión A y un 50% a la versión B. Después de un determinado período de tiempo se comparan los resultados para ver cuál funciona mejor.

Es común encontrarse con grandes sorpresas aquí y muchas veces la versión que la mayoría de los diseñadores considera como “la mejor” termina siendo mucho menos efectiva que la otra.


50 % visitors
see variation A
Variation A
23%
conversion
50 % visitors
see variation B
Variation B
11%
conversion